κDOT: Scaling DOT with Mutation and Constructors

Ifaz Kabir
University of Waterloo
Canada
ikabir@uwaterloo.ca

Ondřej Lhoták
University of Waterloo
Canada
olhotak@uwaterloo.ca

Abstract

Scala unifies concepts from object and module systems by allowing for objects with type members which are referenced via path-dependent types. The Dependent Object Types (DOT) calculus of Amin et al. models only this core part of Scala, but does not have many fundamental features of Scala such as strict and mutable fields. Since the most commonly used field types in Scala are strict, the correspondence between DOT and Scala is too weak for us to meaningfully prove static analyses safe for Scala by proving them safe for DOT.

A DOT calculus that can support strict and mutable fields together with constructors that do field initialization would be more suitable for analysis of Scala. Toward this goal, we present κDOT, an extension of DOT that supports constructors and field mutation and can emulate the different types of fields in Scala. We have proven κDOT sound through a mechanized proof in Coq. We present the key features of κDOT and its operational semantics and discuss work-in-progress toward making κDOT fully strict.

CCS Concepts • **Software and its engineering → Formal language definitions**;

Keywords dependent object types, type safety, mutation

ACM Reference Format:
Ifaz Kabir and Ondřej Lhoták. 2018. κDOT: Scaling DOT with Mutation and Constructors. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium (Scala '18), September 28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3241653.3241659

1 Introduction

Scala aims to be a scalable language where one can easily express ideas both large and small. Toward this goal, Scala unifies concepts from object and module systems by allowing objects to carry type members which are referenced by path dependent types. After a long and elusive search for a sound calculus that could model path-dependent types, the Dependent Object Types (DOT) family of calculi were proposed by Amin et al. [2012], and variants were later proven sound [Amin et al. 2016, 2014; Rompf and Amin 2016b].

While the DOT calculi of Rompf and Amin [2016b] and Amin et al. [2016] have notions of path-dependent types, they eschew many features of Scala to simplify their calculi and type soundness proofs. In particular, objects in the calculus of Rompf and Amin [2016b] only have methods and type members. Field reads are emulated by method calls, which roughly correspond to lazy semantics for field reads. The DOT calculus of Amin et al. [2016], which we call WadlerFest DOT, supports first class functions instead of methods. In this calculus, fields store terms which are evaluated each time the field is read, which again corresponds to lazy semantics. Both of these calculi evaluate function applications in a strict manner, but neither supports any form of field mutation. For this paper, we focus on extending WadlerFest DOT.

The choice of lazy fields simplifies type soundness proofs. By carefully defining the operational semantics, they allow objects to have recursive types without having to worry about field initialization. Scala however, provides many different types of fields:

val strict immutable
var strict mutable
lazy val memoized lazy immutable

The strict field types are much more commonly used than lazy vals. Since DOT uses lazy fields, for many different kinds of static analyses, the correspondence between DOT and Scala is too weak for us to meaningfully prove the analysis safe for Scala by proving it safe for DOT. Furthermore, lazy fields and the lack of field mutation limit the expressiveness of the calculus (see Section 6).

In this paper we present κDOT, an extension of WadlerFest DOT with constructors and field mutation. κDOT adds few typing rules to WadlerFest DOT while substantially improving its expressive power. In particular, our typing rules for mutation are significantly simpler than existing mutation extensions to DOT [Rapoport and Lhoták 2017; Rompf and Amin 2016a] which add syntax and typing rules for ML-style references. The operational semantics of κDOT is defined using an abstract machine with a stack and a heap, which corresponds more closely to the semantics of Scala.

Scala '18, September 28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5836-1/18/09…$15.00
https://doi.org/10.1145/3241653.3241659

```
trait Fruit[T] { type A = T }
trait Tree { tree =>
  type TreeFruit
  val fruit : Fruit[tree.TreeFruit] =
    new Fruit[tree.TreeFruit] {}
}
val mangoTree = new Tree{}
```

(a) Initializing a dependently typed object

```
trait FruityTree extends Tree { tree =>
  val fruits : List[Fruit[tree.TreeFruit]] =
    new Fruit[tree.TreeFruit] {} :: tree.fruits
}
val fruityError = new FruityTree{}
```

(b) An initialization error

Figure 1. A Motivating Example

1.1 A Motivating Example

Our main motivation for designing κDOT was to have a type safe calculus where we could explore initialization systems for Scala. In object-oriented programming languages like Scala, we care about initialization because we want our programs to be free from null-reference exceptions. In Scala, we create objects by first allocating them with fields containing nulls, then filling up the fields with locations of other objects. For instance, the code in Fig. 1b contains an initialization error. If we run this code, the field fruits will be accessed before being initialized, resulting in a null-reference exception. We developed κDOT as an extension of WadlerFest DOT where these problems could be studied.

A secondary motivation is that we wanted to explore the design space of a fully strict DOT calculus without adding sum types, null types, or exceptions to the calculus. Consider the program in Fig. 1a. As we will see in Section 5, WadlerFest DOT is not able to express the heap structure of this program — in WadlerFest DOT the field fruit must be lazy. We wanted to design a DOT calculus which did not have this limitation and we plan to evolve κDOT into a fully strict DOT calculus using initialization systems.

1.2 Contributions

In this paper we make the following contributions:

- We show how mutation and constructors can be added to WadlerFest DOT with minimal new constructs.
- We provide a mechanized proof of type safety for κDOT[1] as an extension to the type safety proof for WadlerFest DOT of Rapoport et al. [2017].
- We discuss the usefulness of such a calculus in relation to initialization.
- We discuss the design choices we made in κDOT.

The rest of the paper is organized as follows. In Section 2, we introduce the syntax and semantics of κDOT. Next, in Section 3, we outline the type safety proof and the changes we made to the type safety proof of WadlerFest DOT. In Section 4, we explore initialization problems in κDOT and discuss possible benefits of tracking initialization in DOT calculi. In Section 5, we compare the expressive power of κDOT to WadlerFest DOT. In Section 6, we discuss the design choices we made in κDOT and compare κDOT to other DOT calculi that have notions of mutation. We discuss related work in Section 7 before concluding in Section 8.

[1] Coq proof available at https://git.uwaterloo.ca/ikabir/dot-public

2 WadlerFest DOT with Constructors

In this section we explain the syntax and semantics of κDOT.

2.1 Syntax

The syntax for κDOT is given in Fig. 2 where we highlighted the parts that are different from Amin et al. [2016].

In κDOT, type labels are used for type members and term labels are used for fields. κDOT uses two different types of variables. Locations are variables that represent items in the heap and abstract variables are used for let bindings and for function and constructor parameters. κDOT has types for recursive objects and their fields, functions, and constructors as well as intersection types, a top type, and a bottom type. Function and constructor definitions are literals in κDOT which, together with objects, can be bound in the heap.

