FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Vlastimil Dort

Read-only Types and Purity for DOT

Department of Distributed and Dependable Systems

Supervisor of the doctoral thesis:	doc. RNDr. Pavel Parízek, PhD.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2024

I am grateful to assoc prof. Pavel Parízek for supervising my PhD studies and this thesis, prof. Ondřej Lhoták for supervising my visit to University of Waterloo in Canada and introducing me to DOT, Yufeng Li for collaborating on mechanizing the roDOT calculus in Coq, and Ifaz Kabir for allowing us to use his version of the mechanization as the baseline and providing us help.

I am also grateful to my colleagues at the Department of Distributed and Dependable System for a friendly environment and support.

Title: Read-only Types and Purity for DOT

Author: Vlastimil Dort

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, PhD., Department of Distributed and Dependable Systems

Abstract: Mainstream object-oriented programming languages, such as Java or Scala, typically allow objects to be mutated by assigning new values to their fields, but it is also common to write code that only accesses objects in a read-only way. Reference mutability is a technique for controlling mutation by distinguishing read-only and mutable references with types. It has been thoroughly studied in Java, and implemented as compiler extensions. Scala is an evolving programming language which integrates many advanced type system features, most notably path-dependent types. To address questions about soundness, the formal Dependent Object Types (DOT) calculus has been developed, which provides a formal proof of soundness for the core of Scala's type system.
In this thesis, we explore the possibility of using DOT's features to integrate reference mutability. We define the roDOT calculus, which is based on a version of DOT with mutable fields, and encodes the mutability of object references using a special type member. This encoding makes it possible to use path dependent types to refer to mutability of a reference, and use intersection and union types to combine mutabilities and implement viewpoint adaptation, ensuring the transitive property of read-only references. In addition to updating the type soundness proof of DOT to this extension, we state and prove the Immutability guarantee, which formally states that objects in roDOT can be only mutated through the use of mutable references.
Modern code also often makes use of pure methods, which have properties of mathematical functions. The ReIm type extension for Java showed the connection of reference mutability to purity of methods, in particular, the side-effect-free property. We explore purity conditions in roDOT and pose the SEF guarantee, by which the type system guarantees that methods that can be typed with read-only parameters are side-effect free. Applying the ideas of ReIm to roDOT required just a few changes to the type system, but necessitated re-working a significant part of the soundness proof. We proved the SEF guarantee by applying the previously stated Immutability guarantee.
In addition to the SEF guarantee, we state a transformation guarantee, which ensures that in a roDOT program, calls to SEF methods can be safely reordered without changing the outcome of the program. The transformation guarantee is proven by applying the SEF guarantee within a framework for reasoning about safe transformations of roDOT programs.
We mechanized the definition of roDOT, the soundness proof and all the guarantees using the Coq proof assistant. As a demonstration of the possibility of bringing the ideas of roDOT to Scala, we provide a patch to the Dotty compiler, which implements basic reference mutability checking.

Keywords: type systems, dependent types, DOT calculus, reference mutability, purity

Contents

1. Introduction

Computer programming is an established endeavor of modern society, making it possible to automate common tasks and achieve higher speed and precision in many fields.

Since the early days, computer programming relies on the use of programming languages, which define textual representations of programs that are both readable by humans and executable by computers.

Over the years, the programming languages have evolved in many directions. The current trends in the evolution of programming languages include:

- Avoiding programmer errors – by the means of ahead-of-time (compile-time) checks which ensure that program code conforms to rules set by the programming language or assertions expressed by the programmer. A typical example is the use of a type system, which ensures that values of some type (such as integers) in the program are used in the intended way, and such values are not confused with different types of values, such as strings.

- Avoiding repetition and boilerplate – by allowing writing code which is reusable in multiple contexts. Examples of that are the polymorphism of objects in object-oriented languages, separation of interface and implementation, and generic programming. Common patterns of design and implementation are getting integrated into the languages – lambda expressions in C++, C# and Java; record types in C# and Java; asynchronous programming in C++ and C#.

- Producing more efficient machine code – by the means of optimizations performed by an ahead-of-time compiler or a JIT compiler. An optimizer can reorganize the program code to allow fast execution, but must operate within the boundaries set by the semantics defined by the programming language.

  On one hand, programming languages rarely provide any guarantee about execution time, letting the programmer focus on good use of provided abstraction and leaving the issue of fast execution to the compiler. On the other hand, it is recognized that high performance code requires changes in programming style, such as using value types instead of reference types. Recently, the expressivity of value types is being expanded in C#, and there is ongoing work to bring value types to Java. In C++, move semantics allowed defining operations in a class that makes handling its instances more efficient, shifting the burden of caring about efficiency from users to implementers of a class.

While in the past and for some languages still today, the languages were steered by private parties (developer decision or companies decision, often as customer demand) nowadays, major languages have put in place a community based process, where language proposals are posted, reviewed and voted on [5, 7, 6].

While design decisions of the past are sometimes regretted, this process allows involvement of multiple parties including researches. Research can be used as a base for, or to evaluate and provide critique for proposals.

When a extension of a type system is proposed (recently explicitly nullable types, safe references), it is expected that the type system will provide certain guarantees to its users, but understanding all the implications is not easy. Formalization of a language or feature can provide hard evidence of the claimed properties.

This thesis aims to advance the development of object-oriented programming languages and their formalization, In particular by extending the DOT calculus, a formalization of the core of type system of Scala, by features associated with read-only references.

In the following sections 1.1 to 1.3, we provide a high-level overview of relevant aspects of object-oriented programming, type systems and formalizations of type systems.

1.1 Object-Oriented Programming

Object-oriented programming is one of the most popular programming paradigms, being at the core of the most popular languages Java, C#, JavaScript and Python.

While being so popular, the languages gathered lots of criticism [30, 105, 32]. Recently, there is an increasing effort to address their deficiencies by improving the languages, or designing and implementing new languages.

In particular, functional programming is gaining popularity, and object oriented languages react by incorporating FP concepts, such as lambda expressions and algebraic types.

The Scala programming language is an object-oriented language compatible with Java includes many concepts from Functional programming [82].

In this thesis, we use Scala as the practical programming language in which we advance the formalization effort.

1.2 Type Systems

Withing the space of programming languages, there is a divide between statically typed and dynamically typed languages [27, 72]. The advantages of static typing are the ability to keep track of the possible values of variables, parameters and fields. If used properly, this may lead to:

- **More efficient code.** The compiler can generate more efficient code for handling statically typed data, because it can handle things more specifically. On the other hand, in a dynamically typed lang, the type must be checked at run-time to determine which action should be taken and if it is valid.

- **Prevention of errors.** Type checking can detect mismatched expectations between components. In dynamic languages, a value of a wrong type can be passed to a function, and such a mistake can cause problem at a later point in the program, making it difficult to find the cause of the problem.

- **Structuring code.** Using types can help divide the problem space by focusing on handling a particular type of values in a given piece of code.
- **More self-documented code.** The types aiding understanding of the code [73] and allowing development tools to provide targeted assistance.

Disadvantages of static types are the following:

- **Lack of expressiveness.** The programmer is limited in their ability to express the intent by the available features in the type system. Such was the case in Java and C# before introduction of generics, where explicit casts to the actual type had to be used when working with collections.
- **Complexity.** When the type system is too powerful, it may become hard to understand. Reported errors may be incomprehensible, such as errors involving templates in C++.
- **Annotation burden.** In order to provide precise typing information, programmers are required to write more code, which may be repetitive and seem unnecessary. This problem is addressed by automatic inference of types, which allows keeping the advantages of static typing without additional effort. Type inference for local variables has been added to C++, C# and Java. Generic and template arguments are also inferred at function calls. In lambda expressions, the types of parameters can also be inferred.

In the simple view, types specify the set of values that can appear at a particular point in the program. Types can also be used in more advanced ways, to check additional properties, and to specify the effects [78]. The type system provides certain guarantees that the programmers can rely on.

Avoiding runtime errors is one of the advantages of a static type system. The type systems of practical programming languages such as Java and Scala are not sound in the sense that it is possible to craft a program that will pass type checking in the compiler, but fail with an error at runtime (in the case of Java and Scala, fail run-time type checks at the level of the JVM).

The above is not surprising. Practical languages usually provide ways to intentionally circumvent type checking by means such as type casts, dynamic invocation, or run-time reflection. Although runtime errors caused by using such constructs are common, it is usually considered to be an acceptable risk tied to using such features. Prevention measures include static analysis tools, general discouragement of using the features unless really needed, and encouraging extra care and testing. It is usually not the plan of language designers to fix such unsoundness at the level of the type system.

In the case of JVM-based languages, there are actually two levels of type systems – of the surface language (Java, Scala) and the JVM itself.

If the type system of Java or Scala is unsound, it can lead to a run-time exception (invalid cast or missing method), but thanks to run-time checks performed by the JVM, it would not lead to a crash or breach of security by corrupting memory. Nevertheless, this situation is highly undesirable, because it goes against the expectation of programmers and could potentially lead to incorrect behavior of the program.

While programmers understand that usage of casts and reflection can lead to run-time type errors, they might expect that programs which contain no casts or reflection will be safe. It has, however, been shown that the type systems of Java and Scala are unsound even for such programs.

Although making such problematic programs may require using various trick requiring good knowledge of the language, their existence means that there is no real guarantee provided by the type system. Also, adding more and more complicated features into the language may open more soundness holes and make it more likely for programmers to stumble upon them.

Having a formal definition with a proof of soundness ensures, that using the features modeled by the calculus will never lead to runtime errors.

With increasing complexity of the type systems, it becomes harder to see if the type system provides the expected guarantees.

```
class Graph{
  // Class Node is a type member of class Graph
  class Node{ ... }
  // Type Node is dependent on the containing object
  def createNode(): Node = ...
  def connectNodes(a: Node, b: Node): Unit = ...
}
class Searcher{
  // Using path-dependent types to refer to a type member
  // of another object
  def shortestPath(g: Graph, source: g.Node, target: g.Node) = ...
}

val g: Graph = new Graph
// Using a dependent type to specify the containing object
val a: g.Node = g.createNode()
val b = g.createNode()
// This call is allowed, because both nodes belong to graph g
g.connectNodes(a, b)
// This call is allowed, because both nodes belong to graph g
new Searcher().shortestPath(g, a, b)

// To demonstrate how dependent types can prevent errors,
// node c belongs to a different graph
val g2: Graph = new Graph
val c = g2.createNode()
// Attempting to connect nodes from different graphs
g.connectNodes(a, c) // Type mismatch g.Node != g2.Node
```

Figure 1.1: Example of dependent types in Scala

1.3 Formalization of Scala

Scala is an object-oriented programming language, built on top of the Java Virtual Machine and integrating improved features of Java with functional concepts.

The type system in Scala includes several modern features, the distinctive feature being dependent types.

Dependent types are based on allowing objects to contain type members, and allow constructing a type by selecting a type member from an object identified by a path. Type members are an object-oriented analogue of generic type parameters.

In the example in Figure 1.1, the inner class `Node` is actually a type dependent on the containing object.

The expressiveness of dependent types allow better specifying the intent, preventing errors. For example in Figure 1.1, type checking prevents passing nodes that do not belong tho the same graph into `connectNodes`.

Also, they can be involved in implicit lookup, allowing programmers to omit

```
// Array types are covariant in Java
Object[] array = new String[1];
array[0] = 1; // Run—time error

// It is normally not possible to create a value of type Nothing
// However, array allocation allows creating
// an arrray with elements of any type initialized to null
val x = new Array[Nothing](1)
// x(0) should be null, but also have type Nothing
val y: Int = x(0) // Run—time error
```

Figure 1.2: Example of run-time type errors in Java and Scala

un-interesting pieces of code which the compiler can fill in automatically based on the types.

However, in Scala, unsoundness has been observed involving dependent types [15]. Simple examples of code producing run-time type errors are shown in Figure 1.2. If the unsoundness is not well understood, that could lead to errors or wrong code.

That lead to formalization efforts, producing the DOT calculus.

The DOT calculus [18] is a formal mathematical calculus which encodes selected and simplified features of the Scala types system and semantics. The development of the calculus and a particular version used in this work will be described in Section 2.3.

1.4 Mechanization

Formalization can be done in traditional “manual” style of definitions and proofs. That is error-prone and long.

For type systems, which are in several variants, leads to repetition and low ability to reuse proofs.

The calculi are usually based on inductive definitions: programs are composed of parts like an abstract syntax tree. Also the process of typing a program can be expressed using inductive rules, where the typing of the program is composed of typing its parts.

Coq [26, 2] is a system for writing and automatically checking proofs, with focus on inductive definitions and proofs. It is common to mechanize program calculi and prove the properties of type systems and automated program analyses in Coq [39, 101, 70].

1.5 Goals and Contribution

The objective of this thesis is to further extend the understanding of advanced type-system features in object-oriented programming languages by means of formalization of program calculi, which can provide proven and mechanically verifiable guarantees.

In particular, this thesis extends the DOT calculus, a formalization of the core of the Scala programming language, with features related to controlling reference mutability. The result is a definition and mechanization of the roDOT calculus and guarantees about type safety, object immutability and side-effect-freedom.

This thesis presents these contributions to the development of DOT calculi:

- Extending the DOT calculus with reference mutability, based on permissions encoded using member types.
- A statement and proof of the Immutability guarantee.
- Encoding side-effect-free methods.
- Side-effect-free guarantee.
- Encoding safe transformations.
- Particular safe transformations based side effect guarantee.
- Mechanization of all the above contributions in Coq.

1.6 Outline

This thesis consist of two major parts, describing the work related to read-only references and side-effect-free methods respectively.

Chapter 2 introduces basic concepts used in this work, gives an overview on the DOT formalization efforts, and presents a full version of a baseline calculus, which is based on prior work and is used as a base for this work.

In Chapter 3, we present the roDOT calculus, which incorporates the feature of reference mutability into DOT. We discuss the requirements, and individual changes that are used to implement it. Then, we update the proof of safety, and present the immutability guarantee (the main guarantee of reference mutability) and prove it. The content of this chapter is based on the paper Reference mutability for DOT by Vlastimil Dort and Ondřej Lhoták, presented at the 34th European Conference on Object-Oriented Programming (ECOOP), 2020 [43].

In Chapter 4, we extend the roDOT calculus with a way to reason about side-effect-free (SEF) methods. We discuss different interpretations of the side-effect-free condition, and identify changes to the roDOT calculus required to provide guarantees about side-effect freedom. Because of the changes, we update the proof of safety once again, and present new guarantees (SEF guarantee and transformation guarantee) and prove them. The content of this chapter is based on the paper Pure methods for roDOT by Vlastimil Dort, Yufeng Li, Ondřej Lhoták and Pavel Parízek, which has been accepted to appear at the 38th European Conference on Object-Oriented Programming (ECOOP) in September 2024 [44].

In Chapter 5 we describe our experience with our attempt to implement reference mutability in the Dotty compiler for Scala. We describe relevant internals of the Dotty compiler related to the implementation, and provide a patch for the Dotty compiler that is able to check reference mutability in simple cases. Finally, we looked at the Scala collection library to identify the patterns that would emerge when applying reference mutability types in Scala code.

2. Background

In this chapter, we present the necessary background for our thesis.

In Section 2.1, we provide a brief overview of modern type system features, which are relevant to this thesis. In Section 2.2, we explain the most important concepts and terms used in this thesis. In Section 2.4, we present the *baseline DOT*, a formal calculus, and that we use as a base for formalizing type system features related to read-only types. In Section 2.5, we discuss how a formal calculus like DOT is mechanized – implemented in a machine-checkable form, where its stated properties can be automatically verified.

2.1 Type Systems

Type systems [90] in programming languages allow generating efficient code dealing with particular kinds of data, and compile-time checking of correct usage of data. In object-oriented languages, the type system is based on a hierarchy of classes.

2.1.1 Advanced Type System Features

In memory-safe object oriented languages, such as Java, C# or Python, it is possible to use an object at runtime without knowing its type at compile time. For example, a list data structure can work with any type of elements. Often, however, the type of the element is known, so the compiler can check correct usage of the elements. To help with these common scenarios, type systems are extended with advanced features that allow more precise information about the types of objects to be expressed within the language.

Intersection types Often, we don't use the precise type of an object, but just one its supertypes, which puts an upper bound on the type, and allows working with the object to the extent of the supertype. If more than one such upper bounds is known, they can be used together in an intersection type. In Java, intersection types are limited to occur in generic bounds.

```
// Intersection type in Java
<T> void m(List<? extends String & T> l){ ... }

// Intersection type in Scala
def m(x: Comparable & Serializable) = { ... }
```

Intersection types are useful in formalization of object oriented languages, because the allow expressing a type containing multiple members as an intersection of single-member declarations. In Java, the full list of members is known at object creation time, but in Compositional Programming disjoint intersection types [86], values can be merged arbitrarily.

Union types In Java, union types are limited to exception handler clauses, where they allow handling multiple exception types with common code. In Scala

3, union types are first-class types. The compiler, however, avoids inferring union types, as that could lead to blow up the size of the type.

```
// Union type in Java
try { ... }
catch(Exception1 | Exception2 e) { ... }

// Union type in Scala
def m(x: Int | String) = { ... }
```

Union types may have further properties of interest: **disjointness**, when the branches of the union are disjoint types, so the actual type of the object uniquely determines the active branch of the union, and **exhaustivity**, when a union covers all possible subtypes of a superclass in a sealed class hierarchy.

Top and Bottom types In a type system with subtyping, it is useful to have two special types – one that is a subtype of all types, and one which all types are a subtype of.

The top type is a supertype of all types, meaning all values have this type. In Scala, this type is called `Any`.

The bottom type is a subtype of all types. In Scala, such type is named `Nothing`. If there existed a value of such type, it could be casted to any other type, meaning it would have to behave like all classes in the type system at the same time. That is not possible, but the use of the bottom is different – it allows expressing that evaluation of a piece of code cannot ever end normally. For example, a method declared to return `Nothing` can only be implemented by throwing an exception, entering an infinite recursion, or exploiting unsoundness in the type system.

Another use of the top and bottom types is in bounds of type parameters and type members, where the upper bound being `Any` and/or the lower bound being `Unknown` means the type or parameter is unbounded in that direction.

Type members In Scala, type members can be abstract and overriden in subclasses.

```
abstract class C{
    // Abstract type member
    type T
}
class D extends C{
    // Type memeber T is overridend by class definition
    class T{...}
}
class E extends C{
    // Type memeber T is overridend by alias definition
    type T = String
}
```

Each type member has a lower and an upper bound, which can be specified independently. If bounds are not specified, then they are the bottom and top types `Unknown` and `Any`. For concrete type members, the upper and lower bounds are the same.

```
abstract class C{
    // Equivalent to type T
    type T >: Unknown <: Any
}
class E extends C{
    // Equivalent to type T = String
    type T >: String <: String
}
```

Type members can be used in *path-dependent types*, which are types formed by a path (a variable and zero or more field selections) and a type member selection.

```
// Path-dependent type referring to a parameter variable
def m(x: C)(y: x.T) = { ... }

class Node{
    type T
    // Path-dependent type involving self-reference
    val value: this.T
    val next: Node
    // Path-dependent type involving field selections
    def m(): this.next.next.T
}
```

In Scala, type members have a lot in common with generic type parameters, a major difference being that type parameters cannot be used in path-dependent types.

Refinement types Refinement types allow making a type more precise by narrowing the bounds of members or adding new members to an existing type. They can be considered a structural-typing version of the more usual class inheritance.

```
class C{
    type A
}
// The type of x is a refinement of type C,
// where the bounds of A are narrowed down to Int and
// the declaration of method m is added
val x : C {type A = Int ; def m(): Int } = ...
```

Type annotations Annotations allow attaching additional information to types, that is not a part of the type system used by the compiler, but can be used by external tools. In Java, annotations are limited to constant values composed of simple types (strings, numbers). In Scala, an annotation can contain an arbitrary value.

```
// Type annotation in Java
void m(@Nullable String s){ ... }
```

```
// Type annotation in Scala
def m(s: List[String @unchecked]) = { ... }
```

Checker framework [87] provides a convenient way of extending the type system with new features by means of type qualifiers attached to normal types as type annotations. The checking is run as a compiler plugin, which accesses the type annotations, checks proper usages of the qualifiers, infers qualifiers for local variables. Custom type hierarchies can be implemented to target specific problems, ranging from simple ones for checking for nullability [42] or integer signedness [71], to complex type systems with dependent types, such as checking locking discipline with a Lock Checker [46] or array accesses using the Index checker [66]. The following example shows how the index checker uses a dependent qualifier attached to a method parameter, to ensure that accessing an element of an array will not throw an `IndexOutOfBoundsException`.

```
void m(Object[] array, @IndexFor("array") int index){
    // The follwing access is guaranteed to succeed
    // by the Index checker
    Object o = array[index];
    ...
}
```

2.1.2 Type System Formalization

Formal program calculi have been developed alongside programming languages for decades, in order to answer theoretical questions, show correctness and verify designs.

The most ubiquitous calculus, the λ-calculus [35, 23], has the advantage of syntactic simplicity, which made it suitable for much of research on computability, type systems, decidability, typing algorithms. It has been used as a base for a plethora of extensions and more complex systems, including the DOT calculus.

The Java programming language [53] had originally a simple type system based on classes. Later, it was extended with generics, which made it more complicated. Due to the popularity and relative simplicity, Java has been used as a base for multiple experimental language features.

Featherweight Java [60] is a formal calculus that models classes, Featherweight Generic Java also supports generic parameters. It is being used to prototype and evaluate language extensions [102].

Minimal calculi have been also defined for an earlier versions of Scala [40], and other languages such as Go [54], Swift [92], and Rust [89].

There is usually a degree of disconnect between real code, formalized calculi and mechanization. The formalization usually omits features, which are deemed not relevant to the properties that are being examined in the formalization. This helps the formalization be more understandable and generalizable. On the other hand, the mechanization includes technical details which would be distracting in the formal presentation, and are therefore simplified or omitted.

2.2 Basic Concepts and Terminology

In the rest of the thesis, we will use concepts related to formal calculi. The meaning is usually standard within the field of programming-language research, but some terms may have specific connotation within DOT or Coq, which we explain in the following list.

For concepts which correspond to formal syntactic elements, we provide the variable names in parenthesis, which we use for variables.

Formal calculus A formal definition consisting of syntax, typing rules and semantics. Modeling selected aspects of a programming language. The purpose is to enable formal and verified reasoning about the language or proposed extensions.

In this thesis, we define two formal calculi: the baseline DOT calculus based on previous work in Section 2.4 and the roDOT calculus in Section 3.3.

Syntax A specification of how a program or its parts can be constructed. It is presented in BNF (Backus-Naur form) style, non-terminals are italic Roman or Greek letters.

In this thesis, we define the syntax of the baseline DOT in Section 2.4.2 (Figure 2.1) and of roDOT in Section 3.3.1 (Figure 3.2).

Term (t) Formalized representation of program code. Not all syntactically formed terms are meaningful. In DOT, terms have two roles: to represent the input program, and the state of execution in operational semantics.

Type (T) Representation of known properties of a piece of program code (term, object). Some types may be meaningful but uninhabited, such as the bottom type $\bot$, which represents a lower bound in the lattice of types.

Variable (x, z, s, y, w) A simple syntactic entity. Can be substituted with another variable (or a term, but that is not used in DOT). In DOT, all variables are term-level. In DOT, variables have two roles: 1) Referring to function parameters and local declarations, self-references in the source program. 2) Representing object locations and references at runtime.

Typing context (Γ) A typing context is a list of variable-type pairs, assigning a type to each variable. That type is used at the top of the typing derivation involving this variable, and can be considered the most precise type for that variable. In DOT, types in the context can contain free variables introduced earlier in the context.

Typing judgment ($\Gamma \vdash X : T$) A relation between a program element X (in DOT, a term or an object) and its type, within a given typing context. In DOT, the type of a term is not determined uniquely – a term can have multiple types at the same time.

Subtyping ($\Gamma \vdash T_1 <: T_2$) A relation (partial order) between types, associated with the ability to convert a element of the subtype to the super type.

The type T_1 on the left is more precise than the type T_2 on the right. In the presence of a **subsumption** rule, if a term has some type, it also has all of its supertypes, so typically subtyping is not used in premises of other rules.

Typing rules Specification of assigning types to terms or whole program.

A typing rule has zero or more premises (above line) and one conclusion (below line). All premises must be satisfied to derive the conclusion.

Derivation A proof of a judgment applied to specific parameters, in the form of a sequence or a tree, starting with rules that have not judgment premises, and ending with a concluding rule that produces the target expression.

Well-typed A piece of a program is well-typed, when a type can be derived for it using the appropriate typing rules.

Program (t) A self-contained piece of code that can be executed. In the case of a calculus without modeling side effects, the program has no input or output. In the case of DOT, a program is represented by a term which is well-typed under an empty context.

Member Definition (d) A field, a method or a type member. In DOT, member definitions have two roles – they constitute object literals in the program, which are blueprints for objects that should be instantiated at that point. Also, they constitute each instantiated object on the heap, storing the code of the object's methods and current values of all its fields.

Declaration (D) A type of a field, type member or a method definition. In the calculus presented here, it is defined with type, but in the mechanization as a mutually inductive definition.

Object An entity consisting of member definitions. Putting together data (fields) and behavior (methods).

In Scala and DOT, objects can also contain type definitions as members.

Heap (Σ) A collection of objects instantiated during program execution. The heap stores the object that can be referred to from multiple places by its location.

Location (y) A unique identifier of an object on the heap during execution. In DOT with mutable fields, it is represented by a variable.

Stack (σ) A part of a configuration, containing **frames**, which are parts of terms to be evaluated later. In DOT, used to keep continuations of let-in terms.

Configuration (c) A state of program evaluation. Initial configuration has empty stack and heap and a term representing the program. An answer has empty stack and single variable.

Dependent type A type referring to a variable. In DOT, containing a type selection ($x.A$).

Semantics Definition of the behavior of programs [51].

Small-step semantics Semantics defined by transformation of program configuration from initial state to answer, with intermediate steps related by reduction relation.

Reduction Relation between a configuration and the next configuration.

Soundness A property of a calculus ensuring that all typed programs will always have a step to execute, or reach answer configuration.

In the formal definition this can be as "never get stuck", but in real programming language, this would typically result in run-time type error (Java) or undefined behavior (crash or data corruption).

Judgment A relation, typically between a syntactic element, a typing context and a type, defined inductively using rules.

Decidable typing A typing judgment is decidable, if every instance of the typing judgment can be proven or disproven. With inductive definition, the positive is proved by forming a derivation by applying typing rules, the negative by inversion. Typing in DOT is conjectured to be undecidable.

Inductive definition A definition of a (often infinite) set or a relation, where the members are defined by a finite number of (often parametric) rules. And the members are produced by application of these rules in the form of a finite derivation. Infinite derivation does not produce a member of the set. Both the syntax and the typing are defined using inductive definitions.

The finite nature of the derivation allows employing proof by induction.

Inductive sets are similar to algebraic data types in functional programming or, more loosely, to sealed class hierarchies in object-oriented programming.

If multiple inductive definitions refer to each other, they are **mutually inductive**.

2.3 DOT Calculi

The increasing complexity of Scala's type system and discoveries of unsoundness in Scala demanded that a formal calculus be defined, in which Scala's core type system feature scan be modeled and questions about soundness can be verifiably answered.

The most desirable property is the proof soundness, because having it gives confidence that the type system of the language and the type checking implemented in the compiler will prevent unexpected runtime errors in the compiled programs.

The formalization effort was successful and resulted in the DOT calculus. Over the years, several versions of DOT calculi were published in research papers, starting from initial concepts, to fully developed calculi with soundness proofs, integrating various language features, and adjusted to answer different research questions.

An advantage of having a formal calculus like DOT is that can be extended with new features before implementing them in the language, to study their properties and to guide their development. For example, a soundness proof of such extension will show that implementing the feature will not break the expectations of type safety of the language. A new feature may also provide additional desired guarantees about the programs, which can be shown formally in the calculus.

In this thesis, we extend one of the versions of dot with features pertaining to reference mutability. To put our work in the context of this development, we briefly overview the history and the various version of DOT calculi. We will not go to details about the way language features are encoded here – a detailed example of a full calculus is given later in Section 2.4, which will be used as a baseline for the following development in this thesis.

2.3.1 Formalization and Soundness

The ideas for the DOT calculus were proposed in 2012 [16]. The first version represented types of objects differently than later DOT calculi, with refinement types. This calculus did not include a proof of soundness yet.

The key challenge in proving soundness of DOT is the possibility of *bad bounds*. A type member with bad bounds, for example with the lower bound being the top type and the upper bound being the bottom type, can cause a collapse of the type hierarchy, where all types are subtypes of each other.

The reason DOT can still be sound is that while it is possible to use a type with bad bounds, it is not possible to create a value of such type, much like as it is not possible to have a value of the bottom type. However, the existence of bad bounds makes reasoning about typing difficult in general.

Difficulties in proving soundness of the DOT calculus resulted in the need to scale back first, and led to the creation of a simplified μDOT [17] calculus, published in 2014. This calculus omitted lattice-related type constructors (the top and bottom types, unions, and intersections).

In 2016, a variant nicknamed WadlerFest DOT [18], was published and became the first "full" DOT for which a soundness proof [96] was completed. The soundness proof was mechanized in Coq.

WadlerFest DOT was simpler than the original proposal in that it constructs types of objects from more basic components. Instead of refinement types, it uses intersections with member declarations, and a separate recursive type construct is used to allow self-reference in member types. The type of an object is a recursive type wrapped around an intersection of member individual declarations. Instead of distinguishing between method and field members, methods are encoded as fields initialized to contain a lambda-function value.

However, the proof was considered too complicated for it to be used in further development of DOT calculi. Subsequently, a simpler proof for the same calculus was made [95], based on the idea of inert contexts which cannot have bad bounds and simplified definitions of typing that work in inert contexts – tight and invertible typing. The baseline DOT that we use in this thesis is derived from this version.

2.3.2 Extensions with Additional Scala Features

WadlerFest DOT was designed with the focus on soundness proof for Scala's dependent types.

WadlerFest DOT has a limitation on how the paths in path dependent types can be formed. While in Scala, the path can contain multiple field selections and type selections, in DOT, the path has only two parts, a variable and a type member selection. The extension pDOT [94] addresses this by allowing longer paths. A limitation of pDOT is that it requires object filed be declared with a precise type of the value they are initialized with. The support for longer paths also significantly complicated the soundness proof of the calculus.

The restrictions on typing member declaration is lifted in gDOT [50], instead weakening the type of self-references within objects to prevent circular derivations. The soundness proof uses a different approach than the previous calculi. It is based on logical relations and implemented in the Iris framework for Coq [4, 62],

While Scala has mutable objects, the objects in WadlerFest DOT and most DOT calculi are immutable – all fields of the object are set at the time of its creation and cannot be changed later.

This approach brings the calculus closer to functional programming, and allows the calculus to be simpler because objects can be directly embedded in the program as values, without a need for defining a heap of objects.

Mutable WadlerFest DOT [93] was the first DOT with soundness proof and mutability. Here, the mutability was implemented using mutable slots and mutable and read-only references to such slots.

A more direct approach, where objects on the heap can have fields reassigned, was used in κDOT [64], which also introduced a concept of object constructors to model gradual object initialization, as part of an effort to formalize safe object initialization, preventing null pointer errors due to uninitialized object fields. This was finished in ιDOT [65].

These calculi were mechanized in Coq, and the code is available on Github. After publication, authors improved the Coq proofs and made several version. A version with mutable fields but without constructors is feature-wise on par with our baseline and was used as a base for our Coq implementation.

A simplified version [63] with mutable fields, but without the specific kDOT feature of constructors, was used as a base for the mechanization of roDOT.

The rich type system of Scala is complemented with the ability to use implicits – values that do not have to be written in the code, but are automatically inserted by the compiler, based on the type of the desired type. Moving from Scala 2 to Scala 3, implicits were mostly redesigned to avoid common pitfalls and encourage preferred practices. With the redesign, a new possibilities are brought by implicit function types [83]. Implicit function types are formalized in the DIF calculus (DOT with implicit functions) [61].

2.3.3 Theoretical Grounding

Another interesting direction in which the family of DOT calculi spans, is away from Scala by removing features such as fields and recursive types, moving closer towards foundational calculi of functional programming – System F and System $F_{<:}$, but keeping the distinguishing feature of the DOT family – type members.

The $D_{<:}$ calculus [18] is DOT without intersection and recursive types, so it is not possible to construct objects. If was designed as a generalization of System $F_{<:}$ [31] with added path dependent types. It includes type members and dependent function types, which make type-checking hard by involving the issue of bad bounds.

The type-level functions in various calculi based on System F can be encoded using type members in a analogous calculus in the DOT family. The calculi between System F and DOT were organized into the System D Square [14]

The encoding of System $F_{<:}$ in System $D_{<:}$ is useful for investigating the issue of decidability of typing in DOT calculi, which is important for the possibility of implementing efficient type-checking algorithms.

It has been shown that typing and subtyping in $D_{<:}$ is undecidable [57, 58], and therefore it is conjectured that they are undecidable in DOT calculi too.

Restricted versions of $D_{<:}$ has been designed, that has decidable subtyping [79, 80]. As an alternative calculus that is closer to Scala while having decidable subtyping, jDOT [58] supports recursive types, but restricts recursion to single member declarations.

Another property of interest is strong normalization – whether well typed programs are guaranteed to terminate. While DOT was designed to be Turing complete (therefore not strong normalizing), System $D_{<:}$ has been shown to be strong normalizing [107].

2.4 Baseline DOT

In this section, we present the baseline DOT calculus that we will later extend with reference mutability.

Implementing a new feature in a DOT calculus requires choosing a baseline version of the calculus, to which the new feature will be added. The baseline should contain already well studied features, so one can focus on the new feature and possible interactions with the existing features.

Our baseline is not identical to any version of the calculus published in research papers, but it is close to kDOT [64]. Because the focus of our work is reference mutability, our baseline DOT uses kDOT's implementation of mutable objects. However, we removed constructor feature in baseline DOT, because constructors have specific purpose of object initialization and are not necessary to model reference mutability.

2.4.1 Structure of a DOT calculus

The DOT calculi based on WadlerFest DOT, including our baseline and the following work, share a common structure that we describe here.

The syntax describes how types and terms are formed. A term is a representation of a program or a piece of program. Terms can contain object literals, where objects consist of one or more member definitions.

Valid terms are assigned a type by applying typing and subtyping rules. To give types to terms containing free variable, a typing context is used which assigns types to variables.

$x ::=$	**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**
$\mid \cdot$	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 2.1: 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
$c ::= \langle t; \sigma; \Sigma \rangle$	**Configuration**

$Q ::=$	**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

$$\frac{}{\mathbf{inert}\,\cdot}\text{(BInert-Empty)} \qquad \frac{\mathbf{inert}\,\Gamma}{\mathbf{inert}\,\Gamma, y : S}\text{(BInert-Bind)}$$

Figure 2.2: Baseline DOT run-time syntax

Semantics is defined in a small-step fashion, with reduction rules each applying to a particular kind of a term. A machine configuration, apart from the term being executed, contains additional context, such as a stack and a heap. Execution starts from an initial configuration with empty stack, heap, etc, and ends with an answer configuration, where the focus of execution has been reduced into a single value.

2.4.2 Syntax and Semantics of the Baseline DOT

The syntax of the baseline DOT is in Figure 2.1 and Figure 2.2.

A program is expressed as a term, which, if correctly typed, reduces to a location of a single item on the heap.

Variables All variables in DOT and the baseline DOT are term-level.

$$\frac{x \notin \mathrm{dom}\ \Gamma_2}{\Gamma_1, x : T, \Gamma_2 \vdash x : T}\text{(BVT-Var)}$$

$$\frac{\Gamma \vdash x : \mu(s : T)}{\Gamma \vdash x : [x/s]T}\text{(BVT-RecE)}$$

$$\frac{\Gamma \vdash x : [x/s]T}{\Gamma \vdash x : \mu(s : T)}\text{(BVT-RecI)}$$

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

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

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash \mathsf{v}x : T}\text{(BTT-Var)}$$

$$\frac{\Gamma \vdash x_1 : \forall(z : T_1)T_2 \quad \Gamma \vdash x_2 : T_1}{\Gamma \vdash x_1 x_2 : [x_2/z]T_2}\text{(BTT-Apply)}$$

$$\frac{\Gamma \vdash x : \{a : T_2..T_3\}}{\Gamma \vdash x.a : T_3}\text{(BTT-Read)}$$

$$\frac{\Gamma \vdash x_1 : T_1 \quad \Gamma \vdash x : \{a : T_1..T_2\}}{\Gamma \vdash x.a := x_1 : T_2}\text{(BTT-Write)}$$

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

$$\frac{\Gamma, s : T_1 \vdash d : T_1 \quad \Gamma, z : \mu(s : T_1) \vdash t : T_2 \quad z \notin \mathrm{fv}\ T_2}{\Gamma \vdash \mathsf{let}\ z = \nu(s : T_1)d\ \mathsf{in}\ t : T_2}\text{(BTT-New)}$$

$$\frac{\Gamma, z_1 : T_1 \vdash t_1 : T_2 \quad \Gamma, z : \forall(z_1 : T_1)T_2 \vdash t_2 : T_3 \quad z_1 \notin \mathrm{fv}\ T_1 \quad z_2 \notin \mathrm{fv}\ T_3}{\Gamma \vdash \mathsf{let}\ z_2 = \lambda(z_1 : T_1)t_1\ \mathsf{in}\ t_2 : T_3}\text{(BTT-Fn)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, z : T_1 \vdash t_2 : T_2 \quad z \notin \mathrm{fv}\ T_2}{\Gamma \vdash \mathsf{let}\ z = t_1\ \mathsf{in}\ t_2 : T_2}\text{(BTT-Let)}$$

Figure 2.3: Baseline DOT typing rules

We distinguish several kinds of variables, based on where they are declared or where they emerge from. Although this distinction is not important for constructing and typing a program, using different symbols for each kind of variable will make the origin of the variable more apparent in definitions and examples, and in the later extension, we will use the variable kind to restrict which variables can occur in certain places.

Abstract variables are bound by term elements, of which *local* variables z are bound by let terms and lambdas, and *self* variables s that are bound by object literals.

Heap *locations* y represent addresses in the runtime heap. Locations cannot appear in the surface syntax – the initial configuration of a program uses only abstract variables. Locations created only in runtime configurations as the program executes and creates objects on the heap.

Typing rules for variables are in Figure 2.3.

$$\frac{}{\Gamma \vdash T <: \top}\text{(BST-Top)} \qquad \frac{}{\Gamma \vdash T_1 \wedge T_2 <: T_1}\text{(BST-And1)}$$

$$\frac{}{\Gamma \vdash \bot <: T}\text{(BST-Bot)} \qquad \frac{}{\Gamma \vdash T_1 \wedge T_2 <: T_2}\text{(BST-And2)}$$

$$\frac{}{\Gamma \vdash T <: T}\text{(BST-Refl)} \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 <: T_2 \\ \Gamma \vdash T_1 <: T_3\end{array}}{\Gamma \vdash T_1 <: T_2 \wedge T_3}\text{(BST-And)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 <: T_2 \\ \Gamma \vdash T_2 <: T_3\end{array}}{\Gamma \vdash T_1 <: T_3}\text{(BST-Trans)}$$

$$\frac{\Gamma \vdash x : \{A : T_1..T_2\}}{\Gamma \vdash T_1 <: x.A}\text{(BST-SelL)} \qquad \frac{\begin{array}{c}\Gamma \vdash T_3 <: T_1 \\ \Gamma \vdash T_2 <: T_4\end{array}}{\Gamma \vdash \{A : T_1..T_2\} <: \{A : T_3..T_4\}}\text{(BST-Typ)}$$

$$\frac{\Gamma \vdash x : \{A : T_1..T_2\}}{\Gamma \vdash x.A <: T_2}\text{(BST-SelU)} \qquad \frac{\begin{array}{c}\Gamma \vdash T_3 <: T_1 \\ \Gamma \vdash T_2 <: T_4\end{array}}{\Gamma \vdash \{a : T_1..T_2\} <: \{a : T_3..T_4\}}\text{(BST-Fld)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T_3 <: T_1 \\ \Gamma, z : T_3 \vdash T_2 <: T_4\end{array}}{\Gamma \vdash \forall(z : T_1)T_2 <: \forall(z : T_3)T_4}\text{(BST-Fn)}$$

Figure 2.4: Baseline DOT subtyping rules

Terms In DOT, each variable *is* also a term. In the baseline DOT, we always make a distinction between terms and variables. This makes it possible to separate typing rules for terms from typing rules for variables. When a variable is used as a term, we use the notation $\mathsf{v}x$. This kind of a term signifies a part of a program that is fully evaluated.

A *let term* evaluates one term and substitutes the result into another term.

Terms in DOT and the baseline DOT are in A-normal form (ANF). Only let terms can have arbitrary subterms, other terms can only reference variable, not 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* reads the value of a field. It evaluates either to the term value given to the field when the object was created, or (because fields can be reassigned) the last value written to a field, An *apply* term applies a lambda substituting the argument into its body, which is retrieved from the heap.

Typing rules for terms are in Figure 2.3.

Example 1 (Simple term in baseline DOT). $\mathsf{let}\, x = x_1.a_1 \,\mathsf{in}\, x_2.a_1 := x$ is a term which first reads the value of field a_1 from the object referenced by x_1 into a local variable x, then writes the value to field a_2 of an object referenced by x_2. The equivalent code in Scala would be `x2.a2 = x1.a1`.

$$\frac{}{\Gamma \vdash \{A = T\} : \{A : T..T\}}\text{(BDT-Typ)} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T..T\}}\text{(BDT-Fld)}$$

$$\frac{\begin{array}{c}\Gamma \vdash d_1 : T_1 \\ \Gamma \vdash d_2 : T_2 \\ d_1 \text{ and } d_2 \text{ have distinct member names}\end{array}}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2}\text{(BDT-And)}$$

Figure 2.5: Baseline DOT definition typing rules

Heap items and Literals Programs in kDOT and the baseline DOT can create heap *items*, which are either lambda abstractions or objects.

The heap items are constructed from *literals*, which can only appear in a let of the form $\mathsf{let}\, z = l \,\mathsf{in}\, t$, ensuring that every literal can be referred to by some variable z.

Types must be explicitly specified in lambda and object literals. *Lambdas* have a function type, which specifies the type of the parameter and allows the lambda to be applied to a argument of such type in an apply term.

An *object* has a *self parameter* s (modeling 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.

Typing rules for object definitions are in Figure 2.5.

Example 2 (Using an object in baseline DOT)*.* If variables x_a and x have type T, then $\mathsf{let}\, x_o = \nu(s : \{a = x_a\} \wedge \{m = \lambda(z : T)s.a := z\}) \,\mathsf{in}\,\mathsf{let}\, x_m = x_o.mx \,\mathsf{in}\, x_m.x$ is a term which first creates an object which stores the value of x_m in field a and in field m, stores a function which reassigns the field a. Then, it reads the function from the field m and applies it to argument x. The equivalent code in Scala would be the following:

```
class C(var a:T){
    def m(z:T) = {a = z}
};
val xo = new C(xa);
xo.m(x)
```

The field containing a function implements the concept of object methods and the action of reading a field and applying it in turn implements a method call.

Types 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

$$\frac{y_1 \to \nu(s:T)\ldots_1 \{a = t\} \ldots_2 \in \Sigma}{\langle y_1.a; \sigma; \Sigma\rangle \longmapsto \langle t; \sigma; \Sigma\rangle}\text{(BR-Read)}$$

$$\frac{\begin{array}{c} y_1 \to \nu(s:T)\ldots_1 \{a = t\} \ldots_2 \in \Sigma_1 \\ \Sigma_2 = \Sigma_1[y_1 \to \nu(s:T)\ldots_1 \{a = \mathsf{v}y_2\} \ldots_2] \end{array}}{\langle y_1.a := y_2; \sigma; \Sigma_1\rangle \longmapsto \langle \mathsf{v}y_2; \sigma; \Sigma_2\rangle}\text{(BR-Write)}$$

$$\frac{y_1 \to \lambda(z:T)t \in \Sigma}{\langle y_1 y_2; \sigma; \Sigma\rangle \longmapsto \langle [y_2/z]t; \sigma; \Sigma\rangle}\text{(BR-Apply)}$$

$$\frac{\Sigma_2 = \Sigma_1, y \to \lambda(z_1:T)t_1}{\langle \mathsf{let}\ z_2 = \lambda(z_1:T)t_1\ \mathsf{in}\ t_2; \sigma; \Sigma_1\rangle \longmapsto \langle [y/z_2]t_2; \sigma; \Sigma_2\rangle}\text{(BR-LetFn)}$$

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

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

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

Figure 2.6: Baseline DOT reduction (operational semantics)

to by type selection. The top type $\top$ is a supertype of every type; the bottom type $\bot$ is a subtype of every type.

Subtyping rules are in Figure 2.4.

Example 3 (Typing an object in baseline DOT). The object defined in Example 2 has type $\mu(s : \{a : T..T\} \wedge \{m : \forall(z:T)T..\forall(z:T)T\})$.

Operational semantics Evaluation of a program is defined by small-step semantics, by step-wise reduction of an initial configuration, until an answer configuration is reached. The *initial configuration* consists of a term (representing the full program), an empty stack, and an empty heap.

A *configuration* represents a full state of evaluation of a program. During evaluation, items can be added to the heap, frames can be pushed to and popped from the stack, and the term representing the *focus of execution* changes according to the reduction rules.

