κDOT: A DOT Calculus with Mutation and Constructors

by

Ifaz Kabir

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Scala is a functional and object-oriented programming language which 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. [2016] 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.

This thesis presents the κDOT calculus, a calculus in the DOT family which supports mutable fields and constructors. κDOT can emulate both lazy and strict fields, and the constructor calls in κDOT emulate how objects are created in Scala. We present the key features of κDOT, the key ideas required for type safety, and discuss how the operational semantics of κDOT relates to that of Scala.

κDOT is proven type safe via a mechanized proof in Coq.

Acknowledgements

I would like to thank everyone who made this thesis possible:

My advisor Ondřej Lhoták, for accepting me into the Masters program at the University of Waterloo and for being patient with me and guiding me as I stumbled around in the darkness that is Dependent Object Types.

Prabhakar Ragde for, among many things, introducing me to Coq and letting me TA his amazing courses.

Marianna Rapoport for being an amazing friend and for sharing her insights into DOT calculi and Coq with me.

My "bois", Ellen Arteca and Alexi Turcotte, for all the coffee runs, the ranting sessions about Coq, and the memes!

Magnus Madsen and Abel Nieto for bringing their own flavour of humour into my life.

Peter Buhr and Gregor Richards for their back-and-forth that made lunch interesting.

Thierry Delisle, Werner Dietl, Aaron Moss, Jeff Luo, and the rest of the PLG for sharing my love and interest in Programming Languages.

Vijay Ganesh for introducing me to SAT and SMT solvers. While SAT/SMT solvers do not make an appearance in this thesis, they were a source of intellectual joy and excitement consistently throughout my Master's.

Murphy Berzish and Dmitry Blotsky for showing me that there is graduate life outside of Programming Languages.

Srishti Gupta for being my swimming buddy and my closest friend during my Master's.

Anthony Brennan for being a consistent part of my life at Waterloo outside of the PLG.

Uday Barar for taking the time get to know me and everyone else around him.

Fatema Boxwala, Luke Franceschini, William Gertler, Alex Gomez, Caelin Jackson, Evy Kassirer, Jahanzeb Khan, Ru Li, Anna Lorimer, Kelly McBride, Neil Parikh, Imran Saleh, Laura Song, Melissa Tedesco, Alex Tomala, Charlie Wang, Hanna Wong, Iris Wen, Jennifer Zhou, and Alex Zvorygin for putting up with me at the CSC.

Ilia Chtcherbakov, Joseph (Doc) Musleh, Rana Saleh, Harry Sivasubramaniam, Shelly Wu, and Samuel Yusim for being the PMC do didn't graduate and leave.

Ian Davidson, Brandon Doherty, Eric Guo, Parham Hamidi, Anthony McCormick, Grant Simms, and the Crossroads Board Game Cafe for the late nights filled with joy and excitement.

Chris Hawthorn, Edward Lee, Michael Wang, Rutger Campbell, and Sean Harrap for remembering to celebrate the important moments of our lives.

Sarah Sun for the magical moments during my Master's when our schedules aligned and we could hangout in Toronto.

Victor Fan, Linlin Li, David McLaughlin, and Jim Wallace for being the friends who graduated but didn't leave!

Yifan Li for coming back into our lives after finishing med-school.

David Shi and Christina Tan for always believing in me, especially during my most doubtful moments.

Steven Arnott, Sean Aubin, Faryal Diwan, Heming Zhang, Brian Lin, and the rest of the learning night community for sharing their life and knowledge with me.

Lilly Horne and Daniel Resnick for sticking with me from Orientation 2010 till the very end.

Gayle Goodfellow for all she did and all she does.

Rose and the Math C&D for all the coffee and doughnuts.

Gregor Richards and Prabhakar Ragde (again) for agreeing to read and review this thesis and for their helpful comments in making this the best thesis it can be.

My parents for doing their best to raise me and providing me with an academic life.

I also cannot thank Nomair Naeem enough for his comments which ultimately convinced me to choose the University of Waterloo for my Master's. I had an amazing time during my Master's and this would have been a very different thesis at a very different institution if it weren't for Nomair!

Dedication

This thesis is dedicated to all the students at Waterloo who fought for change over my 8 years at the University of Waterloo.

Table of Contents

List of Figures

Chapter 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. In languages such as ML, higher level ideas are expressed using modules and functors separately from lower level ideas of function application and data creation, but in Scala the same language of objects and functions is used to express both high and low level ideas. Instead of writing functors that manipulate type-carrying modules, in Scala, we write functions that manipulate type-carrying objects the same way we write functions that manipulate data.

1.1 Path-Dependent Types in Scala

Fig. 1.1 shows how a module signature can be defined in Scala. An `OrderedModule` encapsulates method members on ordered elements as well as a *type member* `Elem` for the type of elements we are comparing. Fig. 1.2 shows how we implement `OrderedModule` for integers and strings. The modules are instantiated the same way as objects using the `new` keyword. If we try to compare elements of different types using an `OrderedModule`, we get a path-dependent type mismatch error as shown in Fig. 1.3.

The example in Fig. 1.3 shows how path-dependent types can distinguish between objects of different base types. However, with path-dependent types we can distinguish between objects of the same base type. In Fig. 1.4, `mangoTree` and `appleTree` are both instances of `Tree`, and are even created using the same constructor. However, we get a path-dependent type mismatch error if we try to put the fruit of one tree into another. The field

```scala
trait OrderedModule {
  type Elem
  val zero : Elem
  def lessThan( left : Elem, right : Elem): Boolean
  def moreThan(left: Elem, right : Elem): Boolean = {
    lessThan( right , left )
  }
}
```

Figure 1.1: Modules in Scala

```scala
trait IntOrdered extends OrderedModule {
  type Elem = Int
  val zero = 0
  def lessThan( left : Int , right : Int ): Boolean = {
    left < right
  }
}

trait StringOrdered extends OrderedModule {
  type Elem = String
  val zero = ""
  def lessThan( left : String , right : String ): Boolean = {
    if ( left . length != right. length ) {
      false
    } else {
      left < right
    }
  }
}

val intOrdered : OrderedModule = new IntOrdered {}
val stringOrdered : OrderedModule = new StringOrdered {}
```

Figure 1.2: Module Implementation in Scala

```
33 val err : Boolean = intOrdered.lessThan(intOrdered.zero, stringOrdered.zero)
34 // OrderedModule.scala:33: error: type mismatch;
35 // found   : this.stringOrdered.zero.type (with underlying type this.stringOrdered.Elem)
36 // required: this.intOrdered.Elem
37 // val err : Boolean = intOrdered.lessThan(intOrdered.zero, stringOrdered.zero)
38 //                                         ^
39 // one error found
```

Figure 1.3: Path-Dependent Type Mismatch in Scala

```
1  trait Fruit[T] {
2    type A = T
3  }
4
5  trait Tree { tree =>
6    type TreeFruit
7    val fruit : Fruit[tree.TreeFruit] =
8      new Fruit[tree.TreeFruit] {}
9    var fruits : List[Fruit[tree.TreeFruit]] = Nil
10 }
11
12 val mangoTree = new Tree{}
13 val appleTree = new Tree{}
14
15 appleTree.fruits = mangoTree.fruit :: appleTree.fruits
16 // Trees.scala:15: error: type mismatch;
17 //  found   : this.Fruit[this.mangoTree.TreeFruit]
18 //  required: this.Fruit[this.appleTree.TreeFruit]
19 // appleTree.fruits = mangoTree.fruit :: appleTree.fruits
20 //                                       ^
21 // one error found
```

Figure 1.4: Type Mismatch Between Objects of the Same Type

`mangoTree.fruit` is path-dependent on `mangoTree`, whereas `appleTree.fruits` is a list of objects which are path-dependent on `appleTree`. Since `appleTree` and `mangoTree` are different objects, the path-dependent types `appleTree.TreeFruit` and `mangoTree.TreeFruit` are different, resulting in a type mismatch.

