Pure Methods for roDOT

Vlastimil Dort
Charles University, Prague, Czech Republic

Yufeng Li
University of Cambridge, UK

Ondřej Lhoták
University of Waterloo, Canada

Pavel Parízek
Charles University, Prague, Czech Republic

Abstract

Object-oriented programming languages typically allow mutation of objects, but pure methods are common too. There is great interest in recognizing which methods are pure, because it eases analysis of program behavior and allows modifying the program without changing its behavior. The roDOT calculus is a formal calculus extending DOT with reference mutability. In this paper, we explore purity conditions in roDOT and pose a SEF guarantee, by which the type system guarantees that methods of certain types are side-effect free. We use the idea from ReIm to detect pure methods by argument types. Applying this idea to roDOT required just a few changes to the type system, but necessitated re-working a significant part of the soundness proof. In addition, we state a transformation guarantee, which states that in a roDOT program, calls to SEF methods can be safely reordered without changing the outcome of the program. We proved type soundness of the updated roDOT calculus, using multiple layers of typing judgments. We proved the SEF guarantee by applying the Immutability guarantee, and the transformation guarantee by applying the SEF guarantee within a framework for reasoning about safe transformations of roDOT programs. All proofs are mechanized in Coq.

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

Keywords and phrases type systems, DOT calculus, pure methods

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.13

Related Version *Extended Version including Appendix*: `https://d3s.mff.cuni.cz/files/publications/dort_pure_report_2024.pdf` [13]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.6`

Funding This work was supported by the Czech Science Foundation project 23-06506S, and by the Czech Ministry of Education, Youth and Sports project LL2325 of the ERC.CZ programme. This research was also supported by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

A feature common to many object-oriented programming languages is that execution of a method can have important side effects such as creating new objects on the heap or modifying (mutating) existing objects. 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 [14, 10] and Code Contracts [15]), 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 [19] 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 and consistency of such annotations.

Scala favours 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 main objective is to design a type system that guarantees side effect freedom for Scala methods and supports advanced language features present in Scala.

Previous formalization efforts for Scala resulted in the Dependent Object Types (DOT) calculus [2], which captures the essence of Scala's type system. However, the original DOT calculus does not model mutation of objects, so purity cannot be addressed there, but some variants that do allow mutation have been developed. roDOT [12] is an existing core calculus for Scala with reference mutability. It has mutable fields and a type system feature to distinguish read-only and mutable references. An important rationale behind the design of roDOT, when compared to other possible approaches, is to use existing features of Scala, including its rich type system, as much as possible rather than introducing new forms of types only for reference mutability, to ease adoption of such a type system into the Scala language. In particular, roDOT expresses the mutability of a reference using a specially designated type member in the type of that reference. The type system of roDOT also provides an *immutability guarantee*: an object can only be mutated if there is a path of mutable references to it from the code being executed.

In this paper, we extend the core roDOT calculus from [12] with the concept of side-effect-free methods. Before going into details, we want to emphasize that it was not possible to simply adapt ReIm [19] from Java, because of several challenges specific to Scala and roDOT that we discuss below. However, we use the idea proposed by the authors of ReIm that side-effect-free methods are recognized based on the types of their parameters.

The general concept of purity is, in addition to (1) side-effect-freedom, sometimes understood to comprise more properties: (2) determinism – returning the same value for the same arguments [14], and (3) termination. In this paper, we focus only on the side-effect-free (SEF) property. We will just mention that in regards to determinism, mutable DOT calculi including roDOT have semantics that is deterministic except for instantiation of objects.

Within the context of roDOT, we look at the SEF property from three different perspectives – what a SEF method does, how it can be recognized using the type system, and how it can be used in programs. We define SEF methods in roDOT as those that do not modify any objects that existed on the heap before the method was called.

As the main result of this paper, we prove the *side-effect-free guarantee* (SEF guarantee), which says that methods with read-only parameters do not modify existing objects.

One important related 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 existing (original) roDOT type system from [12]. Therefore, one of our contributions is an extension of roDOT that makes it possible to recognize read-only types.

Another challenge was defining the SEF guarantee formally and proving it in a calculus that supports a mutable heap (like roDOT). 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.

Finally, as a corollary of the SEF guarantee, we state and prove a guarantee of safety of a particular code transformation. The transformation guarantee states that swapping two calls to SEF methods anywhere in a program does not affect the result of its execution.

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

We mechanized all of our formal results, in particular the soundness proof of the extended roDOT calculus and the SEF guarantee, in Coq to enable future formal reasoning to build on them. Note that the soundness of the original roDOT calculus was proved by hand in [12]. We have made our formalization in Coq public as an artifact for this paper.

1.1 Contribution

In summary, the main contributions of this paper are the following:

- a modification of the original roDOT calculus that makes it possible to test whether a type is read-only, which is necessary to state and prove the SEF guarantee;
- a formal definition of side-effect-free methods in the context of roDOT, statement and proof of the SEF guarantee;

- a general framework for defining transformations of roDOT programs and proving that some of them are safe in that they do not change the result of program execution, statement and proof of a transformation guarantee, which states that re-ordering calls to SEF methods is safe in that sense;
- the first mechanization of roDOT and its immutability guarantee, with addition of the SEF and transformation guarantees, and all the proofs in Coq – provided as an artifact.

1.2 Outline

The paper is organized as follows. Section 2 gives an overview of the roDOT calculus, which has been mechanized in Coq and within which we define the SEF condition. Section 3 discusses the definition of pure and SEF methods, looking at several variants. It defines the SEF guarantee and identifies necessary changes to the roDOT type system in order for the guarantee to work. In Section 4 we describe the changes to the calculus in more detail, and discuss a new proof of type soundness of the calculus. In Section 5, we describe how we proved the SEF guarantee, and in Section 6 we define and prove the transformation guarantee within a framework for safe transformations. An appendix containing full definitions and more detailed discussion is available in the extended version of this paper [13].

2 Background – The roDOT calculus

In this section, we present the summary of the roDOT calculus [12], which we use as the baseline for this work. The DOT calculus [2, 33, 30] is a formal calculus, designed to formalize the essence of the types of the Scala programming language. In the basic versions of the DOT calculus, objects have read-only fields (so the objects are immutable), but there are also several versions that allow changing values of the fields of objects (mutation).

The roDOT calculus [12] evolved from DOT with mutable fields. The goal was to extend DOT with the ability to control mutation of objects using the type system, while using the existing features of the DOT calculus, dependent types.

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 $\{\mathsf{M} : \bot..\bot\}$. Thanks to that, we can refer to the mutability of a variable x using type selection $x.\mathsf{M}$.

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.

2.1 Syntax and typing

The syntax of terms and types in roDOT is in Figure 1. It uses the A-normal form [36] 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. Objects are represented by the $\nu(s : R)d$ constructor, appearing as literals in the programs and as items on the heap (R is the type of the object and d is a list of member definitions).