The heap binds locations y to literals l. The heap is modified by creating a new item using the let-lit term, or by changing the value of a field using the write term.

In a final *answer* configuration, the stack is empty and the term is in normal form – it is a variable term referring to a location of a single item on the heap. The reduction rules are in Figure 2.6.

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \cdot : T_1, T_2}\text{(BCT-EmptyS)}$$

$$\frac{}{\Gamma \vdash_{\sim} \cdot}\text{(BCT-EmptyH)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma : T_2, T_3 \\ \Gamma, z : T_1 \vdash t : T_2 \\ z \notin \text{fv } T_2\end{array}}{\Gamma \vdash \mathsf{let}\ z = \square\ \mathsf{in}\ t :: \sigma : T_1, T_3}\text{(BCT-LetS)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \Gamma_2 \sim \Sigma \\ \Gamma_1, z : T_1 \vdash t : T_2 \\ z \notin \text{fv } T_1\end{array}}{\Gamma_1 \vdash \Gamma_2, y : \forall(z : T_1)T_2 \sim \Sigma, y \to \lambda(z : T_1)t}\text{(BCT-FnH)}$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash \Gamma_2 \sim \Sigma \\ \Gamma_1 \vdash d : [y/s]R\end{array}}{\Gamma_1 \vdash \Gamma_2, y : \mu(s : R) \sim \Sigma, y \to \nu(s : R)d}\text{(BCT-ObjH)}$$

$$\frac{\Gamma \vdash \Gamma \sim \Sigma}{\Gamma \sim \Sigma}\text{(BCT-CorrH)} \qquad \frac{\begin{array}{c}\Gamma \sim \Sigma \\ \Gamma \vdash t : T_1 \\ \Gamma \vdash \sigma : T_1, T_2\end{array}}{\Gamma \vdash \langle t; \sigma; \Sigma\rangle : T_2}\text{(BCT-Corr)}$$

Figure 2.7: Baseline DOT configuration typing rules

2.4.3 Type Soundness

The type soundness property of the calculus 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 declaration of a type member $\{A : T..T\}$ 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 [95].

Typed configurations 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. Typing rules for configurations are in Figure 2.7.

$$\begin{tikzcd}
\begin{array}{c}\text{General subtyping}\\ \Gamma \vdash S <: T\end{array} \arrow[d, leftrightarrow] & \begin{array}{c}\text{Tight subtyping}\\ \Gamma \vdash_{\#} S <: T\end{array} \arrow[l, "\Leftrightarrow"'] \arrow[r] & \begin{array}{c}\text{Precise typing}\\ \Gamma \vdash_{!} x : T\end{array} \\
\begin{array}{c}\text{General typing}\\ \Gamma \vdash x : T\end{array} \arrow[r, Leftrightarrow] & \begin{array}{c}\text{Tight typing}\\ \Gamma \vdash_{\#} x : T\end{array} \arrow[u] \arrow[r, Leftrightarrow] & \begin{array}{c}\text{Invertible typing}\\ \Gamma \vdash_{\#\#} x : T\end{array} \arrow[u] \arrow[ul] \arrow[ull]
\end{tikzcd}$$

Figure 2.8: Dependencies ($\rightarrow$) and equivalence ($\Leftrightarrow$) between definitions of typing in DOT

Soundness theorems 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.

Theorem 1 (Baseline Safety).

If $\vdash t_0 : T$,	The initial term t_0 is well typed,
then either $\exists y, j, \Sigma$: $\langle t_0; \cdot; \cdot\rangle \longmapsto^j \langle \mathsf{v}y; \cdot; \Sigma\rangle$	then execution terminates in j steps with answer y,
or $\forall j$: $\exists t_j, \sigma_j, \Sigma_j$: $\langle t_0; \cdot; \cdot\rangle \longmapsto^j \langle t_j; \sigma_j; \Sigma_j\rangle$.	or continues indefinitely.

Proofs of soundness theorems The proof of Theorem 1 is a direct adaptation of the proof from kDOT (and WadlerFest DOT). It uses auxiliary definitions of variable typing and subtyping, which are equivalent under inert typing contexts.

The relations of these definitions is depicted in Figure 2.8.

2.4.4 Invertible Typing

Invertible typing ($\Gamma;\rho \vdash_{\#\#} x : T$) has the following essential properties:

- It is equivalent to general typing $\Gamma;\rho \vdash x : T$ in an inert context.
- Derivations of declaration types can be inverted. That means, for $\Gamma;\rho \vdash w : D$, where D is a declaration type such as $\{a : T\}$, it allows us to show that the type assigned to w in Γ is an object type containing D or a more precise declaration of the same member. Because the types in Γ correspond to the object on the heap ($\Gamma \sim \Sigma$), the actual object referred to by w must contain a corresponding member definition in Σ, and therefore it is safe to access that member.

Because configurations during execution are typed in inert contexts, this guarantees that a well-typed program does not attempt to access a non-existing member of an object while being executed. Additionally, we use invertible typing in a new role, to prove Lemma 22.

Invertible typing in DOT [95], also adopted in roDOT, has two layers. The first layer, **precise typing**, does not use subtyping, and only deals with declaration types precisely as they appear in the typing context.

For each reference w, its type in the typing context is a recursive type containing an intersection of declarations, and a mutability declaration. Precise

$$\frac{\Gamma;\rho \vdash_{\#\#} x : T_1 \quad \Gamma;\rho \vdash_{\#\#} x : T_2}{\Gamma;\rho \vdash_{\#\#} x : T_1 \wedge T_2}\,(\text{VT}_{\#\#}\text{-AndI}) \qquad \frac{\Gamma;\rho \vdash_{\#\#} x : [x/s]T \quad T \ \mathbf{indep}\ s \quad \Gamma;\rho \vdash [x/s]T\ \mathbf{ro}\ [x/s]T}{\Gamma;\rho \vdash_{\#\#} x : \mu(s : T)}\,(\text{VT}_{\#\#}\text{-RecI})$$

$$\frac{\Gamma;\rho \vdash_{\#\#} x : T_1}{\Gamma;\rho \vdash_{\#\#} x : T_1 \vee T_2}\,(\text{VT}_{\#\#}\text{-Or1}) \qquad \frac{\Gamma;\rho \vdash_{\#\#} x : T_2}{\Gamma;\rho \vdash_{\#\#} x : T_1 \vee T_2}\,(\text{VT}_{\#\#}\text{-Or2})$$

Figure 2.9: Example rules of invertible typing

typing allows opening this recursive type and destructing the intersection types to extract one of the declarations.

The second layer, **invertible typing** combines both variable typing and subtyping into a single layer. In DOT and the original roDOT, it has fewer rules than general typing and subtyping, because it only has rules that construct the target type syntactically "bottom-up", such as closing recursive types with $\text{VT}_{\#\#}$-RecI, or deriving intersection and union types. Examples of those rules are shown in Figure 2.9. Thus the derivations of invertible typing are unambiguously guided by the syntax of the target type.

Figure 2.8 shows relations of the different versions of typing. The equivalence of general and invertible typing requires showing both directions. In DOT and the original roDOT, the direction from invertible to general typing is very easy, because each rule in invertible typing corresponds to a straightforward application of just a few rules from general typing. The direction from general to invertible typing is more involved. It uses tight typing as an intermediate step, and a significant part of the proof is showing that invertible typing is closed under tight subtyping, stated here as Lemma 2.

Lemma 2 (Invertible typing is closed under subtyping)**.** *If* $\Gamma;\rho \vdash_{\#\#} x : T_1$*, and* $\Gamma;\rho \vdash_{\#} T_1 <: T_2$*, where* $\Gamma \sim \rho$*, then* $\Gamma;\rho \vdash_{\#\#} x : T_2$*.*

2.5 Mechanization of DOT Calculi

The numerous existing variants of DOT calculi, mentioned in Section 2.3, were published in research papers. To avoid ambiguities and ensure that there all proof are correct, the publication of each DOT calculus is usually accompanied with a mechanized version of its definitions and proofs.

Th basic definitions and theorem are provided in the text, but due to space constraints, a certain level of syntactic simplification must be employed. Not all definitions can be included, and the proofs are only described at a high level, because a full proof of the presented theorems would be very long.

The original DOT was mechanized using the Coq proof assistant [96]. A mechanization for a DOT calculus with mutable fields, which is reasonably close to our baseline calculus, was done by Ifaz Kabir [63]. We use this version as the base for mechanization of the calculi defined in this thesis.

2.5.1 Baseline DOT mechanization

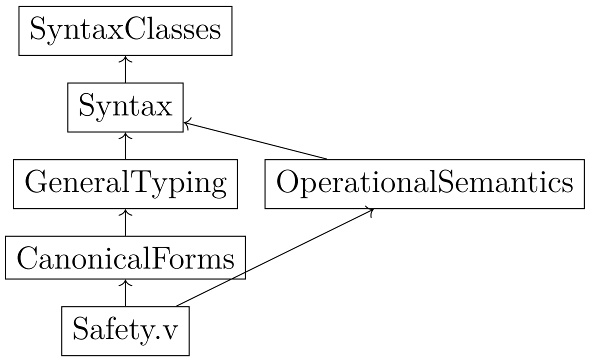

Figure 2.10: Structure of the Coq definitions and soundness proof for DOT

The mechanization is a Coq project using the TLC library [34], which extends Coq's logic with axioms of classical logic. The Coq project is structured into parts, which are shown in Figure 2.10 and described below. The files referenced in this chapter can be found in the baseline mechanization [63] under the directory `dot-simpler/dot-field-mut-stage-1`. Our mechanization (Attachment A.1) follows the same basic structure.

Syntax classes The syntactic elements of DOT calculi (and in general) have several several common properties independent of the particular meaning of those elements within the calculus. We may call them "common sense" properties, regarding free variables, or various forms of substitution. For example, if a variable is not free in a syntactic element, then substituting for that variable does not change anything.

In order to be able to reason about such properties for any element of syntax, these properties are implemented using type classes.

These definitions are in the file `Lib/SyntaxClasses.v`.

Each type class defines a particular operation on or a property of a syntactic element. Type classes `Openable`, `SubstVar` and `Closing` define different variants of substitution in the locally-nameless representation. Class `FreeVar` defines collecting the set of free variables.

```
Class Openable (A : Set) := open_rec : nat —> var —> A —> A.
Class FreeVar (A : Set) := fv : A —> vars.
Class SubstVar (A : Set) := subst_var : var —> var —> A —> A.
```

The common sense properties are each defined as a class with a single proposition. For example the property above is defined as class `SubstFresh`.

```
Class SubstFresh ‘(AS : AbstractSyntax A) :=
    subst_fresh : forall x y X,
    x \notin fv X —> subst_var x y X = X.
```

Syntax

In Figure 2.1, terms and types are defined inductively – terms and types a re formed using constructors, where each constructor may have parameters of various types such as terms, types, variables, labels, etc.

In Coq, these definitions represented by inductive sets. Such inductive definition consists constructors analogous to the definition as presented in Figure 2.1. The presented syntax uses various glyphs aimed at conciseness, readability and a familiar look similar to other formal calculi, but the mechanized version replaced uses a with unified functional application (curried).

Locally nameless representation Another difference is that DOT mechanizations use a *locally nameless notation*[33] to represent variables. The basic definitions of language calculi (including the lambda calculus) represent variables using a set of names, where a variable is used in a binder construct, and then the same name can be used in the scope to refer to this variable. An example of this in the baseline DOT are function terms, let terms, object literals and recursive types.

This representation makes it necessary to deal with possible conflicts of variable names, such as when a binder (constructor, which creates a variable binding) is nested within a scope of another binder that uses the same variable name. Constructing complex terms requires ensuring that conflicts do not emerge, which may require generating unique names and substituting a conflicting name with another name.

These problems can be avoided by representation of variables using De Bruijn indices[41].

In this representation, a binder does not specify any name for the variable, but a use of a variable is represented by an integer index, which specifies which of the enclosing binder it refers to, 0 being the closest binder, 1 the next and so on.. This way, terms can be safely composed without the need for any modifications. On the other hand, this representation is not well suited for working with variables that are not bound in the term itself, such as the locations of objects on the heap. The locally nameless representation combines these two approaches, where a variable can be either free or bound, where free variables are represented by names and bound variables by De Bruijn indices.

Which constructors are binding is not determined by the definition of the recursive set itself, but by the means of an opening operation, which can convert bound variables to free variables. Whenever descending into sub-term is required, such as when typing a subterm, then the term is “opened” – de Bruijn indices that refer to the top level binding are replaced by a named variable.

A reverse operation of closing, which replaces named variables by indices, is used much less often. The syntax, nor the typing rules forbid creation of terms with invalid indices, which larger than the number of binding constructors above. A term which does not contain such indices, and therefore is valid stand-alone, is referred to as “closed”.

Note that there is a slight confusion of terminology. A “closed term” is a term which contains no unbound indices. If one level of binding was stripped, then the “opening” operation can be used to get a “closed” term, while the "closing" operation will produce a term that is not closed.

Syntax The syntax is defined with inductive and mutually inductive definitions in files `Syntax/Vars.v`, `Syntax/Types.v`, and `Syntax/Terms.v`.

The following example shows a mutually inductive definition of types (T, `typ`) and declarations (D, `dec`).

```
Inductive typ : Set :=
| typ_top  : typ
| typ_bot  : typ
| typ_rcd  : dec -> typ
| typ_and  : typ -> typ -> typ
| typ_sel  : avar -> typ_label -> typ
| typ_bnd  : typ -> typ
| typ_all  : typ -> typ -> typ
with dec : Set :=
| dec_typ  : typ_label -> typ -> typ -> dec
| dec_trm  : trm_label -> typ -> typ -> dec.
```

Each such definition is followed by instantiating the syntactic classes described above.

In `Lib/SyntaxClasses.v`, there is a framework for lifting properties such as `SubstFresh` from individual variables to terms and types. All is needed is to prove this property for variables, and prove properties `Transform` and `Collect` for the particular syntax. These properties state how generic mapping and collection operations are distributed through the syntax.

Then every property that is expressed in terms of those two operations (such as `TransformSubstFresh` in `SyntaxClasses/TransformCollect.v`) are automatically lifted to that syntax.

The run-time syntax – stacks, heaps and configurations are defined in `Syntax/AbstractMachine.v`.

General Typing The typing relation is defined in `GeneralTyping/GeneralTyping.v` as a mutually inductive definition of 5 judgments: typing terms, literals, individual definitions, aggregate definitions, and subtyping.

The typing judgment is used through a type class, so that a common syntax `G ⊢ t : T` can be used for terms, literals and definitions.

The following excerpt shows the subsumption rule for typing terms:

```
| ty_sub : forall G (t : trm) T U,
  G ⊢ t : T ->
  G ⊢ T <: U ->
  G ⊢ t : U
```

General typing supports the weakening, narrowing and substitution, which allow manipulating typed terms and changing the typing context. These lemmata are used in the safety proof. They are defined an proven in `GeneralTyping/Weakening.v`, `GeneralTyping/Narrowing.v` and `GeneralTyping/Substitution.v`. The proofs are mutually inductive on the 5 typing judgments.

The Coq proof assistant is based on calculus of inductive definitions [26]. Both the syntax and typing rules of a DOT calculus are represented by inductive definitions in Coq.

Typing judgments

The typing relations are also inductive definitions – parametric propositions, where the parameters are the typing context, the element being typed and the type.

Each typing rule is represented by a constructor, with parameters including the variables and all premises of the rule. A typing derivation is constructed by induction from these constructors. Because of how the typing rules are defined, this induction closely follows the structure of the term, but there are differences - subsumption, opening terms.

Operational semantics Operational semantics is defined by the reduction relation `red` in `OperationalSemantics/OperationalSemantics.v`. The reduction rules are fairly straightforward except the filed assignment rule, which updates a particular field of a particular object on the heap. This updating operation is defined in `OperationalSemantics/HeapUpdate.v`.

Canonical forms The typing relations used in the safety proof – tight, precise and invertible typing – are defined in `CanonicalForms/TightTyping.v`. `CanonicalForms/PreciseTyping.v` and `CanonicalForms/InvertibleTyping.v`. These judgments are used in inert typing contexts, which are defined in `CanonicalForms/RecordAndInertTypes.v`. The equivalence of general, tight and invertible typing under inert contexts is proven in `CanonicalForms/TightTyping.v`. `CanonicalForms/GeneralToTight.v` and `CanonicalForms/InvertibleTyping.v`.

Correspondence of heap with a typing context is defined in `CanonicalForms/HeapCorrespondence.v`.

The file `CanonicalForms/TightTyping.v` contains lemmata which allow inverting typing of a term under an inert context and retrieve the referenced objects from the heap and find their types.

Safety theorem Finally, the entry-point file `Safety.v` defines configuration typing, and proves the safety by the progress and preservation theorems.

Mechanized Proofs

Many properties about typing relations have similar form, where the conclusion and one of the premises of the lemma is a typing judgment:

```
Lemma lemma_about_typing parameters:
TypingConditions1 ->
Preconditions ->
TypingConditions2.
```

Examples of such lemmata are the weakening, narrowing and substitution properties.

```
Lemma weaken_ty_trm: forall G1 G2 (t : trm) T,
    G1 ⊢ t : T ->
    ok (G1 & G2) ->
    G1 & G2 ⊢ t : T.
```

The mechanized proof of such lemma usually conforms to this common structure:

- 1. Induction on the typing premise (using the `induction` tactic).
- 2. Inversion of other premises using the `inversion` tactic.
- 3. Rewriting of the goal and hypotheses, so that they conform to the form expected by the induction hypothesis.
- 4. Application of the induction hypothesis.

The induction tactic applied in step 1 generates a goal for each constructor of the inductive definition. For example, for proving a typing property, a goal is generated for each typing rule. Then, steps 2-4 must be applied to each such goal. In some proofs, many of such goals can be handled by applying the same tactics, but sometimes, each goal is solved separately.

Induction A proof method based on reducing the problem into smaller subproblems until reaching trivial problems, or, viewed the other way around, producing solutions to a general problem starting from trivial problems and composing solutions of smaller problem to solve bigger problems. The proof is finite, ensured by decreasing size of the problem based on the inductive definition. The induction hypothesis is a solution of a subproblem used to solve larger problems. It has the same form as the conclusion. Choosing the proper form of the induction hypothesis is key; often the problem must be stated in more general form for the induction to work. In Coq, induction is employed using the induction tactic, which applies an **induction scheme**, which is a lemma that Coq generated from the inductive definition.

Inversion A proof step deriving the premises of an inductive rule from the conclusion.

Sometimes only one rule of the definition matches the hypothesis. In case multiple rules match, multiple cases must be solved. For example, inversion can be applied on a reduction judgment where the source term has known form, resulting in one variant for each. For general typing, the subsumption rule causes inversion to be practically useless. In Coq, inversion is employed using the inversion tactic.

Mutually inductive definitions The sets of terms, literals, and definitions are mutually inductive, because a term can contain an object literal, which can contain a member definition and such definition can contain a term. Such mutually inductive definitions are supported in Coq, when multiple inductive definitions are defined together.

Similarly, the typing judgments for terms, literal and definitions must be mutually inductive.

While the mutually inductive definitions look like normal inductive definitions, mutually inductive proofs in Coq are harder to do than when working with single induction. The goal must have the form of conjunction of the properties for each of the set in the mutual induction.

Backward reasoning A proof method based on having a goal (starting from conclusion) and applying lemmas that match the goal, requiring one or more premises that become the new goals. This way, the proof is constructed backwards, from the conclusion towards the premises. This can be well automated in Coq. Often, there is only a small number of lemmas that can be applied to a given goal, and the parameter of the lemmas are often determined by the goal. This allows exploring the space using backtracking.

Forward reasoning A proof method based on constructing new propositions from known propositions, starting from the premises. This way is typically not possible to automate, because typically many rules may be applied with many parameters and the space of possible derivations is too big.

3. The roDOT Calculus

In this chapter, we describe the roDOT calculus, an extension of DOT with reference mutability. This chapter has the following structure:

In Section 3.1, the concepts of reference mutability are introduced, including motivations and existing implementations of this feature in programming languages.

In Section 3.2 the design decisions for incorporating reference mutability to DOT calculi are presented. 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.

Section Section 3.3 describes the roDOT calculus, the implementation of reference mutability in a calculus based on the baseline DOT.

In Section 3.4 and Section 3.5, we define properties of roDOT, namely type soundness and the immutability guarantee, and discuss their proofs. We have mechanized roDOT including its properties in Coq. This process is described in Section 3.6.

In section Section 3.7, roDOT is compared to existing type systems with reference mutability.

The content of this chapter is based on the research paper Reference mutability for DOT [43].

3.1 Introduction

In main-stream object-oriented languages, such as Java, C#, objects are by default mutable, that is, objects contain fields which can be updated.

When that happens, the identity of the object is preserved, but the value is changed.

At the same time, two objects can represent the same value while having separate addresses, and the value represented by one object can change during execution of the program.

This is in contrast with the concept of referential transparency, where the value and identity of objects cannot be separated.

The ability to mutate objects causes difficulties in programming, such as:

- **Confusion between identity and equality.**
- **Object copying.** If two pieces of code are put together where one may mutate the object while the other needs it to be constant, the object may need to be copied. Sometimes not necessarily (defensive copy.)
- **Missing optimization opportunities**, where an object is not intended to change in a piece of code, but the optimizer cannot assume that the value is not changed, because it can be changed from code outside of the optimizer's view.
- **Race conditions.** If an object is mutated and at the same time accessed from another thread, the result is undefined.

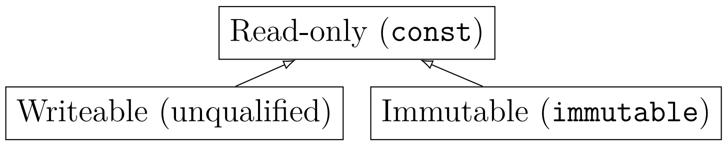

Figure 3.1: Hierarchy of reference mutability qualifiers in D

Several programming languages support constructs that limit mutability in different ways:

- In C++, `const` denotes immutable variables and read-only pointers references. The constness qualifier is attached to the underlying type.
- In Java, `final` denotes immutable variables and fields. For fields, `final` requires definite initialization. Final fields allow certain optimization (caching, moving accesses). Final fields provide guarantees even in presence of data races. There are exceptions to these rules: Reflection does not allow setting final fields, but allows removing the final modifier. Serialization assigns final fields. There are special exception for access to `System.in` and `System.out`, which restrict write access outside of the rules of the language.
- In C#, `readonly` denotes immutable fields [8].
- Field accesses can be hidden behind getter and setter methods. If only a getter is available for a private field, the field is effectively read-only.

Difference between immutable objects and read-only references A clear distinction must be made between read-only references and immutable objects.

A read-only reference can point to a mutable object. While the object cannot be changed through the read-only reference, it can still be changed through other references. That can happen from the same or another thread. If an object is immutable, its value will never change, and all references to it are effectively read-only.

The difference between read-only and immutable references is nicely visible in the D language [3], where the `immutable` pointer qualifier guarantees that the referred object is immutable, unqualified pointers allow mutation, and both are convertible to `const` pointers as shown in Figure 3.1.

Transitively read-only references Read-only references can apply to the first level of indirection, or be transitive. In that case, the result of an access through the indirection are also read-only.

3.1.1 Type Systems for Reference Mutability

Reference mutability types systems (called reference immutability) [106, 59], have 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.

Our original goal was to bring read-only references to Scala. An empirical study [55] 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. 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. Therefore, we switched our focus from implemenation to formal definition of reference mutability in the context of the DOT calculus.

3.2 Design of roDOT

In this section, we discuss how the baseline DOT system can be extended to support read-only references, and explain individual design decision, that together lead to roDOT.

3.2.1 Requirements

First, we identify a list of requirements, which any extension of an existing type system with reference mutability should satisfy:

Keeping expressiveness of the original calculus The extended type system should admit programs that are valid in the baseline calculus (possibly with simple adaptations).

Mutability constraints The type system should provide a way to distinguish between read-write and read-only references to objects. All read-write references should be convertible to read-only references, but not the other way. This can be achieved by having a read-only and read-write version for each type, and making the read-write type a subtype of the corresponding read-only type.

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 – 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 [59].

Mutability polymorphism Previous work [106, 59] 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.

3.2.2 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 the following 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 the following let statement:
$\mathsf{let}\, z = \nu(s : T_\mathrm{o})\{a = x\} \wedge \{m_\mathrm{s} = \mathsf{let}\, z_1 = \lambda(z : T)s.a := z \,\mathsf{in}\, z_1\} \wedge \{m_\mathrm{g} = s.a\} \,\mathsf{in}\, t$,
where
$T_\mathrm{o} \triangleq \mu(s : \{a : T\} \wedge \{m_\mathrm{s} : \forall(z : T)\top\} \wedge \{m_\mathrm{g} : T\})$ and x is an initial value of type T.

The method m_s mutates the contents, so a reference mutability type system should prevent calling it on a read-only reference. The method m_g 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 3.3.5.

3.2.3 Representing Mutability Types

The core feature of roDOT is the ability to distinguish read-only and read-write references. In this section, we describe how this distinction can be achieved in an extension of the baseline DOT.

Mutability Marker

First, we must decide how to distinguish read-write and read-only types.

Previous approaches, such as ReIm, use a type qualifier which is a part of a type, but is kept separate from the usual types. This makes sense in a relatively simple type system such as Java.

However, in a complex type system such as DOT's, there is another straightforward way to represent mutability and other type capabilities. We can define a special marker type $\mathcal{M}$, which will designate read-write references. Then, any type can be made read-write by intersecting it with $\mathcal{M}$.

We define an operation $\mathsf{rw}(T) \equiv T \wedge \mathcal{M}$, which for any reference type T creates its read-write counterpart.

This satisfies our requirement that a read-write type should be a subtype of the read-only version of the same type, because by usual subtyping rules, $T \wedge \mathcal{M} <: T$.

With this definition, we can test whether a type is read-write by testing whether it is a subtype of the marker type $\mathcal{M}$. For a reference x, we define an operation $\Gamma \vdash \textbf{isrw}\ x$ as a typing judgment $\Gamma \vdash x : \mathcal{M}$. In a type system with reference mutability, this judgment will be used as a precondition for a write operation using this reference.

Capability Type Members

Second, we need some mechanism to implement mutability polymorphism.

In ReIm and Checker Framework, polymorphism is supported by a special polymorphic qualifier. This polymorphic qualifier acts as an implicit type parameter, where every occurrence of the polymorphic qualifier can be instantiated to the same concrete qualifier.

This approach conveniently supports simple cases, but has several disadvantages:

- It is not possible to specify bounds for the polymorphic qualifier.
- There can only be one polymorphic qualifier in a declaration.
- It is not possible to combine qualifiers from multiple declarations.

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 $\mathcal{M} \equiv \{\mathsf{M} : \bot..\bot\}$.

With this definition, the mutability marker is a declaration of a special type member with tight bounds equal to the bottom type. A similar declaration with loose bounds such as $\{\mathsf{M} : \bot..\top\}$ does not enable mutability. Therefore, we can construct types for which the mutability is controlled by this bound: $T \wedge \{\mathsf{M} : \bot..S\}$ is mutable if $S <: \bot$.

Them mutability polymorphism can be achieved by using types with the same bound, but also, these capabilities can be combined together.

Dependent Capabilities

This choice of the mutability marker, together with how dependent types are defined in DOT, 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 $\{\mathsf{M} : \bot..\bot\}$).

Using this dependent mutability, we can define a method type where the mutability of the result is defined to be the same as its parameter.

Viewpoint Adaptation

Third, the type system requires an operation $\Gamma \vdash x \triangleright T \to T'$ that performs the viewpoint adaptation described above.

Given a reference x and a type T of a field, $\Gamma \vdash x \triangleright T \to T'$ should be a type equivalent to T if x is a read-write reference, but if x is a read-only reference, the viewpoint-adapted type should be a read-only version of T .

This operation can be composed from two simpler operations: making a read-only version of T and combining the mutability of T with the mutability of x.

While a read-write version of a given type can be made with a simple intersection, the opposite operation 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 type-level operator **ro** that makes a read-only version of a type. We will define $\Gamma \vdash T\,\mathbf{ro}\,T'$ as a binary relation of types 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 3.3.2.

Using these new type operators, we can define viewpoint adaptation as $\Gamma \vdash x \triangleright T \to T_{\mathrm{r}} \wedge \{\mathsf{M} : \bot..T_{\mathrm{m}} \vee x.\mathsf{M}\}$, where $\Gamma \vdash T\,\mathbf{ro}\,T_{\mathrm{r}}$ and $\Gamma \vdash T\,\mathbf{mu}\,T_{\mathrm{m}}$.

Union Types

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 [96, 17], 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.

3.2.4 Additional Changes to the Calculus

Above we showed how to define mutability of a type and related type operations. To make this work within the type system in a sound way, we need additional changes to other features of the type system.

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.

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 reference 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 3.3.1.

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.

Receiver Parameters of Type Declarations

The use of the receiver parameter in method declarations introduces one inconvenience: For a method with a dependent return type, its return type referring to r cannot be abstracted using a type member, because the type member cannot refer to this parameter, and in the baseline DOT, type members cannot be parameterized.

To overcome this, we allow type member to have a parameter, which can be used in the bounds of the type member. We do not however enable specifying bounds for this parameter, to avoid bringing in the complexity of higher order types into the calculus.

For most type declarations shown here, this parameter is unused and not relevant, so we omit it. It is however supported by the full definition of the calculus and by the mechanized version.

Reference Variables

In the baseline DOT, at runtime, a reference value is a heap location y, which is a 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 reference variables w. Two references w, w' may designate the same location y, but can have different mutabilities in the typing context Γ. To track the correspondence between references and locations, 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.

3.3 Full Description of the roDOT Calculus

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

$x ::=$	**Variable**	$\Gamma ::=$	**Context**
$\mid u$	abstract	$\mid \cdot$	empty
$\mid v$	global	$\mid \Gamma, x : T$	binding
$u ::=$	**Abstract**	$\mid \Gamma, !$	hide
$\mid z$	local	$d ::=$	**Definition**
$\mid s$	self	$\mid \{a = x\}$	field
$\mid r$	receiver	$\mid \{m(z, r) = t\}$	method
$v ::=$	**Global**	$\mid \{A(r) = T\}$	type
$\mid y$	location	$\mid d_1 \wedge d_2$	aggregate
$\mid w$	reference	$T ::=$	**Type**
$t ::=$	**Term**	$\mid \top$	top
$\mid \mathsf{v}x$	var	$\mid \bot$	bottom
$\mid \mathsf{let}\ z = t_1\ \mathsf{in}\ t_2$	let	$\mid \mu(s : T)$	recursive
$\mid \mathsf{let}\ z = \nu(s : T)d\ \mathsf{in}\ t$	let-literal	$\mid \{a : T_1..T_2\}$	field decl
$\mid x_1.a := x_2$	write	$\mid \{m(z : T_1, r : T_3) : T_2\}$	method decl
$\mid x.a$	read	$\mid \{B(r) : T_1..T_2\}$	type decl
$\mid x_1.m\, x_2$	call	$\mid x_1.B(x_2)$	projection
$B ::=$	**Type name**	$\mid T_1 \wedge T_2$	intersection
$\mid A$	type member	$\mid T_1 \vee T_2$	union
$\mid \mathsf{M}$	mutability	$\mid \mathsf{N}$	read-only $\bot$

Figure 3.2: roDOT syntax

The roDOT calculus [43] evolved from DOT with mutable fields. In roDOT, write access to a field is controlled by a reference mutability permission. It is based on an idea of a reference capability represented by a special type member M. A reference can only be used to mutate an object if the type of the reference includes this capability, in the form of a type member declaration $\{\mathsf{M} : \bot..\bot\}$. Without this capability, the field can only be read, but with it, the field can also be written to. The permission applies transitively, in the sense that reading from a read-only reference always produces read-only references.

The roDOT calculus has the type safety property (Theorem 10) – a term that has a type in an empty context can be executed and either reduces to an answer, or executes indefinitely.

The essential property of roDOT is the immutability guarantee (Theorem 11): in order for an object to be mutated, a writeable reference to it must exist, or it must be possible to reach it by a path of writeable fields, starting from a writeable reference – the object must be mutably reachable, as defined in Figure 3.17.

3.3.1 Syntax

The syntax of roDOT is defined in Figure 3.2; the shading highlights changes from the baseline DOT (Figure 2.1). It uses the A-normal form of terms from DOT. To avoid ambiguity, if a variable is used in the position of a term, it is marked as $\mathsf{v}x$. Unlike other versions, the roDOT calculus does not have λ values, but methods are a kind of object member (and cannot be reassigned), so there is a more explicit relationship of a method, the containing object and the reference used to call the method.

Terms

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

When typing the program or a part of it, free variables are assigned a type in a typing context Γ. There are several kinds of variables. *Abstract variables* are variables bound in terms such as let-in terms and method definitions. When the program executes, objects are created on the heap, and variables referring to concrete objects on the heap are substituted in place of the abstract variables. Each object on the heap has a unique location y and one or more references w. In an object on the heap, the values of fields are locations of other objects. In terms, only references may appear. The kind of the variable has no effect on execution or typing.

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.

Types

The types form a lattice, with the top, bottom, union and intersection types. Objects can contain multiple members – fields, methods and type members. Types of objects are formed by intersection of individual declaration types for each member. The declarations are wrapped in a recursive type, so several declarations in one object type can reference each other, using a member type selection $s.A(r)$. The parameter r in type declarations and selections allows parameterizing a type member. This feature is not used in this work and the parameter will be omitted in the text.

The special type member M denotes the mutability of a reference. When accessing an object through a reference which does not have this capability, for example $\{a : T..T\}$, the field can only be read. With it, for example $\{a : T..T\} \wedge \{\mathsf{M} : \bot..\bot\}$, the field can also be written to.

In the declaration of the type member M, the lower bound is always $\bot$, and the upper bound determines the mutability. If the upper bound is also $\bot$, it means the reference is mutable. Otherwise, it is read-only. This way, mutable references are subtypes of read-only references, so a mutable reference can be used anywhere a read-only reference is expected, but not vice versa. We will use $\mathcal{M}_T$ as a shorthand for the type member declaration $\{\mathsf{M} : \bot..T\}$, or just $\mathcal{M}$ when the bound is not important.

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 3.4), which makes it possible to combine multiple mutability declarations into one. A new type N is a read-only version of $\bot$.

An example of a type of an object with a type member A, a field a, method m and a mutability declaration is

$$\mu(s : \{A : T_A..T_A\} \wedge \{a : T_a..T_a\} \wedge \{m(r : T_r, z : T_z) : T_m\}) \wedge \{\mathsf{M} : \bot..\bot\}$$

Declaration of a method allows specifying a more precise type of the receiving reference. This allows the type of the method to require that the receiver be writeable, or allow it to be read-only. It is similar to the ability to annotate the type of the this parameter in Java, used by the Checker Framework [53, 9].

The ability to create dependent types $x.A(r)$ is the defining feature of the DOT calculus. In roDOT, this feature is also used to express the mutability of a reference, by selecting the special type member M.

Certain operations in roDOT require using a read-only version of a type. N is a special type which acts as a read-only version of the bottom type.

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.

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 $\{A(r) = \{\mathsf{M} : \bot..r.\mathsf{M}\}\}$.

3.3.2 Typing

The typing rules in Figure 3.6 describe correctly formed programs. In addition to the typing context Γ, which assigns types to variables, the left side of the

$$\begin{tikzcd}
 & & \boxed{\top} & & \\
 & \boxed{\{\mathsf{M} : \bot..\top\}} \arrow[ur] & & \boxed{\mu(s : R)} \arrow[ul] & \\
\boxed{\{\mathsf{M} : \bot..\bot\}} \arrow[ur] & & \boxed{\mu(s : R) \wedge \{\mathsf{M} : \bot..\top\}} \arrow[ul] \arrow[ur] & & \boxed{\mathsf{N}} \arrow[ul] \\
 & \boxed{\mu(s : R) \wedge \{\mathsf{M} : \bot..\bot\}} \arrow[ul] \arrow[ur] & & & \\
 & & \boxed{\bot} \arrow[ul] \arrow[uurr] & &
\end{tikzcd}$$

Figure 3.3: Type hierarchy in roDOT

typing judgment includes an environment ρ that connects references in the terms to locations of objects on the heap. There is one typing rule for each kind of a term and object member definition. The write term is guarded by a check of the mutability permission on the receiving reference. Reading a field is possible using all references, but to enforce the transitivity of read-only references, the type of the resulting reference is changed, to not be writeable if the source reference is read-only. This is done by making a read-only version of the field type, and adding a mutability declaration with a bound that is a union of the mutability of the source reference and the mutability of the field type.

The type-level operations of extracting a read-only version of a type and its mutability are defined in Figure 3.7.

The typing and subtyping rules are shown in Figures 3.4, 3.6 and 3.8. 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.

Subtyping Rules

The rules of subtyping (in Figure 3.4) can be divided into several categories – rules that make subtyping a partial order (reflexivity and transitivity); rules for the lattice types (top, bottom, union, intersection), including a distributivity rule, and two additional rules for intersection types; rules for the special N type; subtyping of member declarations (fields, methods and types); rules for selection types.

To understand the subtyping relationship between read-only and read-write types, Figure 3.3 shows a part of the type hierarchy.

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

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

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

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1 <: T_2 \\ \Gamma;\rho \vdash T_2 <: T_3\end{array}}{\Gamma;\rho \vdash T_1 <: T_3}\text{(ST-Trans)}$$

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_3 <: T_1 \\ \Gamma;\rho \vdash T_2 <: T_4\end{array}}{\Gamma;\rho \vdash \{a : T_1..T_2\} <: \{a : T_3..T_4\}}\text{(ST-Fld)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1 <: T_3 \\ \Gamma;\rho \vdash T_2 <: T_3\end{array}}{\Gamma;\rho \vdash T_1 \vee T_2 <: T_3}\text{(ST-Or)}$$

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

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

$$\frac{\begin{array}{c}\Gamma;\rho \vdash T_1 <: T_2 \\ \Gamma;\rho \vdash T_1 <: T_3\end{array}}{\Gamma;\rho \vdash T_1 <: T_2 \wedge T_3}\text{(ST-And)}$$

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

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

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

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

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

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

Figure 3.4: roDOT subtyping rules

$$\frac{\Gamma = \Gamma_1, x : T, \Gamma_2}{\Gamma;\rho \vdash x : T}\text{(VT-Var)} \qquad \frac{\begin{array}{c}\Gamma;\rho \vdash x : \mu(s : T) \\ T\ \mathbf{indep}\ s\end{array}}{\Gamma;\rho \vdash x : [x/s]T}\text{(VT-RecE)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash x : T_1 \\ \Gamma;\rho \vdash T_1 <: T_2\end{array}}{\Gamma;\rho \vdash x : T_2}\text{(VT-Sub)} \qquad \frac{\begin{array}{c}\Gamma;\rho \vdash x : [x/s]T \\ T\ \mathbf{indep}\ s \\ \Gamma;\rho \vdash [x/s]T\ \mathbf{ro}\ [x/s]T\end{array}}{\Gamma;\rho \vdash x : \mu(s : T)}\text{(VT-RecI)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash x : T_1 \\ \Gamma;\rho \vdash x : T_2\end{array}}{\Gamma;\rho \vdash x : T_1 \wedge T_2}\text{(VT-AndI)} \qquad \frac{\Gamma;\rho \vdash x : T}{\Gamma;\rho \vdash x : \{\mathsf{M}(r_0) : \bot..\top\}}\text{(VT-MutTop)}$$

Figure 3.5: roDOT variable typing 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 [57]. 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.

Variable Typing Rules

Variables appearing in terms and definitions have types given by the typing rules in Figure 3.5. These rules allow opening and closing recursive types, and deriving intersection types. More types can be derived using subsumption and subtyping.

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

$$\dfrac{\begin{array}{c}\Gamma;\rho \vdash x : T \\ \Gamma\ \mathbf{vis}\ x\end{array}}{\Gamma;\rho \vdash \mathsf{v}x : T}\,(\text{TT-Var})$$

$$\dfrac{\begin{array}{c}\Gamma;\rho \vdash x_1 : \{m(z : T_1, r : T_3) : T_2\} \\ \Gamma;\rho \vdash x_1 : [x_2/z]T_3 \\ \Gamma;\rho \vdash x_2 : T_1 \\ \Gamma\ \mathbf{vis}\ x_1 \qquad \Gamma\ \mathbf{vis}\ x_2\end{array}}{\Gamma;\rho \vdash x_1.m\, x_2 : [x_1/r][x_2/z]T_2}\,(\text{TT-Call})$$

$$\dfrac{\begin{array}{c}\Gamma;\rho \vdash t : T_1 \\ \Gamma;\rho \vdash T_1 <: T_2\end{array}}{\Gamma;\rho \vdash t : T_2}\,(\text{TT-Sub})$$

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

$$\dfrac{\begin{array}{c}\Gamma;\rho \vdash x_1 : T_1 \\ \Gamma;\rho \vdash x : \{a : T_1..T_2\} \\ \Gamma;\rho \vdash x : \{\mathsf{M}(r) : \bot..\bot\} \\ \Gamma\ \mathbf{vis}\ x_1 \qquad \Gamma\ \mathbf{vis}\ x\end{array}}{\Gamma;\rho \vdash x.a := x_1 : T_2}\,(\text{TT-Write})$$

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

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

Figure 3.6: roDOT term typing rules

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 $\{\mathsf{M} : \bot..\top\}$.

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 3.2.4. For each such occurrence, there is a premise $\Gamma\,\mathbf{vis}\,x$ 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.

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-Call 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 $\Gamma \vdash x \triangleright T_2 \to T_5$.

Viewpoint Adaptation

In Section 3.2.4, we described how viewpoint adaptation can be expressed in terms of two new type relations $\Gamma \vdash T\,\mathbf{ro}\,T_\mathrm{r}$ and $\Gamma \vdash T\,\mathbf{mu}\,T_\mathrm{m}$.

The relations are defined together in Figure 3.7. The operation $\Gamma \vdash T\,\mathbf{ro}\,T_\mathrm{r}$ 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 $\Gamma \vdash T\,\mathbf{ro}\,T_\mathrm{r}$, 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.

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.

Definition Typing

Definition typing, shown in Figure 3.8 (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.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.7: roDOT type splitting relations

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.

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

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

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

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

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

Figure 3.8: roDOT definition typing rules

$\Sigma ::=$	**Heap**
$\mid \cdot$	empty heap
$\mid \Sigma, y \rightarrow d$	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
$\mid \rho, w \rightarrow y$	assignment
$c ::=$	**Configuration**
$\mid \langle t; \sigma; \rho; \Sigma\rangle$	
$\Gamma_{\text{h}} ::=$	**Heap Context**
$\mid \Gamma, y/s : R$	

Figure 3.9: roDOT run-time configuration syntax

3.3.3 Runtime Configuration

The syntax of runtime configurations is shown in Figure 3.9. 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.

Type safety and other properties are based on the fact that during execution, the type of the configuration is preserved. The rules for typing a machine config-

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

$$\frac{}{\textbf{inert}\ \cdot}\text{(Inert-Empty)} \qquad \frac{\textbf{inert}\ \Gamma}{\textbf{inert}\ \Gamma, y : S}\text{(Inert-Bind)}$$

Figure 3.10: roDOT inert contexts

uration are in Figure 3.11. As the program executes and new objects are added to the heap, new locations and reference variables are used to refer to the objects. To give the configurations a type, these variables are added to the typing context. Their type is the type of the object, and has a fixed form – it is a recursive type containing declarations of all the object's members, intersected with a declaration of mutability. A typing context that only contains types of this form is called *inert context*. Under an inert context, stronger claims can be made about pes of variables than in a general context [95], and it plays an important role in the proof of safety.

Valid configurations are given a type under an inert context Γ. The rules for typing configurations are given in Figure 3.11. 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 Γ and have the same type except for mutability. The heap must correspond to Γ, which requires that for every location y in Γ, 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.

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

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

$$\frac{}{\Gamma;\rho \vdash \sim .}\text{(CT-EmptyH)}$$

$$\frac{}{\Gamma \sim .}\text{(CT-EmptyE)} \qquad \frac{\Gamma_1;\rho \vdash \Gamma_2 \sim \Sigma}{\Gamma_1;\rho \vdash \Gamma_2, w : T \sim \Sigma}\text{(CT-RefH)}$$

$$\frac{\begin{array}{c}\Gamma_1;\rho \vdash \Gamma_2 \sim \Sigma \\ \Gamma_1, y/s : R;\rho \vdash d : [y/s]R \\ R\ \textbf{indep}\ s\end{array}}{\Gamma_1;\rho \vdash \Gamma_2, y : \mu(s : R) \wedge \{\mathsf{M}(r) : \bot..\bot\} \sim \Sigma, y \to d}\text{(CT-ObjH)}$$

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

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

Figure 3.11: roDOT configuration typing rules

mutability of the field and of the reference w_1.

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 $\Gamma(y) = \mu(s : T) \wedge \{\mathsf{M} : \bot..\bot\}$.

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 3.8. 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

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

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

$$\frac{\begin{array}{c}\Gamma;\rho \vdash x : T\\ \Gamma \textbf{ vis } x\end{array}}{\Gamma, y/s : T_4;\rho \vdash \{a = x\} : \{a : T..T\}}\text{(HT-Fld)}$$

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

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

Figure 3.12: roDOT heap definition typing rules

context and substitutes y for s in T_4. The reason for this is so that in the HT-Met rule shown in Figure 3.8, 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.