Path-dependent types allow us to define records where the type of one field is path-dependent on the type of another field. In Fig. 1.5, objects of type `TypeCarry` carry the type members `A` and `B` and fields `x` and `y` of those types. Objects of type `PathDependentPair` have a `left` field of type `TypeCarry` and a `right` field which is a function which takes in inputs of type `left.A` and `left.B` and produces an output of type `left.A`.

Fig. 1.5 also shows how path-dependent types provide a syntactically lightweight way to encode existential types in Scala. In the definition of `IntStringDep.right`, the types `left.A` and `left.B` are abstract, yet we are able to easily use them for type annotations without needing any heavy machinery to deconstruct the type of `IntStringDep`. In addition, Fig. 1.5, together with Fig. 1.6, show how path-dependent types allow for very fine-grained control over execution at the type level. We are allowed to put `PathDependentPair`s with different `left.A` and `left.B` types together in a list and iterate through them as shown by the `printPairs` function, but we may not apply one `right` method to `left.x` and `left.y` of a different `PathDependentPair` as attempted in the `printPairsErr` method in Fig. 1.6.

1.2 DOT calculi

While path-dependent types are expressive, until very recently it was not known whether it was possible to design a type safe calculus with path-dependent types without incurring the overhead of a full dependent type system. After a long and elusive search for a sound calculus that could model the path-dependent types of Scala, the Dependent Object Types (DOT) family of calculi were proposed by Amin et al. [2012], and variants were recently proven sound [Amin et al., 2016, Rompf and Amin, 2016b, Amin et al., 2014].

Proving type safety for DOT calculi is tricky because path-dependent types are very flexible and allow the programmer to create contexts where there are unsound subtyping judgments. While the programmer is allowed to create and type check these unsound contexts, type safety holds because these contexts never appear during execution. We will see examples of this in Chapters 4 and 6.

```scala
trait TypeCarry {
  type A
  type B
  val x: A
  val y: B
}

trait PathDependentPair {
  val left : TypeCarry
  val right : (left.A, left.B) => left.A
}

trait IntStringDep extends PathDependentPair {
  val left  : TypeCarry = new TypeCarry {
    override type A = Int
    override type B = String
    override val x: A = 3
    override val y: B = "Hello!"
  }
  val right  : (left.A, left.B) => left.A = {
    (x: left.A, y: left.B) => x
  }
}

def printPairs(xs: List[PathDependentPair]): Unit = {
  xs match {
    case Nil => Unit
    case depPair :: depPairs =>
      println(depPair.right(depPair.left.x, depPair.left.y))
      printPairs(depPairs)
  }
}
```

Figure 1.5: Path-Dependent Pairs in Scala

```
34 def printPairsErr(xs: List[PathDependentPair]): Unit = {
35   xs match {
36     case Nil => Unit
37     case depPair1 :: depPair2 :: depPairs =>
38       println(depPair1.right(depPair2.left.x, depPair1.left.y))
39       printPairs(depPairs)
40     case depPair :: depPairs =>
41       println(depPair.right(depPair.left.x, depPair.left.y))
42       printPairs(depPairs)
43   }
44 }
45 // DepPair.scala:38: error: type mismatch;
46 // found   : depPair2.left.x.type (with underlying type depPair2.left.A)
47 // required: depPair1.left.A
48 //   println(depPair1.right(depPair2.left.x, depPair1.left.y))
49 //                                  ^
50 // one error found
```

Figure 1.6: Fine-grained Control in Scala

1.3 Laziness in Previous DOT Calculi

The DOT calculi of Rompf and Amin [2016b] and Amin et al. [2016] have notions of path-dependent types, but chose to forgo many fundamental features of Scala in order to simplify the calculi and type soundness proofs. In particular, objects in the calculus of Rompf and Amin only have methods and type members, but not fields. Field reads are emulated by method calls, which roughly correspond to lazy semantics for field reads. We illustrate the laziness of method calls through an example in Scala. In the example in Fig. 1.7, we emulate the code on the left which has a field read `bananaPkg.banana` by the code on the right which uses a method call `bananaPkg.readBanana()`. In the code on the left, the contents of the field `bananaPkg.banana` is evaluated when `new BananaPkg{}` is evaluated. When the field is read, the location of the `Banana` object is fetched. We say that the kind of field read used by the code on the right is lazy, because the contents of field (i.e. body of the `readBanana` method) is not evaluated until the field is read via the method call `readBanana()`.

The DOT calculus of Amin et al. (WadlerFest DOT) supports fields and, instead of methods, first-class functions. In this calculus, fields store terms which are evaluated each

```
class Banana {}
class BananaPkg {
  val banana: Banana = new Banana{}
}

val bananaPkg = new BananaPkg{}
val banana: Banana = bananaPkg.banana
```

```
class Banana {}
class BananaPkg {
  def readBanana(): Banana = new Banana{}
}

val bananaPkg = new BananaPkg{}
val banana: Banana = bananaPkg.readBanana()
```

Figure 1.7: Emulating Field Reads via Method Calls

```
class Orange {}
class OrangePkg {
  lazy val orange: Orange = new Orange{}
}

val orangePkg = new OrangePkg{}
val orange: Orange = orangePkg.orange
val orangeAgain: Orange = orangePkg.orange
```

Figure 1.8: Lazy Field Reads in Scala

time the field is read, which again corresponds to lazy semantics. The closest analogue to this kind of field in Scala are `lazy vals`. In Fig. 1.8, the field `orange` is declared as lazy and its contents are evaluated when the first occurrence of `orangePkg.orange` is evaluated and not when `new OrangePkg{}` is evaluated. Scala memoizes lazy field reads so that the cached value is returned when the second occurrence of `orangePkg.orange` is evaluated.

The choice of lazy fields simplifies type soundness proofs. Lazy fields allow objects to have recursive types without having to worry about field initialization or null reference exceptions if we carefully define the operational semantics. Scala however, provides many different types of fields:

val	strict immutable
var	strict mutable
lazy val	memoized lazy immutable

The strict field types of Scala are much more commonly used than lazy vals. Since the above DOT calculi only support lazy fields, the correspondence between DOT and Scala

is too weak for us to meaningfully prove many static analyses sound for Scala by proving them sound for DOT. Furthermore, lazy fields and the lack of field mutation limit the expressiveness of the calculus (see Chapter 5).

Although field reads are lazy, function and method applications, are evaluated in a strict manner in Scala and both of the above calculi. Neither of these calculi support any form of field mutation. The calculus of this thesis, κDOT, extends WadlerFest DOT with field mutation.

1.4 Mutation in Previous DOT Calculi

While the DOT calculi of Rompf and Amin and WadlerFest DOT [Amin et al., 2016] do not have any notion of mutation, other DOT calculi add mutation via ML-style first-class references to a separate mutable store. Amin and Rompf [2017] and their technical report [Rompf and Amin, 2016a] discuss adding a mutable object store to call-by-value DOT calculi. In these calculi, mutating a reference replaces one object with another in the mutable store. Since WadlerFest DOT is a call-by-reference language, Rapoport and Lhoták [2017] extended WadlerFest DOT with a mutable store of references.

Both of these are very weak approximations to the field mutation we have in Scala. Scala does support mutable local variables, but they are not first-class and immutable local variables are more commonly used. Scala is a primarily call-by-reference language with object fields containing references to other objects. Type systems for the above calculi add a separate store typing context whereas in Scala we only reason about functions and objects.