Objects in κDOT consist of a list of definitions. Note that the grammar for field definitions allows fields to contain arbitrary terms. As we will see in Section 2.2, field reads in this calculus can reduce to arbitrary terms. Field assignment, field reads, constructor calls, and function calls are all written in ANF in κDOT.

The following are some additional differences from Amin et al. [2016] which are not highlighted in Fig. 2.

- We removed objects from the grammar for literals. Objects in κDOT are created by calling a constructor.
- We removed literals from the grammar for terms. Literals in κDOT must be let bound; we cannot directly write a term which represents a literal.
- In WadlerFest DOT, the scope of z in $\nu\,(z\colon T)\,d$ is both T and d. In κDOT, the scope of z is only T.
- Frames, stacks, heaps, and configurations were not used in WadlerFest DOT.
- We define answers to be a location together with a heap and an empty stack. In particular literals are not answers since they must be let bound in κDOT.

2.1.1 New Constructs and Terminology

For a field declaration $\{a\colon S..T\}$, we call S the *setter type* of a and T the *getter type* of a. For a constructor $\kappa\left(\overrightarrow{z\colon T}, z_1\colon U\right)\{d\}\,t$ we call d the set of *default definitions* and t the *body* of the constructor. The variable z_1 can be thought of as the this or self variable of traditional object oriented languages. d and t may both refer to z_1.

Labels and Variables

a, b, c	Term Labels
A, B, C	Type Labels
y	Locations
z	Abstract Variables
$x, k ::= y \mid z$	Variables

Types

$S, T, U ::= \top \mid \bot$	Top and Bottom types
$\mid \forall (z\colon S)\, T$	Dependent Function
$\mid \mu (z\colon T)$	Recursive Type
$\mid \{a\colon S..T\}$	Field Declaration
$\mid \{A\colon S..T\}$	Type Declaration
$\mid x.A$	Type Projection
$\mid S \wedge T$	Type Intersection
$\mid K\left(\overrightarrow{z\colon T}, z_1\colon T\right)$	Constructor Type

Terms

$t, u ::= x$	Variable
$\mid \mathsf{new}\ k\ (\overrightarrow{x})$	Constructor Call
$\mid x.a \mid x.a := x_1$	Field Read, Field Write
$\mid x\, x_1$	Application
$\mid \mathsf{let}\ z = l\ \mathsf{in}\ u$	Literal Binding
$\mid \mathsf{let}\ z = t\ \mathsf{in}\ u$	Let Binding

Literals and Heap Items

$l ::= \lambda (z\colon T)\, .t$	Lambda
$\mid \kappa\left(\overrightarrow{z\colon T}, z_1\colon U\right)\{d\}\, t$	Constructor
$h ::= l$	Literal
$\mid \nu (z\colon T)\, d$	Object

Definitions

$d ::= \{a = t\}$	Field Definition
$\mid \{A = T\}$	Type Definition
$\mid d \wedge d'$	Aggregate Definition

Frames, Stacks, Heaps, and Configurations

$F ::= \mathsf{let}\ z = \square\ \mathsf{in}\ t$	Let Frame
$\mid \mathsf{return}\ y$	Return Frame
$s ::= \varepsilon \mid F :: s$	Stacks
$\Sigma ::= \cdot \mid \Sigma, y = h$	Heap
$c ::= \langle t; s; \Sigma\rangle$	Configuration
$n ::= \langle y; \varepsilon; \Sigma\rangle$	Answer

Figure 2. Syntax of κDOT

$$\frac{y = \nu (z\colon T) \ldots \{a = t\} \ldots \in \Sigma}{\langle y.a;\, s;\, \Sigma\rangle \longmapsto \langle t;\, s;\, \Sigma\rangle} \quad \textsc{(Project)}$$

$$\frac{y = \nu (z\colon T) \ldots \{a = t\} \ldots \in \Sigma \qquad \Sigma' = \Sigma\,[y = \nu (z\colon T) \ldots \{a = y_1\} \ldots]}{\langle y.a := y_1;\, s;\, \Sigma\rangle \longmapsto \langle y_1;\, s;\, \Sigma'\rangle} \quad \textsc{(Assignment)}$$

$$\frac{y = \lambda (z\colon T)\, .t \in \Sigma}{\langle y\, y_1;\, s;\, \Sigma\rangle \longmapsto \langle [y_1/z]\, t;\, s;\, \Sigma\rangle} \quad \textsc{(Application)}$$

$$\frac{\overrightarrow{y_2} = \overrightarrow{y, y_1} \qquad \overrightarrow{z_2} = \overrightarrow{z, z_1} \qquad k = \kappa\left(\overrightarrow{z\colon T}, z_1\colon U\right)\{d\}\, t \in \Sigma}{\langle \mathsf{new}\ k\ (\overrightarrow{y});\, s;\, \Sigma\rangle \longmapsto \left\langle \overrightarrow{[y_2/z_2]}t;\ \mathsf{return}\ y_1 :: s;\ \Sigma,\ y_1 = \nu\left(z_1\colon \overrightarrow{[y/z]}U\right)\overrightarrow{[y_2/z_2]}d\right\rangle} \quad \textsc{(New)}$$

$$\langle y_1;\, \mathsf{return}\ y :: s;\, \Sigma\rangle \longmapsto \langle y;\, s;\, \Sigma\rangle \quad \textsc{(Return)}$$

$$\langle y;\, \mathsf{let}\ z = \square\ \mathsf{in}\ t :: s;\, \Sigma\rangle \longmapsto \langle [y/z]\, t;\, s;\, \Sigma\rangle \quad \textsc{(Let-Loc)}$$

$$\langle \mathsf{let}\ z = l\ \mathsf{in}\ u;\, s;\, \Sigma\rangle \longmapsto \langle [y/z]\, t;\, s;\, \Sigma,\, y = l\rangle \quad \textsc{(Let-Lit)}$$

$$\langle \mathsf{let}\ z = t\ \mathsf{in}\ u;\, s;\, \Sigma\rangle \longmapsto \langle t;\, \mathsf{let}\ z = \square\ \mathsf{in}\ u :: s;\, \Sigma\rangle \quad \textsc{(Let-Push)}$$

Figure 3. Operational Semantics for κDOT

2.1.2 Abbreviations

To simplify our discussion, we will use some abbreviations for declarations. For field declarations of the form $\{a\colon T..T\}$ we will write $\{a\colon T\}$, and for type declarations of the form $\{A\colon T..T\}$ we will write $\{A\colon T\}$.