3.3.4 Reduction

The operational semantics is defined as small step semantics, with machine configurations (Figure 2.2) consisting of a term in the focus of execution t, a stack σ, heap Σ and an environment ρ. The environment ρ maps references to locations and the heap Σ maps locations to objects. The stack σ is used to evaluate let-in terms. A stack frame contains the second part of the term while the first part (represented by $\Box$ in the frame) is being evaluated. The stack is not used to implement method calls.

The execution starts with the program, an empty stack, empty heap and an empty environment, and proceeds by steps defined in Figure 3.13, until it reaches an answer configuration, which has an empty stack and the focus of execution is a single variable. During execution, new items are added to the heap and the environment (there is no garbage collection).

Reduction is defined in Figure 3.13. 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

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

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

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

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

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

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

$$\frac{}{\textbf{answer}\ \langle \mathsf{v}w; \cdot; \rho; \Sigma\rangle}(\text{Ans-Var})$$

Figure 3.13: roDOT reduction (operational semantics)

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-Call 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.

3.3.5 Example

In roDOT, we can rewrite the example from Section 3.2.2 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 $\{\mathsf{M}(r_0) : \bot..r.\mathsf{M}\}$

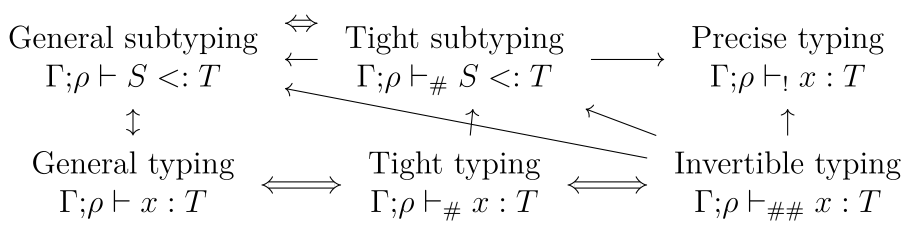
General subtyping
$\Gamma;\rho \vdash S <: T$
Tight subtyping
$\Gamma;\rho \vdash_{\#} S <: T$
Precise typing
$\Gamma;\rho \vdash_{!} x : T$
General typing
$\Gamma;\rho \vdash x : T$
Tight typing
$\Gamma;\rho \vdash_{\#} x : T$
Invertible typing
$\Gamma;\rho \vdash_{\#\#} x : T$

Figure 3.14: Dependencies (→) and equivalence (⇔) between definitions of typing in roDOT

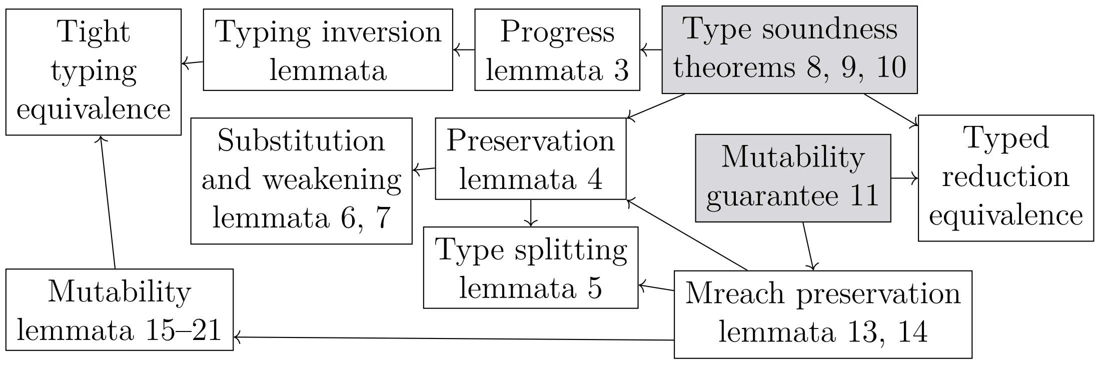

Figure 3.15: Overview of properties and dependencies within proofs of the main theorems

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 $T_{\mathrm{o1}} \triangleq \mu(s : \{a : T \wedge \{\mathsf{M} : \bot..\bot\}\} \wedge \{m_{\mathrm{s}}(z : T \wedge \{\mathsf{M} : \bot..\bot\}, r : \{\mathsf{M} : \bot..\bot\}) : \top\} \wedge \{m_{\mathrm{g}}(z : \top, r : \top) : T \wedge \{\mathsf{M}(r_0) : \bot..r.\mathsf{M}(r_0)\}\})$. Given an initial value x of type $T \wedge \{\mathsf{M} : \bot..\bot\}$, it can be instantiated in $\mathsf{let}\, z = \nu(s : T_{\mathrm{o1}})\{a = x\} \wedge \{m_{\mathrm{s}}(r, z) = r.a := z\} \wedge \{m_{\mathrm{g}}(r, z) = r.a\}\,\mathsf{in}\, t$.

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: $T_{\mathrm{o2}} \triangleq \mu(s : \{A : \bot..\top\} \wedge \{a : T \wedge \{\mathsf{M}(r_0) : \bot..s.A(r_0)\}\} \wedge \{m_{\mathrm{s}}(z : T \wedge \{\mathsf{M}(r_0) : \bot..s.A(r_0)\}, r : \{\mathsf{M} : \bot..\bot\}) : \top\} \wedge \{m_{\mathrm{g}}(z : \top, r : \top) : T \wedge \{\mathsf{M}(r_0) : \bot..r.\mathsf{M}(r_0) \vee s.A(r)\}\})$.

3.4 Type Soundness

Figure 3.14 follows the same scheme as the baseline DOT in Figure 2.8

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 3.13. 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.

An overview of the structure of the proofs is shown in Figure 3.15.

The typed reduction rules for the two interesting cases are shown in Figure 3.16. 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 **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 3 and 4 are examples of progress and preservation for the TR-Read rule.

Lemma 3 (Progress for Read).

If $\Gamma \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0$,	For a configuration with a read term in the focus of execution,
then there exists w_2,	a reference w_2 can be created,
such that $\Gamma \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto$ $\longmapsto \Gamma, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1 \rangle$.	such that a typed step can be taken, where w_2 is added to the typing context and becomes the focus of execution

Lemma 4 (Type preservation for Read).

If $\Gamma \vdash \langle w_1.a; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto$ $\longmapsto \Gamma, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1 \rangle$,	A typed reduction step with a read term in the focus of execution
then $\Gamma, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1 \rangle : T_0$.	results in a configuration with the same type.

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 [64] and [95]. 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 Call terms have the member needed for progress, and preservation. In an inert context, typing and invertible typing are equivalent.

$$\frac{\begin{array}{c} t_0 = w_1.a \\ \Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \qquad \Gamma;\rho_1 \vdash w_1 : \{a : T_4..T_3\} \\ \Gamma;\rho_1 \vdash T_3 \ \mathbf{mu}(r) \ T_7 \\ w_1 \rightarrow y_1 \in \rho_1 \\ \Gamma = \Gamma_3, y_2 : \mu(s_1 : R_1) \wedge \{\mathsf{M}(r_0) : \bot..\bot\}, \Gamma_4 \\ T_2 = \mu(s_1 : R_1) \wedge \{\mathsf{M}(r) : \bot..(T_7 \vee w_1.\mathsf{M}(r))\} \\ y_1 \rightarrow \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma_1 \\ \rho_2 = \rho_1, w_2 \rightarrow y_2 \end{array}}{\Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \Gamma, w_2 : T_2 \vdash \langle \mathsf{v} w_2; \sigma_1; \rho_2; \Sigma_1 \rangle}\,(\text{TR-Read})$$

$$\frac{\begin{array}{c} t_0 = w_1.a := w_3 \\ \Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \\ w_1 \rightarrow y_1 \in \rho_1 \qquad w_3 \rightarrow y_3 \in \rho_1 \\ y_1 \rightarrow \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma_1 \\ \Sigma_2 = \Sigma_1[y_1 \rightarrow \ldots_1 \{a = y_3\} \ldots_2] \end{array}}{\Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \Gamma \vdash \langle \mathsf{v} w_3; \sigma_1; \rho_1; \Sigma_2 \rangle}\,(\text{TR-Write})$$

$$\frac{\begin{array}{c} t_0 = w_1.m\, w_2 \qquad \Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \\ w_1 \rightarrow y_1 \in \rho_1 \\ y_1 \rightarrow \ldots_1 \{m(z, r) = t\} \ldots_2 \in \Sigma_1 \end{array}}{\Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \Gamma \vdash \langle [w_1/r][w_2/z]t; \sigma_1; \rho_1; \Sigma_1 \rangle}\,(\text{TR-Call})$$

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

$$\frac{\begin{array}{c} t_0 = \mathsf{let}\ z = t_1 \ \mathsf{in}\ t_2 \qquad \Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \\ \sigma_2 = \mathsf{let}\ z = \square \ \mathsf{in}\ t_2 :: \sigma_1 \end{array}}{\Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \Gamma \vdash \langle t_1; \sigma_2; \rho_1; \Sigma_1 \rangle}\,(\text{TR-LetPush})$$

$$\frac{\begin{array}{c} t_0 = \mathsf{v} w_1 \qquad \Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \\ \sigma_1 = \mathsf{let}\ z = \square \ \mathsf{in}\ t :: \sigma_2 \end{array}}{\Gamma \vdash \langle t_0; \sigma_1; \rho_1; \Sigma_1 \rangle : T_0 \longmapsto \Gamma \vdash \langle [w_1/z]t; \sigma_2; \rho_1; \Sigma_1 \rangle}\,(\text{TR-LetLoc})$$

Figure 3.16: roDOT typed reduction

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

In Lemma 4, 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 5 , which states that a reference w corresponding to a location y has the read-only version of the type of y.

Lemma 5 (References share read-only types of locations)**.**

If $\Gamma;\rho \vdash y : T_1$,	For a location y with type T_1
and $\Gamma;\rho \vdash T_1$ **ro** T_2,	and a read-only version T_2 of that type
and $\Gamma \sim \rho$ *and* $w \rightarrow y \in \rho$,	and for a reference corresponding to y,
then $\Gamma;\rho \vdash w : T_2$.	the reference shares the type 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 Γ. First, we show that definitions keep their type if references are replaced by locations, by Lemma 6. Then, we use a variant of a substitution Lemma 7 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 6 (Definition dereferencing)**.**

If $\Gamma, s : T_2;\rho \vdash d : T_1$ *and* $\Gamma \sim \rho$,	In a typed definition
then $\Gamma, s : T_2;\rho \vdash [\rho]d : T_1$.	the references can be replaced by locations, preserving the type.

Lemma 7 (Substitution in definition)**.**

If $\Gamma, s : T_3;\rho \vdash d : T_1$,	For a typed definition d in a context with a self variable s of type T_3,
and $s \notin \Gamma$ *and* Γ **vis** y *and* $\Gamma;\rho \vdash y : [y/s]T_3$,	and a location y that has such a type,
then $\Gamma, y/s : T_3;\rho \vdash [y/s]d : [y/s]T_1$.	the location can be substituted for the self variable to get typing under a heap context.

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 8.

Theorem 8 (Typed Progress and preservation)**.**

If $\Gamma_1 \vdash c_1 : T$,	A well typed configuration
then either **answer** c_1,	is either an answer
or exists c_2, Γ_2, *such that* $c_1 \longmapsto c_2$	or can step by typed reduction
and $\Gamma_1, \Gamma_2 \vdash c_2 : T$.	resulting in a configuration with the same type.

$$\frac{\begin{array}{c}\Gamma \vdash \langle t; \sigma; \rho; \Sigma\rangle \textbf{ mreach } y_1 \\ y_1 \to \ldots_1 \{a = y_2\} \ldots_2 \in \Sigma \\ \Gamma;\rho \vdash y_1 : \{a : \bot..\{\mathsf{M}(r) : \bot..\bot\}\}\end{array}}{\Gamma \vdash \langle t; \sigma; \rho; \Sigma\rangle \textbf{ mreach } y_2}\text{(Rea-Fld)}$$

$$\frac{\begin{array}{c}t \textbf{ tfree } w \vee \sigma \textbf{ tfree } w \\ w \to y \in \rho \\ \Gamma;\rho \vdash w : \{\mathsf{M}(r) : \bot..\bot\}\end{array}}{\Gamma \vdash \langle t; \sigma; \rho; \Sigma\rangle \textbf{ mreach } y}\text{(Rea-Term)}$$

Figure 3.17: Mutably reachable objects

By induction on the number of steps, type soundness of typed reduction follows – Theorem 9. 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 10.

Theorem 9 (Typed reduction Safety)**.**
If $\vdash t_0 : T$, — The initial term t_0 is well typed,
then either $\exists w, j, \Sigma, \rho, \Gamma$: — then typed execution terminates in j steps with answer w,
$\vdash \langle t_0; \cdot; \cdot; \cdot\rangle : T \longmapsto^j \Gamma \vdash \langle \mathsf{v}w; \cdot; \rho; \Sigma\rangle$
or $\forall j$: $\exists t_j, \sigma_j, \Sigma_j, \rho_j, \Gamma_j$: — or continues indefinitely.
$\vdash \langle t_0; \cdot; \cdot; \cdot\rangle : T \longmapsto^j \Gamma_j \vdash \langle t_j; \sigma_j; \rho_j; \Sigma_j\rangle$.

Theorem 10 (Safety)**.**
If $\vdash t_0 : T$, — The initial term t_0 is well typed,
then either $\exists w, j, \Sigma, \rho$: — then execution terminates in j steps with answer w,
$\langle t_0; \cdot; \cdot; \cdot\rangle \longmapsto^j \langle \mathsf{v}w; \cdot; \rho; \Sigma\rangle$
or $\forall j$: $\exists t_j, \sigma_j, \Sigma_j, \rho_j$: — or continues indefinitely.
$\langle t_0; \cdot; \cdot; \cdot\rangle \longmapsto^j \langle t_j; \sigma_j; \rho_j; \Sigma_j\rangle$.

3.5 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 Γ 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 3.17. 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.

3.5.1 Proof of the Immutability Guarantee

A new essential property of the type system is the immutability guarantee, expressed by Theorem 11. 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 11 (Immutability guarantee)**.**

If $y \to d \in \Sigma_1$ *and* $\Gamma_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T$,	For an object at some point during well-typed execution,
and $\langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \longmapsto^k \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$,	at any later point,
then either $y \to d \in \Sigma_2$	either the object does not change,
or $\Gamma_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle$ **mreach** y.	or it was reachable by mutable references.

For the proof, we again make use of the typed reduction rules from Figure 3.16. We show two properties of **mreach**: First, Lemma 12 states that if an object is mutated in a reduction step, then it was mutably reachable before the step. Second, Theorem 13 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 12 (Mutated objects are mutably reachable)**.**

If $\Gamma_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T \longmapsto$ $\longmapsto \Gamma_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$,	For any step during execution
and $y \to d \in \Sigma_1$,	and some object on the heap,
then either $y \to d \in \Sigma_2$	either the object is not changed,
or $\Gamma_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle$ **mreach** y.	or it was mutably reachable.

Theorem 13 (Preservation of mutable reachability in one step)**.**

If $\Gamma_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T \longmapsto$ $\longmapsto \Gamma_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$,	For a step of typed execution,
$y \to d \in \Sigma_1$, *and* $\Gamma_2 \vdash \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$ **mreach** y,	and an object mutably reachable after the step,
then $\Gamma_1 \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle$ **mreach** y.	the object must have been reachable before the step.

The proof of Lemma 12 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 13. 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 14, which states that under these conditions, no object becomes mutably reachable.

Lemma 14 (Mutable reachability and focus and stack)**.**

If $\Gamma \vdash \langle t_2; \sigma_2; \rho; \Sigma\rangle$ **mreach** y,
and $\forall x\colon (t_2$ **tfree** $x \vee \sigma_2$ **tfree** $x) \Rightarrow$
$\Rightarrow (t_1$ **tfree** $x \vee \sigma_1$ **tfree** $x)$,
then $\Gamma \vdash \langle t_1; \sigma_1; \rho; \Sigma\rangle$ **mreach** y.

If y is reachable in some cofiguration,

and another configuration differs only in the focus and the stack, but contains all the t-free variables in the focus and on the stack,

then y is reachable in that configuration too.

For TR-Call, Lemma 14 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 15 states that if a variable v_1 is read-write in an inert context Γ with a new location y_2 added, then it was already read-write in the context Γ without y_2. (In other words, it keeps its mutability if the last location y_2 is removed from the context.) We state Lemma 16 for mutability of fields and similar Lemmata 17 and 18 for adding a new reference w_2 to Γ and ρ.

3.5.2 Context Shortening Lemmata

In order to prove mutability in a preceding configuration in the Read and LetNew cases, we need to prove that a variable or a field had a mutable type in a typing context before a new variable has been added. Because adding variables to the typing context can allow more types to be derived for pre-existing variables, we need to show that the new variable does not participate in the typing derivation.

Lemma 15 (Shortening for location mutability)**.**

If $v_1 \neq y_2$, *and* $\Gamma_2 = \Gamma_1, y_2 : T$,
and $\Gamma_2;\rho \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$,
then $\Gamma_1;\rho \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$.

For a variable in a context ending with a different location,

if that variable is mutable in that context,

then it is mutable in a context where that location is removed.

Lemma 16 (Shortening for location field mutability)**.**

If $y_1 \neq y_2$, *and* $\Gamma_2 = \Gamma_1, y_2 : T$,
and $\Gamma_2;\rho \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$,
then $\Gamma_1;\rho \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$.

For a variable in a context ending with a different location,

if that variable has a mutable field in that context,

then it has a mutable field in a context where that location is removed.

Lemma 17 (Shortening for reference mutability).

If $v_1 \neq w_2$, *and* $\Gamma_2 = \Gamma_1, w_2 : T$,	For a variable in a context ending with a different reference,
and $\rho_2 = \rho_1, w_2 \twoheadrightarrow y_2$, *and* $\Gamma_2 \sim \rho_2$,	and a corresponding environment ending with that reference,
and $\Gamma_2;\rho_2 \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$,	if that variable is mutable in that context,
then $\Gamma_1;\rho_1 \vdash v_1 : \{\mathsf{M} : \bot..\bot\}$.	then it is mutable in a context where that reference is removed.

Lemma 18 (Shortening for reference field mutabilitiy).

If $y_1 \neq w_2$, *and* $\Gamma_2 = \Gamma_1, w_2 : T$,	For a variable in a context ending with a different reference,
and $\rho_2 = \rho_1, w_2 \twoheadrightarrow y_2$, *and* $\Gamma_2 \sim \rho_2$,	and a corresponding environment ending with that reference,
and $\Gamma_2;\rho_2 \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$,	if that variable has a mutable field in that context,
then $\Gamma_1;\rho_1 \vdash y_1 : \{a : \bot..\{\mathsf{M} : \bot..\bot\}\}$.	then it has a mutable field in a context where that location is removed.

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 Γ by tight subtyping. That is sufficient to prove Lemma 15, but in Lemma 16, 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 19.

Type Approximation and Restricted Subtyping

In order to prove Lemmas 16 and 18, we need to show that typing which is valid under the typing $\Gamma_1, y_2 : T_1$ is also valid in just Γ. In order to do that, we need to find a derivation that does not use the bounds of any type member of y_2.

To arrive at such a derivation, we define auxiliary, restricted versions of subtyping, in which the use of type member bounds is restricted to just one direction and the use of method subtyping is restricted in order to avoid dealing with noninert contexts. To link the types involved in the original and restricted derivations, we define type approximation relations, which replace type selections and method types with simpler types such as $\top$ and $\bot$.

Type approximations are defined in Figure 3.18, where ϵ can be s, m, or e, and δ is either $\oplus$ or $\ominus$, signifying the direction of the approximation. Restricted subtyping is defined in Figure 3.19, where ϵ can be s or m.

- The *selection inlining approximation* ($\Gamma \vdash T_1 \longmapsto^{\mathrm{s}}_{\delta} T_2$ in Figure 3.19) is a relation between two types which allows replacing type selections by their bounds in one direction δ. The direction $\oplus$ used in covariant positions

$$\frac{}{\top \longmapsto^{\epsilon}_{\delta} \top}(\text{TA}^{\epsilon}\text{-Top}) \qquad \frac{}{\bot \longmapsto^{\epsilon}_{\delta} \bot}(\text{TA}^{\epsilon}\text{-Bot})$$

$$\frac{\Gamma \vdash T_1 \longmapsto^{\epsilon}_{-\delta} T_3 \quad \Gamma \vdash T_2 \longmapsto^{\epsilon}_{\delta} T_4}{\Gamma \vdash \{a : T_1..T_2\} \longmapsto^{\epsilon}_{\delta} \{a : T_3..T_4\}}(\text{TA}^{\epsilon}\text{-Fld})$$

$$\frac{}{\mu(s : T_1) \longmapsto^{\epsilon}_{\delta} \mu(s : T_1)}(\text{TA}^{\epsilon}\text{-Rec}) \qquad \frac{}{v_1.B(x_2) \longmapsto^{\epsilon}_{\delta} v_1.B(x_2)}(\text{TA}^{\epsilon}\text{-Sel})$$

$$\frac{\Gamma \vdash T_1 \longmapsto^{\epsilon}_{\delta} T_3 \quad \Gamma \vdash T_2 \longmapsto^{\epsilon}_{\delta} T_4}{\Gamma \vdash T_1 \wedge T_2 \longmapsto^{\epsilon}_{\delta} T_3 \wedge T_4}(\text{TA}^{\epsilon}\text{-And}) \qquad \frac{\Gamma \vdash T_1 \longmapsto^{\epsilon}_{\delta} T_3 \quad \Gamma \vdash T_2 \longmapsto^{\epsilon}_{\delta} T_4}{\Gamma \vdash T_1 \vee T_2 \longmapsto^{\epsilon}_{\delta} T_3 \vee T_4}(\text{TA}^{\epsilon}\text{-Or})$$

$$\frac{\Gamma \vdash T_1 \longmapsto^{\epsilon}_{-\delta} T_3 \quad \Gamma \vdash T_2 \longmapsto^{\epsilon}_{\delta} T_4}{\Gamma \vdash \{B(r) : T_1..T_2\} \longmapsto^{\epsilon}_{\delta} \{B(r) : T_3..T_4\}}(\text{TA}^{\epsilon}\text{-Typ})$$

$$\frac{\Gamma \vdash_{!} v_1 : \{B(r) : T_1..T_2\} \quad \Gamma \vdash [x_2/r]T_2 \longmapsto^{\text{s}}_{\oplus} T_3}{\Gamma \vdash v_1.B(x_2) \longmapsto^{\text{s}}_{\oplus} T_3}(\text{TA}^{\text{s}}\text{-SelU}) \qquad \frac{\Gamma \vdash_{!} v_1 : \{B(r) : T_1..T_2\} \quad \Gamma \vdash [x_2/r]T_1 \longmapsto^{\text{s}}_{\ominus} T_3}{\Gamma \vdash v_1.B(x_2) \longmapsto^{\text{s}}_{\ominus} T_3}(\text{TA}^{\text{s}}\text{-SelL})$$

$$\frac{}{\mathsf{N} \longmapsto^{\text{m}}_{\oplus} \mathsf{N}}(\text{TA}^{\text{m}}\text{-N}) \qquad \frac{}{\Gamma \vdash T \longmapsto^{\text{s}}_{\delta} T}(\text{TA}^{\text{s}}\text{-Refl})$$

$$\frac{}{\mathsf{N} \longmapsto^{\text{m}}_{\ominus} \bot}(\text{TA}^{\text{m}}\text{-N-Bot})$$

$$\frac{}{\{m(z : T_1, r : T_3) : T_2\} \longmapsto^{\text{m}}_{\oplus} \top}(\text{TA}^{\text{m}}\text{-MetU})$$

$$\frac{}{\{m(z : T_1, r : T_3) : T_2\} \longmapsto^{\text{m}}_{\ominus} \bot}(\text{TA}^{\text{m}}\text{-MetL})$$

$$\frac{}{\mathsf{N} \longmapsto^{\text{e}}_{\oplus} \mathsf{N}}(\text{TA}^{\text{e}}\text{-N}) \qquad \frac{}{v_1.B(x_2) \longmapsto^{\text{e}}_{\oplus} \bot}(\text{TA}^{\text{e}}\text{-SelU})$$

$$\frac{}{\mathsf{N} \longmapsto^{\text{e}}_{\ominus} \bot}(\text{TA}^{\text{e}}\text{-N-Bot}) \qquad \frac{}{v_1.B(x_2) \longmapsto^{\text{e}}_{\ominus} \top}(\text{TA}^{\text{e}}\text{-SelL})$$

$$\frac{}{\{m(z : T_1, r : T_3) : T_2\} \longmapsto^{\text{e}}_{\oplus} \top}(\text{TA}^{\text{e}}\text{-MetU})$$

$$\frac{}{\{m(z : T_1, r : T_3) : T_2\} \longmapsto^{\text{e}}_{\ominus} \bot}(\text{TA}^{\text{e}}\text{-MetL})$$

Figure 3.18: Type approximation

replaces by upper bounds. The result type is a supertype of the original type. The direction $\ominus$ used in contravariant positions replaces by lower bounds. The result type is a subtype of the original type.

- The **method type approximation** ($T_1 \longmapsto^{\text{m}}_{\delta} T_2$ in Figure 3.19) is a relation between two types which replaces method types by $\top$ or $\bot$. The direction $\oplus$ used in covariant positions replaces by $\top$. The result type is a supertype of the original type. The direction $\ominus$ used in contravariant positions replaces by $\bot$. The result type is a subtype of the original type.

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T <: \top}(\mathrm{ST}^{\epsilon}_{\#}\text{-Top})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \bot <: T}(\mathrm{ST}^{\epsilon}_{\#}\text{-Bot})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T <: T}(\mathrm{ST}^{\epsilon}_{\#}\text{-Refl})$$

$$\frac{\rho \vdash T_1 \approx T_2}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_2}(\mathrm{ST}^{\epsilon}_{\#}\text{-Eq})$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_2\\ \Gamma;\rho \vdash^{\epsilon}_{\delta} T_2 <: T_3\end{array}}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_3}(\mathrm{ST}^{\epsilon}_{\#}\text{-Trans})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_1 \vee T_2}(\mathrm{ST}^{\epsilon}_{\#}\text{-Or1})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_2 <: T_1 \vee T_2}(\mathrm{ST}^{\epsilon}_{\#}\text{-Or2})$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_3\\ \Gamma;\rho \vdash^{\epsilon}_{\delta} T_2 <: T_3\end{array}}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 \vee T_2 <: T_3}(\mathrm{ST}^{\epsilon}_{\#}\text{-Or})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \mathsf{N} <: \mu(s : T)}(\mathrm{ST}^{\epsilon}_{\#}\text{-N-Rec})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 \wedge T_2 <: T_1}(\mathrm{ST}^{\epsilon}_{\#}\text{-And1})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 \wedge T_2 <: T_2}(\mathrm{ST}^{\epsilon}_{\#}\text{-And2})$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_2\\ \Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_3\end{array}}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 <: T_2 \wedge T_3}(\mathrm{ST}^{\epsilon}_{\#}\text{-And})$$

$$\frac{\Gamma \vdash_{!} v : \{B(r) : T_1..T_2\}}{\Gamma;\rho \vdash^{\mathrm{s}}_{\oplus} v.B(x_2) <: [x_2/r]T_2}(\mathrm{ST}^{\mathrm{s}}_{\#}\text{-SelU})$$

$$\frac{\Gamma \vdash_{!} v : \{B(r) : T_1..T_2\}}{\Gamma;\rho \vdash^{\mathrm{s}}_{\ominus} [x_2/r]T_1 <: v.B(x_2)}(\mathrm{ST}^{\mathrm{s}}_{\#}\text{-SelL})$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash^{\epsilon}_{-\delta} T_3 <: T_1\\ \Gamma;\rho \vdash^{\epsilon}_{\delta} T_2 <: T_4\end{array}}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \{a : T_1..T_2\} <: \{a : T_3..T_4\}}(\mathrm{ST}^{\epsilon}_{\#}\text{-Fld})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \mathsf{N} \wedge \{\mathsf{M}(r_0) : \bot..\bot\} <: \bot}(\mathrm{ST}^{\epsilon}_{\#}\text{-N-M})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \mathsf{N} <: \{B(r) : T_1..T_2\}}(\mathrm{ST}^{\epsilon}_{\#}\text{-N-Typ})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \mathsf{N} <: \{a : T_1..T_2\}}(\mathrm{ST}^{\epsilon}_{\#}\text{-N-Fld})$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash_{\#} T_3 <: T_1\\ \Gamma, z : T_3;\rho \vdash T_6 <: T_5 \quad \Gamma, z : T_3, r : T_6;\rho \vdash T_2 <: T_4\end{array}}{\Gamma;\rho \vdash^{\mathrm{s}}_{\delta} \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}}(\mathrm{ST}^{\mathrm{s}}_{\#}\text{-Met})$$

$$\frac{T = \{B(r) : T_1 \vee T_3..T_2 \wedge T_4\}}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \{B(r) : T_1..T_2\} \wedge \{B(r) : T_3..T_4\} <: T}(\mathrm{ST}^{\epsilon}_{\#}\text{-TypAnd})$$

$$\frac{}{\Gamma;\rho \vdash^{\epsilon}_{\delta} T_1 \wedge T_2 \vee T_3 <: (T_1 \wedge T_2) \vee (T_1 \wedge T_3)}(\mathrm{ST}^{\epsilon}_{\#}\text{-Dist})$$

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

$$\frac{\Gamma;\rho \vdash^{\epsilon}_{-\delta} T_3 <: T_1 \quad \Gamma;\rho \vdash^{\epsilon}_{\delta} T_2 <: T_4}{\Gamma;\rho \vdash^{\epsilon}_{\delta} \{B(r) : T_1..T_2\} <: \{B(r) : T_3..T_4\}}(\mathrm{ST}^{\epsilon}_{\#}\text{-Typ})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{!} v : \{B(r) : T_1..T_2\}\\ [x_2/r]T_2 \longmapsto^{\mathrm{m}}_{\oplus} T_3\end{array}}{\Gamma;\rho \vdash^{\mathrm{m}}_{\oplus} v.B(x_2) <: T_3}(\mathrm{ST}^{\mathrm{m}}_{\#}\text{-SelU})$$

$$\frac{\begin{array}{c}\Gamma \vdash_{!} v : \{B(r) : T_1..T_2\}\\ [x_2/r]T_1 \longmapsto^{\mathrm{m}}_{\ominus} T_3\end{array}}{\Gamma;\rho \vdash^{\mathrm{m}}_{\ominus} T_3 <: v.B(x_2)}(\mathrm{ST}^{\mathrm{m}}_{\#}\text{-SelL})$$

Figure 3.19: Restricted subtyping

- The **selection type approximation** ($T_1 \longmapsto^{e}_{\delta} T_2$ in Figure 3.19) is a relation between two types which allows replacing type selections types by $\top$ or $\bot$. The direction $\oplus$ used in covariant positions replaces by $\bot$. The direction $\ominus$ used in contravariant positions replaces N by $\top$.

- **One-way tight subtyping** ($\Gamma; \rho \vdash^{s}_{\delta} T_1 <: T_2$ in Figure 3.19) is a limited variant of tight subtyping, which allows using the selection rules only in one direction δ.

- **No-method tight subtyping** ($\Gamma; \rho \vdash^{m}_{\delta} T_1 <: T_2$ in Figure 3.19) is a limited variant of tight subtyping, which allows using the selection rules only in one direction δ, and does not have subtyping between method types.

In Lemma 19, the premise T_2 **nosel** y_2 means that T_2 does not contain type selections involving y_2. The # sign indicates the use of tight subtyping.

Lemma 19 (Shortening for subtyping)**.**

If $\Gamma_2 = \Gamma_1, y_2 : T_1$,	In a context ending with a location,
and $\Gamma_2;\rho \vdash_{\#} T_2 <: \{\mathsf{M} : \bot..\bot\}$,	where T_2 is a mutable type
and T_2 **nosel** y_2,	which does not contain selections from that location,
then $\Gamma_1;\rho \vdash T_2 <: \{\mathsf{M} : \bot..\bot\}$.	then T_2 is mutable in a context where that location is removed.

$$\begin{tikzcd}
T_2 \arrow[rr, dash, "\Gamma_2;\rho \vdash_{\#} T_2 <: M"] \arrow[d, equal] \arrow[dr, "s", "\oplus"'] & & M \arrow[d, "s", "\oplus"'] \\
T_2 \arrow[r, dash, "\Gamma_2;\rho \vdash^{s}_{\#} T_2 <: T_3"'] \arrow[d, equal] & T_3 \arrow[r, dash, "\Gamma_2;\rho \vdash^{s}_{\#} T_3 <: T_4"'] & T_4 \arrow[d, equal] \\
T_2 \arrow[rr, dash, "\Gamma_2;\rho \vdash^{s}_{\#} T_2 <: M"] \arrow[dd, equal] \arrow[dr, "m", "\oplus"'] & & M \arrow[d, "m", "\oplus"'] \\
 & T_5 \arrow[r, dash, "\Gamma_2;\rho \vdash^{m}_{\#} T_5 <: T_6"'] \arrow[d, equal] \arrow[d, bend right, "e", "\oplus"'] & T_6 \arrow[d, equal] \arrow[d, bend right, "e", "\oplus"'] \\
T_2 \arrow[r, dash, "\Gamma_1;\rho \vdash_{\#} T_2 <: T_5"'] \arrow[d, equal] & T_5 \arrow[r, dash, "\Gamma_1;\rho \vdash_{\#} T_5 <: M"'] & M \arrow[d, equal] \\
T_2 \arrow[rr, dash, "\Gamma_1;\rho \vdash_{\#} T_2 <: M"] & & M
\end{tikzcd}$$

Figure 3.20: Diagram of the progression of the proof for Lemma 19

Proof sketch of Lemma 19. 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.

The progression of the proof is shown in Figure 3.20, where M is either $\bot$ or $\{\mathsf{M}(r) : \bot..\bot\}$. In multiple steps, starting with the subtyping $\Gamma_2;\rho \vdash T_2 <: \{\mathsf{M} : \bot..\bot\}$, we approximate the types on both sides to get a derivation in a restricted version of subtyping.

For the first step, we use the type approximation 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 approximation 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 approximation 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 approximation $\longmapsto^{\mathrm{e}}$ replaces the remaining selections on y_2 by $\top$ or $\bot$, in types occurring in intermediate steps of the subtyping derivation.

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 (shown by double lines in Figure 3.20). Therefore after all the steps, we get $\Gamma_1;\rho \vdash T_2 <: \{\mathsf{M} : \bot..\bot\}$.

$\square$

3.5.3 Finishing the Immutability Guarantee Proof

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 17 and 18. 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 20 to show that the field has type $\{a : \bot..\{\mathsf{M} : \bot..T_7\}\}$, and then by Lemma 18, it must have had the same type before.

Lemma 20 (Mutability of a type by subtyping).

If $\Gamma;\rho \vdash T_1\ \mathbf{mu}(r)\ T_2$,	If T_1 has mutability T_2,
then $\Gamma;\rho \vdash \top <: T_2$	then either it is a top-like mutability (meaning read-only),
or $\Gamma;\rho \vdash T_1 <: \{\mathsf{M}(r) : \bot..T_2\}$.	or it can be derived by subtyping with a mutability declaration.

For the object, we need Lemma 21 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 Γ.

Lemma 21 (Mutability of a reference by typing in a shortened context)**.**

If $w_1 \neq w_2$,	For two different references,
and $\Gamma, w_2 : T_2;\rho_2 \vdash_{\#} w_1.\mathsf{M}(r) <: \bot$,	where one is mutable in a context containing the other reference,
and $\rho_2 = \rho_1, w_2 \to y_2$ *and* $\Gamma, w_2 : T_2 \sim \rho_2$,	which is the last one in the context and the corresponding environment,
then $\Gamma;\rho_1 \vdash w_1 : \{\mathsf{M}(r_2) : \bot..\bot\}$.	the mutability of the first reference can be derived by typing in a context without the second reference.

We use the $\longmapsto^{\mathrm{s}}_{\oplus}$ relation from the proof of Lemma 19 above to show this. It simplifies $w_1.\mathsf{M}(r)$ to its bound: $\Gamma, w_2 : T_2 \vdash w_1.\mathsf{M}(r) \longmapsto^{\mathrm{s}}_{\oplus} T_1$. 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 Γ.

3.6 Mechanization

We mechanized the roDOT calculus in Coq, starting from the implementation of field-mutable DOT by Ifaz Kabir. The mechanization was done in collaboration with Yufeng Li from University of Waterloo, and is attached to this thesis as Attachment A.1.

In this section, we describe the main changes that had to be done and the challenges encountered in that process. Finally, Section 3.6 shows how the mechanized definitions and theorems correspond to those provided in this thesis.

Differences Between the Presentation and Mechanization

There are several differences between the text presentation in the thesis and the mechanization, in order to improve readability, organization and extensibility of the code.

- The mechanized syntax uses a locally nameless representation, where variables bound in terms and literals are represented by de Bruijn indices rather than variable names.

- Definitions of an object represented by a list of definitions (while they are structured by a binary intersection in the text definition, the structure is irrelevant).

- The mechanization represents multiple variants of the calculus, where the syntax is shared, but the differences in typing rules are achieved through the typing mode mechanism described above.

- The mechanized syntax contains additional constructor such as `trm_apply`, `lit_fun`, `ctx_stop`, `item_fun`. Hoverer, the typing rules related to these constructors are disabled by the typing mode feature checks explained above. Since the theorems are involve typed terms and machine configurations, these additional constructors are irrelevant.

- The kind of a variable (reference/location, etc.), which is determined by the variable letter in the text version, is kept track of in the typing context in the mechanization.
- The special handling of the self-reference in the typing of object literals and objects on the heap is implemented using the `objctx` parameter of typing. This parameter is a parameter of each typing judgment in addition to the usual typing context. This allows typing object literals and objects on the heap with the same typing rules, passing `objctx_lit` for typing object literals and `objctx_heap` for typing heap items. For term typing, the objctx is empty.
- To ensure no conflicts of variable names, definitions and theorems contain additional well-formedness and freshness premises where necessary, which are omitted in the text.
- For the purpose of generating fresh variables and ensure no conflict of variable names between entries of the machine configuration and types in the typing context, machine configurations carry a an additional entry `Lv`, which stores all the variable names that appeared at any step of the execution so far.
- Definition types and object literals are not part of the definition of terms (`trm`) and types (`typ`) directly, but use separate definitions (`lit`, `def`) used through constructors `typ_rcd`, `lit_obj`.
- We use additional definitions to avoid code repetition and to improve extensibility.
- The definitions of transformations and similarity are structured using the type classes feature of Coq, to allow reusing definitions and lemmas for different syntactic elements.

Inductive Definitions

Term, variable, definition typing and subtyping are defined using mutual induction. Coq allows generating fully or partially mutual induction schemes. The proofs must be specifically structured to conform to this scheme - the conclusion is a conjunction of judgments and the typing condition must come first.

Typing Environments

In DOT, all typing relations use the typing context, which was represented by a list of pairs, where the first item is the variable name and the second item is its type. In roDOT, additional information is needed:

- The subtyping relation uses the equivalence based on the correspondence between references and locations in the runtime environment.
- The typing context for typing definitions contains the self variable which is handled in a specific way when typing method definitions.
- Variables in a typing context may be hidden in method bodies.

Induction by Size

Although, for example term typing closely follows the syntax of terms, there is a difference between induction on term typing and the terms itself. There can be multiple typings for the same term, with different size. Also, in term typing, subterms are opened before typing.

In case where the induction hypothesis cannot be directly applied to the subterm occurrence, one may define a metric such as induction depth, then use inductive scheme on natural numbers.

Typing Modes

Our Coq definitions and theorems support extensibility, which allows extending the calculus with additional typing rules, and verifying multiple versions of the calculus simultaneously. This is achieved by parameterizing the definitions and theorems with the parameter `typing_mode`. Possible values of `typing_mode` are defined in `GeneralTyping/TypingMode.v`. Each value represent a version of the calculus, where each version can have a different set of features. Some typing rules and theorems are only available if a specific feature is enabled. This is achieved by a feature check in the rule or theorem, for example looking like `mode_has_mutability typing_mode ->`.

For mechanization of roDOT, the mode representing this version of the calculus is `rodot`.

Mechanized Definitions and Theorems

The following table shows which definition or lemma in the mechanization represents each lemma and definition in this thesis.

Figure 3.2	Definition `avar` in file `Syntax/Vars.v` Definition `varkind` in file `Syntax/Vars.v` Definition `trm` in file `Syntax/Terms.v` Definition `typ_label` in file `Syntax/Labels.v` Definition `ctx` in file `Syntax/Context.v` Definition `def` and `defs`in file `Syntax/Terms.v` Definition `typ` and `dec`in file `Syntax/Types.v`
Figure 3.4	Mutually inductive definition `subtyp` in file `GeneralTyping/GeneralTyping.v`
Figure 3.5	Mutually inductive definition `ty_var` in file `GeneralTyping/GeneralTyping.v`
Figure 3.6	Mutually inductive definition `ty_trm` in file `GeneralTyping/GeneralTyping.v`
Figure 3.7	Mutually inductive definition `ty_incap`, `ty_capbnd` in file `GeneralTyping/GeneralTyping.v`
Figure 3.8	Mutually inductive definition `ty_def`, `ty_defs` in file `GeneralTyping/GeneralTyping.v`
Figure 3.9	Definition `heap` in file `Syntax/Heap.v` Definition `stack` in file `Syntax/Stack.v` Definition `renv` in file `Syntax/Env.v` Definition `config` in file `Syntax/AbstractMachine.v` Definition `objctx` in file `Syntax/Objctx.v`
Figure 3.10	Definition `record_dec` in file `CanonicalForms/RecordTypes.v` Definition `record_type` in file `CanonicalForms/RecordTypes.v` Definition `inert_type` in file `CanonicalForms/Inert.v` Definition `inert` in file `CanonicalForms/Inert.v`

Figure 3.11	Definition `ty_stack` in file `CanonicalForms/ConfigTyping.v` Definition `ty_config` in file `CanonicalForms/ConfigTyping.v` Definition `renv_corr` in file `CanonicalForms/EnvCorrespondence.v` Definition `heap_correspond` in file `CanonicalForms/HeapCorrespondence.v`
Figure 3.12	Mutually inductive definition `ty_def`, `ty_defs` in file `GeneralTyping/GeneralTyping.v`
Figure 3.13	Inductive definition `red`, `answer` in file `OperationalSemantics/OperationalSemantics.v`
Figure 3.14	Definition `subtyp` in file `GeneralTyping/GeneralTyping.v` Definition `subtyp_t` in file `CanonicalForms/TightTyping.v` Definition `precise_flow` in file `CanonicalForms/PreciseTyping.v` Definition `ty_var` in file `CanonicalForms/GeneralTyping.v` Definition `ty_trm_t` in file `CanonicalForms/TightTyping.v` Definition `ty_var_inv` in file `CanonicalForms/FlatInvertibleTyping.v`
Lemma 3	One of the cases of `typed_progress` handled in `typed_progress_read` in file `OperationalSemantics/TypedProgress.v`
Lemma 4	One of the cases of `typed_preservation` in file `OperationalSemantics/TypedPreservation.v`
Figure 3.16	Inductive definition `typed_red` in file `OperationalSemantics/TypedOperationalSemantics.v`
Lemma 5	Lemma `reference_incap_type` in file `CanonicalForms/ReferenceTypes.v`
Lemma 6	Lemma `ty_deref_defs` in file `OperationalSemantics/HeapInstantiate.v`
Lemma 7	Lemma `instantiation_defs` in file `CanonicalForms/HeapCorrespondence.v`
Theorem 8	Theorem `typed_progress` in file `OperationalSemantics/TypedProgress.v` and Theorem `typed_preservation` in file `OperationalSemantics/TypedPreservation.v`
Theorem 9	Theorem `typed_red_soundness` in file `OperationalSemantics/TypedSoundness.v`
Theorem 10	Theorem `soundness_initial` in file `/Safety.v`
Figure 3.17	Inductive definition `mut_reach` in file `Mutability/MutableReachability.v`
Theorem 11	Theorem `immutability_guarantee` in file `Mutability/ImmutabilityGuarantee.v`
Lemma 12	Lemma `mutated_mreach` in file `Mutability/ImmutabilityGuarantee.v`
Theorem 13	Theorem `mreach_preservation` in file `Mutability/ImmutabilityGuarantee.v`
Lemma 14	Lemma `mreach_preserve_vars` in file `Mutability/ImmutabilityGuarantee.v`
Lemma 15	Lemma `shortening_typ_loc` in file `CanonicalForms/Shortening.v`
Lemma 16	Lemma `shortening_fld_typ_loc` in file `CanonicalForms/Shortening.v`
Lemma 17	Lemma `shortening_typ_ref` in file `CanonicalForms/Shortening.v`
Lemma 18	Lemma `shortening_fld_typ_ref` in file `CanonicalForms/Shortening.v`
Lemma 19	Lemma `shortening_subtyp_typ_loc` in file `CanonicalForms/Shortening.v`
Lemma 20	Lemma `musub` in file `Mutability/ImmutabilityGuarantee.v`
Lemma 21	Lemma `wmu` in file `Mutability/ImmutabilityGuarantee.v`

3.7 Related Work

The distinction of read-only and mutable references is relevant in many programming languages. There is so far not a consensus on handling this issue and each language has a different approach.