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

$x ::= z,\ s,\ r$	**Variable**	$T ::=$	**Type**
$\mid y \mid w$	location, reference	$\mid \top \mid \bot \mid \mathsf{N}$	top, bottom, read-only $\bot$
$t ::=$	**Term**	$\mid \mu(s : T) \mid x_1.B(x_2)$	recursive, selection
$\mid \mathsf{v}x \mid x_1.m\, x_2$	variable, method call	$\mid \{a : T_1..T_2\} \mid \{B(r) : T_1..T_2\}$	field, type decl.
$\mid \mathsf{let}\ z = t_1\ \mathsf{in}\ t_2$	let	$\mid \{m(z : T_1, r : T_3) : T_2\}$	method
$\mid \mathsf{let}\ z = \nu(s : T)d\ \mathsf{in}\ t$	let-literal	$\mid T_1 \wedge T_2 \mid T_1 \vee T_2$	intersection, union
$\mid x.a \mid x_1.a := x_2$	read, write	$B ::=$	**Type member name**
$d ::= d_1 \wedge d_2$	**Definition**	$\mid A \mid \mathsf{M}$	ordinary, mutability
$\mid \{a = x\} \mid \{A(r) = T\}$	field, type	$\sigma ::= \cdot \mid \mathsf{let}\ z = \square\ \mathsf{in}\ t :: \sigma$	**Stack**
$\mid \{m(z, r) = t\}$	method	$\Sigma ::= \cdot \mid \Sigma, y \to d$	**Heap**
$\rho ::= \cdot \mid \rho, w \to y$	**Environment**	$c ::= \langle t; \sigma; \rho; \Sigma \rangle$	**Configuration**

Figure 1 roDOT syntax.

$$\dfrac{\begin{array}{c}\Gamma;\rho \vdash x_1 : \{m(z : T_1, r : T_3) : T_2\} \\ \Gamma;\rho \vdash x_2 : T_1 \qquad \Gamma\ \textbf{vis}\ x_2 \\ \Gamma;\rho \vdash x_1 : [x_2/z]T_3 \qquad \Gamma\ \textbf{vis}\ x_1 \\ T_3\ \textbf{indep}\ z\end{array}}{\Gamma;\rho \vdash x_1.m\, x_2 : [x_1/r][x_2/z]T_2}\ (\text{TT-Call}) \qquad \dfrac{\begin{array}{c}\Gamma;\rho \vdash x_1 : T_1 \qquad \Gamma\ \textbf{vis}\ x_1 \\ \Gamma;\rho \vdash x : \{a : T_1..T_2\} \quad \Gamma\ \textbf{vis}\ x \\ \Gamma;\rho \vdash x : \{\mathsf{M}(r) : \bot..\bot\}\end{array}}{\Gamma;\rho \vdash x.a := x_1 : T_2}\ (\text{TT-Write})$$

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

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

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

Figure 2 Selected rules for typing and reduction in roDOT.

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. In roDOT, references are a separate concept from locations in order to allow references to the same object to have different types (specifically, different mutabilities). While the run-time stack and focus of execution work with references that have their own mutabilities, the heap only works with locations, and mutability is determined by field 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 type members $\{A : T..T\}$ specify lower and upper bounds, and they introduce a new dependent type $x.A$ that has a subtyping relationship with those bounds. This is relevant because roDOT uses a type member for mutability. The ability to create dependent types in this manner is the defining feature of the DOT calculus.

The declarations of an object's members are wrapped in a recursive type, so several declarations in one object type can reference each other, using a member type selection $s.A$ involving the self-reference s. An example of a type of an object without mutability is $\mu(s : \{A : T..T\} \wedge \{a : s.A..s.A\} \wedge \{m(r : T, z : T) : T\})$. An object of this type has a type member A with bounds T, a field a with a self-referential type $s.A$, and a method m.

In roDOT, dependent types are also used to express the mutability of a reference, by selecting the special type member M. 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. The mutability of a reference applies to the whole object – a mutable reference allows writing to all fields.

An example of a type of an object with a type member A, a field a, method m and a mutability declaration is $\mu(s : \{A : T..T\} \wedge \{a : T..T\} \wedge \{m(r : T, z : T) : T\}) \wedge \{\mathsf{M} : \bot..\bot\}$.

A declaration of a method allows specifying a type of the receiving reference $r : T$, which can be more precise than the type of the recursive self parameter s in the defining object. 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 `this` parameter in Java, used by the Checker Framework [18, 9]. For this reason, every method in roDOT has two parameters: a normal parameter z and the receiver r, which is a reference to the object containing the method, like `this` in Scala. In roDOT, the type of r can be dependent on z. The parameter r is special in how it gets its type, but in terms of semantics, behaves the same as z.

Several rules in roDOT need a read-only version of a type. For that, there are two type-level operations: T **ro** U means that U is a readonly version of T, T **mu** U means that U is a mutability bound of type T (rules are shown in Figure 11 in the appendix). A special type N is defined to be the read-only version of the least type in the subtyping lattice, $\bot$.

The typing rules (selected in Figure 2, full set in Figures 8 to 12 in the appendix) describe correctly formed programs. In addition to the typing context Γ, which assigns types to variables, the left side of the typing judgment includes an environment ρ that connects references in the terms to locations of objects on the heap.

The write term, typed by TT-Write, is guarded by a check of the mutability permission on the receiving reference. The premise $\Gamma;\rho \vdash x : \{\mathsf{M} : \bot..\bot\}$ ensures that only mutable references can be used for writing.

Reading a field, typed by TT-Read, is always possible, but the type of the result is changed to read-only if the source reference is read-only. This type operation is called viewpoint adaptation, and ensures that read-onlyness is transitive, which is required for the immutability guarantee of roDOT and for our SEF guarantee. This is achieved by taking a read-only version of the field's type, and adding a mutability that is a union of the mutabilities of the source reference and of the field type. For example, if a reference w has type $\{a : T_1..\mu(s : \dots) \wedge \mathcal{M}_U\}$, then the term $w.a$ has type $\mu(s : \dots) \wedge \mathcal{M}_{w.a \vee U}$.

With the **vis** judgment (Figure 9 in the appendix), roDOT hides captured variables in methods – to access a value from outside, it must be stored in a field of the containing object, so viewpoint adaptation applies to it.

Variables appearing in terms and definitions have types given by the typing and subtyping rules in Figures 9 and 10 in the appendix. Selected rules are shown in Figure 2: ST-Met is a subtyping rule for method declarations. The part highlighted in grey is not part of roDOT, but our modification, which we will describe in Section 4.2. Rules ST-Or and ST-Dist are examples of subtyping rules for union types, which are relevant in Section 4.1.