1.5 Object Initialization in Previous DOT Calculi

In previous DOT calculi, objects were defined by literals which were directly allocated into a store with lazy fields. This is different from Scala where most objects are created by special functions called constructors. As we saw in previous examples, we execute a constructor by calling it with the `new` operator. When executed, a constructor first allocates a new object into the heap with fields containing null references as the initial value of all fields, and then replaces the null references with references to objects. In the example in Fig. 1.9, when `new PeachPkg{}` is executed a `PeachPkg` object is allocated with its `peach` field containing a null-reference. Next `new Peach{}` is executed which allocates a `Peach`

```
class Peach {}
class PeachPkg {
  val peach: Peach = new Peach{}
}

val peachPkg = new PeachPkg{}
```

Figure 1.9: Object Allocation in Scala

object and returns location of the new object. Lastly, the null reference in the `peach` field is replaced with the location of the new `peach` object.

It is generally considered to be a programming error if a constructor does not replace a null reference since it can cause null reference exceptions in the program. We were interested in developing static analysis and type systems for preventing this class of programming errors in Scala.

One of the main motivations for developing a new DOT calculus was that we found existing DOT calculi unsuitable for static analysis and typing restrictions for object initialization. In previous DOT calculi, there is no clear distinction between code that expects an object to be initialized and code that initializes an object. We were also interested in knowing how path-dependent types interacted with object initialization in Scala.

1.6 Thesis Outline

This thesis presents κDOT, a DOT calculus with field mutation and constructors. κDOT is an extension of WadlerFest DOT with a substantial improvement in expressive power while adding only a few syntactic and typing constructs. In particular, the typing rules for mutation in κDOT are simpler than those of DOT calculi with ML-style references since there is no need for a separate store typing context. Furthermore, the operational semantics for κDOT is expressed using an abstract machine with a stack and a heap which is closer to the execution environment of Scala which uses the JVM (Java Virtual Machine) call stack and heap.

The contributions of this thesis are:

- We show how field mutation and constructors can be added to WadlerFest DOT with minimal new constructs.

- We provide a mechanized proof of type safety for κDOT in Coq available at https://git.uwaterloo.ca/ikabir/dot-public.
- We discuss how κDOT is useful for reasoning about interactions between object initialization and path-dependent types.

The type safety proof of κDOT is an extension of the type safety proof of WadlerFest DOT. To make the proof easier to extend we made simplifications to the type safety proof of WadlerFest DOT which were presented at OOPSLA [Rapoport et al., 2017]. The work on κDOT has been accepted for presentation in the upcoming SIGPLAN International Scala Symposium [Kabir and Lhoták, 2018].

The rest of the thesis is outlined as follows. We discuss background work leading to DOT calculi and present WadlerFest DOT and the issues we had with its constructs and operational semantics in Chapter 2. In Chapter 3, we present the κDOT calculus; its syntax, typing constructs, and operational semantics. In Chapter 4, we discuss the type safety proof for κDOT. In Chapter 5, we compare the expressive power of κDOT to WadlerFest DOT, and in Chapter 6, we explore initialization problems using κDOT. In Chapter 7, we discuss the design choices we made for κDOT and provide some concluding remarks in Chapter 8.

As a last note, we would like to point out that it is a very happy coincidence that the letter k, its capitalized form K, and Greek letter κ have something to do with both the "word" konstructor as well as the last name of the author!

Chapter 2

Background

κDOT is an object-oriented calculus with first-class functions and constructors, path-dependent types, top and bottom types, intersection types, and subtyping. Several DOT and DOT-like calculi have appeared before κDOT. The DOT calculus was originally proposed by Amin et al. [2012] as a theoretical foundation where the interactions between the path-dependent types of Scala and other features could be studied, but before DOT, there were other calculi with notions of path-dependent types. In this chapter we discuss general ideas underlying DOT calculi in relation to the calculi that came before. We also discuss WadlerFest DOT in detail so that we can compare it to κDOT in later chapters.

2.1 Related Work

System $F_{<:}$ (F-sub) [Cardelli et al., 1994] is a polymorphic typed lambda calculus with a top type, subtyping, and bounded quantification for function arguments. Polymorphism in this calculus is achieved by type application. DOT calculi, including κDOT, can fully encode System $F_{<:}$, but do not need type application for polymorphism. In DOT calculi, instead of applying types for polymorphism, we apply functions to objects with type members.

On the more object-oriented side, Featherweight Java [Igarashi et al., 2001] is a class-based object-oriented calculus that formalizes Java. Featherweight Java is a simple calculus with objects, subtyping, and method application. The reduction semantics for method application in Featherweight Java is call-by-value in the sense that substitution replaces variables with copies of objects. Objects in Featherweight Java do not have type members, and the calculus does not support any form of path-dependent types.

The νObj calculus [Odersky et al., 2003] was one of the first calculi developed to formalize Scala. It supports first-class classes, objects with type members, and type selection. The key feature of νObj is that it gives unique names to its objects. In νObj, for a type label A, the path-dependent types $x.A$ and $y.A$ are the same type if and only if x and y can be shown to refer to the same object. However, the type members in this calculus only have upper bounds, whereas Scala and DOT calculi have type members with both upper and lower bounds.

Featherweight Scala (FS_{alg}) [Cremet et al., 2006] and Scalina [Moors et al., 2008] are two other calculi that try to formalize Scala and are worth mentioning. FS_{alg} was developed with a focus on algorithmic type checking and to correspond closely to Scala. Scalina was developed to explore higher-kinded types in Scala. FS_{alg} and Scalina both have type members with upper and lower bounds, but neither were proven type safe. In fact, they are large calculi which makes it difficult to understand how their different features interacted with path-dependent types for a proof of type safety.

The DOT calculus was introduced by Amin et al. [2012] as a calculus with only the features required for path-dependent types. They defined a DOT calculus with intersection types, union types, type members with both bounds, type selection, and type refinement and discussed the difficulties in proving preservation, and hence type safety, for a small step operational semantics. Since Amin et al., there have been several other DOT calculi, some of which were proven type safe.

Amin et al. [2014] introduced μDOT, a very simple DOT calculus with methods, path-dependent types, subtyping, and a big step operational semantics. In particular, μDOT did not have a bottom type, union types, or intersection types. The lack of complicated types caused by unions and intersections and their use of big-step semantics allowed them to prove type safety without needing to prove a general environment narrowing lemma. Their type safety proof only required an environment narrowing lemma for types starting with a `this` variable.

In 2016, just before the author started their Master's journey, Amin et al. [2016] proved type safety for a DOT calculus with first-class functions, top and bottom types, intersection types, and a small step operational semantics. They presented the calculus at the workshop commemorating Philip Wadler's 60$^{\text{th}}$ birthday, which is why we call this calculus WadlerFest DOT[1]. κDOT is directly inspired by this DOT calculus, and we discuss it in more detail in the next section.

[1]The Amin et al. [2016] paper refer to the calculus as "the DOT calculus". We call the calculus WadlerFest DOT to distinguish it from other calculi that are also called "the DOT calculus".

Later on in 2016, Rompf and Amin [2016b] presented another DOT calculus with top and bottom types, intersection and union types, and a small step operational semantics. This calculus does not have functions and only supports method calls. However this calculus supports a richer form of subtyping between recursive types.

Amin and Rompf [2017] and Rompf and Amin [2016a] discuss proving type safety for DOT-like calculi under a big step operational semantics using definitional interpreters. In particular, they discuss how to prove type safety for DOT by starting with a mechanized type safety proof for System $F_{<:}$ and slowly generalizing and adding features such as type members while keeping most of the type safety proof intact.

κDOT was developed with the goal of studying interactions between the path-dependent types of Scala and object initialization. Some of the practical considerations include a need to provide Scala programmers with guarantees that their programs will not throw null-pointer exceptions, but we were also interested in enriching the type system of DOT calculi based on initialization. Adding an initialization system to κDOT is the subject of ongoing and future work, but much of the design choices in κDOT were informed by an attempt to port the initialization system of Summers and Mueller [2011] to WadlerFest DOT. We will discuss some of the ongoing work on initialization in Chapter 8.

The Mutable WadlerFest DOT calculus of Rapoport and Lhoták [2017] extends WadlerFest DOT with a mutable store of references and ML-style references, and we originally intended to add an initialization system to this calculus. However, we found that lazy field reads interacted with ML-style references in counterintuitive ways, which led to the development of κDOT. We discuss some of these issues in Section 2.4.