3.7.1 Read-only References in Programming Languages

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 [104]. 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 [13]. 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 [37, 103, 19]. 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 thesis 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.

3.7.2 Reference Mutability Type Systems

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

Javari [106] 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 [60], and an implementation in the Checker Framework [87], 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 3.3.5:

```
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 [59] 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) [110] 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 [38] 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 [52] 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 3.3.5 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 immutability and uniqueness in a multi-threaded context for safe parallelism.

3.7.3 Mutability in DOT Calculi

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 [93] 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.

The lambda values used to represent methods and cells used to represent fields are separate items on the heap. As an example, an object type with a field a of type T, a getter m_g and a setter m_s would have the type $T_\text{o} = \mu(s : \{a : \mathsf{Ref}\ T\} \wedge \{m_\text{g} : T\} \wedge \{m_\text{s} : \forall(x : T)T\})$ and be defined as $\nu(s : T_\text{o})\{a = \mathsf{ref}\ 1\} \wedge \{m_\text{g} = !s.a\} \wedge \{m_\text{s} = \lambda(x : T).(s.a := x)\}\})$.

A move towards a more direct definition of mutable fields has been made in kDOT [64], 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.

Reference mutability has been also defined for System $\text{F}_{<:}$ [68]. In the System $\text{F}_{<:\text{M}}$ calculus, read-only types are created by the $\mathsf{readonly}$ type operator. A reference of a record type $\{a : T\}$ can be used to read and write the field a. However, with a read-only type $\mathsf{readonly}\{a : T\}$, the field can only be read. To ensure transitivity of read-only references, such read produces a read-only type – the typing rule for read-only read is:

$$\frac{\Gamma \mid \Sigma \vdash x : \mathsf{readonly}\ \{a : T\}}{\Gamma \mid \Sigma \vdash x.a : \mathsf{readonly}\ T}$$

The existence of two typing rules for reading a field means that it has to be known whether the access is read-only or mutable. On the other hand, in roDOT there is only one typing rule for field reads, which can be polymorphic in mutability thanks to the use of a dependent type.

4. Method Purity for roDOT

In the previous chapter, we introduced roDOT, a formal calculus which extends the DOT calculus with the support for reference mutability. In this chapter, we will use that as a foundation to explore a closely related topic of *pure methods* – methods that behave like mathematical functions.

As we explained before, in typical practical object-oriented programming languages, such as Java or Scala, references by default allow mutation of objects. Some references in a program are, however, intentionally never used to mutate objects, and are used in a read-only way. This distinction motivates reasoning about reference mutability in object-oriented languages.

Following a similar pattern, methods in object-oriented languages are typically allowed to exhibit several behaviors which are not possible in mathematical functions – methods may cause side effects, have non-deterministic results, and may not terminate. Still, many methods intentionally avoid these behaviors and are designed behave like mathematical functions. Such methods are called *pure*.

Recognizing which methods in a program are pure is of great interest. Because pure methods affect program execution in a limited way, as opposed to non-pure methods, knowing which methods are pure eases analysis of program behavior (both manual and automatic). Furthermore, this knowledge may help improving the program, either in source code form or during compilation, by allowing modifications that are guaranteed to not change the program's behavior, such as caching the results of method calls [56] end eliminating common computations [67].

In this chapter, we explore in which ways method purity can be defined within the context of the roDOT calculus, both from the perspective of static typing and run-time semantics, and also how pure methods can be used in programs.

Generally speaking, purity comprises three distinct properties: (1) side-effect-freedom, (2) determinism, and (3) termination. We particularly focus on a single aspect of method purity – side effect freedom (SEF), as this aspect is most related to roDOT's distinguishing feature of expressing reference mutability with types.

We extend the roDOT calculus with the concept of side-effect-free methods. Informally speaking, in mutable DOT calculi, a method of an object has side effects if calling it can result in modification of the receiving object or other existing objects. On the other hand, side-effect free methods never mutate objects that existed on the heap before the method was called.

We show that the read-only reference types, as defined in roDOT, can be used to statically ensure that certain methods are side-effect free. We use the idea from ReIm [59], which is that methods that have only read-only parameters cannot mutate existing object thanks to the transitivity property of read-only references. Based on that, we pose the *side-effect-free guarantee* (SEF guarantee) by which the type system ensures that methods of certain types are side-effect free.

Applying the idea of recognizing SEF methods by argument types to roDOT required dealing with several challenges specific to Scala and roDOT. One important challenge is that in order to state and prove the SEF guarantee, we needed a way to test whether a given type is read-only. As we will explain, this is not possible in the original roDOT type system as defined in Section 3.3. To make it possible, roDOT's type system required just a few small changes, but necessitated re-working a significant part of the soundness proof. We substantiate the changes to the roDOT type system, and develop a new proof of soundness. This proof is based on new definition of auxiliary typing judgments, defined in multiple layers, where each layer supports different features of roDOT's type system.

Another challenge was formally defining and proving the SEF guarantee in the updated calculus. The roDOT operational semantics says that fresh heap addresses are chosen during method execution. Therefore, after calling a SEF method, these heap addresses can be different, yet the heap still has the same overall structure. We formally define a concept of similarity of heaps in roDOT to describe this relation. We prove the SEF guarantee by simulating the execution of a SEF method with a similar execution, where writeable references are removed, and by applying roDOT's immutability guarantee.

As an application of the SEF guarantee, we show that our definition of side-effect-free methods can also be used to guarantee that programs invoking calls to SEF methods can be safely transformed without changing their behavior. In particular, we state a transformation guarantee, which says that in a roDOT program, calls to SEF methods can be safely reordered, without changing the outcome of the program (assuming execution on single threaded abstract machine as defined in roDOT's semantics).

Formalizing program transformations in roDOT has to deal with specific issues, such as the fact that program code and values are mixed together on the program heap, or the heap similarity mentioned above. In order to deal with these issues, we design a general framework for roDOT, which provides a general way to define program transformations, defines what properties a safe program transformation must have. The framework provides a general theorem about lifting the safety of transformation from execution of a small piece of code to execution of the whole program. We prove the transformation guarantee using the SEF guarantee and the lifting theorem.

We formally defined and proved the SEF and transformation guarantees within the updated roDOT, and extended the mechanization of roDOT from the previous chapter with the new definitions, theorems and proofs.

Outline The rest of the chapter is organized as follows: In Section 4.1, we introduce the concepts of purity and side-effect-freedom. In Section 4.2, we describe our approach for defining these concepts within roDOT. We look deeper into the aspects of side-effect freedom in roDOT from different perspectives, and provide informal definitions of side effect freedom conditions. We show how these perspectives are related to each other, and link them together with informal versions of our guarantees.

In Section 4.3, we motivate and show the necessary updates to roDOT's type system, and describe a new proof of soundness for the updated calculus. In Section 4.4, we formally state the SEF guarantee and present its proof.

In Section 4.5, we look into the safe transformation enabled by the SEF guarantee. We define a general framework for working with safe transformations in roDOT. Then we define a particular transformation, swapping two method calls, which is safe on the condition that both methods are side-effect free. We prove the safety of this transformation using the SEF guarantee.

In Section 4.6, we describe how we updated our mechanization of roDOT to support the new features an properties. In section Section 4.7, we compare our approach to other approaches to purity in object-oriented languages.

The content of this chapter is based on the research paper Pure methods for roDOT [44].

4.1 Side-effect Freedom and Purity in Programming Languages

A feature common to many object-oriented programming languages is that execution of a method can have side effects, such as creating new objects on the heap or modifying (mutating) existing objects. Often, causing side effects is the primary purpose of a method. For example, a setter method modifies a field of the receiving object. Such effects are also the reason why, in general, execution of a method cannot be treated as evaluation of a function in a mathematical sense, because every call of a method with possible side effects can produce different results.

That being said, many methods in object-oriented programs are actually designed as side-effect-free and meant to work like pure mathematical functions, producing the same result on each invocation. An example of such methods are getters, or generally, computations based solely on the arguments passed into the method. Creating methods without side effects is also often considered to be a good practice, because it reduces hidden dependencies, and these methods can be used more freely without the fear of unwanted interaction of their effects.

For example, the program code fragment `val x = computeX() ; val y = computeY()`, which involves two side-effect-free methods, can be transformed to `val y = computeY() ; val x = computeX()` by swapping the order of method calls without any observable change in the program behavior and semantics. Writing side-effect-free methods also enables a greater degree of parallelization (concurrency) and, in general, makes it easier to understand the program behavior. Therefore, the issue of purity is relevant to most mainstream object-oriented programming languages, such as Java, C++, C# and Scala.

However, in common programming languages, pure functions and methods with effects are typically unified under a single concept of a method, and there is no way to express, check and make use of method purity at the language level. The idea that a method is pure can be expressed using an annotation (see, e.g., Checker Framework [45, 10] and Code Contracts [48]), but one must look into the documentation of such an annotation for the exact meaning of purity, and there may be limited possibilities of checking automatically whether the annotation is applied properly.

In the context of Java, ReIm [59] introduced annotations with a formal meaning, which give rise to a type system that allows to recognize side-effect-free methods using the types of their parameters – if all parameters of a method, including the receiver, have read-only types, the method cannot get hold of a writeable reference to an existing object, so it is necessarily side-effect free. The advantage of this general approach, based on the usage of static type systems for reasoning about purity and side-effect freedom, is the possibility to prove soundness of such annotations.

Scala favors a functional programming style, so Scala programs are likely to contain more methods (than Java programs) that can be identified as side-effect-free. Our objective in this chapter is to design a type system that guarantees side effect freedom for Scala methods and supports advanced language features present in Scala.

The original DOT calculus does not model mutation of objects, so purity cannot be addressed there, but roDOT, defined in Chapter 3 has mutable fields and a type system feature to distinguish read-only and mutable references. roDOT also provides an *immutability guarantee*, which we will make use of in this chapter: an object can only be mutated if there is a path of mutable references to it from the code being executed.

4.2 Method Purity for roDOT

We use roDOT, introduced in Chapter 3, as a baseline for this chapter. The reasons to choose roDOT for this task are that unlike previous DOT calculi, it has mutable fields and a type system feature to distinguish read-only and mutable references. It also provides the immutability guarantee (Theorem 11): an object can only be mutated if there is a path of mutable references to it from the code being executed.

In this section, we look at the possible meaning of method purity in roDOT, and informally define several purity conditions for roDOT, which will be formally defined and thoroughly discussed in later sections.

In Section 4.2.1 we look at the three components of purity: side-effect freedom, determinism and termination. In the rest of the chapter, we will focus on the side-effect freedom component.

In Section 4.2.2, we identify three perspectives to look at purity. Each perspective will lead us to different purity conditions, and the main results of this chapter will be theorems linking these perspectives together.

The following sections each describe one of the perspectives: Section 4.2.3 defines conditions for the behavior of SEF methods. To be able to recognize SEF methods statically, in Section 4.2.4 we define a sufficient condition for a method to be SEF based on the types of its parameters, analogous to ReIm [59]. Section Section 4.2.6 describes how a program containing calls to pure or SEF methods can be safely modified without changing its result. In Section 4.2.7, we briefly look at determinism in roDOT (determinism is otherwise not in the focus of this thesis).

4.2.1 Components of Purity

In order for a method to be pure – behave like a mathematical function – it needs to have the following 3 properties.

1. **Side-effect freedom.** Side effects are modifications of global state or IO operations. In roDOT, there is no IO and the global state is the heap.

Methods can have side effects of creating or modifying objects on the heap. We discuss side effect in Section 4.2.3.

2. **Determinism.** The result of the method needs to be determined by the arguments and the global state. In roDOT, the global state is the heap. roDOT does not model non-deterministic choices, but the result of instantiating an object is non-deterministic. We discuss determinism in Section 4.2.7.
3. **Termination.** The method must eventually produce a result and return to the caller – it cannot enter infinite recursion.

In literature, the treatment is inconsistent In this thesis, we will focus on the SEF property.

The discussing in the following section applies to purity, also other properties. We briefly look at the properties. From section Section 4.3 onward, we will only SEF property.

4.2.2 Viewing Purity from Three Perspectives

In this section, we informally define the meaning of side-effect freedom in roDOT, and informally state the main results of this thesis: the SEF guarantee and the transformation guarantee.

We structure our work around an observation that (in any programming language or calculus), we can look at side-effect freedom from different perspectives:

1. Recognize which methods are pure statically at compile time, using types.
2. Define what events can happen or cannot happen at runtime when a pure method is executed.
3. Differentiate SEF methods from general methods based on how they can be safely used in programs, in ways not generally safe for methods that are not known to be SEF.

For each of these perspectives, we will state a *SEF condition*, each giving a different definition of SEF methods in roDOT. First we do it informally in this section, and then formalize the definitions in the following sections. The guarantees then form connections between different SEF conditions.

The first perspective is important because we can see how pure methods are useful. A definition of purity gives us guarantees about what we can do with a pure method. (For example, knowing that inserting a call to a pure method into a correct program will not change the behavior of the program.) It is, however, a view from outside of the method, looking at it as an opaque unit, and does not tell us how to construct a pure method or check it.

The second perspective is looking at the small step semantics of the method, where we can say that pure methods are not allowed to perform certain actions, such as mutating an object. This perspective is most directly related to how the semantics of the roDOT calculus is defined. Purity in roDOT viewed from this perspective is described in Section 4.2.3.

Finally, the third perspective is useful because it provides a way to recognize and check that a method is pure, by just looking at the code. We must, however, accept the fact that statically, it will not be possible to recognize all methods that are pure from the first or second perspective, like we accept that type checking may reject some programs that would actually execute correctly. We give the view from perspective is described in Section 4.2.4.

The practical use of a purity system comes when it allows us to look at the code, and based on what we see (from the third perspective) gives us a guarantee about its behavior (second perspective) and how it can be used (the first perspective). The second perspective can be a connecting step between the first and the third perspectives. The SEF guarantee defined in Section 4.4, makes the connection from the third to the second perspective, while the transformation guarantee, detailed in Section 4.5, links the third perspective with the first.

Because the second perspective is most directly related to the semantics of roDOT, we will look at ways to define purity by defining various conditions from this perspective, and then look at what such conditions imply from the first perspective, and how such conditions can be proven from the third perspective.

4.2.3 Run-time SEF Condition (Perspective 2)

Saying that a method is side-effect-free is informally understood as saying that the execution of the method will not perform any actions that are considered to be side effects. This view corresponds to the second perspective on our list.

This perspective is most directly related to the semantics. In roDOT, this means looking at the small step semantics, defining the beginning and end of execution of a method, and defining the SEF condition in terms of the state of execution or the steps performed between the beginning and the end. When looking at the effects caused by method execution, the only relevant part of the machine configuration is the heap (the focus of execution is the part being evaluated, the stack cannot be changed, and the mapping from references to locations is only relevant for typing). The heap can only be modified by two kinds of execution steps: instantiation of an object and writing a value to a field of an object on the heap.

The condition of side-effect-freedom can be stated in multiple versions of varying strength. In the strictest sense, we could say that a SEF method cannot have any effect on the heap at all. That could be achieved by completely forbidding execution of any statements that may change the heap. In roDOT, those are object creation, which extends the heap, and field write statements, which mutate existing objects. This strict SEF condition would mean that no object instantiations and no field writes are allowed. That would, however, be overly restrictive, as object instantiation is one of the basic operations in object-oriented programming. It is therefore usually (such as in [97, 100, 59, 45]) allowed that a SEF method can instantiate new objects, and also write to the fields of those newly instantiated objects. In turn, the only forbidden action is writing to fields of previously existing objects.

Another choice in the definition is when the change to the heap is detected, which leads to different answers to questions such as: (a) Is it allowed to write to a field of an existing object, if the value written is the same as the current value so the object does not actually change? (b) Is it allowed to write to a field of an existing object, if the field is restored to the previous value before the end of the method execution? We choose to allow (a) but not (b), so our definition observes the state of the heap at every moment during the execution of the method. Allowing (b) would lead to a weaker condition, which would check the state of the heap only at the end of the method call. Forbidding (a) would lead to a stronger condition, defined in terms of allowed steps of execution rather than in terms of the state.

Definition (SEF condition for 2nd perspective, informal statement of 7)**.** *An execution of a method is side-effect free, when at every step of execution until returning from the method, the heap contains all the objects from the start of execution in an unchanged state.*

This condition will be formally defined in Section 4.4.1.

4.2.4 Static SEF Condition (Perspective 1)

The static perspective (the first in our list) is useful because it provides a way to check that a method is SEF by looking at the code. We must, however, accept that statically, it will not be possible to recognize all methods that are pure from the second (and third) perspective, just like type checking may reject some programs that would actually execute correctly.

In ReIm [59], SEF methods are recognized by the mutability of the parameters. roDOT uses the same notion of transitive read-only references, therefore it should be possible to use an analogous condition in roDOT.

Definition (2, static SEF condition, informal statement)**.** *A method has a SEF type, if both its parameter and its receiver parameter have read-only types.*

This condition will be formally defined in Section 4.3.1.

The following examples, Example 4 and Example 5, illustrate its ability to recognize SEF methods.

Example 4. A getter defined as $\{m_{get}(r, z) = z.a\}$ can be typed with $\{m_{get}(r : \top, z : \{a : \top..\top\}) : \top\}$. Both $\top$ and $\{a : \top..\top\}$ are read-only types, and therefore the getter is SEF by Definition 2.

Example 5. The method m_{sef} defined by $\{m_{sef}(r, z_a) = (\mathsf{let}\, x = \nu(r_o : R_o) \ldots \mathsf{in}\; z_a.m_a x)\}$ calls a method of its argument, passing a newly allocated object to it. This method has type $\{m_{sef}(r : \top, z_a : T_z) : \top\}$, where $T_z = \{m_a(r : \top, z : \mu(r_o : R_o) \wedge \{\mathsf{M}(r) : \bot..\bot\}) : \top\}$. By Definition 2, m_{sef} is SEF, because it has read-only parameters, even though it calls m_a, which may mutate the heap.

Example 6 shows how viewpoint adaptation transitively ensures that read-only parameters cannot be used to modify existing objects. Example 7 shows how a dependent type can change whether the method is SEF or possibly not.

Example 6. The method defined by $\{m_{va}(r, z) = (\mathsf{let}\, x = z.a\; \mathsf{in}\; x.b := r)\}$ mutates an object stored in a field of the argument z, and therefore is not SEF. This method cannot be typed with a read-only type for the parameter z, because even if the field a has a mutable type, by viewpoint adaptation of fields in roDOT, the variable x would also have a read-only type, so the subsequent write would not be allowed.

Example 7. A method with a type $\{m_{dep}(r : \top, z_a : \{a : \top..\top\} \wedge x.A) : \top\}$ has a parameter with a type dependent on the variable x, which can decide the mutability. This method is recognized as SEF only in contexts where $\mathsf{N} <: x.A$. When $x.A <: \mathcal{M}_\perp$, then the method can (indirectly) mutate the argument.

4.2.5 SEF Guarantee

For the **SEF guarantee** (Theorem 37), we want to be able to claim that a method is SEF based on the type of the method declaration. The SEF guarantee makes the connection from the first to the second perspective.

Theorem (37, informal statement)**.** *Let c_1 be a well-typed machine configuration just prior to executing a method call step $w_1.mw_2$. If, by typing of the receiving reference w_1, the method m has a SEF type, then the execution of the method will be side-effect free.*

This condition is formally defined and discussed in Section 4.4.2.

The SEF guarantee that we pursue requires the parameters to have read-only types. The notion of read-only types is already present in roDOT in the form of type splitting (Figure 3.7), where its purpose was to make a read-only counterpart of a type to support viewpoint adaptation when accessing a field, and to ensure that recursive types are read-only. We will revise the way to recognize read-only types in Section 4.3.1.

4.2.6 Using Pure Methods in roDOT (Perspective 3)

Finally, the third perspective shows why SEF methods are useful. It is, however, a view from outside of the method, and does not tell us how to construct a SEF method or check it.

The practical use of a type system with SEF methods comes when it allows us to look at the code, and based on what we see (from the first perspective) gives us a guarantee about its behavior (second perspective) and how it can be used (the third perspective). An example of this is allowing safe transformations of the program, which can be applied at coding time using IDE-provided code transformations, or at compile time as optimizations. Certain transformations may be not safe if applied to general methods, but are guaranteed to be safe for pure of SEF methods.

For example, calls to SEF methods can be safely reordered, because they do not have side effect that could interfere with each other. If the method is also deterministic, then duplicate calls of the same method with the same arguments can be de-duplicated, because the method always returns the same value. In that sense, the purity and SEF properties allow more flexibility in how the method can be used.

To keep the problem simple, we will look at one particular case of such transformation: reordering calls to SEF methods, which we can narrow down to swapping two SEF method calls. With SEF methods, the code `x1.m1(); x2.m2()` is equivalent `x2.m2(); x1.m1()`. We will express this swapping as a transformation of a program containing two method calls into another program with the calls swapped.

Definition (Call-swapping transformation of programs, informal statement)**.** *A program t_1 is transformed into t_2 by SEF call-swapping, when the programs are the same except in one place, where t_1 calls two methods in succession, but t_2 calls them in the opposite order. Furthermore, within the contexts of typing these method calls, both methods have the same read-only types, and allow both programs to be typed in the same manner.*

Example 8. A chain of calls $\mathsf{let}\, x_1 = x_{\mathrm{o}1}.m_1 x_{\mathrm{a}1}\, \mathsf{in}\, \mathsf{let}\, x_2 = x_{\mathrm{o}2}.m_2 x_{\mathrm{a}2}\, \mathsf{in}\, t$, can transformed by call swapping into $\mathsf{let}\, x_2 = x_{\mathrm{o}2}.m_2 x_{\mathrm{a}2}\, \mathsf{in}\, \mathsf{let}\, x_1 = x_{\mathrm{o}1}.m_1 x_{\mathrm{a}1}\, \mathsf{in}\, t$.

The static condition from the first perspective is already a part of the definition of the transformation. The transformation guarantee then states that this transformation is safe – it does not change the behavior of the program. By that, the guarantee connects the static condition (first perspective) with the call-swapping transformation (third perspective). We use the run-time condition (second perspective) as a connecting step between them in the proof of this guarantee.

Theorem (52, informal statement)**.** *The call-swapping transformation of programs is safe, in the sense that if for any programs t_1 and t_2 related by this transformation, provided that t_1 terminates with an answer c_1, then t_2 also terminates with an answer c_2, which is the same as c_1, except for certain unavoidable differences in variable names and in method bodies.*

The formal definition of the transformation, formal statement of the transformation guarantee and an outline of its proof is provided in Section 4.5.

4.2.7 Determinism in DOT and roDOT

In this thesis, we focus only on the SEF part of purity, but let's look at the deterministic condition briefly.

For a method to be deterministic, it must always return the same value when called with the same arguments and with the same reachable part of the heap.

DOT calculi generally have deterministic semantics – there is no way to express a nondeterministic choice. However, there is one source of non-determinism: the locations of objects on the heap.

Therefore, not all methods in roDOT are deterministic in this sense – if a SEF method returns an object that was allocated by the method, it will return a different object each time it is called, and therefore is not deterministic. This is an important difference, because the returned object can be subsequently mutated, so for example a call to such non-deterministic method cannot be safely deduplicated.

4.2.8 Termination in DOT and roDOT

The DOT calculus does not have a loop construct, but allows recursive calls, making it possible to enter an infinite loop. An example of a method which simply calls itself is $\nu(s : R)\{m(r, z) = r.mz\}$ The possibility of non-termination is necessarily reflected in the soundness theorems (Theorem 1 and Theorem 10).

DOT calculus was designed to be Turing-complete [107] language, which means termination is undecidable. roDOT is designed to preserve the expressive power of DOT, so conjecture that this holds for roDOT as well.

In this thesis, we focus on the SEF property and do not stat type-system guarantees about termination. Possible non-termination is considered in the design of the transformation framework and Theorem 52 in Section 4.5.

4.3 Recognizing Read-only Types

In this section, we formalize the static SEF condition in roDOT given informally in Section 4.2.4. The notion of read-only types, used by this condition, was already defined in roDOT, we identify issues with that definition in regards to this new use.

We fix them by updating the calculus with small changes, which comprise adding one new subtyping rule and one type splitting rule, and one restriction added to the method subtyping rule. The updated calculus is neither a subset or superset of the original, so it is necessary to update the proof of soundness of the calculus and the immutability guarantee, which were proven by hand for the original roDOT [43].

The soundness proof followed the scheme from [95] and uses an auxiliary definition of invertible typing, which allows doing proofs by induction on the typing of variables. This is possible thanks to eliminating possible cycles in the derivation, by forcing the derivation to follow the syntactic make-up of the target type. One of the new subtyping rules, however, breaks this soundness proof, because it introduces new possibilities to derive types in cycles, which cannot be repaired by simply handling additional cases in the original proof. We implemented a new proof based on a different auxiliary typing definition, which avoids cycles by forcing the derivation to construct the target type by following a given order of type constructors (for example, all union types are handled before intersection types).

Compared to the original invertible typing, which handled most features in a single judgment with many rules, the new approach leads to a definition in several layers, where each layer has just a small number of typing rules. We call this set of judgments *layered typing.* In layered typing, we re-prove important properties of invertible typing, so that the new definition fits into the rest of the existing soundness proof, and also prove new properties required for the SEF guarantee.

The rest of this section is structured as follows: in Section 4.3.1, we formalize static SEF condition which was informally stated in Section 4.2.4.

and discuss the meaning of read-only types in roDOT. In Section 4.3.2, we propose small changes to the roDOT calculus to make definitions work for the SEF guarantee. Although the changes are kept to the minimum, we show how they break the safety proof of roDOT, which used invertible typing adopted from the baseline DOT. In Section 4.3.3, we describe the new layered typing that replaces invertible typing in the updated proof and show its important properties.

4.3.1 Static SEF Condition for roDOT

In the **SEF guarantee** (Theorem 37), we claim that a method is SEF based on the type of the method declaration. Our SEF guarantee follows the approach of ReIm [59] and requires the parameters to have read-only types.

Read-only Types in roDOT

The check if a type is read-only was also present in roDOT, but it had a limited purpose – to ensure that recursive types are read-only in VT-RecI (Figure 3.5). It was not based on subtyping, but rather on the relation **ro**, which makes a read-only version of a type using the syntax-based type splitting (Figure 3.7).

This definition did not guarantee that all supertypes of a read-only type are also read-only. As we will explain below, this would be a critical problem for the SEF guarantee, because it would not be guaranteed that a subtype of an SEF method type is also SEF.

We solve this problem by using a different notion of read-only types, based on subtyping with the "read-only bottom" type N.

The purpose of N in the original roDOT was to be the read-only version of the type $\bot$, which was necessary for defining the **ro** relation. Because the bottom type $\bot$ is a subtype of all types, it is inherently mutable. For that reason, the type N was added and made a lower bound of read-only types. That allows us to define read-only types as supertypes of N.

Definition 1 (Read-only types)**.** *A type T is read-only, if* $\Gamma;\rho \vdash \mathsf{N} <: T$.

With Definition 1 settled, we discovered a few problems related to read-only types, which would not allow us to state the SEF guarantee in the original roDOT.

Our proof of the SEF guarantee, specifically Lemma 43 in Section 4.4.5, relies on the idea that if a reference has some read-only type, then any other reference to the same object has that type too. Note that because of subsumption, a variable of a mutable type also has the corresponding read-only types. This essentially means that in any place where a reference is used by virtue of its read-only type, it can be replaced with a read-only version of that reference. With the new Definition 1 of read-only types, this can be stated as:

Lemma 22 (Read-only types are shared by all references)**.**

If $\Gamma \sim \rho$ *and* $\Gamma;\rho \vdash y : T$	For a location y with a type T,
and $\Gamma;\rho \vdash \mathsf{N} <: T$,	if the type is read-only,
then $\Gamma;\rho \vdash w : T$ *for any* w *such that* $\rho[w] = y$.	then any corresponding reference w has that type.

This, however, does not work in the original roDOT, because of union types. Union types were not a part of the baseline DOT, but were added to roDOT in order to be used to define viewpoint adaptation (union types are already a part of Scala's type system), along with the subtyping rules ST-Or, ST-Or1, ST-Or2, ST-Dist, which are shown in Figure 3.4. Using unions, however, we can construct a type that is a supertype of both N and a mutability declaration:

Example 9 (In the original roDOT, counter-example to Lemma 22 in the original roDOT)*.* Let $T_{\mathrm{am}} \coloneqq \{a : T_{\mathrm{a}}\} \vee \mathcal{M}_{\bot}$[1] be a union of some field declaration with

[1] $\mathcal{M}_T$ is a shorthand for the type member declaration $\{\mathsf{M} : \bot..T\}$.

a declaration of mutability, and $T_{\text{bm}} := \{b : T_{\text{b}}\} \wedge \mathcal{M}_{\perp}$ be a type of a writeable reference to some other field b.

The type T_{am} is not a mutable reference type – the mutability declaration contained in this type is not in an intersection with the field declaration. It is read-only, because $\Gamma;\rho \vdash \mathsf{N} <: T_{\text{am}}$ by the rules of subtyping of union types and by ST-N-Fld.

Let y_1 be a location of type T_{bm}, and w_2 be a reference to y_1 with type $T_{\text{b}} := \{b : T_{\text{bm}}\}$. By subtyping of union and intersection types, $\Gamma;\rho \vdash T_{\text{bm}} <: T_{\text{am}}$, so by subsumption, y_1 has type T_{am}. By Lemma 22, w_2 should have also type T_{am}, but in the original roDOT, it does not.

We observe that the read-only type T_{am} in this counter-example is a union of disjoint declarations, so it does not allow accessing the field a_{am}, or any other member. Therefore, T_{am} is no more useful for typing programs than $\top$. In order to make Lemma 22 work, we decided to extend the type system with new subtyping rules to make types like this equivalent to $\top$. These changes will be described in Section 4.3.2.

The SEF condition

A method is statically SEF if the types of its receiver and parameters are read-only according to Definition 1 i.e., they are supertypes of N. Thanks to subsumption and subtyping of method types, the type $\{m(z : \mathsf{N}, r : \mathsf{N}) : \top\}$ is a type bound for methods named m and requires that both the argument and receiver have read-only types.

Definition 2 (Static SEF condition)**.** *A method is statically SEF if it has a type* $\{m(z : \mathsf{N}, r : \mathsf{N}) : \top\}$

In Section 4.4.2, we will show that this condition works because a method must access all objects through the argument or the receiver (capturing values is modeled using fields of the receiver). This bound requires that both the argument and receiver have read-only types, so any references stored in fields of those objects will also be seen as read-only, because of viewpoint adaptation. Therefore, the method will not be able to get a writeable reference to any existing object.

Typing of Method Calls and Subtyping of Method Types

In order for the SEF guarantee (Theorem 37) to work with Definition 2, it is critical that all subtypes of a SEF method type are also SEF. The reason is at the site of a method call, the observed static type of the method is a supertype of the actual type of the method within its containing object, so this is needed to make the connection from the SEF type at a call site to the SEF type of the actual method.

That is why Definition 1 needs to be based on subtyping, so that all supertypes of read-only types are read-only (method types are contravariant in their parameter types).

Still, the type system required one more change related to a possible dependency between the types of method parameters. In roDOT, the type of the receiver r can be a dependent type referring to the other parameter z of the method. If, however, the receiver type depended on the mutability of the parameter z, then while typing the body of the method, it would be possible to derive that z is mutable, even if its type is read-only in the sense of Definition 1. If r has the type $\{A : z.\mathsf{M}..\bot\}$, one can use the typing rules ST-SelL and ST-SelU to derive $z.\mathsf{M} <: \bot$. The change to the rules TT-Call and ST-Met shown in Figure 4.1 prevent this issue by disallowing using method types where the receiver depends on the mutability of the receiver, and subtyping between method types where the receiver does not depend on the mutability of the parameter and methods where it does.

4.3.2 The Updated roDOT Calculus

$$\frac{\Gamma;\rho \vdash \mathsf{N} <: T}{\Gamma;\rho \vdash T\ \mathbf{ro}\ T}\text{(TS-N)}$$

$$\frac{}{\Gamma;\rho \vdash \top <: \mathsf{N} \vee \{\mathsf{M}(r) : T_1..T_2\}}\text{(ST-NM)}$$

$$\frac{\begin{matrix}\Gamma;\rho \vdash T_3 <: T_1 & \Gamma, z : T_3, r : T_6;\rho \vdash T_2 <: T_4 \\ \Gamma, z : T_3;\rho \vdash T_6 <: T_5 & T_6\ \mathbf{indep}\ z \Rightarrow T_5\ \mathbf{indep}\ z\end{matrix}}{\Gamma;\rho \vdash \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}}\text{(ST-Met)}$$

$$\frac{\begin{matrix}\Gamma;\rho \vdash x_1 : \{m(z : T_1, r : T_3) : T_2\} & \Gamma\ \mathbf{vis}\ x_1 & \Gamma\ \mathbf{vis}\ x_2 \\ \Gamma;\rho \vdash x_1 : [x_2/z]T_3 & \Gamma;\rho \vdash x_2 : T_1 & T_3\ \mathbf{indep}\ z\end{matrix}}{\Gamma;\rho \vdash x_1.m\,x_2 : [x_1/r][x_2/z]T_2}\text{(TT-Call)}$$

Figure 4.1: Updated typing rules for roDOT

In the previous section, we defined the static SEF condition, but identified several reasons why this definition would not work as intended in roDOT as-is. We fix these issues by changes to the roDOT calculus, which amount to two new and one modified typing rule:

- A new subtyping rule ST-NM (Figure 4.1) is added, which makes it so that the union of a mutability declaration and the read-only lower bound N is a top-like type (the other direction of subtyping was already a part of the type system).
- A new rule TS-N (Figure 4.1) is added to type splitting, making it so that all types that are read-only by Definition 1 are unaffected by the splitting operation.
- The typing rule TT-Call subtyping rule ST-Met has a new premise (shown highlighted in Figure 4.1), which disallows introducing a dependency between the receiver type and the parameter in typing method calls and subtyping method of method types. This fixes the problem described in Section 4.3.1.

The new rule ST-NM fixes the counter-example to Lemma 22, because now we have $\Gamma;\rho \vdash \top <: T_{\text{am}}$, derived from $\Gamma;\rho \vdash \top <: \mathsf{N} \vee \mathcal{M}_\top$ and $\Gamma;\rho \vdash \mathsf{N} <: \{a : T_{\text{a}}\}$. By subsumption and $\Gamma;\rho \vdash w_2 : \top$, that also means that $\Gamma;\rho \vdash w_2 : T_{\text{am}}$.

Additionally, because subtyping with N becomes the preferred way to determine that a type is read-only. we can use it improve the type splitting relation $\vdash$ **ro** , by extending it with a new rule TS-N, shown highlighted in Figure 4.1.

With that, the condition in VT-RecI that recursive types are read-only, $\Gamma;\rho \vdash T \ \textbf{ro}\ T$, becomes equivalent to Definition 1:

Lemma 23 (Read-only types)**.** $\Gamma;\rho \vdash \mathsf{N} <: T \Leftrightarrow \Gamma;\rho \vdash T \ \textbf{ro}\ T$.

Type Soundness of the Updated Calculus

The changes described above require updating the type soundness proof of the calculus, to show that the changes did not allow invalid programs to be typed. Also the immutability guarantee has to be proven in the updated calculus.

The new subtyping rule ST-NM has a significant effect on the soundness proof, because it makes it possible to derive many additional union types, such as the now top-like type $\mathcal{M} \vee \mathsf{N}$.

The proof of soundness of roDOT before these changes followed the structure of the "simple" soundness proof for DOT [95]. The core part of this proof is to show that if a reference w has some declaration type D (such as a field $\{a : T\}$), then the type associated with w in the typing context Γ is an object type containing D or a more precise declaration of the same member. That means, for $\Gamma;\rho \vdash w : D$, where D is a declaration type, because the types in Γ correspond to the object on the heap ($\Gamma \sim \Sigma$), the actual object referred to by w must contain a corresponding member definition in Σ, and therefore it is safe to access that member.

This proof is based on two alternative definitions of typing for variables – tight typing and invertible typing (see Figure 3.14 in Section 3.4). These alternative definitions are equivalent to normal typing inert contexts, and make it easier to reason about typing, such as making claims about objects on the heap based, on the types of variables that refer to them. The relations between the different versions of typing were shown in Figure 3.14.

Tight typing is used as an intermediate step in equivalence of general and invertible typing. It is very similar to general typing – it has the same rules, except that subtyping rules involving selection types (ST-SelL and ST-SelU in Figure 3.4) are different. In general typing, they use variable typing to find the bounds of the type member, which makes general subtyping mutually recursive with variable typing. In tight typing, the corresponding rules use precise typing, a simpler version of variable typing, which does not have subsumption.

Updating tight typing for our modified rules is straightforward – we apply the same changes as to general typing, and the proof of equivalence between general and tight typing still works. However, we will show that updating invertible typing poses a challenge, as it cannot be easily extended with the additional rules.

Updating Tight and Invertible Typing

In this section, we revisit relevant aspects of the original soundness proof.

The main utility of invertible typing was providing a simple path of derivation of a variable's type, starting from the type given to it by the typing context, and

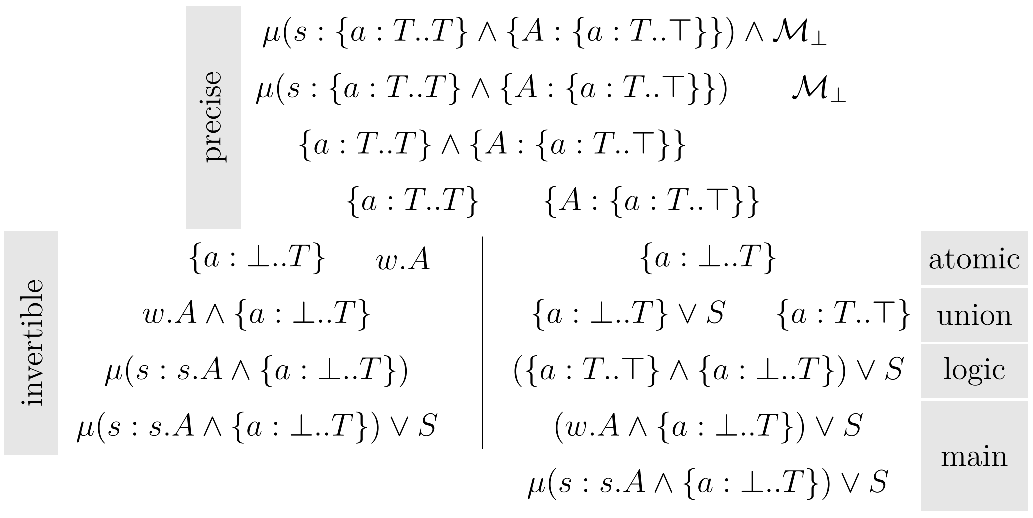
precise
$\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\}) \wedge \mathcal{M}_\perp$
$\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\})$
$\mathcal{M}_\perp$
$\{a : T..T\} \wedge \{A : \{a : T..\top\}\}$
$\{a : T..T\}$
$\{A : \{a : T..\top\}\}$
invertible
$\{a : \perp..T\}$
$w.A$
$w.A \wedge \{a : \perp..T\}$
$\mu(s : s.A \wedge \{a : \perp..T\})$
$\mu(s : s.A \wedge \{a : \perp..T\}) \vee S$
$\{a : \perp..T\}$
atomic
$\{a : \perp..T\} \vee S$
$\{a : T..\top\}$
union
$(\{a : T..\top\} \wedge \{a : \perp..T\}) \vee S$
logic
$(w.A \wedge \{a : \perp..T\}) \vee S$
$\mu(s : s.A \wedge \{a : \perp..T\}) \vee S$
main

Figure 4.2: Example derivation of a type by invertible typing (left) and layered typing (right). The variable w has the type $\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\}) \wedge \mathcal{M}_\perp$ in the typing context and S is an arbitrary type.

ending with a type that was used to access a member at some particular point in the program. This direct path then allowed induction-based proofs of properties of the typing relation.

This task would be especially hindered if the typing rules allowed cycles in the derivation, which would allow the derivation to go through unnecessarily complicated types. For example, with general typing, it is possible to derive $\Gamma \vdash x : T$ from $\Gamma \vdash x : T \wedge T$ and vice versa. Therefore, a derivation of type T can start with $\Gamma \vdash x : T$, go through arbitrarily complicated types such as $(T \wedge T) \wedge T$, and come back to T. This inhibits arguments by induction on the derivation of a general typing.

Invertible typing in DOT prevented this by ensuring that the derivation closely follows the syntactic make-up of the target type.

The original roDOT adopted the invertible typing from DOT [95], where it has two layers, which we present using an example derivation of a type for a variable w in Figure 4.2.

The first layer, **precise typing**, only derives types by deconstructing the type of the variable given by the typing context. For each reference w, its type in the typing context is an intersection of a mutability declaration with a recursive type containing an intersection of declarations. Precise typing allows opening this recursive type and extracting the declarations from the intersection, but does not support subtyping. The top of Figure 4.2 shows individual steps of this process.

The second layer, **invertible typing**, combines both variable typing and subtyping into a single layer. In DOT and the original roDOT, it has fewer rules than general typing and subtyping, because it only has rules that construct the target type syntactically "bottom-up", such as closing recursive types (akin to VT-RecI), or deriving intersection and union types. Thus the derivations of invertible typing are unambiguously guided by the syntax of the target type. The left side of Figure 4.2 shows individual steps of this process in building up the type $\mu(s : s.A \wedge \{a : \perp..T\}) \vee S$ for w.

Because we changed the definition of general typing as described in Section 4.3.2, it is necessary to update invertible and tight typing as well. Updating tight typing is straightforward – the same changes are applied as to general typing.

To adapt invertible typing to the addition of the new rule ST-NM, it is not enough to just extend invertible typing with a rule that derives $\Gamma;\rho \vdash_{\#\#} x : \mathsf{N} \vee \mathcal{M}$, because ST-NM (Figure 4.1) allows deriving many more types, if used together with other subtyping rules.

The distributivity rule ST-Dist (Figure 3.4) actually allows deriving arbitrarily large types from $T_{\mathsf{N}} \vee T_{\mathsf{M}}$, where the T_{N} and T_{M} parts can consist of intersections and unions of various types, containing N and $\mathcal{M}$ somewhere.

Therefore, the invertible typing has to be changed to derive those types, in order to make it equivalent with general and tight typing.

Inversion of typing with union types As per Figure 3.14, invertible typing was equivalent to tight typing. That required invertible typing to be closed under tight subtyping (Lemma 24).

Lemma 24 (In original roDOT, invertible typing is closed under subtyping)**.** *If* $\Gamma;\rho \vdash_{\#\#} x : T_1$*, and* $\Gamma;\rho \vdash_{\#} T_1 <: T_2$*, where* $\Gamma \sim \rho$*, then* $\Gamma;\rho \vdash_{\#\#} x : T_2$*.*

The proof of Lemma 2 in DOT and the original roDOT, is by induction on $\Gamma;\rho \vdash_{\#} T_1 <: T_2$, and relies on "invertibility" of invertible typing: for a given T_1, there is, depending on the top-level type constructor in T_1, only one or a few rules of invertible typing that can possibly derive this type, and the premises of these rules are invertible typing judgments, types simpler than T_1. Inversion can be used to obtain derivations with simpler types, which are used to continue the induction.

In Lemma 24, the case of T_1 being a union type relies on case analysis of deriving union types, which corresponds to inverting the ST-Or rule (Lemma 25).

Lemma 25 (In original roDOT, typing with union types can be inverted)**.**

If $\Gamma;\rho \vdash_{\#\#} x : T_3 \vee T_4$*,*	If a reference has an union type in the original roDOT
then either $\Gamma;\rho \vdash_{\#\#} x : T_3$ *or* $\Gamma;\rho \vdash_{\#\#} x : T_4$*.*	then it has one of the types forming the union.

This lemma is however, not compatible with ST-NM. The addition of ST-NM, together with the existing rules ST-Or and ST-Dist (Figure 3.4), breaks in a significant way that cannot be repaired by simply handling additional cases to the proof.

The rule ST-NM adds new ways of deriving union types such as $\mathsf{N} \vee \mathcal{M}_{\perp}$, and the distributivity rule ST-Dist actually allows deriving arbitrarily large types of the form $T_{\mathsf{N}} \vee T_{\mathsf{M}}$, where the two parts can consist of arbitrary intersections and unions of various types that contain N and $\mathcal{M}_{\perp}$ somewhere within them.

For example, with the variables from Example 9, we have $\Gamma;\rho \vdash w_2 : \{a : T_{\mathrm{a}}\} \vee \mathcal{M}_{\perp}$. Such a type cannot be derived in a syntactically bottom-up manner that invertible typing is based on, because neither part of the union is a type of w_2 ($\Gamma;\rho \not\vdash w_2 : \{a : T_{\mathrm{a}}\}$ and $\Gamma;\rho \not\vdash w_2 : \mathcal{M}_{\perp}$).

Lemma 25 cannot be easily fixed by adding additional cases to account for the new possibilities of deriving union types, because of the ST-Dist rule (shown in Figure 3.4), where the union on the left side appears under an intersection.

As shown in Section 2.4.4, the proof of Lemma 2 relies on the ability to do case analysis of deriving union types (Lemma 25). After the addition of ST-NM, Lemma 25 is not true anymore.

The type $\mathsf{N} \vee \mathcal{M}_\perp$ is top-like, but the addition of the ST-NM rule also allows deriving some non-top-like types, which may be specific to the variable being typed. For example, using ST-NM, w_2 also has the type $\{a : T_\mathrm{a}\} \vee (\{b : T_\mathrm{b}\} \wedge \mathcal{M}_\perp)$.

As an example of a more complex type, consider $((\{a : T_\mathrm{a}\} \vee \{b : T_\mathrm{b}\}) \wedge \{b : T_\mathrm{b}\}) \vee (\{b : T_\mathrm{b}\} \wedge (\mathcal{M}_\perp \vee \{a : T_\mathrm{a}\}))$. With this type, it is not easy to tell the role of each part in the typing derivation, and we definitely cannot do that by inverting just one step of the derivation anymore.

Rather than trying to fix the original invertible typing by adding more rules, we defined a new alternative definition of an auxiliary typing judgment, which is also equivalent to general typing under inert context, but in its design, focusing on the types that can be derived using the new rule ST-NM.

4.3.3 Layered Typing

In *layered typing*, we avoid the need for Lemma 25 by organizing the derivation of a type not bottom-up, but by handling different type constructors which can appear in the target type in separate layers of typing judgments.

The problematic new type derivations use ST-NM and possibly ST-Dist, which work with unions and intersections. For simplicity, let's for a while ignore other aspects of typing, and only focus on unions and intersections. To isolate our problem with inverting derivations of union types, we completely separate derivation of union types from intersection types – all union type constructors are derived before any intersection types.

This approach adapts well to the addition of the ST-NM rule, because this rule is just another way to derive union types (where one of the sides contains $\mathcal{M}$ and the other N). We separate them into two layers that can derive union types.

First, the *basic layer*, derives the newly top-like types possible by the rule ST-NM. Because intersections, recursive types and selections are out of the picture at this layer, these types have a simple form of possibly nested union types, where one of the sides contains N and the other $\mathcal{M}$, where $\mathcal{M}$ is a declaration $\{\mathsf{M} : \perp..T\}$ for some bound T. We will write that as $\vdash \mathsf{N} \lhd T_\mathrm{N}$ and $\vdash \mathcal{M} \lhd T_\mathrm{M}$. Second, the *union layer* derives types possible by the rules ST-Or1 and ST-Or2, allowing nesting a known type of w in a union with any other type. This way, the layers retain the information about how a union type has been derived and those cases can be handled separately when inverting the derivation.