2.2 Operational Semantics

We give the κDOT calculus an operational semantics via an abstract machine (Fig. 3). A configuration of the machine consists of a term t, a stack s, and a heap Σ; where t is the current focus of execution, s is a list of frames representing the current evaluation context, and Σ binds locations to literals and objects. To execute a κDOT term t we initialize the machine with $\langle t;\, \varepsilon;\, \cdot\rangle$ and then start the machine. The machine then either runs indefinitely, finishes executing and returns an answer in the form of a location pointing to a heap item together with an empty stack, or gets stuck.

We note that the operational semantics is deterministic up to renaming of variables. This is because for each possible shape of the term t and stack s in a configuration, there is exactly one possible rule that can apply.

2.2.1 Frames and the Stack

Frames are of two kinds: let frames and return frames. Let frames represent the continuation of executing let terms and return frames represent the continuation of constructor calls.

Frames are pushed onto the stack by either executing a let binding, or by calling a constructor. Executing a let $z =$

t in u pushes the frame let $z = \square$ in u onto the stack via the (Let-Push) rule and starts executing the term t. The $\square$ represents the hole of the let binding's evaluation context. If t executes down to a location y, the (Let-Loc) rule pops the frame from the stack and executes u.

If a constructor $k = \kappa\left(\overrightarrow{z : T}, y : U\right)\{d\}\, t$ is bound in the heap Σ, a constructor call new $k\,(\vec{x})$ does three different things via the (new) rule. Firstly, it allocates a new object y with the default definitions d as its fields in the heap. Secondly, it pushes a return frame return y returning the location of the newly allocated object onto the stack. Lastly, it runs the body of the constructor t. After the body of the constructor finishes executing, the (Return) rule pops the return frame and returns the location of the object.

We note that a return frame can be represented by a let frame of the form let $z = \square$ in y, but we made return frames explicit since we were interested in initialization systems.

2.2.2 Field Reads and Writes

The (Project) rule causes a field read $y.a$ to evaluate to the term t that is bound at $y.a$. Note that if the field is not mutated between subsequent reads, t is reevaluated at each read. Thus field reads are lazy without memoization.

A field write $y.a := y_1$ is evaluated using the (Assignment) rule. This mutates the heap so that $y.a$ contains the location y_1 after execution and returns y_1.

2.2.3 Literals and Applications

A let bound literal is evaluated with the (Let-Lit) rule, which binds the literal to a new location in the store and substitutes references to the location with the new location.

The (Application) rule evaluates Function applications $y\, y_1$. An application evaluates to the body of the function at location y with an appropriate substitution.

3 Type Safety

Fig. 4, Fig. 5, Fig. 6, and Fig. 7 show the typing rules for κDOT. Rules that are new or different from WadlerFest DOT [Amin et al. 2016] are highlighted. Given our operational semantics, type safety says that if we start our abstract machine with a well-typed term then the machine either runs indefinitely or successfully returns an answer.

3.1 Type Safety for κDOT

Theorem 3.1 (Type Safety). *If* $\vdash t : T$, *then either the initial configuration of the abstract machine* $\langle t;\, \varepsilon;\, \cdot\rangle$ *diverges or* $\langle t;\, \varepsilon;\, \cdot\rangle \longmapsto^* \langle x;\, \varepsilon;\, \Sigma\rangle$ *for some answer* $\langle x;\, \varepsilon;\, \Sigma\rangle$.

We prove type safety of κDOT by proving Wright and Felleisen [1994] style progress and preservation lemmas. To express progress and preservation we needed to extend typing to heaps, stacks, and configurations.

Definition 3.2 (Heap Correspondence). For a context Γ and an environment Σ, we say that Γ *corresponds* to Σ, written

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{(Var)}$$

$$\frac{\Gamma \vdash x :\ \{a : T..U\}}{\Gamma \vdash x.a : U}\ (\{\}\text{-E}) \qquad \frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)}\ (\text{Rec-I})$$

$$\frac{\Gamma \vdash x_1 : T \quad \Gamma \vdash x :\ \{a : T..U\}}{\Gamma \vdash x.a := x_1 : U}\ (:=\text{-I}) \qquad \frac{\Gamma \vdash x : \mu(z : T)}{\Gamma \vdash x :\ [x/z]\,T}\ (\text{Rec-E})$$

$$\frac{\Gamma \vdash l : T \qquad x \notin \mathsf{fv}(U) \quad \Gamma, x : T \vdash u : U}{\Gamma \vdash \mathsf{let}\ x = l\ \mathsf{in}\ u : U} \quad \text{(Lit-I)}$$

$$\frac{\Gamma \vdash x : \forall(z : T)\, U \qquad \Gamma \vdash x_1 : T}{\Gamma \vdash x\, x_1 :\ [x_1/z]\,U} \quad \text{(All-E)}$$

$$\frac{\Gamma \vdash k : K\left(\overrightarrow{z : T}, z_1 : U\right) \qquad \Gamma \vdash \overrightarrow{x : T}}{\Gamma \vdash \mathsf{new}\ k\,(\vec{x}) : \mu(z_1 : U)} \quad \text{(K-E)}$$

$$\frac{\Gamma \vdash t : T \qquad x \notin \mathsf{fv}(U) \quad \Gamma, x : T \vdash u : U}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : U} \quad \text{(Let)}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad \text{(And-I)}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \quad \text{(Sub)}$$

Figure 4. Typing in κDOT

$$\frac{\Gamma, x : T \vdash t : U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash \lambda(x : T).t : \forall(x : T)\, U} \quad \text{(All-I)}$$

$$\frac{\Gamma, \overrightarrow{x : T}, x_1 : U \vdash d : U \qquad \vec{x} \notin \mathsf{fv}\left(\vec{T}\right) \quad \Gamma, \overrightarrow{x : T}, x_1 : U \vdash t : T'}{\Gamma \vdash \kappa\left(\overrightarrow{x : T}, x_1 : U\right)\{d\}\, t : K\left(\overrightarrow{x : T}, x_1 : U\right)} \quad \text{(K-I)}$$

Figure 5. Literal Typing in κDOT

$$\Gamma \vdash \{A = T\} :\ \{A : T..T\} \quad \text{(Def-Typ)}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} :\ \left\{a :\ T..\ T\right\}} \quad \text{(Def-Trm)}$$

$$\frac{\Gamma \vdash d_1 : T_1 \qquad \Gamma \vdash d_2 : T_2 \quad \mathsf{dom}(d_1), \mathsf{dom}(d_2)\ \text{disjoint}}{\Gamma \vdash d_1 \wedge d_2 : T \wedge U} \quad \text{(And-Def)}$$