2.2 Semantics

The operational semantics of roDOT is defined as a small step semantics, with machine configurations (Figure 1) consisting of a term in the focus of execution t, a stack σ, a 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 terms of the form $\mathsf{let}\, z = t_1 \,\mathsf{in}\, t_2$. The stack is a list of frames of the form $\mathsf{let}\, z = \square \,\mathsf{in}\, t_2$, where $\square$ represents t_1 while it is being evaluated in the focus of execution. When t_1 is evaluated to a value, that value is substituted for the square in the top frame of the stack, and the t_2 from that frame then becomes the new focus of execution.

Execution starts with the program, an empty stack, empty heap and an empty environment, and proceeds by steps defined in Figure 13 in the appendix, 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). Calling a method copies its body to the focus of execution, while the receiver and argument are substituted.

The semantics is generally deterministic – there is no way to express a nondeterministic choice. However, there is one source of non-determinism: locations of objects on the heap. Allocating objects must be regarded as a non-deterministic operation because even if the new objects are initially equal, they may take on different values due to subsequent mutation.

2.3 Properties

The roDOT calculus has the type soundness property (Theorem 1, Theorem 7 in [12]) – a term that has a type in an empty context can be executed and either reduces to an answer, or executes indefinitely. DOT and roDOT do not include explicit checks for error conditions, but trying to access (read, write or call) a non-existing member of an object is an error. In such a case, a reduction step is not defined and the execution "gets stuck". The soundness theorem guarantees this does not happen for typed programs.

▶ **Theorem 1** (Type Soundness).

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

Type soundness and other properties are based on the fact that during execution, the type of the configuration is preserved. Rules for typing a machine configuration are in Figure 14 in the appendix. 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 an *inert context*. Under an inert context, stronger claims can be made about types of variables [30], and it plays an important role in the proof of soundness.

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

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

Figure 3 roDOT mutable reachable references.

The essential property of roDOT is the immutability guarantee (Theorem 2, Theorem 9 in [12]): 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, defined formally in Figure 3.

▶ **Theorem 2** (Immutability Guarantee).

If $y \rightarrow d \in \Sigma_1$ *and* $\Gamma \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle : T$,	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 $y \rightarrow d \in \Sigma_2$,	either the object does not change,
or $\Gamma \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1\rangle$ **mreach** y.	or it was reachable by mutable references.

3 Method Purity for roDOT

Here we informally define the meaning of side-effect freedom in roDOT, and informally state the main results of this paper: 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. (Static) Recognize which methods are SEF statically at compile time, using types.
2. (Runtime) Define what events can (or cannot) happen when a SEF method is executed.
3. (Usage) Differentiate SEF methods from general methods based on how they can be safely used in programs.

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.

3.1 Runtime SEF condition

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, meaning no instantiations and no writes. 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 [34, 38, 19, 14]) 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.

▶ **Informal statement of Definition 15** (Run-time SEF condition, in Section 5.1)**.** 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.

3.2 Static SEF condition

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.

In ReIm [19], 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.

▶ **Informal statement of Definition 11** (static SEF condition)**.** 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 4.1. Example 3 and Example 4 illustrate its ability to recognize SEF methods.

▶ **Example 3.** 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.

▶ **Example 4.** 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 $T_z = \{m_a(r : \top, z : \mu(r_o : R_o) \wedge \{\mathsf{M} : \bot..\bot\}) : \top\}$. By Definition 11, m_{sef} is SEF, because it has read-only parameters, even though it calls m_a, which may mutate the heap.

Example 5 shows how viewpoint adaptation transitively ensures that read-only parameters cannot be used to modify existing objects. Example 6 shows how a dependent type can change whether the method is SEF or possibly not.

▶ **Example 5.** 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 6.** 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.

3.3 SEF guarantee

For the **SEF guarantee** (Theorem 16), 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.

▶ **Informal statement of Theorem 16** (SEF guarantee, in Section 5.2)**.** 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.

3.4 Using pure methods in roDOT

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. For example, calls to SEF methods can be safely reordered.

To keep the problem simple, we will look at one particular case of such reordering: swapping two calls to SEF methods. With SEF methods, the code `x1.m1(); x2.m2()` is equivalent `x2.m2(); x1.m1()`.

▶ **Informal definition** (Call-swapping transformation of programs)**.** 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 7.** A chain of calls $\mathsf{let}\, x_1 = x_{\text{o}1}.m_1 x_{\text{a}1}\, \mathsf{in}\, \mathsf{let}\, x_2 = x_{\text{o}2}.m_2 x_{\text{a}2}\, \mathsf{in}\, t$, can transformed by call swapping into $\mathsf{let}\, x_2 = x_{\text{o}2}.m_2 x_{\text{a}2}\, \mathsf{in}\, \mathsf{let}\, x_1 = x_{\text{o}1}.m_1 x_{\text{a}1}\, \mathsf{in}\, t$.

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.

▶ **Informal statement of Theorem 27.** The call-swapping transformation 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 are provided in Section 6.

4 Recognizing SEF methods by type in modified roDOT

In this section, we formalize the static SEF condition in roDOT given informally in Section 3.2. Although 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 nor a superset of the original, so it is necessary to update the proof of soundness and the immutability guarantee, which were proven by hand for the original roDOT [12]. The soundness proof followed the scheme from [30] 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 structure 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. In the presence of cycles, we cannot use the straightforward inductive hypothesis to prove properties necessary for type safety, because a derivation for typing a variable can involve derivations of arbitrarily complex types.

We implemented a new proof based on a different auxiliary typing definition, which avoids cycles by forcing the derivation to arrive at the target type by adding type constructors in a fixed order (for example, all unions in the type are handled before intersections). Compared to the original invertible typing, which was single typing judgment with many rules, the new approach leads to a definition in several layers, where each layer has 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 4.1, we formalize static SEF condition, and discuss the meaning of read-only types in roDOT. In Section 4.2, we propose small changes to the roDOT calculus to make definitions work for the SEF guarantee. We give a short overview of the structure of the original soundness proof for roDOT, and show how this proof breaks with the new changes. In Section 4.3, we describe the new layered typing that replaces invertible typing in the updated proof and show its important properties.

4.1 Static SEF condition in roDOT

In the **SEF guarantee** (Theorem 16), we claim that a method is SEF based on the type of the method declaration. Our SEF guarantee follows the approach of ReIm [19] and requires the parameters to have read-only types.

4.1.1 Read-only types in roDOT

The check whether 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 9 in the appendix). It was not based on subtyping, but rather on the relation **ro**, which makes a read-only version of a type using a syntax-based type splitting.

This definition did not guarantee that all supertypes of a read-only type are also read-only. As we will explain in Section 4.1.3, this would be a critical problem for the SEF guarantee.

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 $\bot$ 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 8** (Read-only types). *A type T is read-only, if* $\Gamma;\rho \vdash \mathsf{N} <: T$.