Intersection types can be handled in analogy to how any logical formula can be derived by starting from conjunctive normal form (CNF) and pushing conjunctions down. Any type constructed from a mixture of union and intersection types can be derived by starting from an intersection of union types and pushing the intersections down.

The *logic layer* sitting above the union layer can derive any mixture of unions and intersections using the LTL-And rule shown in Figure 4.7. It takes derivations of two types that may have some parts in common but differ in one place. The common part $C^\vee$ is a syntactical context which combines the argument into a union with other types. For example, we can write the two types $\{a_1 : T_1\} \vee \{a_2 :$

$T_2\}$ and $\{a_1 : T_1\} \vee \mathcal{M}_\perp$ as $C^\vee[\{a_2 : T_2\}]$ and $C^\vee[\mathcal{M}_\perp]$. If we view these two types as an intersection, then the rule pushes the intersection down to the place where the two types differ. In the example derivation on the right of Figure 4.2, we derived two union types on the union layer. (The type $\{a : T..\top\}$ was derived on the previous layer and S is an arbitrary type.) On the logic layer, we combined them into one type, pushing the intersection down to the left.

For example, the variable y_1 from Example 9, has the type $\{a_1 : T_1\} \vee (\{a_2 : T_2\} \wedge \mathcal{M}_\perp)$, which is derived as follows: By precise typing, y_1 has types $\{a_2 : T_2\}$ and $\mathcal{M}_\perp$. On the union layer, it has types $\{a_1 : T_1\} \vee \{a_2 : T_2\}$ and $\{a_1 : T_1\} \vee \mathcal{M}_\perp$, and on the logic layer, it has the type $\{a_1 : T_1\} \vee (\{a_2 : T_2\} \wedge \mathcal{M}_\perp)$.

So far, we ignored some features of the type system – subtyping between declaration types, and ways of deriving selection types and recursive types. The rest of type constructors are handled either below the basic layer or above the logic layer. Subtyping between declarations is handled in an *atomic layer* positioned between precise typing and the basic layer. This layer only deals with types that are single declarations.

Recursive type of the form $\mu(s : T)$ and selection types $(x.B(y))$ have a common property that in a typing derivation, they can "wrap around" or replace any part of the derived type.

If the type is being derived bottom-up like in invertible typing, recursive types are derived by a rule analogous VT-RecI, which wraps around a previously derived type and replaces T by $\mu(s : T)$. Similarly, type selections are derived by a rule analogous to using VT-Sub and ST-SelL, replacing a part T of the target type by $x_1.B(x_2)$, when T is the bound of B in x_1.

If, however, we look at the final type of a whole derivation, the recursive type might appear under unions and intersections, and may contain unions and intersections within it. For example, in Figure 4.2, the left side of the union is wrapped under a recursive type in the last step.

Therefore, in layered typing, these constructors must be handled after the logic layer. We do it for both of them in a final, *main layer.*

The rules on this layer allows any part T of the type under unions and intersection, to be replaced by $\mu(s : T)$. For example, y_1 also has type $\mu(s : \{a_1 : T_1\}) \vee \{a_2 : T_2\}$ on the main layer.

4.3.4 Typing Layers

$C^\vee ::=$	**Union context**
$\mid \square$	type hole
$\mid T \vee C_1^\vee$	right union
$\mid C_1^\vee \vee T$	left union

$C^{\wedge\vee} ::=$	**Logic context**
$\mid \square$	type hole
$\mid T \vee C_1^{\wedge\vee}$	right union
$\mid C_1^{\wedge\vee} \vee T$	left union
$\mid T \wedge C_1^{\wedge\vee}$	right intersection
$\mid C_1^{\wedge\vee} \wedge T$	left intersection

Figure 4.3: Syntactic context

$$\frac{\Gamma \vdash_! x : T_1 \quad \rho \vdash T_1 \approx T_2}{\Gamma;\rho \vdash_{\mathrm{a}} x : T_2}\text{(LTA-Prec)} \qquad \frac{\Gamma;\rho \vdash_{\mathrm{a}} x : T_1}{\Gamma;\rho \vdash_{\mathrm{a}} T_1 <: T_2}\text{(LTA-Sub)} \quad \Gamma;\rho \vdash_{\mathrm{a}} x : T_2$$

Figure 4.4: Rules of atomic typing

$$\frac{\Gamma;\rho \vdash_{\mathrm{a}} x : T}{\Gamma;\rho \vdash_{\mathrm{b}} x : T}\text{(LTB-Atom)} \qquad \frac{\vdash_{\mathrm{c}} \mathsf{N} \lhd T_1 \quad \vdash_{\mathrm{c}} \mathsf{M} \lhd T_2}{\Gamma;\rho \vdash_{\mathrm{b}} x : T_1 \vee T_2}\text{(LTB-NM)} \qquad \frac{\vdash_{\mathrm{c}} \mathsf{M} \lhd T_1 \quad \vdash_{\mathrm{c}} \mathsf{N} \lhd T_2}{\Gamma;\rho \vdash_{\mathrm{b}} x : T_1 \vee T_2}\text{(LTB-MN)}$$

Figure 4.5: Rules of basic typing

$$\frac{\Gamma;\rho \vdash_{\mathrm{b}} x : T}{\Gamma;\rho \vdash_{\mathrm{c}} x : T}\text{(LTC-Basic)} \qquad \frac{}{\Gamma;\rho \vdash_{\mathrm{c}} x : \top}\text{(LTC-Top)}$$

$$\frac{\Gamma;\rho \vdash_{\mathrm{c}} x : T_1}{\Gamma;\rho \vdash_{\mathrm{c}} x : T_1 \vee T_2}\text{(LTC-Or1)} \qquad \frac{\Gamma;\rho \vdash_{\mathrm{c}} x : T_2}{\Gamma;\rho \vdash_{\mathrm{c}} x : T_1 \vee T_2}\text{(LTC-Or2)}$$

Figure 4.6: Rules of union layer typing

$$\frac{\Gamma;\rho \vdash_{\mathrm{c}} x : T \quad x \in \operatorname{dom}\Gamma}{\Gamma;\rho \vdash_{\mathrm{l}} x : T}\text{(LTL-Or)} \qquad \frac{\Gamma;\rho \vdash_{\mathrm{l}} x : C^{\vee}[T_1] \quad \Gamma;\rho \vdash_{\mathrm{l}} x : C^{\vee}[T_2]}{\Gamma;\rho \vdash_{\mathrm{l}} x : C^{\vee}[T_1 \wedge T_2]}\text{(LTL-And)}$$

Figure 4.7: Rules of logic typing

$$\frac{\Gamma;\rho \vdash_{\mathrm{l}} x : T}{\Gamma;\rho \vdash_{\mathrm{m}} x : T}\text{(LTM-Logic)} \qquad \frac{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[[x/s]T] \quad T \textbf{ indep } s \quad \Gamma;\rho \vdash_{\#} \mathsf{N} <: [x/s]T}{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[\mu(s : T)]}\text{(LTM-Rec)}$$

$$\frac{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[[v_3/r]T_1] \quad \Gamma \vdash_! v_2 : \{B(r) : T_1..T_2\}}{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[v_2.B(v_3)]}\text{(LTM-Sel)} \qquad \frac{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[\mathsf{N}]}{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[\mu(s : T)]}\text{(LTM-N)}$$

Figure 4.8: Rules of main typing

The layers of invertible and precise typing are summarized in Table 4.1, along with an example of a type that can be derived on each layer, and a selection of typing rules that are used to derive those types in general typing, and therefore show the connection between layered and general typing. Full definitions are shown in Figures 4.4–4.8.

The layers of the new invertible typing (above precise typing) are the following:

- **Atomic layer** ($\Gamma;\rho \vdash_{\mathrm{a}} x : T$, Figure 4.4). Typing on this layer only gives variables single declaration types – the declarations derived by precise typing, and their supertypes (equivalence and subtyping rules between declarations are handled here).
- **Basic layer** ($\Gamma;\rho \vdash_{\mathrm{b}} x : T$, Figure 4.5). This layer gives all variables the top-like union types of the form $T_{\mathsf{N}} \vee T_{\mathsf{M}}$ and $T_{\mathsf{M}} \vee T_{\mathsf{N}}$, where $\vdash \mathsf{N} \lhd T_{\mathsf{N}}$ and $\vdash \mathsf{M} \lhd T_{\mathsf{M}}$. For example, the top-like types $\mathsf{N} \vee \{\mathsf{M}(r_0) : T_1..T_2\}$ and $\{\mathsf{M}(r_0) : T_1..T_2\} \vee \mathsf{N}$.
- **Union layer** ($\Gamma;\rho \vdash_{\mathrm{u}} x : T$, Figure 4.6). This layer handles union types and $\top$. The union types are formed putting a known type of the variable into a union with another, completely arbitrary type.
- **Logic layer** ($\Gamma;\rho \vdash_{\mathrm{l}} x : T$, Figure 4.7). This layer handles intersection types and distributivity. Intersection types are handled by treating the unions derived by the union layer as CNF, and pushing intersections down. The (LTL-And) rule takes derivations of two types that may have some parts in common but differ in one place. For example, we can write the two types $\{a_1 : T_1\} \vee \{a_2 : T_2\}$ and $\{a_1 : T_1\} \vee \mathcal{M}_{\perp}$ as $C^{\vee}[\{a_2 : T_2\}]$ and $C^{\vee}[\mathcal{M}_{\perp}]$, where $C^{\vee}$ is the common part – a syntactical context which combines the argument into a union with other types. The rule takes two such types, preserves their common part, and combines the differing parts using an intersection type – pushing the intersection down from the top to its target place.
- **Main layer** ($\Gamma;\rho \vdash_{\mathrm{m}} x : T$, Figure 4.8). Derives types containing the recursive type constructors and type selections, by closing the recursive type around a part of the type, or replacing a part of the type by a type selection. Here, the syntactic context $C^{\wedge\vee}$ can consist of a mixture of unions and intersections. The rules LTM-Sel and LTM-Rec have additional conditions, which correspond to the conditions in the corresponding rules in tight typing and subtyping.

  The LTM-N rule handles the fact that $\Gamma;\rho \vdash \mathsf{N} <: \mu(s : T)$.

Additional definitions

The layered typing uses additional typing judgments:

- **Atomic subtyping** (Figure 4.9) is tight subtyping restricted to simple declarations.
- The definition of N- and $\mathcal{M}$-**supertypes** (used in the basic typing layer) is also layered. The basic layer (Figure 4.10) recognizes declarations and recursive types, and the union layer (Figure 4.11) adds top and union types.

Layer	Example type	Relevant rules
Precise typing	$\{a : T..T\}$, $\{A : T..T\}$, $\{m(S,T) : U\}$, $\{\mathsf{M} : \bot..\bot\}$	ST-And1, ST-And2, VT-RecE
Atomic layer	$\{a : T..U\}$, $\{A : T..U\}$, $\{m(S,T) : U\}$	ST-Met, ST-Fld ST-Typ, ST-Eq
Basic layer	$\mathsf{N} \vee \mathcal{M}_\bot$	ST-NM
Union layer	$\top$, $T \vee U$, $\{a_1 : T_1..T_2\} \vee \mathcal{M}_\bot$	ST-Or1, ST-Or2, ST-Top
Logic layer	$T \wedge U$, $\{a_1 : T_1..T_2\} \wedge \{\mathsf{M} : \bot..T\}$	ST-Dist, ST-And, VT-AndI
Main layer	$\mu(s : T)$, $x.A$, $\mu(s : \{a_1 : T_1..T_2\} \wedge s.A(s))$	VT-RecI, ST-SelL, ST-N-Rec

Table 4.1: Layers of new invertible typing

$$\frac{\begin{array}{c}\Gamma;\rho \vdash_{\#} T_3 <: T_1 \\ \Gamma;\rho \vdash_{\#} T_2 <: T_4\end{array}}{\Gamma;\rho \vdash_{\mathrm{a}} \{B(r) : T_1..T_2\} <: \{B(r) : T_3..T_4\}}\text{(LSA-Typ)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash_{\#} T_3 <: T_1 \\ \Gamma, z : T_3;\rho \vdash T_6 <: T_5 \\ \Gamma, z : T_3, r : T_6;\rho \vdash T_2 <: T_4\end{array}}{\Gamma;\rho \vdash_{\mathrm{a}} \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}}\text{(LSA-Met)}$$

$$\frac{\begin{array}{c}\Gamma;\rho \vdash_{\#} T_3 <: T_1 \\ \Gamma;\rho \vdash_{\#} T_2 <: T_4\end{array}}{\Gamma;\rho \vdash_{\mathrm{a}} \{a : T_1..T_2\} <: \{a : T_3..T_4\}}\text{(LSA-Fld)}$$

Figure 4.9: Rules of atomic subtyping

$$\frac{}{\vdash_{\mathrm{b}} \mathsf{M} \lhd \{\mathsf{M}(r_0) : T_1..T_2\}}\text{(LNB-M)}$$

$$\frac{}{\vdash_{\mathrm{b}} \mathsf{N} \lhd \mathsf{N}}\text{(LNB-N)} \qquad \frac{}{\vdash_{\mathrm{b}} \mathsf{N} \lhd \{m(z : T_1, r : T_5) : T_2\}}\text{(LNB-Met)}$$

$$\frac{}{\vdash_{\mathrm{b}} \mathsf{N} \lhd \{a : T_1..T_2\}}\text{(LNB-Fld)} \qquad \frac{}{\vdash_{\mathrm{b}} \mathsf{N} \lhd \{A(r) : T_1..T_2\}}\text{(LNB-Typ)}$$

Figure 4.10: N and $\mathcal{M}$ supertypes

4.3.5 Properties of Layered Typing

For this new layered invertible typing, we proved the properties laid out in Section 2.4.4.

- If a location has a declaration type by layered typing, then it also has a declaration type type by precise typing, with the same or tighter bounds. This property has three variants, for type declarations (Lemma 26), field

$$\frac{\vdash_b \mathsf{N} \lhd T}{\vdash_c \mathsf{N} \lhd T}\text{(LNC-Basic)} \qquad \frac{}{\vdash_c \mathsf{N} \lhd \top}\text{(LNC-Top)}$$

$$\frac{\vdash_c \mathsf{N} \lhd T_1}{\vdash_c \mathsf{N} \lhd T_1 \vee T_2}\text{(LNC-Or1)} \qquad \frac{\vdash_c \mathsf{N} \lhd T_2}{\vdash_c \mathsf{N} \lhd T_1 \vee T_2}\text{(LNC-Or2)}$$

$$\frac{\vdash_b \mathsf{M} \lhd T}{\vdash_c \mathsf{M} \lhd T}\text{(LMC-Basic)} \qquad \frac{}{\vdash_c \mathsf{M} \lhd \top}\text{(LMC-Top)}$$

$$\frac{\vdash_c \mathsf{M} \lhd T_1}{\vdash_c \mathsf{M} \lhd T_1 \vee T_2}\text{(LMC-Or1)} \qquad \frac{\vdash_c \mathsf{M} \lhd T_2}{\vdash_c \mathcal{M} \lhd T_1 \vee T_2}\text{(LMC-Or2)}$$

Figure 4.11: N and $\mathcal{M}$ supertypes union layer

(Lemma 27) and method declarations (Lemma 28).

- Layered typing is equivalent to general typing. As in the original proof, we use tight typing as a step between general and layered typing, then Lemma 29 and Lemma 35.
- We also use layered typing to prove Lemma 22 – if a location has some read-only type in layered typing, then all references to that location have that type too.

With these properties, the safety proof from roDOT, with invertible typing replaced by the new layered typing definition, works as a safety proof of the updated calculus.

Lemma 26 (Type declaration inversion).

If $\Gamma;\rho \vdash_m x : \{B(r) : T_1..T_2\}$,	If a variable has a type member type at a main layer,
then there exist T_3, T_4 *such that* $\Gamma;\rho \vdash_! x : \{B(r) : T_3..T_4\}$,	then it has the same type member by precise typing
and $\Gamma;\rho \vdash_\# T_1 <: T_3$ *and* $\Gamma;\rho \vdash_\# T_4 <: T_2$.	with possibly tighter bounds.

Proof. Declaration types are only affected by the atomic layer, so we can simply invert all rules on the layers above, and the rules on the atomic layer give us the desired result. □

Lemma 27 (Field declaration inversion). *If* $\Gamma;\rho \vdash_m x : \{a : T_1..T_2\}$, *then there exist* T_3, T_4 *such that* $\Gamma;\rho \vdash_! x : \{a : T_3..T_4\}$, $\Gamma;\rho \vdash_\# T_1 <: T_3$ *and* $\Gamma;\rho \vdash_\# T_4 <: T_2$.

Lemma 28 (Method declaration inversion). *If* $\Gamma;\rho \vdash_m x : \{m(z : T_1, r : T_5) : T_2\}$, *then there exist* T_3, T_4, T_6, *such that* $\Gamma;\rho \vdash_! x : \{m(z : T_3, r : T_6) : T_3\}T_4$, $\Gamma;\rho \vdash_\# T_3 <: T_1$, $\Gamma, z : T_3;\rho \vdash_\# T_6 <: T_5$ *and* $\Gamma, z : T_3, r : T_6;\rho \vdash_\# T_2 <: T_4$.

Lemma 29 (Tight to layered typing).

If $\Gamma;\rho \vdash_\# x : T$,	Tight typing
then $\Gamma;\rho \vdash_m x : T$.	implies main layer typing.

The implication from tight typing to invertible typing requires us to show that invertible typing is closed under the rules of tight typing and subtyping. Because the main layer deals with parts of a type under some syntactic context $C^{\wedge\vee}$, it is useful to formulate the property in this way:

Lemma 30 (Layered typing closed under subtyping)**.**
If $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_1]$*,*
and $\Gamma;\rho \vdash_{\#} T_1 <: T_2$*,*
then $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_2]$*.*

A part of a type given by layered typing under $\wedge$ and $\vee$, can be replaced by any supertype.

Similarly to how it was done for invertible typing in the original roDOT, the proof involves inverting the rules of layered typing. Here though, the inversion has to to go through multiple layers of typing, until it reaches the layer where the type system feature involved in the subtyping rule is handled.

Below, we show examples of properties of typing on the main layer that are analogous to subtyping rules.

The subtyping rules for intersection types (ST-And1, ST-And2 and ST-And) must be handled at the logic layer by the rule LTL-And. On the main layer, we show that the features of the main layer do not interfere with intersection types, so we can derive and invert them in the same manner:

Lemma 31 (Intersection at the main layer)**.**
If $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_1]$*,*
and $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_2]$*,*
then $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_1 \wedge T_2]$*.*

If x has two types by main layer typing

then it also has the type made by intersection of the two types.

Lemma 32 (Inversion of intersection at the main layer)**.**
If $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_1 \wedge T_2]$*,*
then $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_1]$
and $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T_2]$*.*

If x is given a type containing an intersection by main-layer typing,

then it also has type with only one of the branches of the intersection.

The rule ST-Or can be handled, thanks to induction on the tight subtyping and to the syntactic context $C^{\wedge\vee}$, by a simpler property:

Lemma 33 (Inversion of union types at the main layer)**.**
If $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T \vee T]$*,*
then $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T]$*.*

If x has a type containing a union of equal types,

then the union can be replaced by a single part.

On the main layer, the N type can be replaced by any read-only type:

Lemma 34 (Replacement of N at the main layer)**.**
If $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[\mathsf{N}]$*,*
and $\Gamma \vdash_{\mathrm{m}} \mathsf{N} \lhd T$*,*
then $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T]$*.*

If x has a type containing the read-only bottom type N,

then that part can be replaced by any other read-only type.

Finally, the other implication (invertible typing implies tight typing) is expressed by Lemma 35.

Lemma 35 (Layered to tight typing)**.**
2.5in If $\Gamma;\rho \vdash_{\mathrm{m}} x : T$*,*
then $\Gamma;\rho \vdash_{\#} x : T$*.*

Layered typing

implies tight typing.

This was very easy in the original DOT, because there, each rule of invertible typing can be mirrored by application of just a few rules of tight typing and subtyping.

In layered typing though, the same approach only works up to the logic layer. On the main layer, there is a problem that the rule LTM-Rec allows closing a recursive type anywhere under a syntactic context $C^{\wedge\vee}$.

However, general and tight typing do not have subtyping rules for recursive types. The rule that closes recursive types in tight typing, which has the same form as VT-RecI, can only close the whole type, and not just a part of it.

That is because this rule is part of a typing judgment, not subtyping. Therefore, it cannot be combined with other subtyping rules (subtyping of unions, intersections, transitivity).

To solve this problem, we can make use of the requirement that the inside of a recursive type must be read-only. Then, we can look at what importance this part of the type has in the typing derivation. We can actually show that any read-only part of a type is either derived from the type of the variable, or it can be replaced by N:

Lemma 36 (Replacement of N supertypes at the main layer)**.**

If $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[T]$,	If x has a type containing
and $\Gamma \vdash_{\mathrm{m}} \mathsf{N} \triangleleft T$,	a read-only type T,
then either $\Gamma;\rho \vdash_{\mathrm{m}} x : T$	then either x has type T
or $\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[\mathsf{N}]$.	or T can be replaced by N in the type of x.

This lemma allows us, for a type $C^{\wedge\vee}[T]$, to either bring the type T to the top level and use VT-RecI to derive the recursive type, or to replace T by N and use subtyping with $\Gamma;\rho \vdash_{\#} \mathsf{N} <: \mu(s : T)$.

This approach has one issue, however: we cannot prove Lemma 35 by induction on the main-layer typing derivation, because when we use Lemma 36, it gives a derivation of just T, which is not a sub-derivation of the derivation of $C^{\wedge\vee}[T]$, so we cannot use the induction hypothesis on it.

Instead of doing the proof directly by induction on the derivation, we do the proof by first using induction on the number of main-layer rules applied in the derivation, and second induction on the number of union and intersection constructors within the type.

We handle the main-layer rule applications one by one, so this metric is decreasing, except the case when Lemma 36 needs to be used. It is only needed if $C^{\wedge\vee}$ has at leas one union/intersection constructor. In that case, we can find a derivation of T that has the same number of main-layer rule application as $C^{\wedge\vee}[T]$ and the number of union/intersection constructors is lowered.

4.3.6 Variants of Subtyping Rules

In addition to fixing the soundness proof, the layered-typing infrastructure also allows further extensions of typing with respect to mutability tags and N. We implemented two independent optional extensions.

- Making $\mathcal{M}_\bot$ and N complementary, so that $\vdash \mathsf{N} \wedge \mathcal{M}_\bot <: \bot$ and $\vdash \top <: \mathsf{N} \vee \mathcal{M}_\bot$.

- Generalizing the variable typing rule VT-MutTop (shown in Figure 3.5) into a subtyping rule, so that $\vdash \top <: \mathcal{M}_\top$.

$$\frac{}{\Gamma;\rho \vdash \top <: \mathsf{N} \vee \{\mathsf{M}(r) : \bot..\bot\}}\text{(ST-NMC-Top)}$$

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

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

Figure 4.12: Variant of subtyping rules for roDOT

The first extension formalizes the idea that read-only and mutable types are complementary. The top-side subtyping $\vdash \top <: \mathsf{N} \vee \mathcal{M}_\bot$ is already ensured by ST-NM added in Section 4.3.2. We can add the bottom-side subtyping as a new rule, ST-NMC-Bot shown in Figure 4.12. This will make $\mathsf{N} \wedge tMbot$ equivalent by subtyping to $\bot$.

However, adding ST-NMC-Bot would make the type system unsound, because the ST-NM rule is formulated without any restrictions on the bounds of the mutability member. By using ST-NM with bad bounds, the whole type hierarchy will collapse as explained in Example 10.

Example 10 (Counter-example for soundness with both ST-NM and ST-NMC-Bot). First, we will show a derivation of $\vdash \{\mathsf{M} : \bot..\bot\} <: \{\mathsf{M} : \top..\bot\}$. By the properties of lattice types, we have $\vdash \{\mathsf{M} : \bot..\bot\} <: \top \wedge \{\mathsf{M} : \bot..\bot\}$ and $\{\mathsf{M} : \top..\bot\} \vee \bot <: \{\mathsf{M} : \top..\bot\}$. By the rules ST-NM and ST-NMC-Bot, we have $\top \wedge \{\mathsf{M} : \bot..\bot\} <: (\{\mathsf{M} : \top..\top\} \vee \mathsf{N}) \wedge \{\mathsf{M} : \bot..\bot\}$ and $(\{\mathsf{M} : \top..\top\} \wedge \{\mathsf{M} : \bot..\bot\}) \vee (\mathsf{N} \wedge \{\mathsf{M} : \bot..\bot\}) <: \{\mathsf{M} : \top..\bot\} \vee \bot$. Finally by distributivity of $\wedge$ and $\vee$, we have $(\{\mathsf{M} : \top..\top\} \vee \mathsf{N}) \wedge \{\mathsf{M} : \bot..\bot\} <: (\{\mathsf{M} : \top..\top\} \wedge \{\mathsf{M} : \bot..\bot\}) \vee (\mathsf{N} \wedge \{\mathsf{M} : \bot..\bot\})$ and the desired subtyping follows by transitivity.

Because of subsumption, with this we can derive $\Gamma \vdash x : \{\mathsf{M} : \top..\bot\}$ for any x that is mutable in Γ, thus getting a value of a type with bad bounds.

As soon as that happens, the type hierarchy collapses, because $\Gamma \vdash \top <: x.\mathsf{M} <: \bot$.

To fix this problem, the bounds in the ST-NM rule must be restricted – this is shown as rule ST-NMC-Top in Figure 4.12.

As the second extension, we can revisit rule VT-MutTop from in Figure 3.5, which expresses the idea that a mutability declaration with the upper bound being $\top$ does not represent any useful information. This can be generalized into a subtyping rule ST-M-Top shown in Figure 4.12, which can be used in more contexts than directly for typing a variable. This allows recognizing more methods as SEF, because with these rules, $\mathsf{N} <: \{\mathsf{M}(r) : \bot..\top\}$.

For example, in the method type $\{m(r : \top, z : \{\mathsf{M}(r) : \bot..x.A(y)\}) : \top\}$, the type of the parameter z, contains a declaration of the mutability member M. In a context where $x.A(y) <: \bot$, this parameter is a mutable reference, but another context where $\top <: x.A(y)$, it is a read-only type. With the original definition, this type is not recognized as read-only even in such a context, because the rule does not consider the bounds in the type declaration. With the variant definition, it is recognized as read-only, and therefore the method has a SEF type.

4.4 The SEF Guarantee

In Section 4.2.5, we informally stated the SEF guarantee, which provides the connection between a static typing condition (Definition 2) and run-time behavior of the method.

In this section, we present the formal definitions of the run-time SEF condition (Definition 7 in Section 4.4.1) and the SEF guarantee (Section 4.4.2). We then outline the proof of the guarantee in Section 4.4.3 and discuss the details of the proof in Section 4.4.5. In Section 4.4.4, necessary lemmas about similarity are provided, and the proof is finished in Section 4.4.5.

4.4.1 The Runtime SEF Condition

We informally stated the run-time SEF condition in Section 4.2.3, where we mentioned that several possible versions of such a condition could be defined. In our approach, we allow a pure method to create new objects and to modify just these new objects, which are under full control of the method.

The main SEF condition is that the method must not modify any **existing objects** that are already on the heap when the method starts executing. We can state such a condition in three variants, depending on the way that it is checked that an object was not modified. Here we will use the variant that guarantees that existing objects on the heap do not change. In such a case, we say that a given execution of a method, starting from a method call start and reaching method call end in k steps, has the Sef-I property (Definition 7).

To arrive at the formal definition, we can start with a very strict definition which prevents any side effect by forbidding any creation or modifications of any objects.

Definition 3 (Sef-N0). *A method execution $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^k \langle \mathsf{v} w_3; \sigma; \rho_2; \Sigma_2\rangle$ is Sef-N0 when for every $j \leq k$ and $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^j \langle t_3; \sigma_3; \rho_3; \Sigma_3\rangle$ the term t_3 is not an object creation or write term.*

Instead of looking at the steps being executed, we can define a similar condition by observing the state of the heap at every moment during the execution of the method. This weaker condition, Sef-I0 defined in Definition 4, allows writes to the heap, as long as the value being written is the same as the value that already exists at the affected heap location.

Definition 4 (Sef-I0). *A method execution $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^k \langle \mathsf{v} w_3; \sigma; \rho_2; \Sigma_2\rangle$ is Sef-I0 when for every $j \leq k$ and $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^j \langle t_3; \sigma_3; \rho_3; \Sigma_3\rangle$ we have $\Sigma_3 = \Sigma_1$.*

As an even slightly weaker condition, we can only check the state of the heap at the end of the method call. This condition, Sef-O0 defined in Definition 5, allows the method to modify the heap as long as it restores the heap to its original state before it returns.

Definition 5 (Sef-O0). *A method execution $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^k \langle \mathsf{v} w_3; \sigma; \rho_2; \Sigma_2\rangle$ is Sef-O0 when $\Sigma_2 = \Sigma_1$.*

In a system executing in a single thread and where we are only interested in the result of execution, the differences between the three are not observable.

In a language where creating objects is a core feature, forbidding creation of a new object is very restrictive. Also, although creating a new object affects the heap, it does not affect any other existing objects, so execution of code that is not using the newly created object is also unaffected. Therefore, it makes sense to allow pure methods to create new objects. Because the method is in full control of such objects, it also makes sense to allow it to modify these new objects.

With this approach, the main SEF condition is that the method must not modify any **existing objects** that are already on the heap when the execution of the method starts. We can state such a condition in all the three variants:

Definition 6 (Sef-N)**.** *A method execution* $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^k \langle \mathsf{v}w_3; \sigma; \rho_2; \Sigma_2\rangle$ *is Sef-N when for every* $j \leq k$ *and* $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^j \langle t_3; \sigma_3; \rho_3; \Sigma_3\rangle$ *the term* t_3 *is not a write term.*

Definition 7 (Sef-I)**.** *A method execution* $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^k \langle \mathsf{v}w_3; \sigma; \rho_2; \Sigma_2\rangle$ *is Sef-I when for every* $j \leq k$ *and* $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^j \langle t_3; \sigma_3; \rho_3; \Sigma_3\rangle$, Σ_1 *is a prefix of* Σ_3*.*

Definition 8 (Sef-O)**.** *A method execution* $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle \longmapsto^k \langle \mathsf{v}w_3; \sigma; \rho_2; \Sigma_2\rangle$ *is Sef-O when* $\Sigma_1 \subseteq \Sigma_2$*.*

Definition 7 uses *heap prefix* to express that a new items may be added to the heap. It is defined in Definition 9.

Definition 9 (Heap prefix)**.** *A heap* Σ_1 *is a prefix of* Σ_2, $\Sigma_1 \subseteq \Sigma_2$, *when* $\forall y \in dom\Sigma, \Sigma_1[y] = \Sigma_2[y]$

From these three definitions, Definition 6 is unnecessarily restrictive, because it forbids modifications of existing objects. The difference between Definition 7 and Definition 8 is that Definition 8 allows modifying an object, as long as it is changed back before returning from the method. Making use of this possibility would require tracking the values assigned to the fields, which is out of scope of roDOT's type system. Also, although we only consider single-threaded execution in this thesis, Definition 8 would not adapt well to possible extensions to multi-threaded environments, because any changes to the object within the method would be observable from other threads. For these reasons, in this thesis, we use the Definition 7 property as the definition of a SEF method and the basis for our SEF guarantee.

Method Call Limits

Because we are going to define conditions about what can be happening while a method is executing, we need to first define what it means in roDOT that a method starts and ends its execution. The semantics of DOT is defined in such a way that it is easy to find when the method starts, but it is not immediately obvious when it ends.

In roDOT, a method is called by a term $w_1.m\, w_2$. Execution of a method starts with the step R-Call. A *method call start* is a configuration of the form $\langle w_1.m\, w_2; \sigma; \rho_1; \Sigma_1\rangle$, where w_1 is the receiver, m is the called method, w_2 is the argument,

σ is the *continuation stack*, and Σ_1 is the *existing heap* (the environment ρ_1 does not have a special significance here).

The execution proceeds by replacing the call term $w_1.m\,w_2$ with the body of the method. Then, the body is executed, which can possibly involve adding objects to the heap, and pushing and popping frames from the stack. However, frames are only popped from the stack by R-LetLoc, when the focus of execution has been reduced to a single variable w. Also, the frames on the stack do not affect the execution in any way until they are popped from the stack.

It all means that unless there is an infinite loop, the body of the method will eventually evaluate to a single value. The machine will reach a configuration $\langle \mathsf{v}w_3; \sigma; \rho_2; \Sigma_2\rangle$, where w_3 is the result of the call and σ is the same stack as at the method call start.

The **first** such configuration after a method call start is the corresponding *method call end*. Another such configuration could possibly occur later in a completely unrelated way (i.e., the following execution may restore the stack to the same state), but only the first such configuration is the method call end.

When a method call end is reached, the execution will either terminate, or proceed by popping a frame from the stack.

4.4.2 The SEF Guarantee

The SEF guarantee, which we informally stated in Section 4.2.5, says that calling a SEF method (i.e. a method where both the receiver and parameter have read-only types) does not modify existing objects in the heap during its execution. Theorem 37 is based on the Sef-I property (Definition 7), and speaks about the state of the heap at every point during the call. It is not the strongest possible purity guarantee, because this allows writing the value that already is in the field. On the other hand, it does not allow the value of fields to be changed and then changed back.

Theorem 37 (Sef-I Guarantee)**.**

If $\Gamma \vdash c_1 := \langle w_1.m\ w_2; \sigma_1; \rho_1; \Sigma_1\rangle : T$	c_1 is a well-typed configuration with a method call in focus
and $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$,	which is a read-only methods,
then for any k:	then for any number of steps of execution
Either $\exists j < k, c_1 \mapsto^j \langle \mathsf{v}w_3; \sigma_1; -; -\rangle$,	either the method has finished execution within k steps,
or $c_1 \mapsto -; -; -; \Sigma_2$	or after k steps, the heap Σ_2
and $\Sigma_1 \subseteq \Sigma_2$.	contains the all the items from Σ_1 unchanged.

4.4.3 Overview of the Proof

The SEF guarantee talks about objects not being modified during the execution of methods, based on the mutability of method parameters. We base our proof of the SEF guarantee on the immutability guarantee (IG, Theorem 11), which guarantees that individual objects can only be modified through mutable references. More precisely, it says that an object can only be modified if there is a path of writeable references leading from the executed code to the modified object. This guarantee was proven for roDOT [43] and is included of our mechanization in Coq.

However, the immutability guarantee cannot be applied at the start of the method, because there may be many mutable references to objects on the heap. Also, IG guarantees immutability until the end of execution of the whole program, but the SEF guarantee only until the end of the method.

These differences can be bridged by taking the machine configuration at the method start, and constructing a different configuration that will execute the same way until the end of the method, but removing the parts that prevent the IG from applying.

The first thing to note is that at the method start, the stack is not relevant to how the method executes and stays the same until the end of the method. We therefore remove this stack entirely, and get an execution of the method isolated from the rest of the program. This execution proceeds through the same steps as the method execution in the original state, but stops at the method end. By removing the stack, we rid the configuration of any references to objects that might be used after the method call returns. But if we apply the IG to this configuration, it will guarantee that objects are not modified until the end of the method, which is exactly what is needed for the SEF guarantee.

Removing the stack is not enough for the IG to apply though, because a SEF method can be called with arguments that are mutable references. We do not want to prevent that from happening during normal execution, because even when a method is SEF, it can be useful to pass mutable references to the method and have the method return one of these references with its mutability intact.

What is special about a SEF method is that (because of the types of its declared parameters) it cannot use the mutability during its execution. Therefore, even if it is called with mutable arguments, it should execute in exactly the same way as if it were called with read-only arguments.

So the second modification to the configuration after removing the stack is to change the mutability of the arguments to read-only. That way, the alternative configuration contains no writeable references, and therefore IG guarantees that no objects that were on the heap at the start will be modified. Still, this alternative configuration executes the same steps as the original execution from the method call start up to the method call end, therefore the original method execution also does not modify any existing object on the heap until the method call end.

The rest of this section contains text written in collaboration with Yufeng Li, from [44].

4.4.4 Similarity

Because during execution of a roDOT program, fresh variables are used for new location and references, two executions of the same program may use different variable. In order to be able treat different executions of the same program as equivalent, we define the *similarity* relation.

Similarity relates structurally equivalent syntactic elements such as terms, objects or whole configurations, which can differ in their free variables. The correspondence of variable names on two sides of the equivalence is specified by a renaming.

Definition 10 (Renaming)**.** *Renaming is a pair of lists of variables* $X := x_1, ..., x_L$ *and* $Y := y_1, ..., y_L$ *which renames* x_i *on the left to* y_i *on the right for each* i *in* $1..L$.

For simple syntactic elements, similarity is defined by the ability to get both sides of the equivalence by substitution starting from a common element.

Definition 11 (Similar variables, objects, terms and stacks)**.** *Two terms* t_1 *and* t_2 *(or variables, objects and stacks) are similar by renaming* X *to* Y*, written* $t_1 \overset{X\,Y}{\approx} t_2$*, if there exists a term* t_3 *and with list of variables* $W := w_1, ..., w_L$*, such that* $\operatorname{fv} t_3 \subseteq \{w_i\}_{i=1,...,L}$ *and* $[x_i/w_i]_i t_3 = t_1$ *and* $[y_i/w_i]_i t_3 = t_2$.

The definition of similarity for configurations is complicated by variables in the domains Σ and environment ρ. Because the domains cannot contain duplicates, but we want to have duplicates in the renaming used for the term, there are two lists of variables – one used for domains of Σ and ρ and the other for their codomain, for the term and the stack.

Definition 12 (Similar configurations)**.** *Two configurations* $\langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle$ *and* $\langle t_2; \sigma_2; \rho_2; \Sigma_2\rangle$ *are* similar up to renaming V to W in the input and X to Y in the output*, written* $- \underset{U\,V}{\overset{X\,Y}{\approx}} -$*, when*

- Terms and stacks are similar by renaming X to Y: $t_1 \overset{X\,Y}{\approx} t_2$ *and* $\sigma_1 \overset{X\,Y}{\approx} \sigma_2$.
- Heaps and environments are similar with respect to renaming inputs. *For each* u_k *from* U *and* v_k *from* V*, either* $\Sigma_1[u_k] \overset{X\,Y}{\approx} \Sigma_2[v_k]$ *or* $\rho_1[u_k] \overset{X\,Y}{\approx} \rho_2[v_k]$.
- Heap correspondence in similarity. *If* u *is a heap location in* Σ_1 *and* v *a heap location in* Σ_2 *such that* $u \overset{X\,Y}{\approx} v$*, then* $u \overset{U\,V}{\approx} v$

Moreover, similarity of configurations is preserved by reduction (Lemma 38) and ensures the existence of reduction for similar configurations (Lemma 39).

Lemma 38 (Similarity is preserved by reduction)**.**

If $c_1 \underset{U\,V}{\overset{X\,Y}{\approx}} c_2$ *for* c_1,c_2 *well-typed*	Given two similar well-typed configurations,
and $c_i \mapsto^n c_i'$,	if both of them reduce in n steps,
then $\exists K := (y_n)_{n=1,...,N}$	then there exists a renaming
and $L := (y_n')_{n=1,...,N}$	
such that K *and* L *are fresh in* c_i,	of new variables
and $c_1' \underset{U,K\ V,L}{\overset{X,K\ Y,L}{\approx}} c_2'$.	by which the results are similar.

Proof. It suffices to consider the case of a single-step reduction (i.e. when $n = 1$) because one can just use subject reduction and induction for the multi-step case. In the single-step case, we solve each case of $c_i \mapsto c_i'$ (defined in Figure 3.13), taking advantage of the fact that $t_1 \overset{X\,Y}{\approx} t_2$, where t_i is the term of c_i, to ensure that the same reduction rule is used to reduce both $c_i \mapsto c_i'$.

The freshness of the variables $y_{i,k}$ follows from the fact that new in all reduction rules, the domains of Σ and ρ are either unchanged or extended with fresh variables. The cases of field read and writes are justified by the second and third conditions of Definition 12 along with the fact that configuration typing means heap correspondence. □

Lemma 39 (Similarity ensures reduction)**.**

If $c_1 \overset{X\,Y}{\underset{U\,V}{\approx}} c_2$ *for* c_1,c_2 *well-typed*	Given two similar well-typed configurations,
and $c_1 \mapsto^n c_1'$,	if one of them reduces in n steps,
then there exists c_2'	then the other configuration
for which $c_2 \mapsto^n c_2'$.	also reduces in n steps to a similar configuration.

Proof. If c_2 is an answer configuration, then by similarity c_1 must be an answer configuration too. Because there are no reduction steps defined for answer configurations, it is possible only when $n = 0$. In the inductive case, use the subject reduction property and the similarity part of Lemma 38. □

The final part of carrying over the SEF property to c_2 is to formalize the phenomenon that reduction only changes fields.

Definition 13 (Objects identical except fields)**.** *For objects* o_1 *and* o_2, *we write* $o_1 \overset{\mathsf{fld}}{\approx} o_2$ *to mean they are identical except for possibly the values of fields.*

Lemma 40 (Reduction only changes fields)**.**

If $\langle -; -; -; \Sigma\rangle \mapsto^n \langle -; -; -; \Sigma'\rangle$	If the heap Σ can evolve into Σ' by execution,
and $y \in \text{dom}\ \Sigma$,	and y is a location of an object in Σ,
then $\Sigma[y] \overset{\mathsf{fld}}{\approx} \Sigma'[y]$.	then the corresponding object in Σ' is identical except fields.

Proof. The only reduction rule that changes existing objects is the field-write reduction rule, which changes just the fields. □

4.4.5 Proof of the SEF Guarantee

The strategy of the proof of Theorem 37 is to focus on the second case of the SEF guarantee by using the immutability guarantee to show the theorem for a configuration c_2 obtained by temporarily truncating the stack of c_1 from Theorem 37.

Lemma 41 (SEF guarantee without stack)**.**

If $\Gamma \vdash c_1 := \langle w_1.m\ w_2; \sigma_1; \rho_1; \Sigma_1\rangle : T$,	For c_1 satisfying the conditions of Theorem 37,
and $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$,	
$c_1' := \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1\rangle$,	if the stack is removed
and $c_1' \mapsto^n c_2'$	then for all steps of reduction
then $\Sigma_1 \subseteq \Sigma_2$.	the heap of c_2' contains all objects of c_1' without modification.

It is easy to prove the full SEF theorem with this result for c_2.

The premise of the immutability guarantee is that c_2 is well-typed in some context Γ_2 and there are no mutably reachable objects in c_2 with respect to the typing of Γ_2. Clearly c_2 is well typed in the original context Γ. But for the part about mutably reachable objects, we cannot just take Γ as Γ_2 because for this Γ_2 must assign read-only types to w_i. Even though we have $(r : \mathsf{N})$ in the typing $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$, this does not necessarily mean $\Gamma[w_1]$ is a read-only type. For example, w_1 might be mutable in Γ but m just does not make use of mutability.

For this reason, instead of using Γ as Γ_2, we construct Γ_2 and show that c_2 is well-typed in Γ_2 like so:

1. *Reference elimination.* Remove bindings for w_i from Γ and replace all occurrences of w_i with the corresponding location y_i in both Γ and c_2.

2. *Read-only weakening.* Add back bindings for w_i, where the new type bound to w_i is the type bound to y_i except with the mutable part set to read-only.

The two steps above correspond to the following two lemmas.

Lemma 42 (Reference elimination)**.**

If $\Gamma \vdash c_1 := \langle w_1.m\ w_2; \sigma_1; \rho_1; \Sigma_1\rangle : T$	If c_1 and Γ satisfy the conditions of *Theorem* 37,
and $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$, *and* $\rho_1[w_i] = y_i$	
$\Gamma' := [w_i/y_i]\Gamma \setminus w_i$	Γ' is a context obtained from Γ by first removing bindings for w_i and then replacing w_i with y_i.
$\rho' := \rho \setminus w_i$	ρ' is an environment obtained from ρ_1 by removing bindings for w_i
Then $\Gamma';\rho' \vdash y_1.m\ y_2 : \top$	Term typing: $y_1.m\ y_2$ is well-typed under Γ' and ρ'
$\Gamma';\rho' \sim [y_i/w_i]_i\Sigma_1$.	Heap correspondence:

Proof. Because w_i is a reference to location y_i, types assigned to y_i and w_i by Γ differ only by mutability, and y_i has a mutable type. So $\Gamma[y_i]$ is a subtype of $\Gamma[w_i]$, and the result follows by substitutivity. □

Lemma 43 (Read-only weakening)**.**

If $\Gamma \vdash c_1 := \langle w_1.m\ w_2; \sigma_1; \rho_1; \Sigma_1\rangle : T$	Let c_1 be a well-typed configuration with a method call in focus
where $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$,	which satisfies the conditions of *Theorem* 37 (the method is SEF)
and $\rho_1[w_i] = y_i$. *Then there exist* T_i, *where* $\mathsf{N} <: \Gamma_2[T_i]$	Then there is a context Γ_2 binding w_i to read-only types
$c_1'' := \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i\Sigma_1\rangle \vdash \top$.	such that the configuration is well-typed in Γ_2.

Proof. Take Γ' and ρ' as from Lemma 42. Let T_i be the read-only version of $\Gamma[y_i]$ so that T_i differs from $\Gamma[w_i]$ only by mutability. The goal is to take $\Gamma_2 := \Gamma'(w_i : T_i)_i$.

The hardest part of showing $\Gamma_2 \vdash c_1' : \top$ is the part of typing $w_1.m\ w_2$. Note that the only difference between T_i and $\Gamma[w_i]$ is that T_i is by construction read-only while $\Gamma[w_i]$ is possibly mutable. So the idea is to use the same derivation of $\Gamma' \vdash y_1.m\ y_2 : \top$, in the process taking advantage of the fact that $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$ means nowhere in the derivation did one need to use the fact that $\Gamma[w_i]$ is potentially mutable.

In the derivation of $\Gamma' \vdash y_1.m\ y_2 : \top$, one sees there are sub-derivations giving read-only types T_i to y_i. Lemma 22 says that for any context Γ and environment ρ that are in correspondence, if a location y_i can be derived to have a read-only type T_i then one may give the same type T_i to any references w_i to the location y_i. The result of Lemma 43 follows. $\square$

By Lemma 43 along with the immutability guarantee, we have SEF established for $c_1'' \coloneqq \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i \Sigma_1 \rangle$. However, notice in particular that we need SEF for $c_1' \coloneqq \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1 \rangle$ in Lemma 41, where there is no substitution $[y_i/w_i]_i$ in the heap. Nevertheless, the substitution $[y_i/w_i]_i$ can be ignored in the sense that execution can only change fields of objects, but fields are always locations while w_i are references. Because c_1' and c_1'' are the same everywhere except for $[y_i/w_i]_i$ on the heap, the SEF property of c_1'' can be carried over to c_1'. The first part of carrying the SEF property over to c_1' is to relate each k-th step of execution starting from c_1'' and the k-th step of execution starting from c_1'. With the definition of similarity from Section 4.4.4, Lemma 44 formalizes the idea that c_1' is similar to c_2 up to renaming $W \coloneqq w_1, w_2$ to $Y \coloneqq y_1, y_2$:

Lemma 44 (Similarity for eliminated references)**.**

Let $c_1' \coloneqq \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1 \rangle$	Let c_1' satisfy the conditions of Lemma 41
and $c_1'' \coloneqq \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i \Sigma_1 \rangle \vdash \top$	and c_1'' the conditions of Lemma 43.
then $c_1' \overset{W\,Y}{\approx} c_1''$	These configurations are similar by renaming w_i to y_i for $i = 1, 2$.

Proof. By definitions of c_1', c_1'' and Definition 12. $\square$

Finishing the proof

We are now ready to prove Lemma 41.

Proof of Lemma 41. We use Lemmas 38, 39 and 44 to get

$$\begin{tikzcd}
c_1' \arrow[r, dash, "\overset{W\,Y}{\approx}" description] \arrow[d, mapsto, "n"] & c_1'' \arrow[d, mapsto, "n"] \\
c_2' \arrow[r, dash, "\underset{K\,L}{\overset{W,K\,Y,L}{\approx}}" description] & c_2''
\end{tikzcd}$$

where K and L are fresh in both c_1' and c_1''. Let Σ_2 be the heap of c_2' and Σ_2'' be the heap of c_2''. Let y be a heap location of Σ_1, so that Lemma 40 applied to the left edge means that $\Sigma_1[y] \overset{\mathsf{fld}}{\approx} \Sigma_2[y]$. The immutability guarantee applies to the right edge because of Lemma 43 and so we know $\Sigma_2''[y] = \Sigma_1[y]$. The freshness of K and L, and the bottom edge then implies $\Sigma_2[y] \overset{W\,Y}{\approx} \Sigma_2''[y] \overset{W\,Y}{\approx} \Sigma_1[y]$. But $\Sigma_1[y]$ can only differ from $\Sigma_2[y]$ in field values and w_i are references, so they never occur as values of fields, therefore $\Sigma_2[y] = \Sigma_1[y]$. $\square$

And with this, Theorem 37 follows straightforwardly.

Proof of Theorem 37. By classical reasoning, assume the first condition is false so that the goal is to prove the second condition. That is, assume that there is no $j < k$ such that the top-most frame of c_1 is popped after execution by j steps: $c_1 \mapsto^j$

$\langle \mathsf{v}w_3; -; -; \Sigma_2 \rangle$. Then, the sequence of reductions $c_1 \mapsto ... \mapsto c_4$ by Lemma 39 corresponds to a sequence of reductions $c_2 \mapsto ... \mapsto c_3$ because even though c_2 has no awaiting frames, there are no frame pops in this execution sequence by the current assumption. By Lemma 41, the second condition follows. □

4.5 Transformations

In Section 4.2.6, we informally stated the transformation guarantee, which connects the static SEF condition with a practical application – any two calls to SEF methods can be safely swapped (the code `x1.m1(); x2.m2()` is equivalent to `x2.m2(); x1.m1()`). In other words, the absence of side effects allows changing the order of execution.

This guarantee is a specific case of how methods known to be SEF can be used in more ways than general methods that may have side effects. For example, safe transformations such as this could be applied to at compile time as an optimization, or while writing program code in a text editor, and be guaranteed to not break the program.

Proving the transformation guarantee for roDOT shows also that the SEF guarantee (Theorem 37) is strong enough to guarantee that swapping two such method calls will not change the behavior of a roDOT program.

In order to define the call-swapping transformation and the associated guarantee in a formal way, we have to deal with several technicalities particular to the roDOT calculus. These concerns would equally apply to other safe transformations that we can imagine, such as reordering field reads or removing dead code.

In order to separate the common problems from the specific case of swapping calls, but build a general framework that can be used to define various transformations of roDOT programs and to reason about safety of those transformations. We instantiate it here only with the call swapping transformation, but it could prove the general part of a proof of safety for other transformations, such as reordering field reads or removing dead code.

In Section 4.5.1, we present the framework for defining and reasoning about safe program transformations in roDOT and similar calculi, including a general Theorem 48 about safety of transformations. In Section 4.5.2, we define the transformation that swaps two calls to methods that are statically determined to be side-effect free (Definition 21). In Section 4.5.3 we present a statement and proof of the transformation guarantee, which guarantees that the call-swapping transformation is safe – does not change the result of a program.

4.5.1 Transformation Framework

In the transformation framework, we define a general form for roDOT program transformations, the precise meaning of a *safe transformation* that "does not affect the behavior of the program".

The framework is based around the idea of type-respecting equivalence relations [91], where, generally speaking, two programs can be treated as equivalent under a given typing context, if they both have the same type under that typing context and evaluate to the same value.

There are, however, specific issues in roDOT that need to be considered:

- Programs in DOT are *terms* in λ-calculus style and A-normal form. They are not formed by sequences of statements, but rather by nesting let-in terms.

- The program can contain object literals and method definitions, so a transformation such as swapping two calls can also be located inside a body of a method.

- The transformation can be conditioned on local typing information, such as the method calls being recognized as SEF. In roDOT, this necessitates looking at the typing context Γ that is used for typing that part of the program.

- The execution of a program is defined using small-step operational semantics, starting from an initial configuration and applying reduction steps until reaching an answer.

  For discussion of what a safe transformation is, we will consider two programs, an original program and a transformed program, which only differ in ways allowed by the transformation. Such a transformation is safe if executions of those two programs reach the "same" answer. We cannot require the answers of the two programs to be identical, because of the following points.

- In the initial program, a transformation, such as swapping calls, can be located anywhere, including inside a body of a method of an object literal, such as $\mathsf{let}\, x = \nu(r : R)\{m(r, z) = t_m\}\, \mathsf{in}\, t_2$.

  During execution of an roDOT program, objects are created on the heap. Each object on the heap includes a copy of the code of its methods, which can be affected by the transformation.

  Therefore, if we look at execution of the two programs in parallel, the transformation affects not only the program being executed, but also the bodies of these methods in objects on the heap.

  In DOT, the code (terms) and values (objects) are mixed with each other. In order to allow reasoning about the transformation, we must consider that during execution, the transformation may be located at multiple places in the focus of execution, stack and heap. This also applies to the answer of the program.

  It would be too restrictive to require that a transformed program produce the exact same output value as the original program since the output value may be an object that may contain the transformed code. To facilitate this, we define each transformation using a local relation which relates two terms that differ only locally, and the framework provides lifting operators, which allow this transformation to occur anywhere in a program or in a machine configuration.

- The locations of objects on the heap are chosen non-deterministically, therefore the framework must deal with the fact that in the answer and in intermediate states of two executions, location names can different.
- Otherwise the execution is deterministic. Programs in DOT and roDOT do not read any input and do not make non-deterministic choices. This makes the situation easier, as we can assume that all possible executions of one program will reach answers that are similar to each other. Therefore, the differences in executions of the original and the transformed program are only caused by the transformation of the initial program.

The general approach in the framework is to define a transformation that applies to an initial program, and prove it safe by showing that if the original and transformed program are executed side-by-side, they will either eventually reach the "same" answer, or both not terminate.

The general safety Theorem 48 is based on executing the two programs and observing that the intermediate states are also related by the transformation (lifted to whole configuration and allowed to occur at multiple places), except the moments when the directly affected part of the program is executing. When the two answers are reached, they will be similar except that bodies of methods on the heap may differ as the transformation permits.