Figure 6. Definition Typing in κDOT

$$\Gamma \vdash T <: \top \quad (\textsc{Top}) \qquad \Gamma \vdash T \wedge U <: T \ (\textsc{And}_1\text{-}<:)$$

$$\Gamma \vdash \bot <: T \quad (\textsc{Bot}) \qquad \Gamma \vdash T \wedge U <: U \ (\textsc{And}_2\text{-}<:)$$

$$\Gamma \vdash T <: T \quad (\textsc{Refl}) \qquad \frac{\Gamma \vdash S <: T \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \ (<:\text{-}\textsc{And})$$

$$\frac{\Gamma \vdash x\colon \{A\colon S..T\}}{\Gamma \vdash S <: x.A} \ (<:\text{-}\textsc{Sel}) \qquad \frac{\Gamma \vdash x\colon \{A\colon S..T\}}{\Gamma \vdash x.A <: T} \ (\textsc{Sel}\text{-}<:)$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{a\colon T_1..U_1\} <: \{a\colon T_2..U_2\}} \ (\textsc{Fld}\text{-}<:\text{-}\textsc{Fld})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A\colon S_1..T_1\} <: \{A\colon S_2..T_2\}} \ (\textsc{Typ}\text{-}<:\text{-}\textsc{Typ})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x\colon S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall\,(x\colon S_1)\,T_1 <: \forall\,(x\colon S_2)\,T_2} \ (\textsc{All}\text{-}<:\text{-}\textsc{All})$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \ (\textsc{Trans})$$

Figure 7. Subtyping in κDOT

$$\frac{\Gamma \vdash S <: U}{\Gamma \vdash \varepsilon\colon S, U} \ (\textsc{Stack Empty})$$

$$\frac{\Gamma \vdash s\colon T, U \qquad x \notin \mathsf{fv}(T) \qquad \Gamma, x\colon S \vdash u\colon T}{\Gamma \vdash \mathsf{let}\ x = \square\ \mathsf{in}\ u :: s\colon S, U} \ (\textsc{Stack Let})$$

$$\frac{\Gamma \vdash s\colon T, U \qquad \Gamma \vdash x\colon T}{\Gamma \vdash \mathsf{return}\ x :: s\colon S, U} \ (\textsc{Stack Return})$$

Figure 8. Stack Typing in K-DOT

$\Gamma \sim \Sigma$, if Γ and Σ have the same domain, and for all $x : T \in \Gamma$ and $x = h \in \Sigma$

- if $h = \lambda\,(z\colon S)\,.t$, then $\Gamma \vdash h\colon T$ using the (All-I) rule.
- if $h = \kappa\left(\overrightarrow{z\colon S}, z_1\colon U\right)\{d\}\,t$, then $\Gamma \vdash h\colon T$ using the (K-I) rule.
- if $h = \nu\,(z\colon U)\,d$ for some object $\nu\,(z\colon U)\,d$, then $T = \mu\,(z\colon U)$ and $\Gamma \vdash d\colon [x/z]\,U$

Fig. 8 shows the typing rules for stacks. Stacks represent evaluation contexts and are given two types $\Gamma \vdash s\colon S, U$. Here S is the type that the focus of execution must have for the overall evaluation context to have type U. The use of subtyping in the (Stack Empty) rule allows us to avoid defining subtyping between stacks and configurations. The (Stack Let) rule closely mirrors the (Let) typing rule. The (Stack Return) rule ensures that, after a constructor call, the stack is typed with the type of the allocated object.

Given heap correspondence and stack typing, we define configuration typing as follows.

Definition 3.3 (Configuration Typing). $\Gamma \vdash \langle t; s; \Sigma\rangle : U$ if $\Gamma \sim \Sigma$ and for some type T, $\Gamma \vdash t\colon T$ and $\Gamma \vdash s\colon T, U$.

Configuration Typing is defined for proving preservation for a small step operational semantics. In the above definition, U is the type of the term that the machine is started with, T is the type of the current focus of execution, and stack typing ensures that U is preserved when pushing and popping stack frames.

Our proof of progress and preservation extends the Rapoport et al. [2017] proof of progress and preservation for WadlerFest DOT. Following Rapoport et al., we define inert types, inert contexts and then prove progress and preservation.

Definition 3.4 (Record Type). A type T is a *record type* if T is the intersection of tight field declarations $\{a\colon S..S\}$, and tight type $\{A\colon S..S\}$ declarations of distinct labels. A field (type) declaration $\{a\colon S..U\}$ ($\{A\colon S..U\}$) is *tight* if its bounds S and U are the same.

Definition 3.5 (Inert Type). A type U is *inert* if

- U is a dependent function type $\forall\,(x\colon S)\,T$, or
- U is a constructor type $K\left(\overrightarrow{x\colon S}, y\colon T\right)$ for a record type T, or
- U is a recursive type $\mu\,(x\colon T)$ for a record type T.

Definition 3.6 (Inert Context). Γ is an *inert context* if the type $\Gamma\,(x)$ that it binds to each variable x is inert.

Lemma 3.7 (Progress). *If Γ is inert and $\Gamma \vdash c\colon U$, then either c is an answer or there exists c' such that $c \longmapsto c'$.*

Lemma 3.8 (Preservation). *If Γ is inert, $\Gamma \vdash c\colon U$, and $c \longmapsto c'$ then there exists Γ' such that the concatenation Γ++Γ' is inert and Γ++$\Gamma' \vdash c'\colon U$.*

An interesting feature of the Rapoport et al. preservation lemma is that it adds an inertness condition to the conclusion. We will explore runtime consequences of this in Section 4.3.

We get type safety from the above progress and preservation lemmas by noting that for $\vdash t\colon T$ the empty context is inert and types the initial configuration as $\vdash \langle t; \varepsilon; \cdot\rangle : T$.

3.2 Cofinite Quantification and Field Mutation

In WadlerFest DOT there are no constructors and objects are literals. In the mechanized proof of Amin et al. [2016], objects are typed using the following rule (cofinite quantification [Aydemir et al. 2008] is made explicit).

$$\frac{\forall x \notin L \qquad \Gamma, x\colon [x/z]\,U \vdash [x/z]\,d\colon [x/z]\,U}{\Gamma \vdash \nu\,(z\colon T)\,d\colon \mu\,(z\colon T)} \ (\{\}\text{-I})$$

To prove safety, Amin et al. [2016] used an idea similar to heap correspondence which they called *store correspondence*.

Definition 3.9 (Store Correspondence)**.** For a context Γ and an environment Σ, we say that Γ *corresponds* to Σ as a store, if Γ and Σ have the same domain, and for all $x : T \in \Gamma$ and $x = l \in \Sigma$, either

- $l = v\,(z\colon U)\,d$ and $\Gamma \vdash l\colon T$ using the ({}-I) rule, or
- $l = \lambda\,(z\colon S)\,.t$ and $\Gamma \vdash l\colon T$ using the (All-I) rule.

If we allow mutation in WadlerFest DOT, the main problem that we run into is that store correspondence may not be preserved by executing an assignment. We illustrate this through the following example (some types elided).