With Definition 8 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 19 in Section 5.4, 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 8 of read-only types, this can be stated as:

▶ **Lemma 9** (Read-only types are shared by all references). *If $\Gamma \sim \rho$ and $\Gamma;\rho \vdash y : T$ and $\Gamma;\rho \vdash \mathsf{N} <: T$, then $\Gamma;\rho \vdash w : T$ for any w such that $\rho(w) = y$.*

This key lemma, however, does not hold in the original roDOT, because of union types.

Union types were not a part of 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 10 in the appendix. However, using unions, it is possible to construct a type that is a supertype of both N and a mutability declaration:

▶ **Example**[†] **10** (In the original roDOT, counter-example to Lemma 9). Let $T_{\mathrm{am}} \coloneqq \{a : T_{\mathrm{a}}\} \vee \mathcal{M}_{\bot}$ be a union of some field declaration with a declaration of mutability, and $T_{\mathrm{bm}} \coloneqq \{b : T_{\mathrm{b}}\} \wedge \mathcal{M}_{\bot}$ be a type of a writeable reference to some other field b.

The type T_{am} is not mutable, because it is not a subtype of $\mathcal{M}_{\bot}$. It is read-only, because $\Gamma \vdash \mathsf{N} <: T_{\mathrm{am}}$, by the rules of subtyping of union types and by rule ST-N-Fld (Figure 10 in the appendix).

Let y_1 be a location of type T_{bm}, and w_2 be a reference to y_1 with type $T_{\mathrm{b}} \coloneqq \{b : T_{\mathrm{bm}}\}$. By subtyping of unions and intersections, $\Gamma \vdash T_{\mathrm{bm}} <: T_{\mathrm{am}}$, so by subsumption, y_1 has type T_{am}. By Lemma 9, w_2 should have also type T_{am}, but in the original roDOT, it does not.

Example[†] 10 is marked with the [†] sign, which we use in this chapter to identify properties of the original roDOT from prior work, in contrast to the modified roDOT in this paper.

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 9 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 4.2.

4.1.2 The SEF condition

A method is statically SEF if the types of its receiver and parameters are read-only according to Definition 8 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 11** (Static SEF condition). *A method is statically SEF if it has a type* $\{m(z : \mathsf{N}, r : \mathsf{N}) : \top\}$

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

Figure 4 New rules for roDOT.

In Section 5, 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), so the method will not be able to get a writeable reference to any existing object.

4.1.3 Subtyping of method types

In order for the SEF guarantee (Theorem 16) to work with Definition 11, 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 8 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 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 8. If r has the type $\{A : z.\mathsf{M}..\bot\}$, one can use the typing rules ST-SelL and ST-SelU (Figure 10 in the appendix)) to derive $z.\mathsf{M} <: \bot$. The change to the rules TT-Call and ST-Met in Figure 2 prevents this issue by disallowing using method types where the receiver depends on the mutability of the parameter.

4.2 The updated roDOT calculus

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) is added, which makes the union of a mutability declaration and the read-only lower bound N a top-like type (the other direction of subtyping was already a part of the type system).
- A new rule TS-N (Figure 4) is added to type splitting, making it so that all types that are read-only by Definition 8 are unaffected by the splitting operation.
- The typing rule TT-Call and subtyping rule ST-Met have a new premise (shown highlighted in Figure 2), which disallows introducing a dependency between the receiver type and the parameter in method subtyping. This fixes the problem described in Section 4.1.3.

The new rule ST-NM fixes the counter-example to Lemma 9, 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, we can now improve the type splitting relation $\vdash\ \ \mathbf{ro}\ $, by extending it with a new rule TS-N, shown in Figure 4. With that, the condition in VT-RecI (Figure 9 in the appendix) that recursive types are read-only, $\Gamma;\rho \vdash T\ \mathbf{ro}\ T$, becomes equivalent to Definition 8:

$$\begin{tikzcd}
 & \begin{array}{c}\text{General subtyping}\\ \Gamma;\rho \vdash S <: T\end{array} \arrow[r, Leftrightarrow, shift left] \arrow[dl, leftrightarrow] & \begin{array}{c}\text{Tight subtyping}\\ \Gamma;\rho \vdash_{\#} S <: T\end{array} \arrow[l, shift left] \arrow[dr] & \\
\begin{array}{c}\text{General typing}\\ \Gamma;\rho \vdash x : T\end{array} \arrow[r, Leftrightarrow] & \begin{array}{c}\text{Tight typing}\\ \Gamma;\rho \vdash_{\#} x : T\end{array} \arrow[r, Leftrightarrow] \arrow[ur] & \begin{array}{c}\text{Invertible typing}\\ \Gamma;\rho \vdash_{\#\#} x : T\end{array} \arrow[r] \arrow[u] \arrow[ul] & \begin{array}{c}\text{Precise typing}\\ \Gamma;\rho \vdash_{!} x : T\end{array}
\end{tikzcd}$$

Figure 5 Dependencies ($\rightarrow$) and equivalences ($\Leftrightarrow$) between definitions of typing in roDOT.

▶ **Lemma 12** (Read-only types). $\Gamma;\rho \vdash \mathsf{N} <: T \Leftrightarrow \Gamma;\rho \vdash T \ \mathbf{ro}\ T$.

4.2.1 Updating the safety proof

The changes described above require updating the type safety proof of the calculus, to show that the changes did not allow invalid programs to be typed. 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 proof of DOT [30]. 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.

The proof was based on two alternative definitions of typing for variables – tight typing and invertible typing. Figure 5 shows the relations between the different versions of typing.

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 10 in the appendix) 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.

4.2.2 Invertible Typing[†]

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 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 structure of the target type.

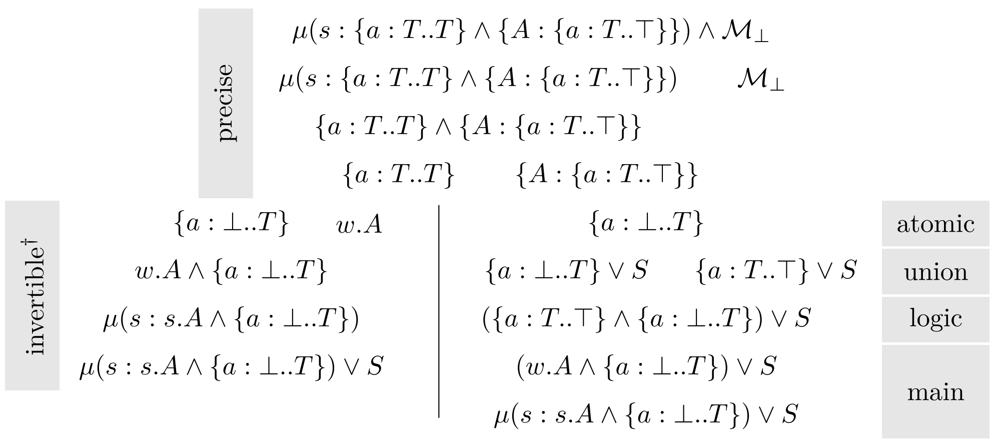
precise
$\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\}) \wedge \mathcal{M}_\bot$
$\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\})$
$\mathcal{M}_\bot$
$\{a : T..T\} \wedge \{A : \{a : T..\top\}\}$
$\{a : T..T\}$
$\{A : \{a : T..\top\}\}$
invertible†
$\{a : \bot..T\}$
$w.A$
$w.A \wedge \{a :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

Figure 6 Example derivation of a type by invertible typing (left) and layered typing (right). Assuming that a variable w has the type $\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\}) \wedge \mathcal{M}_\bot$ in the typing context, w has all the types shown here, ordered from types that are simple to derive at the top, to more complex derivations, which make use of derivations above. S is an arbitrary type.