The transformation framework consists of the following parts:

- Definition of transformations of syntactic elements such as terms and configurations
- Definition of lifting transformations from local term transformations to transformations of whole programs and run-time machine configurations
- Definition of a *similarity transformation*, which uses similarity defined in Section 4.4.4 to deal with differences in variable names.
- Definition of important properties of local transformations, which are relevant to showing that those transformations are safe.
- Lemmas about lifting such properties from local transformations to whole programs.
- Definition of a safe transformation – that does not change the result of execution of a program.
- Theorem 48, about safety of transformations which have the required properties and are lifted to whole programs.

Theorem 48 states that if a local term transformation does not change typing of the term, is compatible with properties such as weakening, narrowing and substitution, does not change whether the term is an answer or not, and if execution of just the transformed term will eventually reach similar configurations, then transforming a program by this transformation anywhere will not change its result.

Transformations of roDOT Programs in General

The purpose of the transformation framework is to provide a means to show that certain transformations of a roDOT program do not affect the behavior of the program. For that, we need to define what a transformation is and what is the precise meaning of “not affecting the behavior”.

A transformation of a program is defined as a binary relation on terms – the original program and the transformed one. For example, the call-swapping transformation is defined as a relation that relates a program containing two method calls with a program that only differs in the order of those calls.

Because the safety of the transformation depends on typing information, the transformation will actually be a relation between tuples, containing not only the term, but also its type and a typing context.

This can be generalized from terms to other syntactic elements – stacks, heaps and machine configurations, though for each kind of element, the meaning of “typing context” and “type” differs slightly. For terms, the typing context is actually paired with a runtime environment $\Gamma; \rho$.

Definition 14 (Transformation)**.** *A transformation τ is a relation between triples consisting of typing contexts $\Gamma_{1,2}$, types $T_{1,2}$ and typeable elements $X_{1,2}$. We write $\langle \Gamma_1 \vdash X_1 : T_1 \rangle \rightarrow_\tau \langle \Gamma_2 \vdash X_2 : T_2 \rangle$ and say that X_1 is transformed into X_2.*

- A *term transformation* is a transformation $\langle \Gamma_1; \rho_1 \vdash t_1 : T_1 \rangle \rightarrow_\tau \langle \Gamma_2; \rho_2 \vdash t_2 : T_2 \rangle$, where $t_{1,2}$ are terms, $T_{1,2}$ are types, $\Gamma_{1,2}$ are typing contexts (mapping variables to types) and $\rho_{1,2}$ are environments mapping object references to heap locations.

- A *stack transformation* is a transformation $\langle \Gamma_1; \rho_1 \vdash \sigma_1 : T_1, T_3 \rangle \rightarrow_\tau \langle \Gamma_2; \rho_2 \vdash \sigma_2 : T_2, T_4 \rangle$, where $\sigma_{1,2}$ are stacks, $T_{1,2}$ are types of the holes in the top frame of $\sigma_{1,2}$ respectively, $T_{3,4}$ are types of the bottom frame of $\sigma_{1,2}$, $\Gamma_{1,2}$ are typing contexts (mapping variables to types) and $\rho_{1,2}$ are environments mapping object references to heap locations.

- A *heap transformation* is a transformation $\langle \rho_1 \vdash \sigma_1 : \Gamma_1 \rangle \rightarrow_\tau \langle \rho_2 \vdash \sigma_2 : \Gamma_2 \rangle$, where $\Sigma_{1,2}$ are heaps, $\Gamma_{1,2}$ are typing contexts (mapping variables to types) and $\rho_{1,2}$ are environments mapping object references to heap locations. In this definition, the typing context take the position of a type, because the types of variables in $Gamma_{1,2}$ correspond to types in $Sigma_{1,2}$.

- A *configuration transformation* is a transformation $\langle \Gamma_1 \vdash c_1 : T_1 \rangle \rightarrow_\tau \langle \Gamma_2 \vdash c_2 : T_2 \rangle$, where $c_{1,2}$ are configurations, $T_{1,2}$ are types, and $\Gamma_{1,2}$ are typing contexts (mapping variables to types).

Like any binary relation, transformations can be symmetric, reflexive or transitive, and we can construct transformations using iteration, composition, union and inversion.

Additionally, a transformation is *type-safe*, if the syntactic elements on both sides are correctly typed under the respective contexts. Another useful property of a transformation is being *type-identical*, where both the types and typing contexts are the same on both sides.

Definition 15 (Type-identical transformation)**.** *A transformation τ is type-identical, if $\langle \Gamma_1 \vdash X_1 : T_1\rangle \rightarrow_\tau \langle \Gamma_2 \vdash X_2 : T_2\rangle$ implies $\Gamma_1 = \Gamma_2$ and $T_1 = T_2$.*

Definition 16 (Type-safe transformation)**.** *A transformation τ is type-safe, if $\langle \Gamma_1 \vdash X_1 : T_1\rangle \rightarrow_\tau \langle \Gamma_2 \vdash X_2 : T_2\rangle$ implies $\Gamma_1 \vdash X_1 : T_1$ and $\Gamma_2 \vdash X_2 : T_2$.*

The Similarity Transformation

In Section 4.4.4, we defined similarity of configurations as a relation with a list of pairs of variables, to deal with location names in the SEF guarantee. We can treat similarity as an instance of a configuration transformation.

Definition 17 (Similarity transformation)**.** *The similarity transformation $\approx$ is a transformation of configurations that relates $\langle \Gamma_1 \vdash c_1 : T_1\rangle \rightarrow_\approx \langle \Gamma_2 \vdash c_2 : T_2\rangle$ when $\Gamma_1 \vdash c_1 : T_1$, $\Gamma_2 \vdash c_2 : T_2$ and $c_1 \overset{xs}{\approx} c_2$, where xs is a one-to-one mapping of variables from $\operatorname{fv} c_1$ and $\operatorname{fv} c_2$.*

In this definition, both sides must be correctly typed, which enables easier reasoning. The typing contexts may differ, because they do not affect the execution.

Therefore, the similarity transformation is not type-identical, but is trivially type-safe, reflexive, transitive and symmetric.

Lifting Local Transformations

The safe swapping of calls will be defined as a transformation of terms that transforms one program containing two successive calls into another program in which the calls are swapped.

Due to the A-normal form of terms, two successive calls in the program have the form $\mathsf{let}\, x_{c1} = x_{o1}.m_1 x_{a1} \,\mathsf{in}\, \mathsf{let}\, x_{c2} = x_{o2}.m_2 x_{a2} \,\mathsf{in}\, t$, where m_1 and m_2 are methods and t is the continuation of the program.

The swapped calls can be located anywhere in the program, for example inside a method of an object literal such as $\mathsf{let}\, x = \nu(r : R)\{m(r, z) = t_m\} \,\mathsf{in}\, t_2$.

To facilitate the possibility of the transformation being located anywhere in a term, it is useful to define the transformation in two steps: (1) A *local transformation*, which only allows swapping calls at the root of the term. (2) A lifting operator $\mathsf{lift}\ \tau$ which takes a local transformation τ and allows it to be located at one place anywhere in a term.

Such a local transformation τ of a term can be further lifted by $\mathsf{cfg}\ \tau$ to a whole run-time configuration, where τ applies at exactly one place in the focus of execution, in the stack or on the heap.

To allow multiple occurrences, we can apply the iteration operator to the lifted transformation. Having a definition that only allows one occurrence is useful in the proof of Theorem 48 in Section 4.5.1, where we want to look at each occurrence individually. The definitions of the lifting operators are shown in Figure 4.13 and Figure 4.14.

The operator $\mathsf{cfg}\ \tau$ lifts τ to anywhere in the configuration, while $\mathsf{foc}\ \tau$ only lifts to the root of the focus of execution.

Term, stack, heap and configuration transformations can be defined by lifting a local term transformation, as defined in Figure 4.13 and Figure 4.14. This lifting contain premises that ensure that both sides of the transformation are well typed. The rules in these definition mimic the typing rules and heap correspondence rules, but applying to two terms, stacks, heap or configurations at the same time.

The lifting to configurations is used to relate intermediate states of execution of the original program with execution of the transformed program. During execution, which is defined using small-step semantics, the place where a transformation is located will move around as the machine state evolves. For example, when an object is instantiated, if the transformation was applied within a method of that literal, then after instantiation, the transformation will apply in the heap.

Example 11. Let's consider two configurations related by a lifted transformation τ: $\langle\Gamma \vdash \langle \mathsf{let}\, x = \nu(r : R)\{m_1(r, z) = t_1\} \,\mathsf{in}\, t_2; \sigma; \Sigma; \rho\rangle : T\rangle \rightarrow_{\mathsf{cfg}\,\tau} \langle\Gamma \vdash \langle \mathsf{let}\, x = \nu(r : R)\{m_1(r, z) = t_2\} \,\mathsf{in}\, t_2; \sigma; \Sigma; \rho\rangle : T\rangle$, where the bodies of the method are related by τ: $\langle\Gamma, r : R, z : T_z \vdash t_1 : T_t\rangle \rightarrow_\tau \langle\Gamma, r : R, z : T_z \vdash t_2 : T_t\rangle$. After one reduction step, the next configurations will still be related by the lifted transformation. $\langle\Gamma \vdash \langle[t_2/x_2]w; \sigma; \Sigma, y \sim \nu(r : R)\{m(r, z) = t_{12}\}; \rho, w \sim y\rangle : T\rangle \rightarrow_{\mathsf{cfg}\,\tau} \langle\Gamma \vdash \langle[t_2/x_2]w; \sigma; \Sigma, y \sim \nu(r : R)\{m(r, z) = t_{21}\}; \rho, w \sim y\rangle : T\rangle$

Safe Transformations

As pointed out in the list at the beginning of Section 4.5.1, the definition of a safe transformation must allow for different variable names and for the fact that in the program answer, the transformation can still occur at multiple places in the heap. Therefore a local transformation is safe if execution of the transformed program reaches answers related by an iteration of this transformation in union with similarity.

Definition 18 (Safe transformation)**.** *A transformation τ is safe if $\langle\Gamma_1 \vdash c_1 : T\rangle \rightarrow_\tau \langle\Gamma_2 \vdash c_2 : T\rangle$ and $\vdash \langle t_1; \cdot; \cdot; \cdot\rangle : T$ and $c_1 \longrightarrow^k c_3$, where c_3 is an answer typed as $\Gamma_3 \vdash c_3 : T$, implies that there exists c_4, Γ_4 and j such that $c_2 \longrightarrow^j c_4$, $\Gamma_4 \vdash c_4 : T$ and $\langle\Gamma_3 \vdash c_3 : T\rangle \rightarrow_{(\tau\cup\approx)^*} \langle\Gamma_4 \vdash c_4 : T\rangle$.*

Thanks to being able to define a transformation by applying a general lifting to a local transformation, the safety proof of such a transformation can be also divided into a theorem that will apply to any local transformation with certain local properties, and then proving those local properties for the particular local transformation.

This approach makes it possible to state the call-swapping guarantee presented here (Theorem 52), or analogous guarantees for other local transformations.

Such transformation guarantees can be proven by looking at executions of the two programs (original and transformed) in parallel, and showing that the intermediate states are still related by the iterated lifted transformation (plus similarity), until both executions reach answer states. The answers will therefore also be related by that transformation, and so the two programs will have the same result in the sense of Definition 18.

As the two programs execute, the transformation will be moved around in the configuration. The ability to apply the transformation anywhere is ensured by the lifting operator, but in order for the transformation to still apply, the typing conditions need to be preserved as well as the configuration and typing context changes. For that reason, the local transformation has to have additional properties: weakening, narrowing and substitution.

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash t_1 : T\rangle \xrightarrow{\tau} \langle \Gamma; \rho \vdash t_2 : T\rangle \\ \Gamma; \rho \vdash T <: S\end{array}}{\langle \Gamma; \rho \vdash t_1 : S\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash t_2 : S\rangle}\text{(TRFL-Local)}$$

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash t_1 : T_1\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash t_2 : T_1\rangle \\ \Gamma, z : T_1; \rho \vdash t_3 : T_2 \qquad \Gamma; \rho \vdash T_2 <: S\end{array}}{\langle \Gamma; \rho \vdash \mathsf{let}\ z = t_1\ \mathsf{in}\ t_3 : S\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash \mathsf{let}\ z = t_2\ \mathsf{in}\ t_3 : S\rangle}\text{(TRFL-Let1)}$$

$$\frac{\begin{array}{c}\langle \Gamma, z : T_1; \rho \vdash t_1 : T_2\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma, z : T_1; \rho \vdash t_2 : T_2\rangle \\ \Gamma; \rho \vdash t_3 : T_1 \qquad \Gamma; \rho \vdash T_2 <: S\end{array}}{\langle \Gamma; \rho \vdash \mathsf{let}\ z = t_3\ \mathsf{in}\ t_1 : S\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash \mathsf{let}\ z = t_3\ \mathsf{in}\ t_2 : S\rangle}\text{(TRFL-Let2)}$$

$$\frac{\begin{array}{c}\langle \Gamma, s : R; \rho \vdash d_1 : R\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma, s : R; \rho \vdash d_2 : R\rangle \\ \Gamma, z : \mu(s : R) \wedge \mathcal{M}_\bot; \rho \vdash t : T \\ \Gamma; \rho \vdash T <: S \qquad R\ \textbf{indep}\ s\end{array}}{\langle \Gamma; \rho \vdash \mathsf{let}\ z = \nu(s : R)d_1\ \mathsf{in}\ t : S\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash \mathsf{let}\ z = \nu(s : R)d_2\ \mathsf{in}\ t : S\rangle}\text{(TRFL-Lit1)}$$

$$\frac{\begin{array}{c}\langle \Gamma, z : \mu(s : R) \wedge \mathcal{M}_\bot; \rho \vdash t_1 : T\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma, z : \mu(s : R) \wedge \mathcal{M}_\bot; \rho \vdash t_2 : T\rangle \\ \Gamma, s : R; \rho \vdash d : R \\ \Gamma; \rho \vdash T <: S \qquad R\ \textbf{indep}\ s\end{array}}{\langle \Gamma; \rho \vdash \mathsf{let}\ z = \nu(s : R)d\ \mathsf{in}\ t_1 : S\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash \mathsf{let}\ z = \nu(s : R)d\ \mathsf{in}\ t_2 : S\rangle}\text{(TRFL-Lit2)}$$

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash d_1 : T_1\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash d_2 : T_1\rangle \\ \Gamma; \rho \vdash d_3 : T_2 \\ d_1 \text{ and } d_3 \text{ have distinct member names} \\ d_2 \text{ and } d_3 \text{ have distinct member names}\end{array}}{\langle \Gamma; \rho \vdash d_1 \wedge d_3 : T_1 \wedge T_2\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash d_2 \wedge d_3 : T_1 \wedge T_2\rangle}\text{(TRFL-And1)}$$

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash d_1 : T_1\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash d_2 : T_1\rangle \\ \Gamma; \rho \vdash d_3 : T_2 \\ d_1 \text{ and } d_3 \text{ have distinct member names} \\ d_2 \text{ and } d_3 \text{ have distinct member names}\end{array}}{\langle \Gamma; \rho \vdash d_3 \wedge d_1 : T_2 \wedge T_1\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash d_3 \wedge d_2 : T_2 \wedge T_1\rangle}\text{(TRFL-And2)}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma, s : T_4, !, z : T_1, r : T_4 \wedge [r/s]T_4 \wedge T_3 \\ R = \{m(z : T_1, r : T_3) : T_2\} \\ \langle \Gamma'; \rho \vdash t_1 : T_2\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma'; \rho \vdash t_2 : T_2\rangle\end{array}}{\langle \Gamma, s : T_4; \rho \vdash \{m(z, r) = t_1\} : R\rangle \xrightarrow{\text{lift } \tau} \langle \Gamma, s : T_4; \rho \vdash \{m(z, r) = t_2\} : R\rangle}\text{(TRFL-Met)}$$

Figure 4.13: Lifting local transformations to terms

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash t_1 : T_1 \rangle \xrightarrow{\tau} \langle \Gamma; \rho \vdash t_2 : T_1 \rangle \\ \Gamma \sim \rho \qquad \Gamma; \rho \sim \Sigma \\ \Gamma; \rho \vdash T_1 <: T_2 \qquad \Gamma; \rho \vdash \sigma : T_2, S\end{array}}{\langle \Gamma \vdash \langle t_1; \sigma; \Sigma; \rho \rangle : S \rangle \xrightarrow{\text{foc } \tau} \langle \Gamma \vdash \langle t_2; \sigma; \Sigma; \rho \rangle : S \rangle}\text{(TRFF-Focus)}$$

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash t_1 : T_1 \rangle \xrightarrow{\text{lift } \tau} \langle \Gamma; \rho \vdash t_2 : T_1 \rangle \\ \Gamma \sim \rho \qquad \Gamma; \rho \sim \Sigma \\ \Gamma; \rho \vdash T_1 <: T_2 \qquad \Gamma; \rho \vdash \sigma : T_2, S\end{array}}{\langle \Gamma \vdash \langle t_1; \sigma; \Sigma; \rho \rangle : S \rangle \xrightarrow{\text{cfg } \tau} \langle \Gamma \vdash \langle t_2; \sigma; \Sigma; \rho \rangle : S \rangle}\text{(TRFC-Focus)}$$

$$\frac{\begin{array}{c}\langle \Gamma; \rho \vdash \sigma_1 : T_2, S \rangle \xrightarrow{\text{stack } \tau} \langle \Gamma; \rho \vdash \sigma_2 : T_2, S \rangle \\ \Gamma \sim \rho \qquad \Gamma; \rho \sim \Sigma \\ \Gamma; \rho \vdash T_1 <: T_2 \qquad \Gamma; \rho \vdash t : T_1\end{array}}{\langle \Gamma \vdash \langle t; \sigma_1; \Sigma; \rho \rangle : S \rangle \xrightarrow{\text{cfg } \tau} \langle \Gamma \vdash \langle t; \sigma_2; \Sigma; \rho \rangle : S \rangle}\text{(TRFC-Stack)}$$

$$\frac{\begin{array}{c}\langle \rho \vdash \Sigma_1 : \Gamma \rangle \xrightarrow{\text{heap } \tau} \langle \rho \vdash \Sigma_2 : \Gamma \rangle \\ \Gamma \sim \rho \qquad \Gamma; \rho \vdash t : T_1 \\ \Gamma; \rho \vdash T_1 <: T_2 \qquad \Gamma; \rho \vdash \sigma : T_2, S\end{array}}{\langle \Gamma \vdash \langle t; \sigma; \Sigma_1; \rho \rangle : S \rangle \xrightarrow{\text{cfg } \tau} \langle \Gamma \vdash \langle t; \sigma; \Sigma_2; \rho \rangle : S \rangle}\text{(TRFC-Heap)}$$

Figure 4.14: Lifting local transformations to configurations

These properties are analogous to the weakening, narrowing and substitution lemmas, which are a part of the soundness proof for the roDOT calculus and were adapted from the soundness proof for DOT [95]. The properties state that the transformation will relate two terms even if the typing context is extended, when a type in the typing context is refined, or when a variable is substituted in the term and its type. As an example, the weakening property is stated in Definition 19.

Definition 19 (Transformation weakening)**.** *A transformation τ is compatible with weakening, if $\langle \Gamma_1, \Gamma_2 \vdash t_1 : T\rangle \rightarrow_\tau \langle \Gamma_1, \Gamma_2 \vdash t_2 : T\rangle$ implies $\langle \Gamma_1, \Gamma_3, \Gamma_2 \vdash t_1 : T\rangle \rightarrow_\tau \langle \Gamma_1, \Gamma_3, \Gamma_2 \vdash t_2 : T\rangle$.*

If a local transformation has these properties, then so does the lifted transformation. The weakening property can also be lifted to stacks and heaps.

Lemma 45 (Transformation weakening lifting)**.** *If a transformation τ is compatible with weakening, then $\mathsf{lift}\,\tau$ is compatible with weakening.*

Another important property of a transformation is that it *preserves answers*, meaning that it only relates answers with answers and non-answers with non-answers.

With these properties, we can show that as the programs both execute a single step, the transformation will relate the next states, with exceptions stated below. First, the transformation may occur at more than one location after the step. For example, suppose that before a method call, the transformation occurred in the body on the heap of the method to be called. After the call, the transformation occurs both in the heap and in the focus of execution. For that reason, the transformation is iterated in the conclusion of Lemma 46. Second, when the execution has reached an answer, there are no more steps. This case is handled separately. Third, when the transformation occurs in the root of the focus, the executions may diverge for a while, but converge eventually. This case is also handled separately, and will be resolved later in the proof of Theorem 48 using Definition 20.

Lemma 46 (Transformation execution step)**.** *Let τ be a transformation that is type-identical, type-safe, compatible with weakening, narrowing and substitution, and preserves answers.*

If $\langle \Gamma \vdash c_1 : T\rangle \rightarrow_{\mathsf{cfg}\,\tau} \langle \Gamma \vdash c_2 : T\rangle$, then one of these holds:

- *Both c_1 and c_2 are answers.*
- $\langle \Gamma \vdash c_1 : T\rangle \rightarrow_{\mathsf{foc}\,\tau} \langle \Gamma \vdash c_2 : T\rangle$
- *There exist c_3 and c_4, such that $c_1 \longrightarrow c_3$, $c_2 \longrightarrow c_4$, and $\langle \Gamma \vdash c_3 : T\rangle \rightarrow_{(\mathsf{cfg}\,\tau \cup \approx)^*} \langle \Gamma \vdash c_4 : T\rangle$.*

Proof. By the progress lemma, which is a part of the safety proof [43], the configurations c_1 and c_2 can each be an answer or reduce to another configuration. Because τ preserves answers, either both are answers, and the first case applies, or both are not answers, so both can progress. In that case, we do case analysis for the reduction step and the location where the transformation occurs.

- If in the root of the focus, then the second case applies.
- If in a first part of a let-term, then it moves up as the let term is split. If it applies in a first part of a let-lit-term, then it moves to the heap as part of a newly instantiated object. If it applies in a second part of a let-term, then moves to the stack.
- If in the stack, then in case of steps that change the stack, it either moves up the stack, down the stack or into the focus. In other cases, it applies the same way in the same stack, but weakening and similarity is needed in steps that create new variables.
- If in the heap, then it always stays in the heap, where weakening may be applied. In the case of a method call, it will be additionally copied to the focus, while applying substitution.

$\square$

The case when the transformation applies in the root of the focus represents the instant when the two programs start to go in different ways, but the transformation is safe because the programs will still eventually reach similar states.

In the case of call swapping, this means that in one program, the call to m_1 is executed first, then the call to m_2. In the transformed program, it is vice versa. During this time, the intermediate states may differ arbitrarily, but when they return from both of the calls, the results are the same regardless of the order.

Definition 20 (Eventual similarity)**.** *A transformation τ eventually reduces to similarity, if for $\langle \Gamma_1 \vdash c_1 : T\rangle \rightarrow_{foc\,\tau} \langle \Gamma_2 \vdash c_2 : T\rangle$ where c_1 terminates, there exists $n \geq m$ and $c_3, c_4, \Gamma_3, \Gamma_4$, such that*
$c_1 \longrightarrow^n c_3$, $c_2 \longrightarrow^m c_4$, $\Gamma_3 \vdash c_3 : T$, $\Gamma_4 \vdash c_4 : T$, and $\langle \Gamma_3 \vdash c_3 : T\rangle \rightarrow_{\approx} \langle \Gamma_4 \vdash c_4 : T\rangle$.

With swapping calls, both executions reach similarity in the same number steps, but in general, the numbers of steps n and m may differ. The condition $n \geq m$ is needed in order to avoid the case where the execution of the program on the right side would be indefinitely prolonged by increasing the number of steps of the execution over and over.

Additionally, we will use the fact that the $\approx$ transformation is also preserved as the program executes.

Lemma 47 (Similarity compatible with reduction)**.**

If $\langle \Gamma_1 \vdash c_1 : T_1\rangle \rightarrow_{\approx} \langle \Gamma_2 \vdash c_2 : T_2\rangle$	If c_1 is similar to c_2,
and $c_1 \longrightarrow c_3$, $c_2 \longrightarrow c_4$,	and these configurations reduce in one step,
then $\langle \Gamma_1 \vdash c_3 : T_1\rangle \rightarrow_{\approx} \langle \Gamma_2 \vdash c_4 : T_2\rangle$.	then the result is also similar.

The property Definition 20 is only concerned with what happens if the transformation occurs once, in the root of the focus. But with Lemma 46 and Lemma 47, we can show that for a local transformation with the properties above, if this transformation occurs anywhere in the configuration, any number of times, the answers will also differ only by occurrences of this transformation and by similarity.

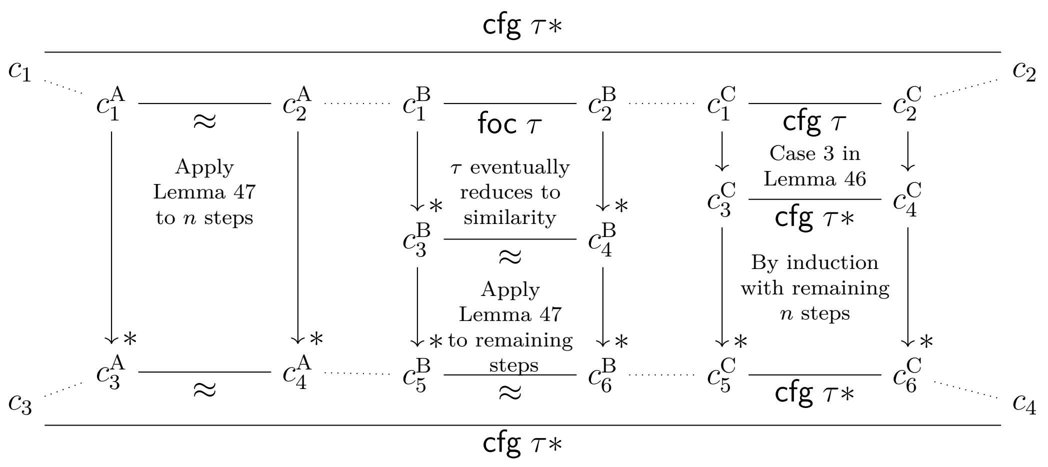

Figure 4.15: Proof of transformation execution theorem, showing three different cases for $n + 1$ steps

Theorem 48 (Transformation execution)**.** *If τ is a transformation that is type-identical, type-safe, compatible with weakening, narrowing and substitution, preserves answers, and eventually reduces to similarity, then $(\mathsf{cfg}\,\tau \cup \approx)^*$ is safe.*

Proof. The conclusion of this theorem is $\langle \Gamma_1 \vdash c_1 : T\rangle \rightarrow_{(\mathsf{cfg}\,\tau\cup\approx)^*} \langle \Gamma_2 \vdash c_2 : T\rangle$ and $\Gamma_1 \vdash c_1 : T_1$ and $c_1 \longrightarrow^k c_3$ where c_3 is an answer and $\Gamma_3 \vdash c_3 : T$, implies there exists c_4, Γ_4 and j such that $\langle t_2; \cdot; \cdot; \cdot\rangle \longrightarrow^j c_4$, $\Gamma_4 \vdash c_4 : T$, and $\langle \Gamma_3 \vdash c_3 : T\rangle \rightarrow_{(\mathsf{cfg}\,\tau\cup\approx)^*} \langle \Gamma_4 \vdash c_4 : T\rangle$.

First, do induction on j, the number of execution steps to reach the answer. For 0 steps, the conclusion trivially holds. For $n+1$ execution steps, do induction on the iteration of the transformation $(\mathsf{cfg}\ \tau\cup \approx)^*$ between c_1 and c_2. Each step of this iteration is either similarity, or application of $\mathsf{cfg}\ \tau$. We will show that every such step of the transformation reduces to answers related by $(\mathsf{cfg}\ \tau\cup \approx)^*$, using different cases which are shown in Figure 4.15. Then the final transformation between c_3 and c_4 will be a concatenation of the transformation produced all transformation steps.

The case when the transformation step is similarity, is shown in Figure 4.15 as the transformation between c_1^A and c_2^A on the left. We can use Lemma 47 on all steps of the execution until the answers are reached. Because similarity is compatible with reduction, such answers are also similar.

For a transformation step that is $\mathsf{cfg}\ \tau$, we apply Lemma 46 and do case analysis:

- In the case where c_1 and c_2 are answers, there cannot be any more steps, which contradicts the assumption that there are $n + 1$ steps.

- The case where the transformation applies in the focus is shown in Figure 4.15 as c_1^B and c_2^B. First we use the fact that τ eventually reduces to similarity, to get intermediate configurations c_3^B and c_4^B which are similar. Then, we use Lemma 47 to apply the similarity to the answers.

- The case where an execution step is made and the transformation is preserved (Case 3 in Lemma 46), is shown in Figure 4.15 as c_1^C and c_2^C reducing to c_3^{C} and c_3^{C} in one step. We apply the first inductive hypothesis of this theorem (with n execution steps) to the rest of the execution.

□

4.5.2 The Call-swapping Transformation

The specific transformation guarantee that we want to achieve should state that swapping two calls will not change the outcome of the program, in the sense of Definition 18.

The call-swapping is defined as a local transformation, where, due to the A-normal form of terms, the calls are in let-in terms. The two calls $x_{o1}.m_1 x_{a1}$ and $x_{o2}.m_2 x_{a2}$ appear in a different order, but the continuation t is the same.

The transformation is only safe if both the methods are side-effect free. For that reason, the transformation requires several typing conditions analogous to the premises of Theorem 37 in Section 4.4.2.

Definition 21 (Local call swapping). *The local call-swapping transformation* csw *is a transformation of terms that relates* $\langle \Gamma \vdash \mathsf{let}\, x_{c1} = x_{o1}.m_1 x_{a1} \,\mathsf{in}\, \mathsf{let}\, x_{c2} = x_{o2}.m_2 x_{a2} \,\mathsf{in}\, t : T \rangle \rightarrow_{\mathsf{csw}} \langle \Gamma \vdash \mathsf{let}\, x_{c2} = x_{o2}.m_2 x_{a2} \,\mathsf{in}\, \mathsf{let}\, x_{c1} = x_{o1}.m_1 x_{a1} \,\mathsf{in}\, t : T \rangle$ *when*

- $x_{c1,2}$ *are distinct from* $x_{a1,2}$ *and* $x_{o1,2}$*,*
- $\Gamma \vdash x_{o1}.m_1 x_{a1} : T$*,*
- $\Gamma \vdash x_{o2}.m_2 x_{a2} : T$*,*
- $\Gamma \vdash x_{o1} : \{m_1(r_1 : \mathsf{N}, z_1 : T_{a_1}) : \top\}$*,*
- $\Gamma \vdash x_{o2} : \{m_2(r_2 : \mathsf{N}, z_2 : T_{a_2}) : \top\}$*,*
- $\Gamma \vdash x_{a1} : T_{a_1}$*,*
- $\Gamma \vdash x_{a2} : T_{a_2}$*,*
- $\Gamma \vdash \mathsf{N} <: T_{a_1}$*, and*
- $\Gamma \vdash \mathsf{N} <: T_{a_2}$*.*

4.5.3 The Call-swapping Transformation Guarantee

To state the transformation guarantee, we first apply Definition 18 to Definition 21:

Lemma 49 (Call swapping is safe). *The transformation* $\mathsf{cfg}\ \mathsf{csw}$ *is safe.*

Thanks to using a local transformation csw lifted to configurations, we can prove this lemma using Theorem 48.

First, we need to to prove the premises of Theorem 48, which are local properties of the local transformation csw defined in Definition 21.

$$\begin{tikzcd}
\begin{array}{c}\langle \mathsf{let}\, x_{\mathrm{a}1} = w_{\mathrm{r}1}.m_1\, w_{\mathrm{z}1}\, \mathsf{in} \\ \mathsf{let}\, x_{\mathrm{a}2} = w_{\mathrm{r}2}.m_2\, w_{\mathrm{z}2}\, \mathsf{in} \\ t(x_{\mathrm{a}1}, x_{\mathrm{a}2}) \\ ;\sigma;\Sigma;\rho\rangle\end{array} \arrow[rr, dash, "\mathsf{foc\ csw}"] \arrow[d] & & \begin{array}{c}\langle \mathsf{let}\, x_{\mathrm{b}2} = w_{\mathrm{r}2}.m_2\, w_{\mathrm{z}2}\, \mathsf{in} \\ \mathsf{let}\, x_{\mathrm{b}1} = w_{\mathrm{r}1}.m_1\, w_{\mathrm{z}1}\, \mathsf{in} \\ t(x_{\mathrm{b}1}, x_{\mathrm{b}2}) \\ ;\sigma;\Sigma;\rho\rangle\end{array} \arrow[d] \\
\begin{array}{c}\langle w_{\mathrm{r}1}.m_1\, w_{\mathrm{z}1} \\ ;\cdots :: \sigma;\Sigma;\rho\rangle\end{array} \arrow[d, "k_1"'] \arrow[dddrr, dash, "\approx"] & & \begin{array}{c}\langle w_{\mathrm{r}2}.m_2\, w_{\mathrm{z}2} \\ ;\cdots :: \sigma;\Sigma;\rho\rangle\end{array} \arrow[d, "k_2"'] \arrow[dddll, dash, "\approx"'] \\
\begin{array}{c}\langle \mathsf{v} w_{\mathrm{a}1} \\ ;\cdots :: \sigma;\Sigma\Sigma_{\mathrm{a}1};\rho\rho_{\mathrm{a}1}\rangle\end{array} \arrow[d] \arrow[dddrr, dash, "\approx"] & & \begin{array}{c}\langle \mathsf{v} w_{\mathrm{b}2} \\ ;\cdots :: \sigma;\Sigma\Sigma_{\mathrm{b}2};\rho\rho_{\mathrm{b}2}\rangle\end{array} \arrow[d] \arrow[dddll, dash, "\approx"'] \\
\begin{array}{c}\langle \mathsf{let}\, x_{\mathrm{a}2} = w_{\mathrm{r}2}.m_2\, w_{\mathrm{z}2}\, \mathsf{in} \\ \ldots \\ ;\sigma;\Sigma\Sigma_{\mathrm{a}1};\rho\rho_{\mathrm{a}1}\rangle\end{array} \arrow[d] & & \begin{array}{c}\langle \mathsf{let}\, x_{\mathrm{b}1} = w_{\mathrm{r}1}.m_1\, w_{\mathrm{z}1}\, \mathsf{in} \\ \ldots \\ ;\sigma;\Sigma\Sigma_{\mathrm{b}2};\rho\rho_{\mathrm{b}2}\rangle\end{array} \arrow[d] \\
\begin{array}{c}\langle w_{\mathrm{r}2}.m_2\, w_{\mathrm{z}2} \\ ;\cdots :: \sigma;\Sigma\Sigma_{\mathrm{a}1};\rho\rho_{\mathrm{a}1}\rangle\end{array} \arrow[d, "k_2"'] & & \begin{array}{c}\langle w_{\mathrm{r}1}.m_1\, w_{\mathrm{z}1} \\ ;\cdots :: \sigma;\Sigma\Sigma_{\mathrm{b}2};\rho\rho_{\mathrm{b}2}\rangle\end{array} \arrow[d, "k_1"'] \\
\begin{array}{c}\langle \mathsf{v} w_{\mathrm{a}2} \\ ;\cdots :: \sigma;\Sigma\Sigma_{\mathrm{a}1}\Sigma_{\mathrm{a}2};\rho\rho_{\mathrm{a}1}\rho_{\mathrm{a}2}\rangle\end{array} \arrow[d] & & \begin{array}{c}\langle \mathsf{v} w_{\mathrm{b}1} \\ ;\cdots :: \sigma;\Sigma\Sigma_{\mathrm{b}2}\Sigma_{\mathrm{b}1};\rho\rho_{\mathrm{b}2}\rho_{\mathrm{b}1}\rangle\end{array} \arrow[d] \\
\begin{array}{c}\langle t(w_{\mathrm{a}1}, w_{\mathrm{a}2}) \\ ;\sigma;\Sigma\Sigma_{\mathrm{a}1}\Sigma_{\mathrm{a}2};\rho\rho_{\mathrm{a}1}\rho_{\mathrm{a}2}\rangle\end{array} \arrow[rr, dash, "\approx"] & & \begin{array}{c}\langle t(w_{\mathrm{b}1}, w_{\mathrm{b}2}) \\ ;\sigma;\Sigma\Sigma_{\mathrm{b}2}\Sigma_{\mathrm{b}1};\rho\rho_{\mathrm{b}2}\rho_{\mathrm{b}1}\rangle\end{array}
\end{tikzcd}$$

Figure 4.16: Reduction and transformation of call swapping in focus

The local transformation csw is trivially symmetric and type-identical, and it preserves answers, because it only applies to let-terms, which are never in the focus of an answer.

It is also type-safe and compatible with weakening, narrowing and substitution, which can be proved by applying weakening, narrowing or substitution to each of the typing and subtyping conditions in the definition of csw.

Lemma 50 (Call-swap weakening)**.** *The transformation csw is compatible with weakening.*

The crucial property that ensures safety of the transformation is that csw satisfies Definition 20.

Lemma 51 (Call swapping eventual similarity)**.** *The transformation csw eventually reduces to similarity.*

Proof. The proof is based on looking at the execution of let-terms and the method calls in the two programs, using the SEF guarantee to relate intermediate states by similarity, and combining the information obtained from that to relate the states after the calls by similarity. This is captured in Figure 4.16.

We have two configurations $c_\mathrm{a} = \langle \mathsf{let}\ x_\mathrm{a1} = w_\mathrm{r1}.m_1\,w_\mathrm{z1}\ \mathsf{in}\ \mathsf{let}\ x_\mathrm{a2} = w_\mathrm{r2}.m_2\,w_\mathrm{z2}\ \mathsf{in}\ t(x_\mathrm{a1}, x_\mathrm{a2}); \sigma; \rho; \Sigma\rangle$ and $c_\mathrm{b} = \langle \mathsf{let}\ x_\mathrm{b2} = w_\mathrm{r2}.m_2\,w_\mathrm{z2}\ \mathsf{in}\ \mathsf{let}\ x_\mathrm{b1} = w_\mathrm{r1}.m_1\,w_\mathrm{z1}\ \mathsf{in}\ t(x_\mathrm{b1}, x_\mathrm{b2}); \sigma; \rho; \Sigma\rangle$. Assuming that they both terminate, we need to show that they reduce to similar configurations.

c_a reduces to method call begin $\langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \sigma_\mathrm{a1}\sigma; \rho; \Sigma\rangle$, where $\sigma_\mathrm{a1} = \mathsf{let}\ x_\mathrm{a1} = \square\ \mathsf{in}\ (\mathsf{let}\ x_\mathrm{a2} = w_\mathrm{r2}.m_2\,w_\mathrm{z2}\ \mathsf{in}\ t(x_\mathrm{a1}, x_\mathrm{a2}))$. Assuming termination, there must exist a method call end $\langle \mathsf{v}w_\mathrm{a1}; \sigma_\mathrm{a1}\sigma; \rho\rho_\mathrm{a1}; \Sigma\Sigma_\mathrm{a1}\rangle$. By the SEF guarantee, in the method call end, the heap has a prefix Σ. The stack does not affect the result, so we can also say that $\langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \cdot; \rho; \Sigma\rangle \longmapsto^{k_1} \langle \mathsf{v}w_\mathrm{a1}; \cdot; \rho\rho_\mathrm{a1}; \Sigma\Sigma_\mathrm{a1}\rangle$.

The execution continues by reducing to $\langle \mathsf{let}\ x_\mathrm{a2} = w_\mathrm{r2}.m_2\,w_\mathrm{z2}\ \mathsf{in}\ t(w_\mathrm{a1}, x_\mathrm{a2}); \sigma; \rho\rho_\mathrm{a1}; \Sigma\Sigma_\mathrm{a1}\rangle$, then to $c_\mathrm{a2} = \langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \sigma_\mathrm{a2}\sigma; \rho\rho_\mathrm{a1}; \Sigma\Sigma_\mathrm{a1}\rangle$, where $\sigma_\mathrm{a2} = \mathsf{let}\ x_\mathrm{a2} = \square\ \mathsf{in}\ t(w_\mathrm{a1}, x_\mathrm{a2})$.

This is a method call begin which will reduce to a method call end $\langle \mathsf{v}w_\mathrm{a2}; \sigma_\mathrm{a2}\sigma; \rho\rho_\mathrm{a1}\rho_\mathrm{a2}; \Sigma\Sigma_\mathrm{a1}\Sigma_\mathrm{a2}\rangle$. Again, by the SEF guarantee, the heap in the end has a prefix $\Sigma\Sigma_\mathrm{a1}$. And again, the stack does not affect the execution, so we also have $\langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \cdot; \rho\rho_\mathrm{a1}; \Sigma\Sigma_\mathrm{a1}\rangle \longmapsto^{k_2} \langle \mathsf{v}w_\mathrm{a2}; \cdot; \rho\rho_\mathrm{a1}\rho_\mathrm{a2}; \Sigma\Sigma_\mathrm{a1}\Sigma_\mathrm{a2}\rangle$. With the stack removed, the method call begin does not have any references to the objects created in the previous call, so it is similar to $\langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \cdot; \rho; \Sigma\rangle$ and the method call end is similar to $\langle \mathsf{v}w_\mathrm{a2}; \cdot; \rho\rho_\mathrm{a2}; \Sigma\Sigma_\mathrm{a2}\rangle$.

The execution continues by reducing to $c_\mathrm{a3} = \langle t(w_\mathrm{a1}, w_\mathrm{a2}); \sigma; \rho\rho_\mathrm{a1}\rho_\mathrm{a2}; \Sigma\Sigma_\mathrm{a1}\Sigma_\mathrm{a2}\rangle$. Now, we will show that c_b reduces to a similar configuration.

c_b reduces to method call begin $\langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \sigma_\mathrm{b2}\sigma; \rho; \Sigma\rangle$, where $\sigma_\mathrm{b2} = \mathsf{let}\ x_\mathrm{b2} = \square\ \mathsf{in}\ (\mathsf{let}\ x_\mathrm{b1} = w_\mathrm{r1}.m_1\,w_\mathrm{z1}\ \mathsf{in}\ t(x_\mathrm{b1}, x_\mathrm{b2}))$.

If we remove the stack, we get a configuration $\langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \cdot; \rho; \Sigma\rangle$ for which we already know that it reduces in k_2 steps to something similar to $\langle \mathsf{v}w_\mathrm{a2}; \cdot; \rho\rho_\mathrm{a2}; \Sigma\Sigma_\mathrm{a2}\rangle$. That means there are $w_\mathrm{b2}, \Sigma_\mathrm{b2}, \rho_\mathrm{b2}$ similar to $w_\mathrm{a2}, \Sigma_\mathrm{a2}, \rho_\mathrm{a2}$, such that $\langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \cdot; \rho; \Sigma\rangle \longmapsto^{k_2} \langle \mathsf{v}w_\mathrm{b2}; \cdot; \rho\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\rangle$.

Adding back the stack, we have $\langle w_\mathrm{r2}.m_2\,w_\mathrm{z2}; \sigma_\mathrm{b2}\sigma; \rho; \Sigma\rangle \longmapsto^{k_2} \langle \mathsf{v}w_\mathrm{b2}; \sigma_\mathrm{b2}\sigma; \rho\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\rangle$.

The execution continues by reducing to $\langle \mathsf{let}\ x_\mathrm{b1} = w_\mathrm{r1}.m_1\,w_\mathrm{z1}\ \mathsf{in}\ t(x_\mathrm{b1}, w_\mathrm{b2}); \sigma; \rho\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\rangle$, then to $c_\mathrm{b1} = \langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \sigma_\mathrm{b1}\sigma; \rho\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\rangle$, where $\sigma_\mathrm{b1} = \mathsf{let}\ x_\mathrm{b1} = \square\ \mathsf{in}\ t(x_\mathrm{b1}, w_\mathrm{b2})$.

If we remove the stack, we get a configuration $\langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \cdot; \rho\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\rangle$. With the stack removed, this method call begin does not have any references to the objects created in the previous call, so it is similar to $\langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \cdot; \rho; \Sigma\rangle$, for which we already know that it reduces in k_1 steps to something similar to $\langle \mathsf{v}w_\mathrm{a1}; \cdot; \rho\rho_\mathrm{a1}; \Sigma\Sigma_\mathrm{a1}\rangle$. That means there are $w_\mathrm{b1}, \Sigma_\mathrm{b1}, \rho_\mathrm{b1}$ similar to $w_\mathrm{a1}, \Sigma_\mathrm{a1}, \rho_\mathrm{a1}$, such that $\langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \cdot; \rho; \Sigma\rangle \longmapsto^{k_1} \langle \mathsf{v}w_\mathrm{b1}; \cdot; \rho\rho_\mathrm{b1}; \Sigma\Sigma_\mathrm{b1}\rangle$.

Adding back the stack and heap, we have $\langle w_\mathrm{r1}.m_1\,w_\mathrm{z1}; \sigma_\mathrm{b1}\sigma; \rho\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\rangle \longmapsto^{k_1} \langle \mathsf{v}w_\mathrm{b1}; \sigma_\mathrm{b1}\sigma; \rho\rho_\mathrm{b2}\rho_\mathrm{b1}; \Sigma\Sigma_\mathrm{b2}\Sigma_\mathrm{b1}\rangle$.

The execution continues by reducing to $c_\mathrm{b3} = \langle t(w_\mathrm{b1}, w_\mathrm{b2}); \sigma; \rho\rho_\mathrm{b1}\rho_\mathrm{b2}; \Sigma\Sigma_\mathrm{b2}\Sigma_\mathrm{b1}\rangle$. Because the variables and heaps in c_b3 were chosen to be similar to the variables and heaps in c_a3, we get $c_\mathrm{a3} \approx c_\mathrm{b3}$.

$\square$

Applying Theorem 48 on csw gives us that $(\mathsf{cfg\ csw} \cup \approx)^*$ is safe. That means $\mathsf{cfg\ csw}$ is safe as well, because $\mathsf{cfg\ csw} \subset (\mathsf{cfg\ csw} \cup \approx)^* = ((\mathsf{cfg\ csw} \cup \approx)^* \cup \approx)^*$. This concludes the proof of Lemma 49.

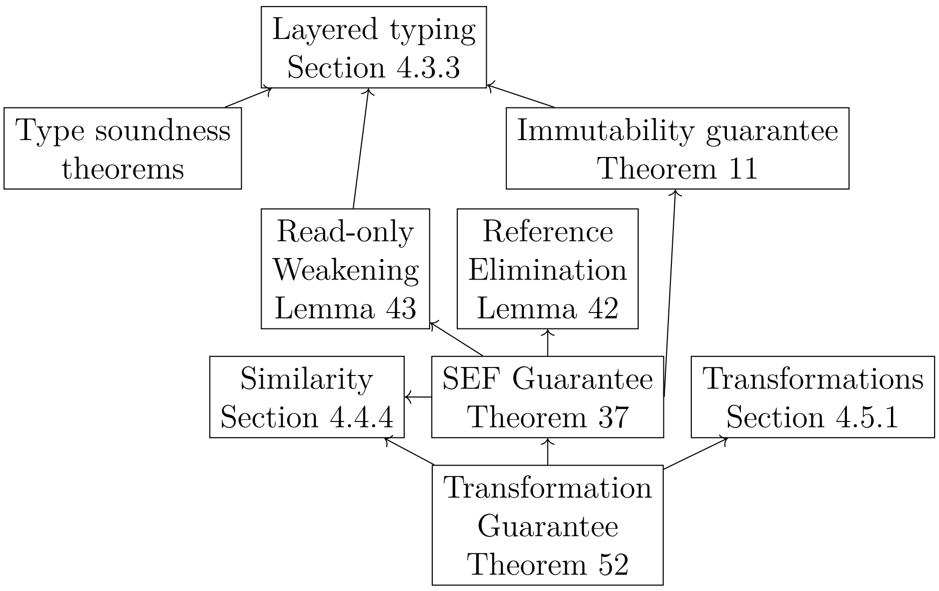

Figure 4.17: Main theorems and lemmata related to Chapter 4 and their dependencies.

As the the final form of the transformation guarantee, we unfold the definitions used in Lemma 49, and specialize the theorem to initial programs:

Theorem 52 (Transformation guarantee)**.**

If $\langle \vdash t_1 : T\rangle \rightarrow_{\mathit{lift\ csw}} \langle \vdash t_2 : T\rangle$	For two programs that differ by swapping call to SEF methods
and $\langle t_1; \cdot; \cdot; \cdot\rangle \longrightarrow^k c_3$,	
where **answer** c_3 *and* $\Gamma_3 \vdash c_3 : T$,	if one of the programs produces an answer,
then there exists c_4, Γ_4 *and* j	
such that $\langle t_2; \cdot; \cdot; \cdot\rangle \longrightarrow^j c_4$,	then the other program also produces
answer c_4, $\Gamma_4 \vdash c_4 : T$	an answer of the same type
$\langle \Gamma_3 \vdash c_3 : T\rangle \rightarrow_{(\mathit{cfg\ csw} \cup \approx)^*} \langle \Gamma_4 \vdash c_4 : T\rangle$.	and the answer only differs in variable names and swapped method calls (in methods of objects on the heap).

4.6 Mechanization

We added the guarantees defined in this chapter to the mechanization of roDOT (Attachment A.1). The main theorems and lemmata relevant to this chapter are summarized in Figure 4.17. We also incorporated the changes described in Section 4.3.2. The mechanization was done in collaboration with Yufeng Li from University of Waterloo.

4.6.1 Typing Modes

As described in Section 4.3.2, we changed a few typing rules in roDOT. The changes work together to achieve the immutability guarantee, but the individual changes are not necessarily dependent on each other. In order to keep the possibility to experiment with various features of the type system, we did not simply update the definitions, but we use the typing mode mechanism introduced in Section 3.6. This allows us to use a single codebase and share most of the definitions and proofs between both variants. Using the `typing_mode` parameter, which is added to all the definitions and theorems related to typing, it is possible to switch between the original and updated definitions and proofs.

We use additional modes to use the variant rules shown in Figure 4.12.

- `rodot` is the original roDOT as described in Chapter 3
- `rodot_top_nm` is roDOT extended with ST-NM.
- `rodot_sef` is roDOT extended as in Figure 4.1.
- `rodot_nm_compl` is extended with ST-NMC-Top and ST-NMC-Bot from Figure 4.12.
- `rodot_nm_muttop` is extended with ST-M-Top from Figure 4.12.
- `rodot_nm_compl_muttop` is extended with all from Figure 4.12.

4.6.2 Mapping of Definitions and Theorems

The following table shows which definition or lemma in the mechanization represents each lemma and definition in this thesis.

Definition 1	Notation `is_ro_type` in file `Mutability/SefGuarantee.v`
Lemma 22	Lemma `reference_nocap_type` in file `CanonicalForms/ReferenceTypes.v`
Definition 2	Used unfolded in the premises of Theorem 37 and Definition 21.
Figure 4.1	Rules `ty_incap_nocap` and `subtyp_nocap_mut_top_boundless` and `subtyp_met` and `ty_trm_call` in file `GeneralTyping/GeneralTyping.v`
Lemma 23	Rule `ty_incap_nocap`, Lemma `ty_incap_diag_nocap_sup` in file `GeneralTyping/GeneralTyping.v`
Lemma 24	Lemma `invertible_flat_typing_closure_tight` in file `CanonicalForms/FlatInvertibleTyping.v`
Lemma 25	Lemma `invertible_flat_typing_or_inv` in file `CanonicalForms/FlatInvertibleTyping.v`
Figure 4.3	Inductive definitions `orctx` in file `Syntax/Orctx.v` and `logctx` in file `Syntax/Logctx.v`
Figure 4.4	Inductive definition `ty_inv_atomic` in file `CanonicalForms/LayeredTyping/LayeredTypingAtomic.v`
Figure 4.5	Inductive definition `ty_inv_basic` in file `CanonicalForms/LayeredTyping/LayeredTypingBasic.v`
Figure 4.6	Definition `ty_inv_basic_closure` in file `CanonicalForms/LayeredTyping/LayeredTypingBasicClosure.v`
Figure 4.7	Inductive definition `ty_inv_logic` in file `CanonicalForms/LayeredTyping/LayeredTypingLogic.v`
Figure 4.8	Inductive definition `ty_inv_main` in file `CanonicalForms/LayeredTyping/LayeredTypingMain.v`
Figure 4.9	Inductive definition `subtyp_atomic` in file `CanonicalForms/LayeredTyping/LayeredSubtypingAtomic.v`
Figure 4.10	Inductive definition `ty_has_N_atomic`, `ty_has_M` in file `CanonicalForms/LayeredTyping/LayeredCapabilityBasic.v`
Figure 4.11	Definition `ty_has_N_closure`, `ty_has_M_closure` in file `CanonicalForms/LayeredTyping/LayeredCapabilityBasicClosure.v`

Lemma 26	Lemma `invertible_main_to_precise_typ_dec` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Lemma 27	Lemma `invertible_main_to_precise_fld_dec` in file `CanonicalForms/InvertibleTyping.v`
Lemma 28	Lemma `invertible_main_to_precise_met_dec` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Lemma 29	Lemma `tight_to_invertible_main` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Lemma 30	Lemma `invertible_main_typing_closure_tight` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Lemma 31	Lemma `ty_inv_main_lc_and` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Lemma 32	Lemma `ty_inv_main_lc_and_invl`, `ty_inv_main_lc_and_invr` in file `CanonicalForms/LayeredTyping/LayeredTypingMain.v`
Lemma 33	Lemma `ty_inv_main_lc_or_diag` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Lemma 34	Lemma `ty_inv_main_lc_nocap_main_replace` in file `CanonicalForms/LayeredTyping/LayeredTypingMain.v`
Lemma 35	Lemma `ty_inv_main_tight` in file `CanonicalForms/LayeredTyping/LayeredTypingMainCount.v`
Lemma 36	Lemma `ty_inv_main_lc_N_cases` in file `CanonicalForms/LayeredTyping/LayeredTypingMainHard.v`
Figure 4.12	Rules `subtyp_nocap_mut_top_complement` and `subtyp_nocap_mut_bot` and `subtyp_mut_top` in file `GeneralTyping/GeneralTyping.v`
Definition 7	Used unfolded in the conclusion of Theorem 37.
Definition 9	Notation `subheap_unchanged` in file `Syntax/Heap.v`
Theorem 37	Theorem `SefG_I` in file `Mutability/SefGuarantee.v`
Definition 10	We use a pair of lists of variables (`list var`).
Definition 11	Definitions `similar_trm`, `similar_avar`, `similar_item`, `similar_stack` in file `Similarity/SyntaxSimilarity.v`
Definition 12	Definition `similar_conf_sef` in file `Similarity/ConfigSefSimilarity.v`
Lemma 38	Lemma `conf_sim_red_append` in file `Similarity/ConfigSefSimilarity.v`
Lemma 39	Lemma `conf_sim_red_create` in file `Similarity/ConfigSefSimilarity.v`
Definition 13	Definition `heap_item_flds_sim` in file `Mutability/SefGuarantee.v`
Lemma 40	Lemma `red_heap_flds_sim` in file `Mutability/SefGuarantee.v`
Lemma 41	Lemma `imm_transport` in file `Mutability/SefGuarantee.v`
Lemma 42	Lemma `sef_neq_elim_ty_hole`, `sef_neq_elim_heap_corr` in file `Mutability/SefGuarantee.v`
Lemma 43	Lemma `sef_neq_ro_ty_config` in file `Mutability/SefGuarantee.v`
Lemma 44	Lemma `ref_elim_similar_conf_sef` in file `Similarity/ConfigSefSimilarity.v`
Definition 14	Definition `Transform` in file `Transformation/Transformation.v`
Definition 15	Definition `TransformCondTypeIdentical` in file `Transformation/Transformation.v` applied using `TransformEnsures`
Definition 16	Definition `TransformCondTyping` in file `Transformation/TransformationTerm.v` applied using `TransformRequires`

Definition 17	Inductive definition `trf_similarity` in file `Transformation/TransformationSimilarity.v`
Figure 4.13	Mutually inductive definitions `trf_local_idtyp_trm` and `trf_local_idtyp_lit` and `trf_local_idtyp_def` and `trf_local_idtyp_defs` in file `Transformation/TransformationLifting.v`
Figure 4.14	Inductive definitions `trf_focus_idtyp_conf`, `trf_lift_idtyp_conf` in file `Transformation/TransformationConfig.v`
Definition 18	Used unfolded in the conclusions of Theorem 48 and Lemma 49
Definition 19	Notation `TransformWeakeningCompat` in file `Transformation/TransformationTerm.v`
Lemma 45	Class instance `WeakeningCompatTransformationTrm` in file `Transformation/TransformationLifting.v`
Lemma 46	Theorem `transformation_preserved` in file `Transformation/TransformationLiftingPreservation.v`
Definition 20	Class definition `TransformConfigReducesToShortIfTerminates` in file `Transformation/TransformationConfig.v` applied to `trf_similarity` in file `Transformation/TransformationSimilarity.v`
Lemma 47	Class instance `TransformConfigReducesTo1Similarity` in file `Transformation/TransformationSimilarity.v`
Theorem 48	Theorem `transformation_preserved_if_terminates` in file `Transformation/TransformationLiftingPreservation.v`
Definition 21	Inductive definition `trf_swap_calls_local` in file `Transformation/TransformationSwapCalls.v`
Lemma 49	Theorem `swap_calls_transformation_preservation` in file `Transformation/TransformationSwapCallsGuarantee.v`
Lemma 50	Class instance `TransformSwapCallsWeakening` in file `Transformation/TransformationSwapCalls.v`
Lemma 51	Lemma `transformation_swap_calls_reduces_to_similarity` in file `Transformation/TransformationSwapCallsPreservation.v`
Theorem 52	Theorem `swap_calls_transformation_guarantee` in file `Transformation/TransformationSwapCallsGuarantee.v`

4.7 Related Work

As far as we know, our work is the first one to consider the issue of purity within a DOT calculus with mutable fields. Other program calculi and programming languages have a diverse approach to purity. We can broadly classify into the following categories:

- **Pure languages** do not allow mutation of objects or the global state, so all functions are pure by default. This approach is typically associated with functional programming, but an object-oriented system can also be pure [12], when the objects are immutable.

  Examples of pure languages are Haskell, Coq. Examples of pure calculi are lambda calculi, or DOT calculi without mutable fields.

- **Pure sub-languages.** Multi-paradigm languages can combine imperative and functional styles in different contexts. For example, C++ has been described as containing multiple sub-languages, such as template meta-programming and the `constexpr` keyword, which do not allow mutation. Notably, these sub-languages are used for **compile-time evaluation**.

- **Purity annotations.** Several programming languages allow programmers to state that a piece of code is pure, as an easy way to enable optimizations or aid analysis. The disadvantage is that such annotations can be incorrectly applied, leading to errors.

- **Purity analysis.** To avoid imposing an annotation burden on the programmer, purity can be inferred by automatic program analysis and side-effect analysis can be used for program optimization.

Additionally, the programming language differ in the properties required for code to be considered pure – what is considered a side effect, and whether determinism and termination is required.

With respect to termination, many languages do not have a way to specify that a method or a function terminates, but allow the opposite – stating that a method does not terminate. In addition to entering an infinite loop, other possible outcomes may include terminating the program, or throwing an exception. In languages with rich type systems, this is achieved by using the bottom type as the return type of the method. Such method cannot return, because it would need to produce a value of the bottom type and the bottom type has no values.

4.7.1 Pure Calculi and Programming Languages

Purity in programming is such an important concept that in many languages, functions are pure by default, thanks to lack of side effects in the semantics and also thanks to referential transparency.

In pure functional languages, effects must typically be explicitly declared in the program using monadic types. This style of programming has been shown to be as powerful as other styles and lies behind practical programming languages.

Lambda calculi Lambda calculi are at the root of theoretical development of functional languages. Although not practical, they inspired practical languages such as Haskell and showed that encoding of many common programming patterns in pure function is possible.

In most lambda calculi, termination is not guaranteed. Simply typed lambda calculus guarantees termination by the strict normalization property.

Object calculi Many theoretical aspects of object calculi do not model object mutation [12], as that is often not relevant, and modeling mutable state complicates the definitions significantly.

A pure object calculus may include a field update operation, which produces a new object value by a shallow copy of the original object except the one update field.

DOT calculi (most of those mentioned in Section 2.3) also usually do not model mutation of objects when it is not relevant to the main feature of the calculus.

Pure functional languages In **pure functional languages**, such as **Haskell**, code is primarily organized into pure functions, which are by construction SEF and deterministic.

Implementing actions that are typically considered side-effects (such as input and output), is achieved with monadic return types.

Haskell functions, however, are not guaranteed to terminate. Because of lazy evaluation, results of non-terminating computations can be manipulated as longs as they do not need to be actually evaluated. An expression that is known to be non-terminating has the the bottom type _|_.

Coq Coq is also a pure functional programming language, though it is not typically used for general programming, It is, however, possible to execute Coq programs and even translate Coq programs to Scala programs that are verifiably correct [20, 21].

Unlike Haskell, Coq requires that all functions terminate. That is because all recursion in Coq has to be guided by an inductive type, so the depth of recursion is limited by a value of finite size.

4.7.2 Imperative Calculi and Languages

Object oriented languages with mutable fields are currently the most popular languages for programming. Many publications [99, 97, 100, 109, 25, 49, 88] focus on Java and languages with similar type systems.

When approached from a practical standpoint, the definition of purity in these languages has to include considerations other than modification of object fields, such as accessing global variables, or synchronization.

This leads to different definitions of purity. The term "pure" is sometimes used to mean the same as "side-effect-free", without requiring determinism.

Java In Java, there are no purity related features in the language and its library, but several tools have been implemented.

Java has been used as a base due to its relatively simple and conservative type system.

ReIm [59] provides both a type system for reference mutability and a way to automatically infer mutability types. It can therefore automatically find pure methods, which have all parameters read-only. We adopted this way of recognizing pure methods by parameter types for roDOT in this work. While in ReIm, mutability is attached to parameter types as a qualifier in the style of the Checker Framework, roDOT uses the special member type M to include the mutability in the parameter type using intersection types. In ReIm, mutability qualifiers are subjected to qualifier polymorphism and viewpoint adaptation. roDOT can express the equivalent of polymorphic qualifiers using dependent types and implements viewpoint adaptation using union and intersection types [43].

To avoid imposing an annotation burden on the programmer, purity can be inferred by automatic program analysis [77, 97], and side-effect analysis can be used for program optimization [36].

Purity is of great use to program verification and specification frameworks, where it enables inserting run-time checks without changing behavior, and allows more precise analysis. JML [11] and Checker Framework [45] allow annotating a method as pure. JML and Checker Framework use simple checks, where pure methods are not allowed to call impure methods. Checker Framework uses the fact that side-effect free methods do not invalidate flow-sensitive types of local variables.