Suppose Γ corresponds to Σ as a store, and Σ only contains the following elements.

$$y = v\,(y\colon \{a\colon \{C\colon y.A\}\})\,(\{A = y.A\} \wedge \{a = t\})$$
$$y_1 = v\,(y_1\colon \_)\,\{C = y.A\}$$

After executing $y.a := y_1$, our heap contains

$$y = v\,(y\colon \_)\,(\{A = y.A\} \wedge \{a = y_1\})$$

Now, for store correspondence the following needs to hold.

$$\forall x \notin L \qquad \Gamma, x\colon \_ \vdash \{a = y_1\} : \{a\colon \{C = x.A\}\}$$

But $\Gamma \vdash y_1 : \{C = y.A\}$ for only y, not cofinitely many x's and hence the above does not hold. The requirement that objects be typeable with cofinitely many variables for store correspondence is too strict and we defined heap correspondence to relax this requirement for our preservation lemma.

The cofinite quantification in the ({}-I) rule essentially types an object by typing its definitions with all possible bindings for the object. WadlerFest DOT reuses object typing for store correspondence. The key insight that allows us to prove preservation for mutation in κDOT is that once an object has already been bound to the heap, we only need to type definitions with the current binding of the object, not all other possible bindings. Thus, unlike store correspondence which uses object typing via the ({}-I) to establish correspondence between objects and the context, heap correspondence only uses definition typing. For the above example, heap correspondence only requires the following.

$$\Gamma \vdash \{a = y_1\} : \{a\colon \{C = y.A\}\}$$

The Rapoport et al. proof uses a similar idea to store correspondence which they call *well-typed evaluation contexts* where similar issues appear.

4 Exploring Initialization in κDOT

We started working on κDOT to design systems that could warn programmers about null-reference errors in Scala. While Scala has null-references, they are absent from κDOT and other WadlerFest DOT variants. This may lead us to think that if we are happy with lazy field reads, we may not have to care about initialization in these calculi. In this section we define a notion of initialization in κDOT and discuss the bad bounds problem for DOT calculi to further motivate initialization in κDOT.

4.1 Initialized Locations

In object-oriented languages with null-references, we consider an object to be fully initialized if we cannot reach null-references by transitively reading fields of the object. We define a slightly different concept, but argue that this concept is analogous (Section 4.2).

Definition 4.1 (Locally Initialized)**.** In a heap Σ, we say that a location x is *locally initialized* if

- x is bound to a literal in Σ, or
- x is bound to an object $v\,(x\colon T)\,d$ in Σ and every term declaration in x is bound to a location.

Definition 4.2 (Fully Initialized)**.** In a heap Σ, we say that a location x is fully initialized if we cannot reach any term that is not a locally initialized location by following a path of field reads starting from x.

4.2 Emulating Nulls in κDOT

We define a bottom typed infinite loop as the notion of null in κDOT.

$$\begin{aligned}\Omega = \mathsf{let}\ k = \kappa\,(x\colon \{null\colon \bot\})\,\{\{null = x.null\}\}\,x\ \mathsf{in}\\ \mathsf{let}\ nPkg = \mathsf{new}\ k\ ()\ \mathsf{in}\\ nPkg.null\end{aligned}$$

Ω has type $\bot$, and executing it causes the abstract machine to bind $nPkg = v\,(nPkg\colon \{null\colon \bot\})\,\{null = nPkg.null\}$ on the heap and then loop indefinitely reading $nPkg.null$. Since Ω has type $\bot$, it also has all other types via subtyping.

Now we can emulate Scala-like constructors in κDOT by always using Ω for the default definitions in κDOT constructors. In this setting, the problem of initialization becomes that fields must be written to before any of the default Ω are read and executed. Here the notion of null pointer exception is executing a default Ω.

4.3 Bad Bounds and Divergent Programs

The bad bounds problem for DOT calculi [Amin et al. 2012] is that for any pair of arbitrary types T and U there exists an environment Γ such that $\Gamma \vdash T <: U$. All that is required for this is $\Gamma \vdash y\colon \{A\colon T..U\}$, and subtyping follows from the (<:-Sel), (Sel-<:), and (Trans) subtyping rules. The requirement of $\Gamma \vdash T <: U$ is easily satisfied if $y\colon \{A\colon T..U\} \in \Gamma$ or $y\colon \bot \in \Gamma$.

This allows us to type crazy terms. For example, the following is a well-typed κDOT term, even though it seems we are reading an arbitrary field of a function. $f.a$ is typed in a context with $\Gamma\,(x) = \bot$ and subtyping allows us to use the bad bounds $(\forall\,(x\colon \top)\,\top..\{a\colon \top\})$ on the left.

$\mathsf{let}\ f = \lambda\,(x\colon \top)\,.x\ \mathsf{in}$	$\Gamma \vdash x\colon \{A\colon \forall\,(x\colon \top)\,\top..\{a\colon \top\}\}$
$\mathsf{let}\ x = \Omega\ \mathsf{in}$	$\Gamma \vdash f\colon \{a\colon \top\}$
$f.a$	$\Gamma \vdash f.a\colon \top$

Rapoport et al. [2017] noticed that the above kind of bad bounds do not occur in inert contexts. Our preservation

theorem says that if we started our machine with a well-typed term, the heap corresponds to some inert context. This means we will never have a machine state $\langle f.a;\ s;\ \Sigma\rangle$, where f is a function in Σ. If f is a function and $f.a$ is the body of a let frame on the stack, the machine diverges without executing $f.a$. For the above term, the machine diverges while trying to execute Ω and never executes $f.a$.

4.4 Path-Dependent Subtyping via Initialization

Bad bounds are one of the reasons why κDOT currently offers only a weak form of path-dependent types: types of the form $x.A$ instead of $x.a.b\ldots c.A$. To illustrate why κDOT disallows full-paths, consider $\mu\,(x\colon \{a\colon \{A\colon T..U\}\})$. Even if we have a location x of this type in our context, κDOT does not have $T <: x.a.A <: U$. To use $T <: U$, we must let bind $x.a$ to a variable in κDOT: let $y = x.a$.