Although the abstract machine used for the operational semantics of κDOT was inspired by more recent work on proving safety of compiler transformations, it can find its origins in the Mark 1 machine Sestoft [1997].

2.2 WadlerFest DOT

We now present the syntax, operational semantics, and typing rules of WadlerFest DOT.

2.2.1 WadlerFest DOT Syntax

The syntax for WadlerFest DOT is given in Fig. 2.1.

Labels and Variables

a, b, c	Term Labels
A, B, C	Type Labels
x, y, z	Variables

Literals

$l ::= \lambda(z\colon T).t$	Lambda
$\mid \nu(z\colon T)\,d$	Object

Definitions

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

Terms

$t, u ::= x$	Variable
$\mid l$	Literal
$\mid x.a$	Field Read
$\mid x\, x_1$	Application
$\mid \mathsf{let}\ z = t\ \mathsf{in}\ u$	Let Binding

Types

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

Evaluation Contexts and Answers

$e ::= \square$	Hole
$\mid \mathsf{let}\ x = \square\ \mathsf{in}\ t$	Let Bound Term
$\mid \mathsf{let}\ x = l\ \mathsf{in}\ e$	Outer Literal Binding
$n ::= x$	Variable Answer
$\mid l$	Literal Answer
$\mid \mathsf{let}\ x = l\ \mathsf{in}\ n$	Let Answer

Figure 2.1: Syntax of WadlerFest DOT

The WadlerFest DOT calculus uses type labels (A, B, C) for *type members* and term labels (a, b, c) for *term members* of objects. Literals in WadlerFest DOT are lambda functions or objects. Objects in WadlerFest DOT consist of a list of definitions.

Terms in WadlerFest DOT are defined using administrative normal form (ANF)[Sabry and Felleisen, 1993]. Terms in WadlerFest DOT are either a variable x, a literal l, a field read $x.a$, a function application $x\,x_1$, or a let binding $\mathsf{let}\ z = t\ \mathsf{in}\ u$.

The types in WadlerFest DOT are one of the following:

- The *top* and *bottom types* correspond to the top and bottom types of the subtyping lattice.
- A *dependent function* type $\forall\,(z\colon S)\,T$ is the type of a function with a parameter z of type S with a return type of T, which can refer to z.
- A *recursive type* type $\mu\,(z\colon T)$ is the type of an object of type T which can refer to its self-variable z.
- A *field declaration* type $\{a\colon T\}$ says that an object has a term member of type T.
- A *type declaration* type $\{A\colon S..T\}$ says that a term member A is a subtype of T and a super type of S.
- A *type projection* $x.A$ refers to the type member A of the object x.
- An *intersection* type $S \wedge T$ refers to the most general subtype of S and T.

The presentation here is slightly different from Amin et al. [2016], who refer to literals as values. We call them literals here for economy of concepts between WadlerFest DOT and κDOT. In κDOT, literals are not values or answers. In addition, we alert the reader that for objects $\nu\,(z\colon T)\,d$, both T and d are in the scope of z, but when we discuss objects in κDOT, definitions will not be in the scope of z.

2.2.2 WadlerFest DOT Type System

The typing and subtyping rules for WadlerFest DOT are given in Fig. 2.2, Fig. 2.3, and Fig. 2.4.

The rules (Var), (All-E), (Let), and ({}-E) are standard typing rules for variables, function applications, let bindings and field reads.

The (All-I) and ({}-I) rules are used to type function and object literals. An object $\nu\,(z\colon T)\,d$ is given the recursive type $\mu\,(z\colon T)$, by checking the definitions d against the declared type T. Note that the initial types given to type members using the (Def-Typ)

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

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

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)} \quad (\textsc{Rec}\text{-I}) \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \quad (\textsc{Sub})$$

$$\frac{\Gamma \vdash x : \mu(z : T)}{\Gamma \vdash x : [x/z]\,T} \quad (\textsc{Rec}\text{-E}) \qquad \frac{\Gamma, x : T \vdash t : U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash \lambda(x : T).t : \forall(x : T)\,U} \quad (\textsc{All}\text{-I})$$

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

Figure 2.2: Typing in WadlerFest DOT

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

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

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

Figure 2.3: Definition Typing in WadlerFest DOT

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

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

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

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

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a\colon T\} <: \{a\colon U\}} \qquad (\textsc{Fld}\text{-}<:\text{-}\textsc{Fld})$$

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

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

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

Figure 2.4: Subtyping in WadlerFest DOT

$$\frac{t \longmapsto t'}{e[t] \longmapsto e[t']} \quad \text{(Term)}$$

$$\frac{l = \lambda\,(z\colon T)\,.t}{\mathsf{let}\ x = l\ \mathsf{in}\ e[x\,y] \longmapsto \mathsf{let}\ x = l\ \mathsf{in}\ e[[y/z]\,t]} \quad \text{(Apply)}$$

$$\frac{l = \nu\,(z\colon T) \ldots \{a = t\} \ldots}{\mathsf{let}\ x = l\ \mathsf{in}\ e[x.a] \longmapsto \mathsf{let}\ x = l\ \mathsf{in}\ e[[x/z]\,t]} \quad \text{(Project)}$$

$$\mathsf{let}\ x = y\ \mathsf{in}\ t \longmapsto [y/x]\,t \quad \text{(Let-Var)}$$

$$\mathsf{let}\ x = \mathsf{let}\ y = s\ \mathsf{in}\ t\ \mathsf{in}\ u \longmapsto \mathsf{let}\ y = s\ \mathsf{in}\ \mathsf{let}\ x = t\ \mathsf{in}\ u \quad \text{(Let-Let)}$$

Figure 2.5: Operational Semantics for WadlerFest DOT

rule have *tight bounds*, i.e. the lower bound and the upper bound are the same. This means that the declared type T of a well-typed object literal must have tight bounds. The recursion introduction (Rec-I), recursion elimination (Rec-E), and intersection introduction (And-I) rules apply only to variables, but the subsumption rule (Sub) applies to all terms.

For subtyping, the (Top) and (Bot) rules establish the top and bottom types of the subtyping lattice. The (Refl) and (Trans) rules define reflexivity and transitivity of subtyping and the (And_1-<:), (And_2-<:), and (<:-And) rules define standard subtyping rules for intersection types. As is standard, subtyping for function types is contravariant in the parameter type and covariant in the return type by the (All-<:-All) rule. Typing for field declarations is covariant by the (Fld-<:-Fld) rule, and typing for type members declarations are contravariant in the lower bound and covariant in the upper bound by the (Typ-<:-Typ) rule. Note that field typing together with the subtyping rules for intersections allows DOT calculi to have both width and depth subtyping for records. The (Sel-<:) and (<:-Sel) rules make a type projection $x.a$ a subtype of its upper bound and a super type of its lower bound. These last two rules are where most of the expressive power of DOT comes from. While making DOT expressive, these two rules are also where much of the complexities for proving type safety come from.

2.2.3 WadlerFest DOT Operational Semantics

The operational semantics for WadlerFest DOT is given in Fig. 2.5.

The (Term), (Apply), (Project) rules apply inside an evaluation context. The

(Term) rule is a simple congruence rule. The (Apply) rule looks up a function in the evaluation context and performs a call-by-reference function application. The (Project) rule performs a field read, replacing references to the variable z (representing the `this` self-reference) with the let binding variable of the object. Note that field reads are call-by-name in WadlerFest DOT — term labels are associated with terms, not values or references, and the terms are reevaluated after each read. The (Let-Var) rule evaluates let bindings for variables. The (Let-Let) rule evaluates nested let bindings by pulling out nested let bindings to larger scopes.