The original roDOT adopted the invertible typing from DOT [30], where it has two layers, which we present using an example derivation of a type for a variable w in Figure 6.

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 6 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 6 shows individual steps of this process in building up the type $\mu(s : s.A \wedge \{a : \bot..T\}) \vee S$ for w.

As per Figure 5, invertible typing was equivalent to tight typing. That required invertible typing to be closed under tight subtyping (Lemma† 13).

▶ **Lemma† 13** (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 addition of ST-NM, together with the rules ST-Or and ST-Dist (Figure 2), breaks this. In Lemma† 13, the case for ST-Or relies on case analysis of deriving union types (Lemma† 14).

▶ **Lemma† 14** (In original roDOT, typing with union types can be inverted)**.**
If $\Gamma;\rho \vdash_{\#\#} x : T_3 \vee T_4$, then either $\Gamma;\rho \vdash_{\#\#} x : T_3$ or $\Gamma;\rho \vdash_{\#\#} x : T_4$.

However, the rule ST-NM adds new ways of deriving union types such as $\mathsf{N} \vee \mathcal{M}_\bot$, 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 M somewhere within them.

For example, with the variables from Example† 10, we have $\Gamma;\rho \vdash w_2 : \{a : T_\mathrm{a}\} \vee \mathcal{M}_\bot$, but $\Gamma;\rho \nvdash w_2 : \{a : T_\mathrm{a}\}$ and $\Gamma;\rho \nvdash w_2 : \mathcal{M}_\bot$. Such a type cannot be derived in a syntactically bottom-up manner that invertible typing is based on. Rather than trying to fix invertible typing by adding complicated rules, we replace it by a new auxiliary typing judgment, which derives types in different way.

4.3 Layered Typing

In *layered typing*, we avoid the need for Lemma† 14 by organizing the derivation of a type not bottom-up, but by handling different type constructors in separate layers of typing judgments. All union type constructors are derived before intersection types, recursive types and type selections. Additionally, we derive union types on two layers:

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 $\{\mathsf{M} : \bot..T\}$ for some bound T. We will write that as $\vdash \mathsf{N} \triangleleft T_\mathrm{N}$ and $\vdash \mathcal{M} \triangleleft T_\mathrm{M}$. 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 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 $\{a_1 : T_1\} \vee \mathcal{M}_\bot$ as $C^\vee[\{a_2 : T_2\}]$ and $C^\vee[\mathcal{M}_\bot]$. 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 6, 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.

The rest of the type constructors are handled either below the basic layer or above the logic layer. Subtyping between declarations is handled in an *atomic layer* positioned before the basic layer. This layer only deals with types that are single declarations.

Recursive types of the form $\mu(s : T)$ and selection types can "wrap around" or replace any part of the derived type (in general typing by VT-RecI and ST-SelL, Figures 9 and 10 in the appendix), which may both appear under unions and intersections, and also contain them within. Therefore, they are handled above the logic layer in a final, *main layer.* In the rule LTM-Sel in Figure 7, the syntactic context $C^{\wedge\vee}$ can consist of a mixture of unions and intersections. The rules have premises that correspond to conditions in the relevant rules of tight typing. For example, in Figure 6, the left side of the union is wrapped under a recursive type in the last step.

The layers are summarized in Table 1, showing the relevant type constructors and the connection to rules of general typing. Selected rules are shown in Figure 7; full definitions are in Figures 15–19 in the appendix.

Table 1 The layers of layered typing.

Typing layer	**Relevant type constructors**	**Relevant rules**
Atomic layer	$\{a : T..U\}$, $\{A : T..U\}$, $\{m(S,T) : U\}$	ST-Met, ST-Fld, ST-Typ
Basic layer	$\mathsf{N} \vee \mathcal{M}$	ST-NM
Union layer	$\top$, $T \vee U$	ST-Or1, ST-Or2, ST-Top
Logic layer	$T \wedge U$	ST-Dist, ST-And, VT-AndI
Main layer	$\mu(s : T)$, $x.A$	VT-RecI, ST-SelL, ST-N-Rec

$$\frac{\begin{array}{c}\Gamma;\rho \vdash_{\mathrm{l}} x : C^{\vee}[T_1] \\ \Gamma;\rho \vdash_{\mathrm{l}} x : C^{\vee}[T_2]\end{array}}{\Gamma;\rho \vdash_{\mathrm{l}} x : C^{\vee}[T_1 \wedge T_2]}\text{(LTL-And)} \qquad \frac{\begin{array}{c}\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[[v_3/r]T_1] \\ \Gamma \vdash_{!} v_2 : \{B(r) : T_1..T_2\}\end{array}}{\Gamma;\rho \vdash_{\mathrm{m}} x : C^{\wedge\vee}[v_2.B(v_3)]}\text{(LTM-Sel)}$$

Figure 7 Selected rules of layered typing.

- Typing on the **atomic layer** ($\Gamma;\rho \vdash_{\mathrm{a}} x : T$) only gives variables single declaration types – the declarations derived by precise typing, and their supertypes (subtyping rules between declarations are handled here).
- The **basic layer** ($\Gamma;\rho \vdash_{\mathrm{b}} x : T$) additionally gives all variables top-like types of the form $T_{\mathrm{N}} \vee T_{\mathrm{M}}$ and $T_{\mathrm{M}} \vee T_{\mathrm{N}}$, where $\vdash \mathsf{N} \lhd T_{\mathrm{N}}$ and $\vdash \mathcal{M} \lhd T_{\mathrm{M}}$.
- The **union layer** ($\Gamma;\rho \vdash_{\mathrm{u}} x : T$) handles $\top$ and unions of known and arbitrary types.
- The **logic layer** ($\Gamma;\rho \vdash_{\mathrm{l}} x : T$) handles intersections and distributivity. The rule LTL-And takes two 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.
- The **main layer** ($\Gamma;\rho \vdash_{\mathrm{m}} x : T$) closes recursive types and handles type selections.