But the above type is inert for any T and U, and we can create objects of this type using Ω.

$$\text{let } k = \kappa\,(x\colon \{a\colon \{A\colon T..U\}\})\,\{\{a = \Omega\}\}\,x \text{ in}$$
$$\text{let } x = \text{new } k\,()$$

The program only diverges when we execute $x.a$ (without assigning to $x.a$). However, consider what happens if we are able to (at least locally) initialize x by assigning a location y to $x.a$. When we assign to $x.a$, we provide evidence that indeed $T <: U$ since we would have $y : \{A\colon S\}$ with $T <: S$, and $S <: U$. Thus we conjecture that if we are able to track initialization, κDOT will be able to support a richer form of subtyping using the following rules. Thus there are metatheoretic reasons to care about initialization even if we are happy with lazy semantics.

$$\frac{x \text{ locally initialized} \quad \Gamma \vdash x.a\colon \{A\colon S..T\}}{\Gamma \vdash S <: x.a.A} \qquad \frac{x \text{ fully initialized} \quad \Gamma \vdash x\ldots a\colon \{A\colon S..T\}}{\Gamma \vdash S <: x\ldots a.A}$$

$$\frac{x \text{ locally initialized} \quad \Gamma \vdash x.a\colon \{A\colon S..T\}}{\Gamma \vdash x.a.A <: T} \qquad \frac{x \text{ fully initialized} \quad \Gamma \vdash x\ldots a\colon \{A\colon S..T\}}{\Gamma \vdash x\ldots a.A <: T}$$

5 Expressive Power of κDOT

In this section, we compare κDOT to WadlerFest DOT and argue that κDOT is more expressive than WadlerFest DOT. We start by showing how κDOT is at least as expressive as WadlerFest DOT by mapping WadlerFest DOT constructs to κDOT and then show that the use of mutation allows κDOT to express ideas that cannot be expressed in WadlerFest DOT. We end the section by discussing how the different kinds of fields in Scala may be expressed in κDOT.

5.1 Mapping WadlerFest DOT to κDOT

κDOT makes the following changes to WadlerFest DOT.

1. Literals must be let bound in κDOT.
2. κDOT replaces object literals with constructors.
3. κDOT adds contravariant setter types and mutation.

Literals l in WadlerFest DOT map to κDOT as

$$\text{let } x = l \text{ in } x$$

Object literals can always be emulated by let binding a constructor for the object and then calling it. The WadlerFest DOT object literal $\nu\,(x\colon T)\,d$ maps to κDOT as

$$\text{let } k = \kappa\,(x\colon T)\,\{d\}\,x \text{ in new } k\,()$$

Mutation is a strict addition to WadlerFest DOT, in the sense that it adds its own syntax and typing rule without changing the rest of the language. Furthermore, a field declaration $\{a\colon S..T\}$ has $\{a\colon \bot..T\}$ as a super type. These two types are equally as expressive in the absence of mutation, since field reads only depend on getter types.

5.2 κDOT Improves upon WadlerFest DOT

Amin et al. [2016] comment that compared to simply typed strict records in other calculi, the records (objects) being lazy in WadlerFest DOT "does not limit expressiveness, as a fully evaluated record can always be obtained by using let bindings to pre-evaluate field values before they are combined in a record". But we cannot always pre-evaluate records with path-dependent types. For example, for the program in Fig. 1a, the type of mangoTree.fruit is path-dependent on mangoTree. In both WadlerFest DOT and κDOT we cannot let bind terms whose types depend on objects that are created at a later stage of evaluation, so we cannot express the program in Fig. 1a by pre-evaluating mangoTree.fruit through a let binding.

In WadlerFest DOT, dependent fields must always be lazy. In κDOT however, fields can start off being lazy as the default definitions allocated by a constructor but later be fully evaluated through assignment. For example, we can express the program in Fig. 1a in κDOT as shown in Fig. 9. In this sense, κDOT is more expressive than WadlerFest DOT since κDOT can express fully evaluated dependent fields.

In Fig. 9, calling $kTree$ allocates an object containing the definition $\{fruit = \Omega\}$ and then runs the body of the constructor. When the body of $kTree$ is executed, the $kFruit$ constructor is called to create a fully evaluated path-dependent object to assign to the $fruit$ field.

5.3 var and lazy val in κDOT

We now discuss how to emulate the different kinds of fields in Scala in κDOT. The immutability of val is an orthogonal consideration, so we only consider var for strict fields.

In Scala, var fields are fully evaluated. Similar to the Scala program in Fig. 10a, in κDOT we can create fully evaluated recursive objects by assigning to them in constructors.

The lazy val fields in Scala are lazily evaluated, but once evaluated the result is memoized. So in Fig. 10b, fruits.mangoes is only evaluated in the first read of fruits.mangoes; in the

$$
\begin{array}{l}
\text{let} \\
\quad kTree = \kappa(y : \{TreeFruit : y.TreeFruit\} \\
\qquad\qquad\qquad \wedge \{fruit : \{A : y.TreeFruit\}\}) \\
\qquad \{\{TreeFruit = y.TreeFruit\} \wedge \{fruit = \Omega\}\} \\
\qquad \text{let } kFruit = \kappa\,(y : \{A : y.TreeFruit\}) \\
\qquad\qquad\qquad \{\{A = y.TreeFruit\}\}\, y \\
\qquad \text{in} \\
\qquad \text{let } x = \text{new } kFruit\,() \text{ in} \\
\qquad y.fruit := x \\
\text{in} \\
\text{let} \\
\quad mangoTree = \text{new } kTree\,()
\end{array}
$$

Figure 9. Fully Evaluated Fields in κDOT

```
trait Mangoes { x =>
  val mango = mangoTree.fruit
  var moreMangoes : Mangoes = x
}
class RecursiveMangoes extends Mangoes { x =>
  override var moreMangoes : Mangoes = new Mangoes {}
  moreMangoes.moreMangoes = x
}
val mangoes = new RecursiveMangoes {}
```

(a) var fields are fully evaluated

```
class Fruits { x =>
  lazy val mangoes = new Mangoes {}
}
val fruits = new Fruits {}
val mangoes2 = fruits.mangoes
val mangoes2Again = fruits.mangoes
```

(b) lazy val fields are lazily evaluated and memoized

Figure 10. Scala Examples

$$
\begin{array}{l}
\text{let } mangoes2 = fruits.mangoes \text{ in} \\
\text{let } \_ = fruits.mangoes := mangoes2 \text{ in} \\
\text{let } mangoes2Again = fruits.mangoes \text{ in} \\
\text{let } \_ = fruits.mangoes := mangoes2
\end{array}
$$

Figure 11. Manual memoization in κDOT

second read the cached value is read. In κDOT, one possible way of emulating this is by adding an immediate write back to every field read. Fig. 11 shows how the field reads in Fig. 10b can be emulated in κDOT.

Since getter and setter types can be different, the above may not be typeable in the context in which the field read occurs. However, we can completely replace lazy semantics in our calculus with a memoized lazy semantics by having the abstract machine push update frames on field reads.

6 Discussion

In this section, we compare some of our design choices for κDOT to other DOT calculi and discuss our experience working with type safety proofs for WadlerFest DOT.

6.1 First-Class Constructors

In κDOT, we replaced the object literals of WadlerFest DOT with constructors since they form a natural place for initialization restrictions and initialization related static analysis. Constructors are first-class in κDOT; constructors can be created at runtime and references to them passed around to functions and even other constructors. This allows us to preserve all the features of WadlerFest DOT's object literals.