2.3 Mutable WadlerFest DOT

The Mutable WadlerFest DOT calculus of Rapoport and Lhoták [2017] adds ML-style references to WadlerFest DOT. It adds syntax for references, dereferencing, and reference updates shown below.

- $\mathsf{ref}\ x\ T$ creates a new reference of type T and initializes it with the variable x.
- $!x$ reads the contents of a reference x.
- $x := y$ updates the reference x with the variable y.

A reference containing a variable of type T is given the type $\mathsf{Ref}\ T$, and subtyping between references is expressed by the following invariance rule.

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

Mutable WadlerFest DOT keeps most of the operational semantics of WadlerFest DOT but adds a store of bindings from locations to variables. The additional rules are shown in Fig. 2.6.

2.4 Issues with WadlerFest DOT

While WadlerFest DOT and Mutable WadlerFest DOT are type safe calculi, we found the (Let-Let) rule difficult to work with when proving type safety. We also found it awkward to think about program operation when terms evaluating in one scope move to a different scope. Furthermore, call-by-name field reads interacted counter-intuitively with ML-style references when we tried to express ideas of field initialization for dependently typed fields in Mutable WadlerFest DOT.

$$\frac{p \notin \mathsf{dom}\,(\sigma)}{\sigma \mid e[\mathsf{ref}\ x\ T] \longmapsto \sigma, p = x \mid e[p]} \quad (\textsc{Ref})$$

$$\frac{\sigma(p) = y}{\sigma \mid \mathsf{let}\ x = p\ \mathsf{in}\ e[!x] \longmapsto \mathsf{let}\ x = p\ \mathsf{in}\ e[y]} \quad (\textsc{Deref})$$

$$\sigma \mid \mathsf{let}\ x = p\ \mathsf{in}\ e[x := y] \longmapsto \sigma[p := y] \mid \mathsf{let}\ x = p\ \mathsf{in}\ e[y] \quad (\textsc{Store})$$

Figure 2.6: Operational Semantics for Mutable WadlerFest DOT

To see how the (Let-Let) rule interacts with type safety, we note that the typing rules allow us to type terms with free variables in their type annotations. For example, the following typing derivation is valid in WadlerFest DOT, although the variable *magic* is free.

$$\frac{z\colon magic.A \vdash z\colon magic.A}{\vdash \lambda\,(z\colon magic.A)\,.z\colon \forall\,(z\colon magic.A)\,magic.A}\ (\textsc{All-I})$$

Now consider the following program which has the type $\forall\,(x\colon magic.A)\,magic.A$. The program reduces to a program which has type $\forall\,(x\colon x_1.A)\,x_1.A$ but not the original type $\forall\,(x\colon magic.A)\,magic.A$.

$$\begin{array}{rl}
 & \mathsf{let}\ x_1 = \lambda\,(z\colon \top)\,.z\ \mathsf{in} \\
 & \mathsf{let}\ x_2 = \mathsf{let}\ magic = x_1\ \mathsf{in}\ magic\ \mathsf{in} \\
 & \lambda\,(z\colon magic.A)\,.z \\
(\textsc{Let-Let}) \longmapsto & \mathsf{let}\ x_1 = \lambda\,(z\colon \top)\,.z\ \mathsf{in} \\
 & \mathsf{let}\ magic = x_1\ \mathsf{in} \\
 & \mathsf{let}\ x_2 = magic\ \mathsf{in} \\
 & \lambda\,(z\colon magic.A)\,.z \\
(\textsc{Let-Var}) \longmapsto & \mathsf{let}\ x_1 = \lambda\,(z\colon \top)\,.z\ \mathsf{in} \\
 & \mathsf{let}\ x_2 = x_1\ \mathsf{in} \\
 & \lambda\,(z\colon x_1.A)\,.z
\end{array}$$

So subject reduction does not hold for WadlerFest DOT. The problem is that to type $\mathsf{let}\ x = t\ \mathsf{in}\ \lambda\,(z\colon magic.A)\,.z$, we type $\lambda\,(z\colon magic.A)\,.z$ assuming that only x may be possibly substituted, but in the above reduction we also substituted the variable *magic*.

Since the type system does not naturally guarantee that terms are closed, subject reduction is proved assuming that the term is closed. If $\mathsf{let}\ x = t\ \mathsf{in}\ u$ is closed, no free variables like *magic* appear in u. This complicates the type safety proof since we have to prove additional closure properties for subject reduction.

Operationally, the (Let-Let) rule makes it difficult to understand exactly when a nested let bound literal becomes part of the evaluation context. In the example below, several reduction steps take place before the let bound value finally becomes part of the evaluation context. This makes reasoning about the operational side of WadlerFest DOT tedious.

$$\begin{aligned}
& \mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \mathsf{let}\ x_3 = \lambda(z\colon \top).z\ \mathsf{in}\ x_3\ \mathsf{in}\ x_2\ \mathsf{in}\ x_1 \\
(\textsc{Let-Let}) \longmapsto\ & \mathsf{let}\ x_2 = \mathsf{let}\ x_3 = \lambda(z\colon \top).z\ \mathsf{in}\ x_3\ \mathsf{in} \\
& \mathsf{let}\ x_1 = x_2\ \mathsf{in}\ x_1 \\
(\textsc{Let-Let}) \longmapsto\ & \mathsf{let}\ x_3 = \lambda(z\colon \top).z\ \mathsf{in} \\
& \mathsf{let}\ x_2 = x_3\ \mathsf{in} \\
& \mathsf{let}\ x_1 = x_2\ \mathsf{in}\ x_1
\end{aligned}$$

In WadlerFest DOT and Mutable WadlerFest DOT, fields of an object can be path-dependent on the object itself, but the contents of the field must be defined inside the object. For example, in the following Mutable WadlerFest DOT program, the type of the field $tree.fruit$ is $\mathsf{Ref}\ \{A\colon tree.Fruit\}$, which is path-dependent on $tree$.

$$\begin{aligned}
&\mathsf{let}\ tree = \\
&\quad \nu(z\colon \{Fruit\colon z.Fruit..z.Fruit\} \wedge \{fruit\colon \mathsf{Ref}\ \{A\colon z.Fruit..z.Fruit\}\}) \\
&\qquad \{Fruit = z.Fruit\} \wedge \\
&\qquad \{fruit = \mathsf{let}\ x = \nu(z_1\colon \{A\colon z.Fruit\})\{A = z.Fruit\}\ \mathsf{in} \\
&\qquad\qquad \mathsf{ref}\ x\ \{A\colon z.Fruit..z.Fruit\}\}
\end{aligned}$$

Notice how the entirety of `tree.fruit` had to be defined inside `tree` because of the path-dependent typing. But now, because of call-by-name semantics, a new reference is created every time `tree.fruit` is read. For example, in the code below, x_2 and x_3 refer to

two different objects.

$$
\begin{array}{l}
\mathsf{let}\ tree = \_\ \mathsf{in} \\
\mathsf{let}\ x_1 = tree.fruit\ \mathsf{in} \\
\mathsf{let}\ x_2 = !x_1\ \mathsf{in} \\
\mathsf{let}\ x_3 = tree.fruit\ \mathsf{in} \\
\mathsf{let}\ x_4 = !x_3
\end{array}
\quad \longmapsto^* \quad
\begin{array}{l}
\mathsf{let}\ tree = \_\ \mathsf{in} \\
\mathsf{let}\ fruit_1 = \nu\,(z_1\colon\ \{A\colon z.Fruit\})\,\{A = z.Fruit\}\ \mathsf{in} \\
\mathsf{let}\ x_1 = p_1\ \mathsf{in} \\
\mathsf{let}\ x_2 = fruit_1\ \mathsf{in} \\
\mathsf{let}\ fruit_2 = \nu\,(z_1\colon\ \{A\colon z.Fruit\})\,\{A = z.Fruit\}\ \mathsf{in} \\
\mathsf{let}\ x_3 = p_2\ \mathsf{in} \\
\mathsf{let}\ x_4 = fruit_2 \\
\text{store: } p_1 = fruit_1,\ p_2 = fruit_2
\end{array}
$$