For layered typing, we also proved the following properties:

- If a location has a declaration type by layered typing, then it also has a declaration type by precise typing, with the same or tighter bounds. This property has three variants, for field, type and method declarations.
- Layered typing is equivalent to general typing. As in the original proof, we use tight typing as a step between general and layered typing, and separately prove both directions of equivalence between tight and layered typing (Lemma 31 and Lemma 37 in the appendix).
- We also use layered typing to prove Lemma 9 – 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. Formal statements of these and other selected properties are given in Section A.3 in the appendix.

5 The SEF Guarantee

In Section 3.3, we informally stated the SEF guarantee, which provides the connection between a static typing condition (Definition 11) and run-time behavior of the method.

In this section, we present the formal definitions of the run-time SEF condition (Definition 15 in Section 5.1) and the SEF guarantee (Section 5.2). We then outline the proof of the guarantee (Section 5.3) and discuss some details of the proof (Section 5.4).

5.1 The run-time SEF condition

We informally stated the run-time SEF condition in Section 3.1, 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 in which 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 15). The other possible variants are stated as Definition 42 and Definition 43 in the appendix.

▶ **Definition 15** (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.*

5.1.1 Method call limits

Because we are defining a condition on what can happen while a method is executing, we need to understand what it means in roDOT that a method starts and ends its execution.

In roDOT, a method is called by a term $w_1.m\, w_2$. 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. 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, 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.

5.2 The SEF guarantee

The SEF guarantee, informally stated in Section 3.3, says that a SEF method does not modify existing objects in the heap during its execution. Theorem 16 is based on Definition 15, 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 16.** *Let the configuration $c_1 := \langle w_1.m\ w_2; \sigma_1; \rho_1; \Sigma_1\rangle$ be well-typed in a context Γ. Further assume that $\Gamma \vdash w_1 : \{m(z : \mathsf{N})(r : \mathsf{N}) : \top\}$. Then for any k steps of execution:*

1. Either the method call has finished executing. *There is $j < k$ for which $c_1 \mapsto^j \langle \mathsf{v}w_3; \sigma_1; -; -\rangle$.*
2. Or, the method call has not finished executing and in this period existing objects in the heap are unchanged. *For each $c_1 \mapsto^k c_2$, all heap locations in c_1 also exist in c_2 and moreover they are unchanged in c_2.*

5.3 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 2), which states that individual objects can only be modified through mutable references. This guarantee was proven for roDOT [12] and is included in 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.

First, note that the stack is not relevant to how the method executes and stays the same from the method start until its end. We therefore remove this stack entirely, and get an execution isolated from the rest of the program. This execution proceeds through the same steps, 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. If we apply the IG to this configuration, it will guarantee that objects are not modified until the end of the method, 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, because even when a method is SEF, it can be useful to pass mutable references to the method and have it return one of these references with its mutability intact.

What is special about a SEF method is that (because of its declared parameter types) it cannot use the mutability during its execution. Therefore, when called with mutable arguments, it should execute in exactly the same way as if 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 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, meaning the original method execution also does not modify any existing object on the heap.

5.4 Proof of the SEF Guarantee

The strategy of the proof of Theorem 16 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 16.

▶ **Lemma 17** (SEF guarantee without stack). *For c_1 satisfying the conditions of Theorem 16, let $c_1' \coloneqq \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1 \rangle$. If $c_1' \mapsto^n c_2'$ for some n and c_2', then 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_1'. The premise of the immutability guarantee is that c_1' is well-typed in some context Γ_2 and there are no mutably reachable objects in c_1' with respect to the typing of Γ_2. Clearly c_1' 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 might not make use of its mutability. Therefore, instead of using Γ as Γ_2, we construct Γ_2 and show that c_1' 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'_1.
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.

We do this instead of changing the types of w_i, because we only know that w_i are used in a read-only way in the focus, while on the heap, w_i might be used as a part of dependent types referring to their mutability. Changing their types would break the typing of the heap.

The second step is essential, because only references can be read-only, while locations always have mutable types. The references w_i are added in the same order as in the original context, to ensure that types in the typing context only refer to preceding variables in case the types are dependent. The two steps above correspond to the following two lemmas.

▶ **Lemma 18** (Reference elimination). *Let c_1 and Γ satisfy the conditions of Theorem 16, and $\rho_1[w_i] = y_i$. Define Γ' as the context obtained from Γ by first removing bindings for w_i and then replacing w_i with y_i. Define ρ' as the environment obtained from ρ_1 by removing bindings for w_i. Then the term $y_1.m\ y_2$ is well-typed under $\Gamma';\rho'$ and we have heap correspondence $\Gamma';\rho' \sim [y_i/w_i]_i\Sigma_1$.*

Proof. Because w_i is a reference to 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 19** (Read-only weakening). *Let c_1 and Γ satisfy the conditions of Theorem 16, and y_1, y_2 be such that $\rho_1[w_i] = y_i$. Then there is a context Γ_2 binding w_i to read-only types such that the configuration $c''_1 := \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i\Sigma_1\rangle$, is well-typed in Γ_2 (formally $\Gamma_2 \vdash c''_1 : \top$).*

By Lemma 19 along with the immutability guarantee, we have SEF established for $c''_1 := \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i\Sigma_1\rangle$. However, we need SEF in particular for the configuration $c'_1 := \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1\rangle$ in Lemma 17, where there is no substitution $[y_i/w_i]_i$ in the heap. Nevertheless, this substitution can be ignored in the sense that execution can only change fields of objects, but in roDOT, fields always store locations while w_i are references. (When a reference is assigned to a field, the corresponding location is stored, in order to make the field type determine the mutability of the stored value.) 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. We need to show that the two executions are almost the same, except some specific variables that appear in the machine configuration can differ. In particular, the references w_i can be replaced by the locations y_i. Additionally, the locations and references of newly created objects can differ, because variable names are not assigned deterministically.

For that reason, we define the *similarity* relation, which formalizes structural equivalence of syntactic elements such as terms, objects or whole configurations that differ only in names of variables. It relates two such elements using a *renaming* relation over variables. A formal definition of similarity and its basic properties is given in Section A.5 in the appendix.

Similarity has two important properties with respect to program execution: (1) it is preserved by reduction, and (2) if a machine configuration can reduce, then all similar configurations can reduce too (and the results are similar). That means that if we start with two similar configurations, where the execution of one reaches an answer state, then the execution of the other will reach a similar answer state. With this definition of similarity, Lemma 20 formalises the idea that c''_1 is similar to c'_1 up to renaming w_i to y_i:

▶ **Lemma 20** (Similarity for eliminated references)**.** *Let c'_1 satisfy the conditions of Lemma 17 and c''_1 the conditions of Lemma 19. Then $c'_1 \overset{(w_1,y_1),(w_2,y_2)}{\approx} c''_1$.*

The final part of carrying over the SEF property to c'_1 is to recognize that reduction only changes values of fields (while the structure of the object, methods and type members are immutable).