```
// This method is deterministic, because given the same
// state of the heap and the arguments, it returns the same value.
// It is not SEF, because it modifies the containg object.
@Deterministic
int increaseFld(int value) {
    return this.fld += value;
}
// This method is side-effect free, but not deterministic, because
// even called with the exact same arguments,
// it may return a different, newly allocated string each time.
@SideEffectFree
String parenthesized(String a) {
    return "(" + a + ")";
}
// A pure method is both @Deterministic and @SideEffectFree
@Pure
int getValue(int limit) {
    return this.fld > limit ? limit : this.fld;
}
```

The Determinism Checker [75] for Checker Framework checks whether collection operations produce deterministic results, with focus on determinism of iteration order (for example, for hash maps the order of iteration is not specified).

Observational purity [76, 24] is a weaker property that allows side effects as long as they are not observable from certain parts of the code. This definition is based on classes and access control, features which are not modeled in DOT calculi.

Scala For Scala, a type system for purity was developed, but not based on the DOT calculus [98].

This type system only SEF property, and uses method annotations to express which parameters a method can modify.

```
    class A {
        // A mutable field
        private var b: Int
        // This method can only modfiy this object and its owned objects
        def setB(int i) @mod(this) = {
            this.b = i;
        }

        // A field holding an owned object
        @local private var a : StringBuilder = new StringBuilder
        // A method returning an owned object
        def getA() @loc(this) = a
    }
```

```
// This method is SEF (pure), because it only modifies new objects
def m(): A @mod() = {
    val a = new A()
    // Ok — modifies a new object
    a.setB(3)
    // Ok — modifies an object owned by a new object
    a.getA().append("end")

    return a
}
```

C/C++ The **GCC** provides attributes that can be used to allow the compiler to apply optimizations.

A function annotated with a `pure` attribute does not change the observable state. It allows elimination of successive calls to such method, when no mutation happens between such calls.

```
size_t strlen (const char *s) __attribute__((pure));
```

A stronger attribute, `const`, states that the result does not depend on any state.

```
int abs (int x) __attribute__ ((__const__));
```

In C++, templates and `constexpr` may be considered pure sub-languages, which allow guaranteed compile-time evaluation.

.NET Framework In the .NET framework and languages on top of it such as C#, purity can be specified by a `[Pure]` method attribute, which is part of the framework library. It is used by Code Contracts [48], which requires contracts to be pure, but Code Contracts do not check that this annotation is correctly applied [47, 1].

The following example shows how a pure method can be used in a CodeContracts precondition, which may or may not be evaluated at run-time based on compilation settings.

```
[Pure]
bool IsEven(int index){
    return (i & 1) == 0;
}
void Process(int index){
    Contract.Requires(IsEven(index));
    ...
}
```

Dafny An interesting approach is taken in Dafny [69], where pure functions (pure) and methods (with possible effects) are separate language constructs. Termination is ensured by supplying a termination metric.

4.7.3 Capability and Effect Systems

There are other ways to express the permitted side-effects of functions using types, which have been developed in recent work on formal type systems.

The principle of **capabilities** [85, 84] is to require every operation that can have a side effect to take an extra value, called a capability, as a parameter. Then, if some function or method does not have the capability value corresponding to a particular effect, we can conclude that it does not perform that effect. Capabilities are well suited for coarse-grained effects, such as performing input/output in general or accessing some specific file, where a single capability value can guard a set of related operations. To apply such an approach to reasoning about a fine-grained effect such as writing to a field of a specific object, we would need large numbers of such capability values, one new capability value for each existing object. For each reference passed to a parameter or stored in a field, a corresponding capability would need to be passed or stored, thus multiplying the number of parameters and fields.

Wyvern's effect system [74] expresses possible effects by type members of objects. That is syntactically similar to how roDOT represents mutability, but the meaning of the type members is different. In roDOT, the type member of an object reference defines the bounds on the mutability of the reference, the knowledge about whether a reference may be used for mutation, in the type of that reference. In contrast, in Wyvern, the effect member represents a permission to perform an effect, such as `file.Write`, where the effect can be independent of the object that contains the effect member. Thus, Wyvern effect members are more similar to the capability-based approach.

Another successful direction is to use types to express sets of possible variables captured or aliased by values in the program. **Capture Types** [28] follow from a capability based approach, and enable reasoning about where capability values may be stored in the heap or captured in closures, in order to more precisely reason about where effects may occur. **Reachability Types** [22] annotate the type of an expression with a set of variables, which are values that are possibly reachable from the result of that expression. This can be used in conjunction with effect qualifiers as in Graph IR [29], where a function type declares a set of variables that can be read or written, describing the possible effects in a fine-grained way. The types can also be extended to support qualifier polymorphism [108]. This work is defined in the context of a higher order functional formalism, whereas roDOT is an object-oriented calculus. Also, both Wyvern and Reachability Types express effects using new constructs added to the type system, while roDOT aims to encode mutability using the existing DOT constructs of dependent types, unions and intersections.

5. Implementation Experience

In this chapter, we describe our experience with experimental implementation a reference mutability type system within the Dotty compiler for Scala.

With the release of the version 3 of the Scala programming language and the Dotty compiler, we explored the idea of implementing mutability checking within the compiler's type checker, using as much of the existing language features as possible.

This prototyping work both helped identify specific problems related to read-only references in Scala, and in a simplified version serves as a limited demonstration of a reference mutability type system in Scala.

5.1 Background – The Dotty Compiler

The prototyping was done in the Dotty compiler, which is the primary compiler for Scala 3.

One of the goals of the authors of Scala 3 was to eliminate soundness issues that were discovered in the type system of Scala 2, by using the concepts from the DOT calculus at its core, which also gave the name to the new compiler – Dotty.

The Dotty compiler was developed alongside with Scala 3, and was used to prototype many other changes, which were then accepted to the language, such as new syntax for existing features, advanced types (higher-kinded types, match types), or significant re-working of the system of implicit values.

One of the changes to the type system was a more regular treatment of intersection and union types. This led to the idea of using these types to implement advanced features such as read-only references, and made us wonder if we could implement a reference mutability system in Dotty.

Support for Explicit Nullable Types

As an example of another type system extension that was prototyped in Dotty at the same time a demonstrated the possibility to express advanced type features using Scala types, is expressing explicitly nullable references with union types [81].

Normally, any reference type includes the `null` value – the `Null` type is a subtype of all reference types. With explicit nullable types enabled, `Null` is detached from the rest of the reference types. With that change, `String` is not nullable, and the corresponding nullable type has to be `String | Null`.

Other languages achieved similar results using special type features (nullable reference types in C#) or external checkers (nullable annotations in Java).

This is now an experimental feature included in the mainline Dotty compiler.

Phases of the Dotty Compiler

In order to implement reference mutability feature, we need to understand the internal workings of the compiler and decide the proper places where to insert the necessary checks, add supporting types and symbols, process additional information provided in the form of annotations, or modify the type system, so that we can use the existing infrastructure in the compiler.

The Dotty compiler processes code in several phases, starting from the source code and resulting in a class file for the JVM. In the following text, we briefly describe selected important phases.

The **Frontend** phase comprises both a parser and a typer.

The **Parser** processes the text input into an internal representation in the form of a abstract syntax tree. The trees are represented by a immutable structure of object from a class hierarchy, and are carried over to the subsequent phases, which can analyze, transform or annotate the tree.

The **Typer** takes the trees, assigns a type to each node and check type correctness. The types are either taken from the source or inferred.

The processing of classes and methods is lazy – the element is type checked only at the moment when it's type is required. This allows heavily relying on type inference while allowing arbitrary ordering of definitions and circular references.

The **RefChecks** phase checks whether overloading and overriding is used correctly.

The **Pickler** serializes typed trees into a TASTY format, which can be embedded within a `.class` file, retaining all type information, which then can be read back by the compiler during separate compilation.

The **Erasure** phase erases most type information, and reduces it to a simple form that can be understood by the JVM. Finally a **Bytecode Generation** generates executable instructions for the JVM.

The phases in Dotty do not execute in a linear order. Rather, each symbol in the program is compiled lazily, as demanded by other symbols referencing it.

Representation of Scala 3 types in Dotty

In order to properly integrate the new special types into Scala's type system, we need to understand the type hierarchy in Scala 3 and the internal representation of programs and types in the compiler.

The program is represented in Dotty by a typed abstract syntax tree. This tree is passed through the phases, and each phase can analyze or modify the tree,

In the typer phase, a type is assigned to each definition, identifier and expression in the tree.

As seen from the language, the types in Scala 3 form a lattice with the special class `scala.Any` at the top and `scala.Nothing` at the bottom. The classes are organized into two groups, based on reference or value semantics, according to the hierarchy shown in Figure 5.1.

Because of the possibility to use dependent and singleton types in Scala, the internal representation of types is more complicated than in Java or C#. For example, for a variable declared as `val s:String = "x"`, the most precise type of `s` is not `String`, but `x.type`. The type of the string literal `"x"` is also `"x"`.

The types are internally organized into a hierarchy shown in Figure 5.2.

The ground types correspond to DOT types – declarations (class or method), intersections and unions. `AndType(A,B)` and `OrType(A,B)` represent intersections and unions, `ClassInfo` represents the definition of a class, and `MethodOrPoly` represents parameterized types – methods and generics.

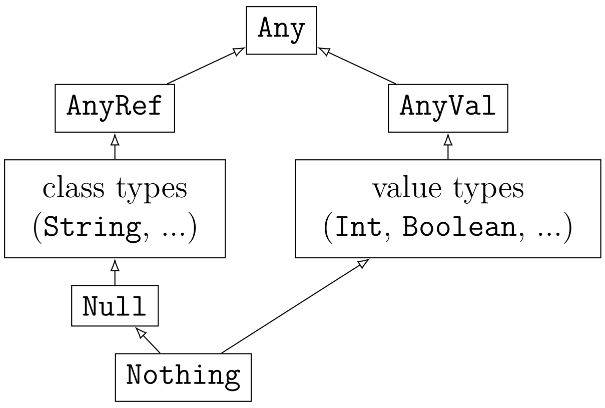

Figure 5.1: Class hierarchy in Scala.

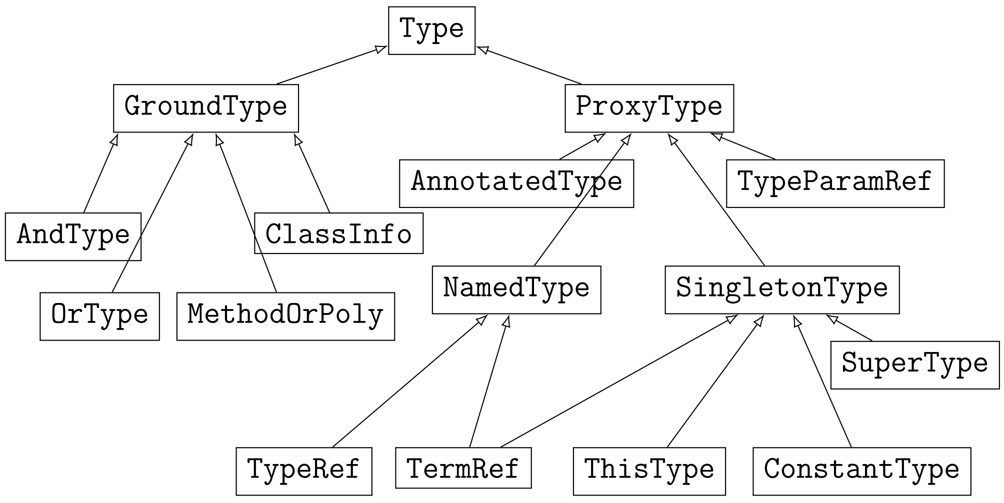

Figure 5.2: Hierarchy of internal representation of Scala types in Dotty (only selected types shown).

Proxy types are such types that have an *underlying type*, to which they add more information. For example, if a variable is declared as `val s:"x" = "x"`, then the underlying type of the singleton type `s.type` is the constant type `"x"` specified in the declaration of `s`. This constant type has the type `scala.String` An example chain of proxies and underlying types is shown in Figure 5.3.

`TermRef(name, owner)` is a singleton type of variables, method parameters and object fields, the underlying type is the declared or inferred type of the variable. It is also used to refer to packages, which are viewed as objects containing class definitions.

`TypeRef(name, owner)` is used to refer to classes by their name. The underlying type is the declaration of the class.

`ThisType(Cls)` is the type of `this` or `Cls.this` within a definition of a class `Cls`.

5.2 Explicit Mutable References

The idea of expressing explicitly mutable references is analogous to the explicit nullable references. While with nullable references, the non-nullable type is weak-

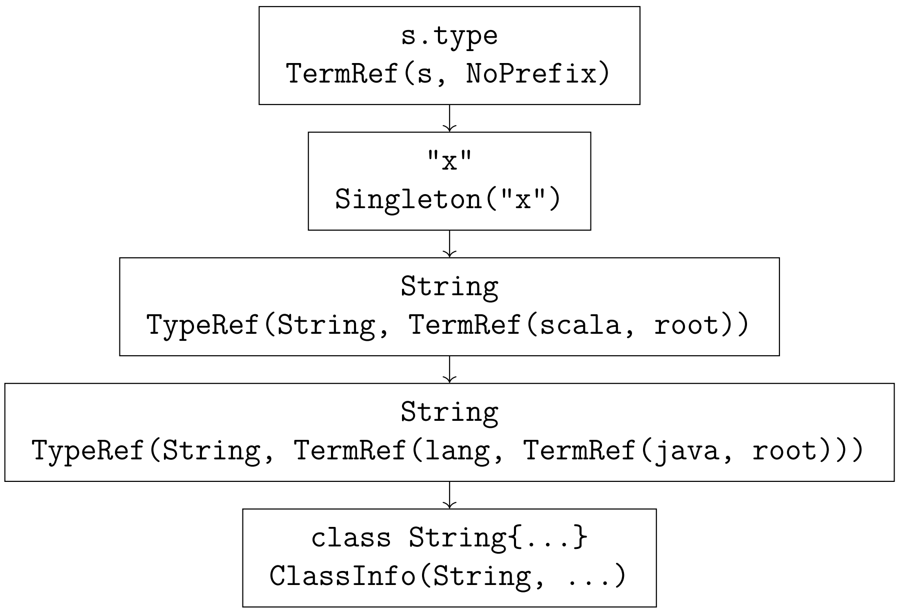

Figure 5.3: Underlying types of variable `val s:"x" = "x"` in Dotty.

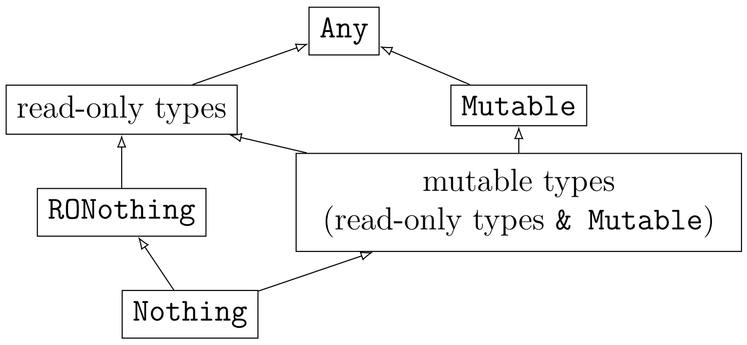

Figure 5.4: Scala type hierarchy with added mutability.

ened by a union with the Null type, a read-only type could be made mutable – strengthened – by an intersection with a mutable trait.

This leads to the basic idea that for a class `C`, the type `C` is read-only and `C & Mutable` is mutable. `Mutable` is a special trait defined in the compiler. Figure 5.4 shows the subtyping relationship between read-only and mutable types formed in this way, analogous to Figure 3.3 in roDOT.

When an field assignment is type-checked by the typer, it is checked that the receiver is of type `Mutable`.

This design allows distinguishing mutable and read-only parameters of a method:

```
// Method which mutates the argument
def mutating(arg: C & Mut): Unit = {
    arg.fld = 1
}
// Method not allowed to mutate the argument
def reading(arg: C): Int = {
    arg.fld
    // arg.fld = 1  would be a type error
    //   (cannot mutate a read-only reference)
    // mutating(arg) would be a type mismatch
    //   (incompatible argument type)
}
// A mutable argument passed through
def mutpass(arg: C & Mut): C & Mut = arg
// A read-only argument passed through
def ropass(arg: C): C = arg
```

We also attempted a system, where references would be by default mutable, with the idea that under such system, all existing code should be able to compile without changes. This could be achieved by making the syntax tree of a type reference such as `C` refer to the type `C & Mutable`, and this operation could be reversed by applying a `@ReadOnly` annotation. There were, however, significant drawback to this approach:

- In a few context, a type must refer to a simple class reference. An example is the `extends` clause of classes, where any type can be written, but must resolve to a class that will be used as a base for the class being defined. Changing the interpretation of type syntax requires stripping the mutability to get back to the basic class type.
- Later phases of the compiler, such as the TASTY pickler, also assumed that syntax trees which represent a class will be typed with a TypeRef to that class. Breaking such unwritten assumptions leads to internal compiler errors, such as a crash with a run-time type error, trying to cast the type to a TypeRef. When accessing the typed abstract syntax tree, the Dotty compiler often uses pattern matching, which is prone to fail if the structure matched on does not follow the expected patterns.
- Making types be mutable by default means that even immutable types such as `String` or `Int` would end up in the mutable intersections `String & Mutable`. While in theory this is not a problem, it is also not any useful, clutters the internal representation, and necessitates dealing with the `Mutable` trait in many places that are not relevant to mutability. For example string literals would have to have type `String & Mutable` in order for `var x: String = "s"` to type-check.

Type of `this`

A special consideration must be given to the type of the `this` keyword, which in the scope of a class refers the containing object, being the analogue of the self-variable s and the receiver parameter r in roDOT.

A method of a class can be allowed or disallowed to mutate the containing object, which means that it must be possible to change the type of `this` to be either mutable or read-only within that method.

Unlike Java, Scala does not allow explicitly writing the `this` parameter, so the mutability has to be written in a different way. We used the `@ReadOnly` annotation on the method.

This corresponds to the type of `this` in class `C` having type `C` or `C & Mutable` within the method. However, the type of `this` is a `ThisType` with `TypeRef(C)` as its underlying type. This lead the underlying type of `ThisType` changing based on the context it appears in.