In particular, updating the reference x_1 has no impact on the reference x_3 or on the field `tree.fruit`. The example below would only change what p_2 points to.

$$
\begin{array}{l}
\mathsf{let}\ tree = \_\ \mathsf{in}\ \mathsf{let}\ tree = \_\ \mathsf{in} \\
\mathsf{let}\ fruit_1 = \nu\,(z_1\colon\ \{A\colon z.Fruit\})\,\{A = z.Fruit\}\ \mathsf{in} \\
\mathsf{let}\ x_1 = p_1\ \mathsf{in} \\
\mathsf{let}\ x_2 = fruit_1\ \mathsf{in} \\
\mathsf{let}\ fruit_2 = \nu\,(z_1\colon\ \{A\colon z.Fruit\})\,\{A = z.Fruit\}\ \mathsf{in} \\
\mathsf{let}\ x_3 = p_2\ \mathsf{in} \\
\mathsf{let}\ x_4 = fruit_2\ \mathsf{in} \\
\mathsf{let}\ fruit_3 = \nu\,(z_1\colon\ \{A\colon z.Fruit\})\,\{A = z.Fruit\}\ \mathsf{in} \\
x_3 \coloneqq fruit_3 \\
\text{store: } p_1 = fruit_1,\ p_2 = fruit_2
\end{array}
$$

We found the above to be counter intuitive when programming in the Mutable Wadler-Fest DOT calculus and a very weak approximation to the field mutation allowed by `var` fields in Scala and the field initialization done by Scala. As we will see in subsequent chapters, κDOT mitigates all of the above issues.

Chapter 3

κDOT

We now present the κDOT calculus, a DOT calculus with support for field mutation and constructors. Conceptually, we get κDOT from WadlerFest DOT in the following steps.

1. Ensure literals are always let bound.
2. Remove evaluation contexts and add a heap.
3. Add field mutation.
4. Remove objects from the literal grammar and constructors.

During the development of κDOT, we proved type safety for the calculus after each of the above steps. In the next few sections, we describe the syntax, typing rules, and operational semantics for κDOT before describing type safety in Chapter 4.

3.1 Syntax

The syntax for κDOT is given in Fig. 3.1, where we highlighted the parts that are different from WadlerFest DOT. We make the following changes to the syntax of terms.

- In κDOT, we separate out variables into two categories. Locations are variables that represent items in the heap and abstract variables are used for let bindings and for parameters of functions and constructors.
- We added *setter types* (lower bounds) to field types to soundly type field assignments.

Labels and Variables

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

Definitions

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

Terms

$t, u ::= x$	Variable
$\mid \mathsf{new}\ k\,(\overrightarrow{x})$	Constructor Call
$\mid x.a$	Field Read
$\mid x.a := x_1$	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

Types

$S, T, U ::= \top$	Top Type
$\mid \bot$	Bottom Type
$\mid \forall\,(z\colon S)\,T$	Dependent Function
$\mid \mu\,(z\colon T)$	Recursive Type
$\mid \left\{a\colon S..T\right\}$	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

Frames, Stacks, Heaps, and Configurations

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

Figure 3.1: Syntax of κDOT

- We added constructors to the grammar for literals and added types for constructors. Since objects in κDOT are allocated via constructor calls, we do not need objects to be part of the syntax of terms in κDOT.
- We removed objects from the grammar for literals and added syntax for literal bindings. In κDOT, literals are always referred to by variables.
- We added syntax for field assignment and constructor calls.

Since we give κDOT an operational semantics based on an abstract machine with a stack and a heap, we also added the following syntax for machine states.

- We added heap items, which are literals or objects.
- Unlike objects in WadlerFest DOT, the definitions of an object are not in the scope of the self-variable in κDOT.
- We added frames, stacks, heaps, and configurations. Configurations are machine states.
- We changed the notion of answer to reflect the new operational semantics.

Note that just like in WadlerFest DOT, the grammar for field definitions allows fields to contain arbitrary terms, so field reads are still call-by-name and can reduce to arbitrary terms. But field assignment in κDOT only allows assigning variables to object fields.

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. When we describe the type system for κDOT, setter types will control what can be written to a field. Getter types are the same as the field typing in WadlerFest DOT and tell us the type of the term contained inside a field. Setter and getter types are separated because field reads are a covariant operation whereas field writes are a contravariant operation.

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. Objects allocated by a constructor are first allocated with the default definitions. The body of the constructor is o computation that happens before the allocated object is available to the callee of the constructor.

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 $\{A\colon T\}$.

Since it is a very common pattern to have constructors that take no arguments and are only used once, we will sometimes use the WadlerFest DOT notation $\nu\,(x\colon T)\,d$ as an abbreviation for a constructor creation immediately followed by a constructor call $\mathsf{let}\ k = \kappa\,(z\colon T)\,\{d\}\ z\ \mathsf{in}\ \mathsf{new}\ k\,()$.

Furthermore, for function types where the output does not depend on the input, we will use $T \to U$ to mean $\forall\,(z\colon T)\,U$.

3.2 Operational Semantics

We give the κDOT calculus an operational semantics via an abstract machine (Fig. 3.2). 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. We execute a κDOT term t by initializing the machine with $\langle t;\, \varepsilon;\, \cdot\rangle$ and then starting the machine. The machine then either runs indefinitely (diverges), finishes executing and returns an answer $\langle y;\, \varepsilon;\, \Sigma\rangle$ with y pointing to an item in the heap, or gets stuck.

For each possible shape of the term t and stack s in a configuration, there is at most one possible rule that can apply. Thus the operational semantics is deterministic up to renaming of variables.

Frames and the Stack

The abstract machine supports two kinds of frames, 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 term, or by calling a constructor. When executing $\mathsf{let}\ z = t\ \mathsf{in}\ u$, the (Let-Push) rule pushes the frame $\mathsf{let}\ z = \square\ \mathsf{in}\ u$ onto the stack 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)

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

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

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

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

$$\langle y_1;\ \mathsf{return}\ y :: s;\ \Sigma\rangle \longmapsto \langle y;\ s;\ \Sigma\rangle \qquad \text{(Return)}$$

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

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

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

Figure 3.2: Operational Semantics for κDOT

rule pops the frame from the stack and executes u. Note that only z is ever substituted in u, leaving other variables intact.

If there is a binding of a constructor $k = \kappa\left(\overrightarrow{z\colon T}, y\colon U\right)\{d\}\, t$ in the heap Σ, a constructor call $\mathsf{new}\ k\,(\vec{x})$ does three different things via the (NEW) rule. Firstly, a new object is allocated with the default definitions d as fields and bound to a fresh location y in the heap. Secondly, a return frame $\mathsf{return}\ y$ returning the location of the newly allocated object is pushed onto the stack. Lastly, the body of the constructor t is run. If 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 $\mathsf{let}\ z = \square\ \mathsf{in}\ y$, but we made return frames explicit since we were interested in designing initialization systems for DOT. Return frames separate computation resulting from different constructors.

Field Reads and Writes

For a field read $y.a$, the (PROJECT) rule looks up the object referred to by y and produces the term t that is bound at $y.a$. If the field is not mutated between subsequent reads, t is reevaluated at each read. Thus field reads are lazy without memoization just like in WadlerFest DOT.

A field assignment $y.a \coloneqq y_1$ is evaluated using the (ASSIGNMENT) rule. The rule firstly mutates the heap so that $y.a$ contains the location y_1 and secondly returns y_1 as the focus of execution.

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. Similar to WadlerFest DOT, function applications in κDOT are call-by-reference.