6.2 The Abstract Machine

Standard small-step operational semantics for lambda calculi define a stepping relation between terms and the standard preservation theorem shows that after a term takes a step, its type is preserved. Amin et al. [2016] express the operational semantics for WadlerFest DOT as evaluating terms inside evaluation contexts where an evaluation context is defined as follows.

$$e ::= \square \mid \text{let } x = \square \text{ in } t \mid \text{let } x = h \text{ in } e$$

In WadlerFest DOT, after a subterm steps inside an evaluation context, the overall term is still typeable. This allows Amin et al. to prove the standard preservation theorem.

In κDOT, field mutation allows objects to refer to objects that were created at a later stage of evaluation. If we try to convert the heap in κDOT back to a set of let bindings, the forward pointers are not typeable. We express the operational semantics for κDOT using an abstract machine instead of evaluation contexts because in κDOT, stepping may cause the overall term to become untypeable. In κDOT, the stepping relation and preservation are expressed between machine states and this is unavoidable.

6.3 Operational Differences Between κDOT and WadlerFest DOT

In WadlerFest DOT, answers are defined to be evaluation contexts containing a variable, a function literal, or an object literal. WadlerFest DOT requires the following as one of its rules for reduction inside evaluation contexts.

$$
\begin{array}{rl}
& \text{let } x = (\text{let } y = t \text{ in } t') \text{ in } u \\
\longmapsto & \text{let } y = t \text{ in } (\text{let } x = t' \text{ in } u) \qquad \text{(Let-Let)}
\end{array}
$$

This changes the focus of execution to t and allows WadlerFest DOT to execute nested let bindings.

During the design process of κDOT, before we added constructors and a stack to our abstract machine, we also had a similar rule and considered literals together with a heap as answers. When we added constructors and constructor frames to distinguish computations happening inside different constructors, we realized that we could use the stack for

evaluating nested let bindings as well, making the (Let-Let) rule unnecessary. We further simplified the calculus by ensuring literals are always let bound so that only variables were answers. This simplifies both the statements and the proofs of progress and preservation. We note however that the two sides of the (Let-Let) rules are observationally equivalent and the κDOT abstract machine can optionally reduce the number of frames used by using the (Let-Let) rule.

The use of evaluation contexts in WadlerFest DOT required object literals to be typeable without referring to any variables used to let bind them. In the following WadlerFest DOT code, $\{a = t\}$ is in the scope of z and evaluation performs a substitution. In the equivalent κDOT reduction, the substitution is performed during allocation and does not need to be performed during field reads. Hence, $\{a = [x/z]\, t\}$ does not need to be in the scope of z in the κDOT heap.

$$\begin{aligned} & \text{let } x = v\,(z\colon \_)\,\{a = t\} \text{ in } x.a \\ \underset{\text{WadlerFest}}{\longmapsto} \; & \text{let } x = v\,(z\colon \_)\,\{a = t\} \text{ in } [x/z]\, t \\ & \langle x.a;\, s;\, x = v\,(z\colon \_)\,\{a = [x/z]\, t\}\rangle \\ \underset{\kappa\text{DOT}}{\longmapsto} \; & \langle [x/z]\, t;\, s;\, x = v\,(z\colon \_)\,\{a = [x/z]\, t\}\rangle \end{aligned}$$

6.4 ML-style References in κDOT

We now compare κDOT to other DOT calculi that use ML-style references. We firstly note that κDOT can emulate ML-style references by creating objects which contain locations in their fields. Then dereferencing and reference mutation can be emulated by field reads and field assignments.

Rapoport and Lhoták [2017] provide a variant of WadlerFest DOT with ML-style references. They firstly extend the calculus with a separate store of references which point to locations and secondly extend typing judgments with a store typing context. This calculus does not have separate setter and getter types for references and subtyping between references is defined by the following invariance rule.

$$\frac{T <: U \qquad U <: T}{\text{Ref } T <: \text{Ref } U} \qquad \text{(Ref-Sub)}$$

In their type safety proof, they use a separate correspondence between the mutable store and the store typing context, which allows them to keep the Amin et al. store correspondence relation. We originally intended to base κDOT on this calculus, but the combination of reference reads and call by name field reads interacted in counterintuitive ways when we tried to express ideas of initialization in this calculus.

Rompf and Amin [2016a] discuss adding ML-style references to DOT-like calculi, and also add a separate mutable store and store typing context. However, their mutable store binds locations to objects and mutation replaces an object in the store with a different object. κDOT only allows writing locations to fields, not values. However since locations can point to objects, replacing one location with another is equivalent to updating one object with another.

One of the distinguishing features of κDOT compared to the above calculi is that κDOT has a single heap and a single typing context. We wanted this simplification because we eventually want to extend the calculus with initialization tracking by adding more contexts to typing relations.

6.5 Working with DOT Type Safety Proofs

Amin et al. [2014] comment that DOT type safety proofs are easier to evolve by adding features in a bottom-up fashion and checking where the type safety proof breaks. We now briefly discuss the bottom-up approach we took for κDOT.

Amin et al. [2016] define an operational semantics based on evaluation contexts and Rapoport et al. prove soundness for this calculus. However, in this calculus, both values and variables were treated as answers. Starting from this calculus, we arrived at the current κDOT in the following steps, producing a proof of type safety at the end of each step.

1. Adapt the notion of evaluation contexts to heaps.
2. Change the operational semantics to reduce literals to a let binding and redefine answers to only be variables.
3. Replace well-typed evaluation contexts with heap correspondence.
4. Add setter types and field mutation.
5. Remove objects from the literal grammar, ensure literals are let bound, and add constructors and frames.

Unlike functions, we designed constructors to take multiple arguments because we eventually want to add an initialization system to κDOT where we differentiate between fully initialized inputs to the constructor and partially initialized inputs to the constructor. This made the last step tedious because the type safety proof and the Coq libraries used had little support for dealing with multiple arguments and many lemmas had to be lifted manually to multiple arguments.

7 Related Work

The κDOT calculus is an extension to the DOT calculus of Amin et al. [2016]. The type safety proof of κDOT is an extension of the proof of Rapoport et al. [2017].

Mackay et al. [2012] developed a Coq formalization for a version of Featherweight Java [Igarashi et al. 2001] with mutable and immutable objects. Field assignment in this calculus is very similar to κDOT; assignment in this calculus also assigns locations to fields. Their heap and typing context design is also very similar, but one difference is that they have separate typing contexts for variables and locations. For their preservation theorem, their variable typing context stays fixed while the location typing context grows. However, Featherweight Java does not have path-dependent types, i.e. types do not depend on variables or locations. One of our main contributions is that field mutation is possible in a setting with path-dependent types. In fact, to the best of our knowledge, no existing approach to a mechanized proof of type safety for DOT supports field assignment.