```
class C{
    var fld: Int
    // Method which mutates the receiver
    @Mutating
    def mutating(): Unit = {
        fld = 1
    }
    // Method not allowed to mutate the receiver
    @ReadOnly
    def reading: Int = {
        fld
        // fld = 1  would be a type error
        //   (cannot mutate a read—only reference)
        // mutating() would be a type mismatch
        //   (cannot call a mutating method
        //   on a read—only reference)
    }
}
```

Overloading and Overriding Methods

In Scala, method calls are resolved in the typer phase using the full available typing information, including mutability. However, when the code is compiled into JVM bytecode, for compatibility with Java code, the Scala types get erased into Java types, and the overload resolution at run-time follows the Java rules.

In the erasure phase, intersection types are erased to either the more precise component, if the two components are in a subtype relationship, otherwise to the first component.

Therefore, it is not possible to have a method overloaded with variants differing in just mutability.

Polymorphism

In order to preserve mutability information throughout the program, it is important to express mutability polymorphism – the ability of a method to be typed both with read-only and mutable types.

A form of this can be directly achieved in Scala using a singleton type referring to a parameter.

```
def methodPoly1(par: C): par.type = par
```

In such case, the method's return type is a reference to its argument. A call `methodPoly1(arg)` would have the same type as `arg`, including mutability. It is, however necessary, that the method always returns its argument and not any other value of the same type.

A more common scenario is when the return type is the same class as the argument, but not necessarily the same value. In such case, we would want to express that the result has the same mutability as the argument. This leads to the idea of using a special type member, which can be selected in order to refer to the mutability of the parameter.

```
def methodPoly2(par: C): C & par.Mut = ...
```

Mutabilities of multiple arguments can be combined using union types, the same way as with viewpoint adaptation:

```
def methodPoly2(par1: C, par2: C):
    C & (par1.Mut | par2.Mut) = ...
```

Another possibility is to express the mutability explicitly as a type argument.

```
def method3[AM >: Mutable](arg: C & AM): C & AM
```

While this works, we observed that applying this pattern leads to unbearable syntactic overhead. The situation is even worse in the presence of higher-kinded types (type lambdas).

In an attempt to reduce this overhead, we implemented annotations, which would transform the code by generating the mutability type parameters. The equivalent of the code above would be:

```
def method3(arg: C @Tag("AM")): C @Tag("AM")
```

The problem of this solution was that the type parameters did not have syntactic representation in the code, which breaks the way type parameters are handled, such as when typing calls to the methods, and fixing this would required adjusting many places in the compiler.

Inference

Scala programs can rely heavily on type inference. While in Java and C# have type inference for local variables and lambda expressions, method parameters and return types must always be specified. In Scala, the return type can also be inferred.

Type inference reduces Annotation overhead when the type is obvious or hard to type. Because mutability integrated into the types, when a type is not specified, it will be inferred including its mutability. However, if the type of a method is explicitly stated, then the mutability also must be included.

External Code

Scala code typically relies on classes from the Scala library, Java library, and other libraries, which were not compiled with explicit mutability support. In order to maintain soundness, such code would have to be annotated to specify mutability of the parameters and return types.

5.3 Demonstration Implementation in Dotty

Attached to this thesis as Attachment A.2 is a simplified demonstration implementation. The implementation showcases the following features:

- The special type member `MUT`, representing reference mutability.

- Checking reference mutability at field assignment sites.
- Viewpoint-adaptation of the types of field at field access sites.
- Setting the mutability of the receiver in methods using the `@Mutating` annotation.
- Adjustment of the type of `this` in methods to the mutability of the receiver.
- Checking the mutability of the receiver at call sites of `@Mutating` methods.

The following snippets are part of the demonstration code, which can be processed by the provided implementation. How to access the demonstration code is described in Attachment A.2.

The implementation represents roDOT's mutability declaration $\{\mathsf{M}(r) : \bot..\bot\}$ with a refinement type `Any {type MUT <: Nothing}`. In order to simplify the syntax, we use a type alias `Mutable` defined as follows:

```
type Mutable = Any {type MUT <: Nothing}
```

5.3.1 Examples

The following pieces of code shows the basic checking of reference mutability and convertibility of reference types.

```
class A{
    var fld: Int // Mutable field
}
// This method takes a read–only reference to A
def m_ro(a: A) = {
    a.fld = 1 // error, a is not mutable
}
// This method takes a mutable reference to A
def m_mut(a: A & Mutable) = {
    a.fld = 1 // ok, a is mutable
    m_ro(a) // ok, mutable is convertible to read–only
}
// This method takes a read–only reference to A
def m_ro2(a: A) = {
    m_mut(a) // error, type mismatch
}
```

The following class demonstrates how mutating and read-only methods are defined, and how mutation is prevented by the type of `this` being either a mutable or a read-only reference.

```
class B{
    var i: Int

    // This method can access the containing object as read–only
    def m_ro_this() = {
        i = 1 // error
    }
    // This method can access the containing object as mutable
    @Mutating
    def m_mut_this() = {
        i = 1 // ok
        m_ro_this(); // ok, calling a read-only method
    }
    // This method can access the containing object as read-only
    def m_ro_this2() = {
        m_mut_this(); // error, calling a mutating method
                      // on a read-only receiver
    }
}
```

The next example shows how viewpoint adaptation changes the type of a field access expression, so that the read-only-ness of a reference applies transitively.

```
class C{
    // Mutable reference field
    var fld_mu: A & Mutable
    // Read-only reference field
    var fld_ro: A

    // This method can access the containing object as read-only
    def m_ro_this() = {
        fld_mu.fld = 1 // Error, type of fld_mu is view-point adapted
                       // to read-only
        fld_ro.fld = 1 // Error, read-only reference
    }
    // This method can access the containing object as mutable
    @Mutating
    def m_mut_this() = {
        fld_mu.fld = 1 // ok
        fld_ro.fld = 1 // Error, read-only reference
    }
}
```

5.3.2 Overview of Changes

Implementing the features described above entailed the following changes to the compiler:

- The special type member `MUT`, the special type `RONothing` and the annotation `@Mutating` are predefined in the compiler.
- The special type `RONothing` is added to the hierarchy as a subtype of all classes except `RONothing` and refinement types defining members other than `MUT`.
- The underlying type of `TermRef` is viewpoint-adapted to combine the mutability of the prefix and the field type.

- The underlying type of `ThisType` is changed to mutable in the context of methods annotated with `@Mutating`.
- The Typer checks that a method receiver is mutable when the methods is annotated with `@Mutating`.
- The Typer checks that the receiver of field assignment has a mutable type.
- The Typer assigns a mutable type to invocations of a class constructor.
- In the Typer phase, it is checked that classes do not override the special `MUT` type member.
- In the RefChecks phase, it is checked that methods annotated with the `@Mutating` annotation cannot override methods that do not have such annotation.

5.4 Obstacles Encountered

We started the implementation with the expectation that Although we did not achieve full implementation of reference mutability in Dotty, we identified several key ares that would need to improve in order to fully implement this feature.

- Formalization. The need to formally specify and validate type system design. This led us to the development of roDOT.
- Receiver variables. Reference mutability heavily relies on the ability to specify and refer to the mutability of the receiver. This is possible in Java, but not in Scala. Adding this feature to Scala would not only improve reference mutability, but also allow other uses such as those implemented for Java in the Checker Framework.
- Specify internal interfaces. The Dotty compiler is a complex software which implements many language features with different design goals, and processing input from different sources, which leads to complex dependencies between code in various stages of the compiler. It would help development of language extensions if the phases of the compiler had clearly defined interfaces with explicitly stated constraints on the input and produced structures.
- Type system regularity: Despite Scala being already influenced by formal designs such as DOT, Scala still contains several irregular features such as functions with variable arguments, which are internally represented as a special type that can only be used in specific places.
- Separation of syntax, symbols and types. Although it is possible to modify the typer to give syntax trees and symbols different type, in the internal data structures of Dotty, the trees, symbols and types are tied together. It is for example not easy to add a new symbol defined by the syntax.

5.5 Case Study: Scala Collections

In order to gain insight into the impact of reference mutability is Scala, we started a cases study on the collection library.

The Scala library comes with an extensive set of collections, including mutable and immutable collections, and a detailed hierarchy of traits which allow writing collection-handling code which is not dependent on a concrete implementation. Note that an immutable collection are immutable in the shallow sense and their elements can be potentially mutated.

Near the top of the collection hierarchy is the `SeqOps` trait, which provides several methods that demonstrate different behaviors with respect to mutability.

The SeqOps trait The trait has three type parameters: the type of elements `A`, the collection type `C`, and the collection constructor `CC`.

```
trait SeqOps[+A, +CC, +C] { ... }
```

As `SeqOps` is a parent to both mutable and immutable collections, it does not provide operations that would mutate the collection object itself. Therefore, all these methods can be defined with a read-only receiver type. What is of concern is the mutability of the elements, and ensuring transitivity of read-only references.

We categorized the methods of `SeqOps` according to their behavior with respect to mutability.

In the following code excerpts, we show the signatures of methods with added mutability annotations. These annotations, which are highlighted by shading, express what the mutability of a type within the methods signature should be. They are, with the exception of making a mutable version of a type, outside of the abilities of the provided demonstration implementation.

- The type `T & Mutable` represents a mutable version of type `T`.
- The type `ro |> T` represents a read-only version of type `T`.
- The type `this |> T` represents the type `T` viewpoint-adapted to the receiver `this`.
- The type `this |» T` represents a higher-kinded type `T` viewpoint-adapted to the receiver `this`.

Read-Only Methods The simplest case are methods that do not access elements at all:

```
def length: Int
def nonEmpty: Boolean
def knownSize: Int
def sizeCompare(otherSize: Int): Int
```

Methods that access elements in read-only manner, such as by calling `equals` or `toString`, and do not return any object, can be handled similarly.

Here we assume that `toString` is declared as read-only.

```
def addString(b: mutable.StringBuilder  & Mutable ):
  mutable.StringBuilder  & Mutable
```

Element Access Methods In the case of methods that return an element of the collection, the result type must be viewpoint-adapted to ensure that a read-only reference to an element is returned when the collection is read-only, even though the type of elements `A` might be mutable.

```
// Returns the first element
def head: this |> A
// Returns the last element
def last: this |> A
// Returns the element at the given position
def apply(i: Int): this |> A
```

When the returned element is wrapped in another object, such as `Option`, the element type is viewpoint-adapted in the same way. It is, however also beneficial to viewpoint adapt the wrapper object, because on that will allow an implementation where the wrapper object is permanently stored in a field of the collection.

```
// Optionally returns the first element
def headOption: this |> Option[this |> A]
// Optionally returns the last element
def lastOption: this |> Option[this |> A]
```

Subsequences and Permutations Methods for filtering elements, which return the same type of collection. The resulting collection has the same type as the receiver. However, the collection type `C` is defined at the top level of the trait and therefore does not have access to the receiver reference. This is equivalent to the difference between s and r in roDOT methods.

Therefore, the type of elements of `C` must be viewpoint-adapted by `this`. We denote this operation by an operator `this |»`, to highlight a difference form the viewpoint-adaptation operator `this |>`. A possible implementation of this operator would be to modify the definition of `C` to be parameterized by a type that viewpoint-adapts the element type, then `this |»` would designate `C[this]`.

```
// Elements of the collection without duplicates
def distinct: this |» C
// Elements of the collection in reverse order
def reverse: this |» C
// Elements of the collection until the given index
def take(n: Int): this |» C
// Elements of the collection starting from the given index
def drop(n: Int): this |» C
def takeRight(n: Int): this |» C
def dropRight(n: Int): this |» C
def slice(from: Int, until: Int): this |» C

def combinations(n: Int):  this |> Iterator[this |» C]
def sliding(size: Int, step: Int): this |> Iterator[this |» C]
```

Predicates Methods, which take a function objects that is applied to elements of the collection a and return a collection containing some of the elements.

The predicates are typically assumed to not modify the elements.

```
def exists(p: ( ro |> A) => Boolean): Boolean

def filter(pred: ( ro |> A) => Boolean): this |» C
def sortWith(lt: ( ro |> A, ro |>  A) => Boolean): this |» C
def takeWhile(p: ( ro |> A) => Boolean): this |»C
def span(p: ( ro |> A) => Boolean): ( this |» C, this |» C)
def partition(p: ( ro |> A) => Boolean): ( this |» C, this |» C)

def find(p: ( ro |> A) => Boolean): Option[ ro |> A]
def findLast(p: ( ro |> A) => Boolean): Option[ ro |> A]
```

Collection type conversion Conversion to a different collection type. If it was guaranteed to return a new object, then the result could be declared mutable. However, as a memory efficient implementation, the method can return `this` if the actual collection type is compatible with the desired collection type. Therefore the resulting, collection must be of the same mutability – viewpoint adapted by the receiver. The element type is also viewpoint adapted in the same way as in other methods that access elements of the collection.

```
def toSeq: this |> Seq[ this |> A]
def view: this |> SeqView[ this |> A]
def toList: this |> immutable.List[ this |> A]
def toSeq: this |> immutable.Seq[ this |> A]
def reverseIterator: this |>  Iterator[ this |> A]
def iterator: this |> Iterator[ this |> A]
def toIterable: this |> Iterable[ this |> A]
```

Comparison Methods that compare elements of the collection with supplied objects. We can assume that there is no mutation involved and the argument has a read-only type.

```
def indexOf[B >: ro |> A](elem: B): Int
def contains[A1 >: ro |> A](elem: A1): Boolean
def diff[B >: ro |> A](that: Seq[B]): this |» C
```

Element transformation Methods that transform elements into a different type, which can be either mutable or read-only without restriction.

```
def map[B](f: ( ro |> A) => B): this |> CC[B]
def flatMap[B](f: ( ro |> A) => IterableOnce[B]): this |> CC[B]
def flatten[B](implicit asIterable:
  ( ro |> A) => IterableOnce[B]): this |> CC[B]
```

Adding elements Methods which add elements to the collection (return a larger collection). In order to allow efficient implementation by returning one of the collections if the other one is empty, the result is viewpoint-adapted not only by the receiver, but also by the other collection.

```
def concat[B >: this |> A](suffix: IterableOnce[B]):
  suffix |> this |> CC[B]
```

Mutable collections The collection library provides immutable and mutable versions of collections. The mutable versions have the same non-mutating operations, as the immutable ones, where changed collections are by copying the content of the original collection. All the properties of these methods listed above equally apply to both immutable and mutable versions of the collections.

Additionally, mutable collections provide mutating operations, which change the contents of the collection in-place. In this case, the receiver reference needs to be checked to be mutable. These methods would not be declared `ReadOnly`.

Because the type of `this` in these methods is mutable, there is no need to viewpoint-adapt the types involved.

Some of the mutating methods return `this` to allow call chaining. In that case, the return type is `this.type`, which in a mutating method is a mutable type.

The following code shows the difference between a mutating and a read-only method from the mutable collection `ArrayBuffer`:

```
@Mutating
def append(elem: A): Buffer.this.type
@ReadOnly
def appended[B >: ro |> A](elem: B): ro |> CC[B]
```

Summary

In summary, this case study showed that for annotating Scala collections with reference mutability, the far most common pattern is a read-only method which has the return type viewpoint-adapted by the receiver.

In some cases, this viewpoint-adaptation is necessary to express the possibility of an efficient implementation returning the receiver. This shows complexity emerging from the field-based reference mutability with transitivity and the combination of mutable and immutable collections under one API.

The use of the type parameter `C` poses a complication for annotating methods which need to return a version of the collection type with viewpoint-adapted element type.

6. Conclusion

Our work has demonstrated that the type system features of Scala, formalized in the DOT calculus, can be used to encode reference mutability, with a small amount of necessary type system extensions. As such, it provides an alternative to existing reference mutability systems, which are based on dedicated type constructs, or capability and effect annotations that are defined separately from the basic type system.

In this thesis, we presented this encoding in the form of the roDOT calculus, which uses a typing judgment to recognize mutable references, and is able to provide a guarantee that objects can only be mutated by using mutable references – the Immutability Guarantee.

With a further extension to roDOT that improves the ability to also recognize read-only types, roDOT can additionally identify methods that are side-effect free (SEF), because they have read-only parameters. We stated this in the roDOT calculus in the form of the SEF Guarantee.

Moreover, this guarantee can be used to justify safe transformations of programs, which can provide a formal background for program optimization or code editing – we prove that in roDOT, changing the order of calls to SEF methods does not change the result of execution of the program.

We mechanized all the definitions and theorems of the roDOT calculus in Coq, which entailed adapting an existing proof of type safety of DOT with mutable fields to use a new approach based on layered type judgments, and designing original proofs for the mutability and SEF guarantees. The safe transformations and the Transformation guarantee are constructed within a general framework for defining transformations of roDOT programs.

Finally, we provide a simplified implementation of roDOT's ideas for the Dotty compiler, and a brief analysis of patterns that would emerge from adopting reference mutability types in the context of the Scala collection library.

6.1 Future Work

We outlined one possible approach to reference mutability in object-oriented languages and calculi. As the treatment of read-only references in programming languages and of type systems continues to evolve, our work could be used as a base for continued development in this direction in several ways, which we briefly envision below.

Implementation We provide only a rudimentary implementation of the ideas of roDOT for the Dotty compiler. This implementation could be extended and enable more practical evaluation of the approach, but would require solving the problems described in Section 5.4.

Encoding Multiple Reference Capabilities In roDOT, we introduced a single mutability type member M, to encode reference mutability. We believe that other reference capabilities could be also encoded in the form of type members.

For example, a similarly formed uniqueness marker $\{\mathsf{U} : \bot..\bot\}$ could designate that a reference is unique (there are no other references pointing to the same object). This would be a step towards implementing a system analogous to that of Pony.

Tracking uniqueness of references is not possible in roDOT, because the type system does not put any restrictions on how many times a field of an object is read, or in how many places a variable is used. In order to make this possible, a calculus would need to be designed, which would integrate roDOT's features with a linear calculus. In such a calculus, reading fields of objects would be destructive – in order to get a value of a field, either a new value needs to be written at the same time, or the field needs to be removed. Each variable could be used only once, unless an explicit construct is used to duplicate the reference, which would remove the uniqueness marker from its type.

Type Bounds In roDOT, the mutability tags have a fixed form $\{\mathsf{M}(r) : \bot..T\}$, where the lower bound is limited to always be $\bot$, and the upper bound determines the mutability. The upper bound is typically either $\top$, $\bot$, a type selection, or a logical combination thereof, and is compared with $\bot$ to check if the reference is mutable.

It could be interesting to see if the bounds could be used to express and check multiple levels of mutability (or other capabilities), by using more complex types to represent those levels in the bounds.

One possible direction could be to encode borrowing and fractional permissions, where a mutable reference x_1 could be split into two references x_2, x_3, which cannot be both mutable at the same time, which would be expressed in its mutability bounds as subtraction of the mutability of the other references: $x_2 : \{\mathsf{M}(r) : \bot..(x_1.\mathsf{M} - x_3.\mathsf{M})\}$,

Recursive Types and Self References One of the points where the DOT calculi and roDOT differ from the Scala language, is the representation of objects and their types by a recursive constructor $\nu(s : R)$ and recursive types $\mu(s : R)$. In these constructs, the variable s allows self-references within the object, by designating the object itself.

This does not closely correspond to how objects and classes are represented in the Scala language, and introduces accidental complexity to the calculus. For example, in roDOT subtyping is not defined for recursive types, which posed hurdles for the definitions and proofs of our layered typing.

More fundamentally, in order to reference mutability to work as intended in roDOT, we had to put restrictions on the usage of $s.\mathsf{M}$ within recursive types. Also, methods require an additional receiver parameter r, which designates the reference that was used to call the method, and is not identifiable with the variable s despite pointing to the same object.

Possibly, a re-imagination of the approach to recursive types in the DOT calculus. where the self-variable s would designate the reference rather than the object itself, could improve this situation, and more closely model Scala. An inspiration for that could be jDOT [58] or the calculi of Abadi and Cardelli [12]

Bibliography

[1] CodeContracts. https://github.com/microsoft/CodeContracts.

[2] The Coq proof assistant. https://coq.inria.fr/. Accessed: 2023-10-24.

[3] const(FAQ) - D Programming Language. https://dlang.org/articles/const-faq.html.

[4] Iris project. https://iris-project.org/.

[5] C# language design. https://github.com/dotnet/csharplang. Accessed: 2022-10-05.

[6] PEP 8000 – Python language governance proposal overview. https://peps.python.org/pep-8000/, . Accessed: 2022-10-05.

[7] Rust RFCs. https://github.com/rust-lang/rfcs, . Accessed: 2022-10-05.

[8] ECMA-334, C# language specification. https://www.ecma-international.org/publications-and-standards/standards/ecma-334/. Accessed: 2022-10-05.

[9] The Checker Framework Manual: Custom pluggable types for Java. https://checkerframework.org/manual/#initialization-checker, 2022. Accessed: 2022-10-05.

[10] The Checker Framework Manual: Custom pluggable types for Java. https://checkerframework.org/manual/#purity-checker, 2022. Accessed: 2022-10-05.

[11] JML reference manual: Class and interface member declarations. https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_7.html#SEC60, 2022. Accessed: 2022-10-12.

[12] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. ISBN 978-0-387-94775-4. doi:10.1007/978-1-4419-8598-9. URL https://doi.org/10.1007/978-1-4419-8598-9.

[13] Andrei Alexandrescu. *The D programming language*. Addison-Wesley, Upper Saddle River, N.J, 2009. ISBN 978-0-321-63536-5.

[14] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. doi:10.1145/3009837.3009866. URL https://doi.org/10.1145/3009837.3009866.

[15] Nada Amin and Ross Tate. Java and Scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 838–848, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344449. doi:10.1145/2983990.2984004. URL `https://doi.org/10.1145/2983990.2984004`.

[16] Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012.

[17] 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*, OOPSLA '14, pages 233–249. ACM, 2014. ISBN 978-1-4503-2585-1. doi:10.1145/2660193.2660216.

[18] 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. URL `https://doi.org/10.1007/978-3-319-30936-1_14`.

[19] Ellen Arvidsson, Elias Castegren, Sylvan Clebsch, Sophia Drossopoulou, James Noble, Matthew J. Parkinson, and Tobias Wrigstad. Reference capabilities for flexible memory management. *Proc. ACM Program. Lang.*, 7 (OOPSLA2), oct 2023. doi:10.1145/3622846. URL `https://doi.org/10.1145/3622846`.

[20] Youssef El Bakouny and Dani Mezher. Scallina: Translating verified programs from Coq to Scala. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2018. doi:10.1007/978-3-030-02768-1_7. URL `https://doi.org/10.1007/978-3-030-02768-1_7`.

[21] Youssef El Bakouny, Tristan Crolard, and Dani Mezher. A Coq-based synthesis of Scala programs which are correct-by-construction. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona, Spain, June 20, 2017*, pages 4:1–4:2. ACM, 2017. doi:10.1145/3103111.3104041. URL `https://doi.org/10.1145/3103111.3104041`.

[22] Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: Tracking aliasing and separa-

tion in higher-order functional programs. *Proc. ACM Program. Lang.*, 5 (OOPSLA), oct 2021. doi:10.1145/3485516. URL `https://doi.org/10.1145/3485516`.

[23] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985. ISBN 978-0-444-86748-3.

[24] Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*, 2004.

[25] William C. Benton and Charles N. Fischer. Mostly-functional behavior in Java programs. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2009. doi:10.1007/978-3-540-93900-9_7. URL `https://doi.org/10.1007/978-3-540-93900-9_7`.

[26] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.

[27] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard E. Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 2014. doi:10.1007/978-3-662-44202-9_11. URL `https://doi.org/10.1007/978-3-662-44202-9_11`.

[28] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. Capturing types. *ACM Trans. Program. Lang. Syst.*, 45(4), nov 2023. ISSN 0164-0925. doi:10.1145/3618003. URL `https://doi.org/10.1145/3618003`.

[29] Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. Graph IRs for impure higher-order languages: Making aggressive optimizations affordable with precise effect dependencies. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023. doi:10.1145/3622813. URL `https://doi.org/10.1145/3622813`.

[30] Luca Cardelli. Bad engineering properties of object-oriented languages. *ACM Comput. Surv.*, 28(4es):150, 1996. doi:10.1145/242224.242415. URL `https://doi.org/10.1145/242224.242415`.

[31] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994. doi:10.1006/INCO.1994.1013. URL `https://doi.org/10.1006/inco.1994.1013`.

[32] David Cassel. Why are so many developers hating on object-oriented programming? https://thenewstack.io/why-are-so-many-developers-hating-on-object-oriented-programming/. Accessed: 2024-05-20.

[33] Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi:10.1007/S10817-011-9225-2. URL https://doi.org/10.1007/s10817-011-9225-2.

[34] Arthur Charguéraud. TLC: a non-constructive library for Coq. https://www.chargueraud.org/softs/tlc/. Accessed: 2024-05-20.

[35] Alonzo Church. The calculi of lambda-conversion. *Bull. Amer. Math. Soc*, 50:169–172, 1944.

[36] Lars Ræder Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997. doi:10.1002/(SICI)1096-9128(199711)9:11<1031::AID-CPE354>3.0.CO;2-O.

[37] 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. URL https://doi.org/10.1145/2824815.2824816.

[38] 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. URL https://doi.org/10.1109/ICSE.2017.52.

[39] Joshua M. Cohen and Philip Johnson-Freyd. A formalization of Core Why3 in Coq. *Proc. ACM Program. Lang.*, 8(POPL):1789–1818, 2024. doi:10.1145/3632902. URL https://doi.org/10.1145/3632902.

[40] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In Rastislav Kralovic and Pawel Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings*, volume 4162 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006. doi:10.1007/11821069_1. URL https://doi.org/10.1007/11821069_1.

[41] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75

(5):381–392, 1972. ISSN 1385-7258. doi:https://doi.org/10.1016/1385-7258(72)90034-0. URL `https://www.sciencedirect.com/science/article/pii/1385725872900340`.

[42] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. Building and using pluggable type-checkers. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 681–690. ACM, 2011. doi:10.1145/1985793.1985889. URL `https://doi.org/10.1145/1985793.1985889`.

[43] Vlastimil Dort and Ondřej Lhoták. Reference mutability for DOT. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 18:1–18:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ECOOP.2020.18. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2020.18`.

[44] Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek. Pure methods for roDOT. In *ECOOP 2024*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024 (forthcoming).

[45] Michael D. Ernst. Annotation type Pure. `https://checkerframework.org/api/org/checkerframework/dataflow/qual/Pure.html`, 2022. Accessed: 2022-10-05.

[46] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. Locking discipline inference and checking. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1133–1144. ACM, 2016. doi:10.1145/2884781.2884882. URL `https://doi.org/10.1145/2884781.2884882`.

[47] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010. doi:10.1007/978-3-642-18070-5_2. URL `https://doi.org/10.1007/978-3-642-18070-5_2`.

[48] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2103–2110. ACM, 2010. doi:10.1145/1774088.1774531. URL `https://doi.org/10.1145/1774088.1774531`.

[49] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David A. Wagner. Verifiable functional purity in Java. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 161–174. ACM, 2008. ISBN 978-1-59593-810-7. doi:10.1145/1455770.1455793. URL `https://doi.org/10.1145/1455770.1455793`.

[50] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.*, 4(ICFP):114:1–114:29, 2020. doi:10.1145/3408996. URL `https://doi.org/10.1145/3408996`.

[51] J. A. Goguen. Semantics of computation. In Ernest Gene Manes, editor, *Category Theory Applied to Computation and Control*, pages 151–163, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg. ISBN 978-3-540-37426-8.

[52] 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. URL `https://doi.org/10.1145/2384616.2384619`.

[53] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java® language specification, Java SE 8 edition. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.1`, 2022. Accessed: 2022-10-05.

[54] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight Go. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi:10.1145/3428217. URL `https://doi.org/10.1145/3428217`.

[55] 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. URL `https://doi.org/10.4204/EPTCS.246.5`.

[56] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 311–320. ACM, 2000. doi:10.1145/349299.349341. URL `https://doi.org/10.1145/349299.349341`.

[57] 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. URL https://doi.org/10.1145/3371077.

[58] Hu, Zhong Sheng. Decidability and algorithmic analysis of dependent object types (dot). Master's thesis, 2019. URL http://hdl.handle.net/10012/14964.

[59] 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*, OOPSLA '12, pages 879–896. Association for Computing Machinery, 2012. ISBN 978-1-4503-1561-6. doi:10.1145/2384616.2384680. URL https://doi.org/10.1145/2384616.2384680.

[60] 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. URL https://doi.org/10.1145/503502.503505.

[61] Alex Jeffery. Dependent object types with implicit functions. In Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom, editors, *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, pages 1–11. ACM, 2019. ISBN 978-1-4503-6824-7. doi:10.1145/3337932.3338811. URL https://doi.org/10.1145/3337932.3338811.

[62] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980.

[63] Ifaz Kabir. themaplelab / dot-public: A simpler syntactic soundness proof for dependent object types. https://github.com/themaplelab/dot-public/tree/master/dot-simpler, 2022. Accessed: 2022-10-10.

[64] 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. URL https://doi.org/10.1145/3241653.3241659.

[65] Ifaz Kabir, Yufeng Li, and Ondřej Lhoták. ιDOT: a DOT calculus with object initialization. *Proc. ACM Program. Lang.*, 4(OOPSLA):208:1–208:28, 2020. doi:10.1145/3428276. URL https://doi.org/10.1145/3428276.

[66] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. Lightweight verification of array indexing. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 3–14. ACM, 2018. doi:10.1145/3213846.3213849. URL `https://doi.org/10.1145/3213846.3213849`.

[67] Anatole Le, Ondřej Lhoták, and Laurie J. Hendren. Using inter-procedural side-effect information in JIT optimizations. In Rastislav Bodík, editor, *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3443 of *Lecture Notes in Computer Science*, pages 287–304. Springer, 2005. doi:10.1007/11406921_22. URL `https://doi.org/10.1007/11406921_22`.

[68] Edward Lee and Ondřej Lhoták. Simple reference immutability for System F<:. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023. doi:10.1145/3622828. URL `https://doi.org/10.1145/3622828`.

[69] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20. URL `https://doi.org/10.1007/978-3-642-17511-4_20`.

[70] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight Java with assignment and immutability using the coq proof assistant. In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 11–19. ACM, 2012. doi:10.1145/2318202.2318206. URL `https://doi.org/10.1145/2318202.2318206`.

[71] Christopher A. Mackie. Preventing signedness errors in numerical computations in Java. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 1148–1150. ACM, 2016. doi:10.1145/2950290.2983978. URL `https://doi.org/10.1145/2950290.2983978`.

[72] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, page 1–10, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329163. doi:10.1145/2617548.2617553. URL `https://doi.org/10.1145/2617548.2617553`.

[73] Dennis Mancl and William Havanas. A study of the impact of C++ on software maintenance. In *Proceedings of the Conference on Software Maintenance, ICSM 1990, San Diego, CA, USA, 26-29 November, 1990*, pages 63–69. IEEE, 1990. doi:10.1109/ICSM.1990.131325. URL `https://doi.org/10.1109/ICSM.1990.131325`.

[74] Darya Melicher, Anlun Xu, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. Bounded abstract effects. *ACM Trans. Program. Lang. Syst.*, 44(1), jan 2022. ISSN 0164-0925. doi:10.1145/3492427. URL `https://doi.org/10.1145/3492427`.

[75] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. Verifying determinism in sequential programs. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 37–49. IEEE, 2021. doi:10.1109/ICSE43902.2021.00017. URL `https://doi.org/10.1109/ICSE43902.2021.00017`.

[76] David A. Naumann. Observational purity and encapsulation. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2005. ISBN 978-3-540-25420-1 978-3-540-31984-9. doi:10.1007/978-3-540-31984-9_15. URL `https://doi.org/10.1007/978-3-540-31984-9_15`.

[77] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process*, 29(12), 2017. ISSN 20477473. doi:10.1002/smr.1889. URL `https://doi.org/10.1002/smr.1889`.

[78] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer, 1999. doi:10.1007/3-540-48092-7_6. URL `https://doi.org/10.1007/3-540-48092-7_6`.

[79] Abel Nieto. Towards algorithmic typing for DOT (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala - SCALA 2017*, SCALA '17, pages 2–7. ACM Press. ISBN 978-1-4503-5529-2. doi:10.1145/3136000.3136003. URL `http://dl.acm.org/citation.cfm?doid=3136000.3136003`.

[80] Abel Nieto. Towards algorithmic typing for $\mathrm{d}_{<:}$. *CoRR*, abs/1708.05437, 2017. URL `http://arxiv.org/abs/1708.05437`.

[81] Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In Robert Hirschfeld and Tobias

Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:26, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-154-2. doi:10.4230/LIPIcs.ECOOP.2020.25. URL `https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.25`.

[82] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with Scala. *Commun. ACM*, 57(4):76–86, 2014. doi:10.1145/2591013. URL `https://doi.org/10.1145/2591013`.

[83] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL): 42:1–42:29, 2018. doi:10.1145/3158130. URL `https://doi.org/10.1145/3158130`.

[84] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. Safer exceptions for Scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*, SCALA 2021, page 1–11, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391139. doi:10.1145/3486610.3486893. URL `https://doi.org/10.1145/3486610.3486893`.

[85] Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. Scoped capabilities for polymorphic effects, 2022.

[86] Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 364–377, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342193. doi:10.1145/2951913.2951945. URL `https://doi.org/10.1145/2951913.2951945`.

[87] 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. URL `https://doi.org/10.1145/1390630.1390656`.

[88] David J. Pearce. JPure: A modular purity system for Java. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2011. doi:10.1007/978-3-642-19861-8_7. URL `https://doi.org/10.1007/978-3-642-19861-8_7`.

[89] David J. Pearce. A lightweight formalism for reference lifetimes and borrowing in Rust. *ACM Trans. Program. Lang. Syst.*, 43(1):3:1–3:73, 2021. doi:10.1145/3443420. URL `https://doi.org/10.1145/3443420`.

[90] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

[91] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN 0262162288.

[92] Dimitri Racordon and Didier Buchs. Featherweight Swift: a core calculus for Swift's type system. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2020, page 140–154, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381765. doi:10.1145/3426425.3426939. URL `https://doi.org/10.1145/3426425.3426939`.

[93] 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. ACM Press, 2017. ISBN 978-1-4503-5098-3. doi:10.1145/3103111.3104036. URL `https://doi.org/10.1145/3103111.3104036`.

[94] Marianna Rapoport and Ondřej Lhoták. A path to DOT: formalizing fully path-dependent types. *Proc. ACM Program. Lang.*, 3(OOPSLA):145:1–145:29, 2019. doi:10.1145/3360571. URL `https://doi.org/10.1145/3360571`.

[95] Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *Proc. ACM Program. Lang.*, 1(OOPSLA):46:1–46:27, 2017. doi:10.1145/3133870. URL `https://doi.org/10.1145/3133870`.

[96] 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*, OOPSLA '16, pages 624–641. ACM, 2016. ISBN 978-1-4503-4444-9. doi:10.1145/2983990.2984008. URL `https://doi.org/10.1145/2983990.2984008`.

[97] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 82–91. IEEE Computer Society, 2004. ISBN 978-0-7695-2213-5. doi:10.1109/ICSM.2004.1357793. URL `https://doi.org/10.1109/ICSM.2004.1357793`.

[98] Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In Werner Dietl, editor, *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*, FTfJP '13, pages 4:1–4:7.

ACM, 2013. ISBN 978-1-4503-2042-9. doi:10.1145/2489804.2489808. URL https://doi.org/10.1145/2489804.2489808.

[99] Alexandru Salcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2004. URL https://dspace.mit.edu/handle/1721.1/30470.

[100] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2005. ISBN 978-3-540-24297-0 978-3-540-30579-8. doi:10.1007/978-3-540-30579-8_14. URL https://doi.org/10.1007/978-3-540-30579-8_14.

[101] Noé De Santo, Aurèle Barrière, and Clément Pit-Claudel. A Coq mechanization of JavaScript regular expression semantics. *CoRR*, abs/2403.11919, 2024. doi:10.48550/ARXIV.2403.11919. URL https://doi.org/10.48550/arXiv.2403.11919.

[102] Ulrich Schöpp and Chuangjie Xu. A generic type system for featherweight Java. In David R. Cok, editor, *FTfJP 2021: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, Virtual Event, Denmark, 13 July 2021*, pages 9–15. ACM, 2021. doi:10.1145/3464971.3468419. URL https://doi.org/10.1145/3464971.3468419.

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

[104] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Upper Saddle River, NJ, 2013. ISBN 978-0-321-56384-2.

[105] Elliot Suzdalnitski. Object-oriented programming – the trillion dollar disaster. https://betterprogramming.pub/object-oriented-programming-the-trillion-dollar-disaster-92a4b666c7c7. Accessed: 2024-05-20.

[106] 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. URL https://doi.org/10.1145/1094811.1094828.

[107] Fei Wang and Tiark Rompf. Towards strong normalization for dependent object types (DOT). page 25 pages. doi:10.4230/LIPICS.ECOOP.2017.27. URL http://drops.dagstuhl.de/opus/volltexte/2017/7276/. Artwork Size: 25 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

[108] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. Polymorphic reachability types: Tracking freshness, aliasing, and separation in higher-order generic programs. *Proc. ACM Program. Lang.*, 8 (POPL), jan 2024. doi:10.1145/3632856. URL `https://doi.org/10.1145/3632856`.

[109] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In Manuvir Das and Dan Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, pages 75–82. ACM, 2007. ISBN 978-1-59593-595-3. doi:10.1145/1251535.1251548. URL `https://doi.org/10.1145/1251535.1251548`.

[110] 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. URL `https://doi.org/10.1145/1287624.1287637`.

List of Figures

List of Tables

List of Abbreviations

CNF Conjunctive Normal Form

DOT Dependent Object Types

JVM Java Virtual Machine

SEF Side-Effect Free

List of Publications

Published paper at peer-reviewed conferences

- Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. Lightweight verification of array indexing. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 3–14. ACM, 2018.
- Vlastimil Dort and Ondřej Lhoták. Reference mutability for DOT. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 18:1–18:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

Papers accepted at peer-reviewed conferences The following paper is on the track to appear at the 38th European Conference on Object-Oriented Programming (ECOOP 2024) in September 2024.

- Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek. Pure methods for roDOT. In *ECOOP 2024*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024 (forthcoming).

Other (non-peer-reviewed)

- 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.

A. Attachments

A.1 Mechanization of roDOT in Coq

The first attachment to is the full mechanization of roDOT, its soundness proof and proofs of the guarantees from this thesis, as explained in Section 3.6 and Section 4.6.

The source code is provided directly in the attached directory `rodot`, or, for convenience, as a Docker image containing the necessary software to verify the proofs.

The code is based on the mechanization of Field mutable DOT by Ifaz Kabir [63], and roDOT features were implemented in collaboration with Yufeng Li.

A.1.1 Using the Attached Source Code

Compiling the code requires Coq[1] version 8.10.2, and the TLC library[34] version 20181116.

The following commands can be used to install Coq and TLC using the OCaml Package Manager [2][63]:

```
opam init --compiler=4.09.1 --disable-sandboxing -a
opam pin add coq 8.10.2 -y
opam repo add coq-released http://coq.inria.fr/opam/released
opam pin add coq-tlc 20181116 -y
```

A.1.2 Using the Docker Image

The Docker image is available on DockerHub as `rodotcalculus/rodot-thesis`[3]. Running the image requires a x86_64 machine with the Linux operating system.

To download and run the docker image on your machine, execute the following command:

```
sudo docker run -it --rm rodotcalculus/rodot-thesis
```

When you run the docker image, a shell will start in the directory `/root`. Enter the directory `rodot` with the command:

```
cd rodot
```

The `*.v` files in this directory and its subdirectories constitute the source code.

A.1.3 Verifying the Proofs

To build the project and verify the correctness of the proof, run:

```
make -j4
```

[1]`https://coq.inria.fr/`

[2]`https://opam.ocaml.org/`, `https://coq.inria.fr/opam-using.html`

[3]`https://hub.docker.com/r/rodotcalculus/rodot-thesis`

You can omit the `-j4` option or use a different number to control the level of parallelism of the build. During the checking process, names of individual source files will be printed. On successfully verifying all the files, the text `make[1]: Leaving directory '/root/rodot'` is printed.

Checking theorem assumptions Coq code can use additional axioms, which are assumed true, so the correctness of the evaluation depends on correctness of the used axioms.

The proof use the following axioms of classical logic:

```
LibAxioms.prop_ext (from TLC library)
LibAxioms.indefinite_description (from TLC library)
LibAxioms.fun_ext_dep (from TLC library)
Classical_Prop.classic
```

In order to check what axioms are used (and that no additional axioms are used), we provide the script `print-assumptions.sh`. The first argument names a module, the second argument names a theorem to check.

To check the axioms used by the main theorems, run the following commands after building the project:

```
./print-assumptions.sh Safety soundness_initial
./print-assumptions.sh \
Mutability.ImmutabilityGuarantee immutability_guarantee
./print-assumptions.sh Mutability.SefGuarantee SefG_I
./print-assumptions.sh \
Transformation.TransformationSwapCallsGuarantee \
swap_calls_transformation_guarantee
```

A.2 Demonstration of Reference Mutability Checking in Dotty

The second attachment is a patch for the source code of the Dotty compiler, which shows a simplified implementation of reference mutability checking for Scala as explained in Section 5.3.

To use the patched compiler, either apply the patch attached to this thesis to a copy of the source code of Dotty[4], or use the Docker image available for convenience.

A.2.1 Using the Attached Source Patch

The patch must be applied to the source code of the Dotty compiler at Git commit `cdfc76e2e5093dd8f70e1d582e38d2537494c0bb` [5].

[4]`https://github.com/scala/scala3`

[5]`https://github.com/scala/scala3/tree/cdfc76e2e5093dd8f70e1d582e38d2537494c0bb`

Running the compiler requires Java SE JDK version 8 [6] (versions newer than 8 are not supported). and SBT [7].

The source code can be downloaded and the patch applied using the following commands:

```
git clone https://github.com/scala/scala3
cd scala3
git checkout cdfc76e2e5093dd8f70e1d582e38d2537494c0bb
git switch —c reference—mutability—demo
git am < reference—mutability—demo.patch
```

A.2.2 Using the Docker Image

The Docker image is available on DockerHub as `rodotcalculus/rodot-thesis`[8]. Running the image requires a x86_64 machine with the Linux operating system.

To run the docker image on your machine, execute the following command.

```
sudo docker run —it ——rm rodotcalculus/thesis
```

When you run the docker image, a shell will start in the directory `/root`. Enter the dotty directory by the command:

```
cd dotty
```

This is the root directory of the Dotty compiler, with the patch already applied.

Using the Reference Mutability Demo

To run the compiler with the reference mutability checker, first start SBT with the command `sbt`, which will enter an interactive prompt.

Running tests To run the checker on a set of tests that show the implemented features, use the command `testMutCompilation`. On a successful run, the tests will print:

```
[info] Test dotty.tools.dotc.mut.MutCompilationTests.negAll started
[=======================================] completed (4/4, 0 failed, 3s)
[info] Test dotty.tools.dotc.mut.MutCompilationTests.pos started
[=======================================] completed (2/2, 0 failed, 0s)
[info] Test run finished: 0 failed, 0 ignored, 2 total, 3.976s
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
```

The test source codes can be found under the directory `tests/pos-mut` (files that should type-check without errors), and `tests/neg-mut` (files that should type-check with errors, on lines indicated by comments),

[6]https://www.oracle.com/cz/java/technologies/javase/javase8u211-later-archive-downloads.html

[7]https://www.scala-sbt.org/download/

[8]https://hub.docker.com/r/rodotcalculus/rodot-thesis

Type-checking a Scala file To type-check a Scala file, type the command `dotc -Ym -Ystop-after:refchecks SourceFile.scala`. The option `-Ym` turns on the mutability checks (options prefixed with `-Y` generally enable experimental features), while `-Ystop-after:refchecks` stops compilation early to avoid entering later stages of compilation that are not adjusted to the changes.

The code examples shown in Section 5.3 are provided in the directory `reference-mutability-demo-examples`.