3.3 Type System

Fig. 3.4, Fig. 3.3, Fig. 3.5, and Fig. 3.6 show the typing rules for κDOT. Most of the rules are the same as WadlerFest DOT. Rules that are new or different from WadlerFest DOT

$$\frac{\Gamma,\, z\colon T \vdash t\colon U \qquad z \notin \mathsf{fv}\,(T)}{\Gamma \vdash \lambda\,(z\colon T)\,.t\colon \forall\,(z\colon T)\,U} \tag{All-I}$$

$$\frac{\Gamma,\, \overrightarrow{z\colon T},\, z_1\colon U \vdash d\colon U \qquad \overrightarrow{z} \notin \mathsf{fv}\left(\overrightarrow{T}\right) \qquad \Gamma,\, \overrightarrow{z\colon T},\, z_1\colon U \vdash t\colon T'}{\Gamma \vdash \kappa\left(\overrightarrow{z\colon T}, z_1\colon U\right)\{d\}\,t\colon K\left(\overrightarrow{z\colon T}, z_1\colon U\right)} \tag{K-I}$$

Figure 3.3: Literal Typing in κDOT

$$\frac{\Gamma\,(x) = T}{\Gamma \vdash x\colon T} \tag{Var}$$

$$\frac{\Gamma \vdash x\colon \{a\colon T..U\}}{\Gamma \vdash x.a\colon U} \tag{\{\}-E}$$

$$\frac{\Gamma \vdash x_1\colon T \qquad \Gamma \vdash x\colon \{a\colon T..U\}}{\Gamma \vdash x.a \coloneqq x_1\colon U} \tag{$\coloneqq$-I}$$

$$\frac{\Gamma \vdash x\colon T}{\Gamma \vdash x\colon \mu\,(x\colon T)} \tag{Rec-I}$$

$$\frac{\Gamma \vdash x\colon \mu\,(z\colon T)}{\Gamma \vdash x\colon [x/z]\,T} \tag{Rec-E}$$

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

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

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

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

$$\frac{\Gamma \vdash x\colon T \qquad \Gamma \vdash x\colon U}{\Gamma \vdash x\colon T \wedge U} \tag{And-I}$$

$$\frac{\Gamma \vdash t\colon T \qquad \Gamma \vdash T <\colon U}{\Gamma \vdash t\colon U} \tag{Sub}$$

Figure 3.4: Term Typing in κDOT

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

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

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

Figure 3.5: Definition Typing in κDOT

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

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

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

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

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

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

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

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

Figure 3.6: Subtyping in κDOT

are highlighted.

Since literals are not terms, we type literals separately using literal typing and type let bound literals using the (Lit-I) rule. Constructors are typed using the (K-I) rule. The default definitions d, output type U, and the constructor body may all refer to x_1.

Field assignments are typed using the (:=-I) rule, where a variable may be assigned to a field if it conforms to its setter type. Constructors calls are typed using the (K-E) rule, which produces recursive object types.

For definition typing, we modify the (Def-Trm) rule so that term members are typed with the setter type being the same as the getter type. For a well-typed constructor literal $\kappa\left(\overrightarrow{z\colon T}, z_1\colon T_1 \wedge \cdots \wedge T_n\right)\{d\}$, this means that the bounds in T_i must be tight regardless of whether T_i is a type member or a term member. We modify the (Fld-<:-Fld) subtyping rule to be contravariant in setter types.

We will discuss the proof of type safety in Chapter 4.

3.4 Bringing Scala to DOT

We end the chapter by briefly comparing the fields in Scala, WadlerFest DOT and κDOT.

In κDOT, we tried to preserve as much of WadlerFest DOT as possible while trying to add features inspired by Scala. Scala has `var`s, `val`s, and `lazy val`s and depth-subtyping for `val`s and `lazy val`s, and WadlerFest DOT offers a non-memoized version of `lazy val`s with depth-subtyping.

Fields in κDOT are lazy, but allow for mutation. However, although fields can contain arbitrary terms, we only allow writing fully evaluated locations to fields. This is directly inspired by Scala, where only fully evaluated locations are allowed to be written to `var` fields. Furthermore, κDOT allows depth-subtyping on getter types. As we will see in Chapters 5 and 6 this kind of field, together with constructors, allows us to express all programming patterns available with `var`s and `lazy val`s.

In κDOT, we can even provide some immutability guarantees using subtyping to make the setter type $\bot$. This is slightly different from the type of immutablity guaranteed by Scala. Scala disallows all writes to `val` fields, whereas if we have an object y with a field a of type $\{a\colon \bot..\top\}$ and a variable x of type $\bot$, κDOT will allow the write $y.a := x$. However, the type safety proof in Chapter 4 will show that a fully evaluated location can never have type $\bot$, so $y.a := x$ will never be executed.

Chapter 4

Type Safety for κDOT

In this chapter we outline the type safety proof for κDOT. We prove type safety for κDOT by proving progress and preservation lemmas in the style of Wright and Felleisen [1994]. We will start by defining what type safety means for our operational semantics. We will then look at the challenges in proving type safety for DOT calculi. Next, we will look at the proof infrastructure that we developed and used for proving the canonical forms lemmas. Lastly, we will discuss details of our mechanized proof of type safety.

4.1 Configuration Typing for Type Safety

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.

Theorem 4.1.1 (Type Safety). *If* $\vdash t : T$*, then either the initial configuration of the abstract machine* $\langle t; \varepsilon; \cdot \rangle$ *diverges or* $\langle t; \varepsilon; \cdot \rangle \longmapsto^* \langle y; \varepsilon; \Sigma \rangle$ *for some answer* $\langle y; \varepsilon; \Sigma \rangle$*.*

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

The proofs were mechanized and checked in Coq and are available at https://git.uwaterloo.ca/ikabir/dot-public.

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

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

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

Figure 4.1: Stack Typing in K-DOT

Definition 4.1.2 (Heap Correspondence). *For a context* Γ *and an environment* Σ*, we say that* Γ corresponds *to* Σ*, written* $\Gamma \sim \Sigma$*, if* Γ *and* Σ *have the same domain, and for all* $x : T \in \Gamma$ *and* $x = h \in \Sigma$*,*

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

Heap Correspondence is essentially literal typing (Fig. 3.3) together with objects typed using definition typing.

Fig. 4.1 shows the typing rules for stacks. Stacks represent evaluation contexts and are given two types $\Gamma \vdash s : S, U$. Here, S is the type that the focus of execution must have in order for the overall evaluation context to have type U.

The use of subtyping in the (Stack Empty) rule allows us to avoid defining subtyping explicitly between stacks and configurations. When a term t has type S and $S <: U$, then t also has type U. Similarly, when a configuration pairs t with an empty stack the overall configuration has type U.

The (Stack Let) rule closely mirrors the (Let) typing rule. For $x \notin \mathsf{fv}(S)$ and $\Gamma, x : S \vdash u : T$, if a term t has type S, then $\mathsf{let}\ x = t\ \mathsf{in}\ u$ has type T. Similarly, if t is used as the focus of execution and $\mathsf{let}\ x = \square\ \mathsf{in}\ u$ is added on top of a stack s of type $\Gamma \vdash s : T, U$, they behave as if $\mathsf{let}\ x = t\ \mathsf{in}\ u$ is paired with s.

The (Stack Return) rule ensures that, after a constructor call, the stack is typed with the type of the allocated object while allowing the constructor body to have any type.

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

Definition 4.1.3 (Configuration Typing). $\Gamma \vdash \langle t; s; \Sigma \rangle : U$ *if* $\Gamma \sim \Sigma$ *and for some type* T, $\Gamma \vdash t : T$ *and* $\Gamma \vdash s : T, U$.

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.