There have been several calculi that have tried to model the path-dependent types of Scala before Amin et al. [2012] introduced DOT. Odersky et al. [2003] introduced the *νObj* calculus and proved it type safe. In Scala, type members have both upper and lower bounds, but the type members in *νObj* are less expressive and only have upper bounds. Featherweight Scala [Cremet et al. 2006] (FS_{alg}) was developed with a focus on algorithmic type checking and to correspond closely to Scala. Scalina [Moors et al. 2008] was developed to explore higher-kinded types in Scala. They were not proven to be type safe, but had type members with both bounds.

FS_{alg} and Scalina were large calculi and it was unclear how their features interacted with path-dependent types for type safety. The DOT family of calculi were first introduced by Amin et al. [2012] as a calculus containing only the features of Scala that were needed for path-dependent types. In their paper, they define a DOT calculus with intersection types, union types, type selection, and type refinement and explore the difficulties in proving preservation for a small step semantics. Since its introduction, there have been several mechanized type safety proofs for calculi in the DOT family for either small step or big step operational semantics.

Amin et al. [2014] introduce μDOT. μDOT is a very simple calculus with methods, path dependent types, subtyping, and a big step operational semantics. μDOT does not have a bottom type, union types, or intersection types.

Rompf and Amin [2016b] present a DOT calculus with top and bottom types, intersection and union types, and a small step operational semantics. Amin and Rompf [2017] and their earlier technical report [Rompf and Amin 2016a] discuss proving type safety for DOT-like calculi under a big step operational semantics using definitional interpreters. Their paper discusses proving type safety for these calculi by starting with a proof of type safety for System $F_{<:}$ and then slowly generalizing or adding features such as ML-style references while keeping most of the type safety proof intact.

Amin et al. [2016] present the WadlerFest DOT calculus, which is a DOT calculus with top and bottom types, intersection types, and a small step operational semantics. Rapoport and Lhoták [2017] extend WadlerFest DOT with a mutable store and ML-style references. Rapoport et al. [2017] present a more extendable proof of the WadlerFest DOT calculus. In their paper, they introduce a syntactic condition on contexts where bad bounds do not occur, called inertness, and provide a general technique (a "proof recipe") for proving canonical forms for WadlerFest DOT-like calculi in inert contexts.

Without field assignment, path-dependent types make it difficult to reason about initialization in DOT calculi. Without constructors, there is no clear distinction between code that is meant to initialize an object and code that assumes that an object is initialized and no longer contains null-references. None of the above calculi supported constructors or field assignment. This is why we developed κDOT, which extends the WadlerFest DOT calculus with constructors and field assignment. For our type safety proof, we extended the WadlerFest DOT type safety proof of Rapoport et al. [2017].

8 Future Work and Conclusion

We now discuss the planned steps for the future of κDOT. Much of the following is guided by a need to slowly build proof infrastructure to support the changes we want to make to our mechanized type safety proof. We are currently in the process of adding initialization types to κDOT. Once we have initialization types, we plan to restrict field reads to only locations which have been initialized. As the very last step, we will restrict fields to only contain variables, achieving a DOT calculus which is completely strict.

Recursive functions in WadlerFest DOT are written by binding them to a field and dereferencing the field inside the body of the function. Restricting field reads of uninitialized values will restrict writing recursive functions in this way. We plan to remove this restriction by adding an operator and reduction rule of the following form.

$$\frac{x = v\,(x\colon T)\ldots\{a = t\}\ldots \in \Sigma \qquad \Sigma' = \Sigma\,[x = v\,(x\colon T)\ldots\{a = y\}\ldots],\ y = l}{\langle x.a := l;\ s;\ \Sigma\rangle \longmapsto \langle y;\ s;\ \Sigma'\rangle}\ (\textsc{Lit-Assn})$$

A literal is directly assigned to a field by pushing the literal to the heap and mutating the field with the location of the literal. For typing, l will be typed assuming x is initialized, allowing it to read fields of x, in particular $x.a$.

We added mutation and constructors to DOT because we believed them necessary to "scale DOT to Scala" [Odersky 2016]. DOT was developed to be a simple calculus, a calculus containing only the essential parts for path-dependent types and a foundation on which features of Scala could be slowly added in a bottom up approach till all the fundamental features could be proven safe. DOT also acts as a safe space for experimentation where features can be experimented with before being added to Scala. In that spirit, we developed κDOT to guide the design of initialization systems for Scala.

There is an existing body of work on initialization [Fahndrich and Xia 2007; Qi and Myers 2009; Summers and Mueller 2011] to take inspiration from. However, these systems are often complicated and deal with large languages with all their features and quirks. For example, to adapt to the nulls in their language, many of these systems add nullable types on top of their existing type systems and awkward types such as Nullable Option types emerge in these settings. DOT gives us the luxury to design a system without nulls from the ground up rather than having to adapt to a system where we already have null-references. If we evolve DOT without ever introducing null-references or exceptions, we can achieve a Scala which stands on a strong null-free foundation.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.), Vol. 9600. Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent Object Types. In *19th International Workshop on Foundations of Object-Oriented Languages*.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. https://doi.org/10.1145/3009837

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *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*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 233–249. https://doi.org/10.1145/2660193.2660216

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A Core Calculus for Scala Type Checking. In *Mathematical Foundations of Computer Science 2006*, Rastislav Královič and Paweł Urzyczyn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23.

Manuel Fahndrich and Songtao Xia. 2007. Establishing Object Invariants with Delayed Types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/1297027.1297052

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. 2012. Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP '12)*. ACM, New York, NY, USA, 11–19. https://doi.org/10.1145/2318202.2318206

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Safe type-level abstraction in Scala. In *International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*.

Martin Odersky. 2016. Scaling DOT to Scala — Soundness. http://www.scala-lang.org/blog/2016/02/17/scaling-dot-soundness.html.

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A Nominal Theory of Objects with Dependent Types. In *ECOOP 2003 – Object-Oriented Programming*, Luca Cardelli (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–224.

Xin Qi and Andrew C. Myers. 2009. Masked Types for Sound Object Initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 53–65. https://doi.org/10.1145/1480881.1480890

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A Simple Soundness Proof for Dependent Object Types. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 46 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133870

Marianna Rapoport and Ondřej Lhoták. 2017. Mutable WadlerFest DOT. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs (FTFJP'17)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3103111.3104036

Tiark Rompf and Nada Amin. 2016a. From F to DOT: Type Soundness Proofs with Definitional Interpreters. *CoRR* abs/1510.05216v2 (2016). http://arxiv.org/abs/1510.05216v2

Tiark Rompf and Nada Amin. 2016b. Type soundness for dependent object types (DOT). In *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*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 624–641. https://doi.org/10.1145/2983990.2984008

Alexander J. Summers and Peter Mueller. 2011. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 1013–1032. https://doi.org/10.1145/2048066.2048142

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.