▶ **Definition 21** (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 22** (Reduction only changes fields)**.** *If $\langle -; -; -; \Sigma\rangle \mapsto^n \langle -; -; -; \Sigma'\rangle$ and y is a location in Σ, then $\Sigma(y) \overset{\mathsf{fld}}{\approx} \Sigma'(y)$.*

And with this, we can finish the proof of Theorem 16.

Proof. By classical reasoning, assume the condition 1 is false so that the goal is to prove the condition 2. 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 $c_1 \mapsto ... \mapsto c_2$ corresponds to a sequence of reductions $c'_1 \mapsto ... \mapsto c'_2$ because even though c'_1 has no awaiting frames, there are no frame pops in this execution sequence by the current assumption. By Lemma 17, the condition 2 follows. ◀

6 Transformations

In Section 3.4, we informally stated the transformation guarantee, which connects the static SEF condition with a practical application – that calls to SEF methods can be safely swapped.

Defining the call-swapping and the guarantee in a formal way requires dealing with several technicalities particular to the roDOT calculus (or DOT calculi in general). In order to separate the common problems from the specific case of swapping calls, we build a general framework in which various transformations of roDOT programs can be defined and proven safe. We instantiate it here only with the call swapping transformation, but it could represent the general part of a proof of safety for other transformations, such as reordering field reads or removing dead code.

In Section 6.1, we present the framework for defining and reasoning about safe program transformations in roDOT and similar calculi, including a general Theorem 25 about safety of transformations. In Section 6.2, we define the transformation that swaps two calls to methods that are statically determined to be side-effect free (Definition 26) and state the transformation guarantee.

6.1 Transformation framework

The framework defines a general form for roDOT program transformations, defines the precise meaning of a *safe transformation* that "does not affect the behavior of the program", and provides a way of proving this property, while helping to deal with common technical issues of DOT formalizations.

Our general approach 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.

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 $\mathsf{let}\, x = \nu(r : R)\{m(r, z) = t_m\}\, \mathsf{in}\, t_2$. We must consider that in roDOT, terms are in A-normal form, and that a term can have multiple different types. Also code (terms) and values (objects) are mixed with each other during execution of a program – the program contains object literals, and methods on the heap contain program code.

During execution, objects are created on the heap, including copies of the code of their methods, which can be affected by the transformation. 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 general safety Theorem 25 is based on executing the two programs and observing that the intermediate states are also related by the transformation (lifted to whole configurations 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.

A more detailed description of the framework design and its definitions are given in Section A.7 in the appendix. The following text describes the most important parts.

6.1.1 Transformations of roDOT programs in general

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, we define the transformation as a binary relation between triples: the term, its type and a typing context. This generalizes to other syntactic elements – stacks, heaps and machine configurations, though for each kind of element, the exact meaning of “typing context” and “type” may differ. For terms, the typing context is actually paired with a runtime environment $\Gamma; \rho$.

▶ **Definition 23** (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.*

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.

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 lift τ which takes a local transformation τ and allows it to be located at one place anywhere in a term.

Such a local transformation of a term can be further lifted by cfg τ 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 25 in Section A.7.4 in the appendix, where we want to look at each occurrence individually. More details about lifting are in Section A.7.3; the definitions of the lifting operators are shown in Figure 23 and Figure 24.

6.1.2 Safe transformations

The definition of a safe transformation must allow the answers of the two programs to contain different variable names and allow for the fact that the transformation can occur in the heap of the program answer, possibly at multiple places. Therefore a local transformation is safe if execution of the transformed program reaches answers related by an iteration of this transformation. To deal with non-deterministic location names, the transformation is in union with a *similarity transformation* $\approx$, which allows one-to-one variable renaming.

▶ **Definition 24** (Safe transformation). *A transformation τ is safe if $\langle \Gamma_1 \vdash c_1 : T\rangle \to_\tau \langle \Gamma_2 \vdash c_2 : T\rangle$ and $c_1 \longrightarrow^k c_3$, where c_3 is an answer typed as $\Gamma_3 \vdash c_3 : T$, implies that there exist c_4, Γ_4 and j such that $c_2 \longrightarrow^j c_4$, $\Gamma_4 \vdash c_4 : T$ and $\langle \Gamma_3 \vdash c_3 : T\rangle \to_{(\tau\cup\approx)^*} \langle \Gamma_4 \vdash c_4 : T\rangle$.*

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 27), or analogous guarantees for other local transformations.

Theorem 25 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. Full definitions of the premises are in Section A.7 in the appendix as Definitions 49, 50, 52, 56 and 54.

▶ **Theorem 25** (General safety for local transformations). *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.*

6.2 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 24.

Call-swapping is defined as a local transformation 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 $\mathsf{let}\, x_{c1} = x_{o1}.m_1 x_{a1}\,\mathsf{in}\,\mathsf{let}\, x_{c2} = x_{o2}.m_2 x_{a2}\,\mathsf{in}\, t$. In the transformation, the two calls $x_{o1}.m_1 x_{a1}$ and $x_{o2}.m_2 x_{a2}$ appear in the opposite order, but the continuation t is the same.

The transformation is only safe if both the methods are SEF, so it has several typing premises, analogous to the ones of Theorem 16 in Section 5.2.

▶ **Definition 26** (Local call swapping). *The local call-swapping transformation csw is a transformation of terms that relates $\langle \Gamma \vdash \mathsf{let}\, x_{c1} = x_{o1}.m_1 x_{a1}\,\mathsf{in}\,\mathsf{let}\, x_{c2} = x_{o2}.m_2 x_{a2}\,\mathsf{in}\, t : T\rangle \to_{\mathsf{csw}} \langle \Gamma \vdash \mathsf{let}\, x_{c2} = x_{o2}.m_2 x_{a2}\,\mathsf{in}\,\mathsf{let}\, x_{c1} = x_{o1}.m_1 x_{a1}\,\mathsf{in}\, t : T\rangle$ when*

- *$x_{c1,2}$ are distinct from $x_{a1,2}$ and $x_{o1,2}$,*
- *$\Gamma \vdash x_{o1}.m_1 x_{a1} : T_{c1}$, $\Gamma \vdash x_{o2}.m_2 x_{a2} : T_{c2}$, and $\Gamma, x_{c1} : T_{c1}, x_{c2} : T_{c2} \vdash t : T$,*
- *$\Gamma \vdash x_{o1} : \{m_1(r_1 : \mathsf{N}, z_1 : T_{a_1}) : \top\}$, and $\Gamma \vdash x_{o2} : \{m_2(r_2 : \mathsf{N}, z_2 : T_{a_2}) : \top\}$,*
- *$\Gamma \vdash x_{a1} : T_{a_1}$, and $\Gamma \vdash x_{a2} : T_{a_2}$,*
- *$\Gamma \vdash \mathsf{N} <: T_{a_1}$, and $\Gamma \vdash \mathsf{N} <: T_{a_2}$.*

As the the final form of the transformation guarantee, we apply Definition 24 to Definition 26, and specialize the theorem to initial programs. The proof is given in Section A.8 in the appendix.

▶ **Theorem 27** (Transformation guarantee). *If* $\langle \vdash t_1 : T\rangle \rightarrow_{\mathsf{lift\ csw}} \langle \vdash t_2 : T\rangle$ *and* $\langle t_1; \cdot; \cdot; \cdot\rangle \longrightarrow^k c_3$*, where* c_3 *is an answer typed as* $\Gamma_3 \vdash c_3 : T$*, then there exists* c_4*,* Γ_4 *and* j *such that* $\langle t_2; \cdot; \cdot; \cdot\rangle \longrightarrow^j c_4$*,* c_4 *is an answer typed as* $\Gamma_4 \vdash c_4 : T$ *and* $\langle \Gamma_3 \vdash c_3 : T\rangle \rightarrow_{(\mathsf{cfg\ csw} \cup \approx)^*} \langle \Gamma_4 \vdash c_4 : T\rangle$ *.*

7 Related work

Since the topic of this work includes both the DOT calculus and method purity, here we discuss previous work related to these concepts. Prior to this work, many variants of the DOT calculus were published, some including mechanized proofs. Also the issue of purity in object-oriented languages is of great research interest, and it is approached from different angles of automation and precision. We give details about the existing work in the following subsections. As far as we know, our work is the first one to consider the issue of purity within a DOT calculus.

7.1 Mechanizations of DOT calculi

The first appearance of a DOT calculus [3] did not include a proof of soundness, but was followed by several versions with proofs in Coq [33, 30] and Iris [17]. In particular, WadlerFest DOT [2], thanks to its simplicity and its proof of soundness based on invertible typing [30], was used as a baseline for numerous extensions [32, 22, 31, 23], including roDOT. While objects are immutable in WadlerFest DOT, it was extended to support mutation using mutable slots in Mutable WadlerFest DOT [32], and more directly by allowing changing values of fields in kDOT [22]. A simplified version [21] with mutable fields, but without the specific kDOT feature of constructors, was used as a base for the mechanization of roDOT.

The differences between the mechanization of roDOT and those of previous DOT calculi mainly stem from the differences in how roDOT handles variables – namely, typing of variables and terms being separated from each other, using different definitions of typing contexts to support variable hiding, using the runtime environment to map references to locations, and using typing information in its definition of operational semantics.

The mechanization of roDOT includes a feature to ease further extensions to the calculus. The definitions and theorems are parameterized by a "typing mode", which allows selecting type system features that are supported. Using this feature, our proofs work for roDOT both with and without the changes described in this paper.

7.2 Purity in other languages

Purity in programming is such an important concept that in many languages, functions are pure by default. This approach is typically associated with functional programming, but an object-oriented system can also be pure [1] when the objects are immutable. That is also the case in the basic DOT calculus.

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 is used in practical programming languages such as Haskell.

Regarding purity in object oriented languages with mutable fields, many publications [37, 34, 38, 40, 6, 16, 29] focus on Java and languages with similar type systems, such as C#. For Scala, a type system for purity was developed, but not based on the DOT calculus [35].

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.

Observational purity [25, 5] 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.

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. Code Contracts [15], JML [20] and Checker Framework [14] allow annotating a method as pure. Code Contracts do not check that this annotation is correctly applied, and JML and Checker Framework use simpler 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.

To avoid imposing an annotation burden on the programmer, purity can be inferred by automatic program analysis [26, 34], and side-effect analysis can be used for program optimization [11].

ReIm [19] 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 [12].

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** [28, 27] 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 [24] 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** [7] 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** [4] 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 [8], 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 [39]. 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.

8 Conclusion

To conclude, our paper confirms that the reference mutability system provided by roDOT can be mechanically proven sound, and with a few changes can be used to guarantee side-effect freedom of methods, and to justify safe transformations of programs.

References

1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. doi:10.1007/978-1-4419-8598-9.

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

3 Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, OOPSLA '14, pages 233–249. ACM, 2014. doi:10.1145/2660193.2660216.

4 Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: Tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485516.

5 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.

6 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.

7 Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. Capturing types. *ACM Trans. Program. Lang. Syst.*, 45(4), November 2023. doi:10.1145/3618003.

8 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), October 2023. doi:10.1145/3622813.

9 The Checker Framework Manual: Custom pluggable types for Java. URL: https://checkerframework.org/manual/#initialization-checker, 2022.

10 The Checker Framework Manual: Custom pluggable types for Java. URL: https://checkerframework.org/manual/#purity-checker, 2022.

11 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.

12 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.

13 Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek. Pure methods for roDOT (an extended version). Technical Report D3S-TR-2024-01, Dep. of Distributed and Dependable Systems, Charles University, 2024. URL: https://d3s.mff.cuni.cz/files/publications/dort_pure_report_2024.pdf.

14 Michael D. Ernst. Annotation type Pure. https://checkerframework.org/api/org/checkerframework/dataflow/qual/Pure.html, 2022.

15 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.

16 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. doi:10.1145/1455770.1455793.

17 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.

18 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.

19 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. doi:10.1145/2384616.2384680.

20 JML reference manual: Class and interface member declarations. https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_7.html#SEC60, 2022.

21 Ifaz Kabir. themaplelab / dot-public: A simpler syntactic soundness proof for dependent object types. https://github.com/themaplelab/dot-public/tree/master/dot-simpler.

22 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.

23 Ifaz Kabir, Yufeng Li, and Ondrej Lhoták. ιDOT: a DOT calculus with object initialization. *Proc. ACM Program. Lang.*, 4(OOPSLA):208:1–208:28, 2020. doi:10.1145/3428276.

24 Darya Melicher, Anlun Xu, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. Bounded abstract effects. *ACM Trans. Program. Lang. Syst.*, 44(1), January 2022. `doi:10.1145/3492427`.

25 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. `doi:10.1007/978-3-540-31984-9_15`.

26 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. `doi:10.1002/smr.1889`.

27 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, pages 1–11, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3486610.3486893`.

28 Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. Scoped capabilities for polymorphic effects, 2022. `arXiv:2207.03402`.

29 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`.

30 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`.

31 Marianna Rapoport and Ondrej 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`.

32 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. `doi:10.1145/3103111.3104036`.

33 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. `doi:10.1145/2983990.2984008`.

34 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. `doi:10.1109/ICSM.2004.1357793`.

35 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. `doi:10.1145/2489804.2489808`.

36 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 288–298, New York, NY, USA, 1992. Association for Computing Machinery. `doi:10.1145/141471.141563`.

37 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`.

38 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. doi:10.1007/978-3-540-30579-8_14.

39 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), January 2024. doi:10.1145/3632856.

40 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. doi:10.1145/1251535.1251548.