4.2 Inert Types for Type Safety

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

Definition 4.2.1 (Record Type). *A type* T *is a* record type *if* T *is the intersection of tight field declarations* $\{a : S..S\}$ *and tight type* $\{A : S..S\}$ *declarations of distinct labels. A field (type) declaration* $\{a : S..U\}$ *(*$\{A : S..U\}$*) is* tight *if its bounds* S *and* U *are the same.*

Definition 4.2.2 (Inert Type). *A type* U *is* inert *if*

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

Definition 4.2.3 (Inert Context). Γ *is an* inert context *if the type* $\Gamma(x)$ *that it binds to each variable* x *is inert.*

Note that definition typing always produces record types. Since a constructor is typed by ensuring the default definitions correspond to the declared output type, a well-typed constructor must have a record type as its declared output type. Thus, an object created by calling a well-typed constructor must have an inert type. Inert types essentially capture precise types of objects and literals in the heap that result from executing a well-typed κDOT program. When a well-typed program is executed, the focus of execution is always typed under an inert context. Inert contexts are free from bad bounds and do not allow unsound subtyping judgments such as $K(a : T) <: \forall (z : T)\, T$ and $\forall (z : T)\, T <: \{a : T\}$. Using inertness, we define the progress and preservation lemmas as follows.

Lemma 4.2.4 (Progress). *If* Γ *is inert and* $\Gamma \vdash c : U$*, then either* c *is an answer or there exists* c' *such that* $c \longmapsto c'$*.*

Lemma 4.2.5 (Preservation). *If* Γ *is inert,* $\Gamma \vdash c : U$*, and* $c \longmapsto c'$*, then there exists* Γ' *such that the concatenation* $\Gamma +\!\!+ \Gamma'$ *is inert and* $\Gamma +\!\!+ \Gamma' \vdash c' : U$*.*

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

An interesting feature of our preservation lemma is that it adds an inertness condition to the conclusion. We use inertness to ensure all objects in our heap and typing context are typed using tight bounds, which makes proving the canonical forms lemmas easier. In the next section we will discuss the bad bounds problem for DOT calculi which makes proving canonical forms lemmas difficult. We will explore runtime consequences of inertness in Section 6.5.

4.3 Bad Bounds Break Canonical Forms

In general, canonical forms lemmas tell us that the type of a term informs us of its shape. For example, the simply typed lambda calculus has a canonical forms lemma for function types [Pierce and C, 2002, p. 105].

Lemma 4.3.1 (Canonical Forms for Functions in STLC). *If* v *is a value of type* $T \to U$*, then there exists a* t *such that* $v = \lambda\,(x : T)\,.t$*.*

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 $\Gamma \vdash y : \{A : T..U\}$, and subtyping follows from the (<:-Sel), (Sel-<:), and (Trans) subtyping rules. The requirement of $\Gamma \vdash T <: U$ is easily satisfied if $y : \{A : T..U\} \in \Gamma$ or $y : \bot \in \Gamma$.

This means that there are contexts where canonical forms does not hold and terms which would be ill-typed under more traditional type systems can be typed in DOT. For example, the following context gives a variable bound to a function literal a constructor type and then allows the new operation to be called on that variable.

$$y : \{A : \forall\,(z : \top)\,\top..K\,(z : U)\} \vdash \mathsf{let}\ k = \lambda\,(z : \top)\,.z\ \mathsf{in}\ \mathsf{new}\ k\,() : \mu\,(z : U)$$

Here, although $\lambda(z\colon \top).z$ is given the type $\forall(z\colon \top)\top$ through literal typing, $\mathsf{new}\ k()$ is typed in the context $y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top$. Here k is given a constructor type via the following typing and subtyping rules.

$$y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top \vdash \forall(z\colon \top)\top <\colon y.A \qquad (<\colon\text{-Sel})$$

$$y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top \vdash y.A <\colon K(z\colon U) \qquad (\text{Sel-}<\colon)$$

$$y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top \vdash \forall(z\colon \top)\top <\colon K(z\colon U) \qquad (\text{Trans})$$

$$y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top \vdash k\colon \forall(z\colon \top)\top \qquad (\text{Var})$$

$$y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top \vdash k\colon K(z\colon U) \qquad (\text{Sub})$$

$$y\colon \{A\colon \forall(z\colon \top)\top..K(z\colon U)\}, k\colon \forall(z\colon \top)\top \vdash \mathsf{new}\ k()\colon \mu(z\colon U) \qquad (\text{K-E})$$

In the lambda calculus, we get canonical forms by an inversion of the typing relation, but a similar inversion does not work for DOT calculi. However, Amin et al. [2012] noticed that since objects are always created with tight bounds, if we start from a well typed term, bad bounds do not occur in any context where reduction will take place. For the type safety proof for κDOT, we insist on inert contexts because bad bounds do not occur in these contexts. In an inert context, we are able to invert the typing relation for canonical forms.

4.4 Substitution Lemma for Path-Dependence

The standard substitution lemma for a call-by-value lambda calculus is as follows [Pierce and C, 2002, p. 106].

Lemma 4.4.1 (Call-by-value Substitution Lemma). *If* $\Gamma, x\colon T \vdash t\colon U$ *and* $\Gamma \vdash v\colon T$ *for a value* v*, then* $\Gamma \vdash [v/x]\,t\colon U$.

The above lemma is used in the abstraction elimination case of the preservation lemma. It tells us that since an abstraction is typed using the following rule, when we apply the abstraction to a value v of the correct type and produce $[v/x]\,t$, the type is preserved.

$$\frac{\Gamma, x\colon T \vdash t\colon U}{\Gamma \vdash \lambda(x\colon T).t\colon T \to U} \qquad (\text{Abs-I})$$

Since we are in a call-by-reference setting, the type of substitution we perform is simpler; we only replace an abstract variable with a location (both are variables) rather than substitute a value for a variable. This simplifies the required substitution property to the following form.

Property (Call-by-reference Substitution). *If* $\Gamma, x\colon T \vdash t\colon U$ *and* $\Gamma \vdash y\colon T$ *for a location* y*, then* $\Gamma \vdash [y/x]\, t\colon U$.

However, the above property does not hold in the presence of path-dependent types. We illustrate this with the following example. Suppose we have the following for $x \neq y$:

$$\Gamma, x\colon \mu\,(z\colon \{A\colon \top\}) \vdash x\colon x.A$$
$$\Gamma \vdash y\colon \mu\,(z\colon \{A\colon \top\})$$

Here, $\Gamma, x\colon \mu\,(z\colon \{A\colon \top\}) \vdash \top <\colon x.A$ by recursion elimination and the ($<\colon$-Sel) rule. Thus, x can be given the path-dependent type $x.A$ through subtyping. Here x and y have the same type as required by the substitution property, but if we remove the reference x from the typing context, then we remove the subtyping relation $\top <\colon x.A$, so we cannot type y as $\Gamma \vdash y\colon x.A$. The problem can occur even when the variable being substituted does not occur in the type or the term.

$$\Gamma, x : \mu\,(z\colon \{A\colon \bot\}), x_1 : x.A, x_2 : \forall\,(z_1\colon \bot)\,\top \vdash x_2\, x_1\colon \top$$
$$\Gamma \vdash y\colon \mu\,(z\colon \{A\colon \bot\})$$
$$\Gamma, x_1 : x.A, x_2 : \forall\,(z_1\colon \bot)\,\top \not\vdash x_2\, x_1 : \top$$

Similar to the above example, we have $\Gamma, x\colon \mu\,(z\colon \{A\colon \bot\}) \vdash x.A <\colon \bot$ and x_1 can be given the type $\bot$ by subtyping. If we remove the reference x from the typing context, then we remove the subtyping relation $x.A <\colon \bot$, and so we cannot type x_1 as $\bot$. Hence we cannot use x_1 as an input to x_2 since inputs to x_2 have to have type $\bot$.

Since the simple substitution property does not hold, for path-dependent types we need a more general substitution lemma.

Lemma 4.4.2 (Substitution Lemma). *If* $\Gamma_1, x : T, \Gamma_2 \vdash t\colon U$ *and* $\Gamma_1 \vdash y\colon [y/x]\, T$ *for a location* y*, then* $\Gamma_1, [y/x]\, \Gamma_2 \vdash [y/x]\, t\colon [y/x]\, U$.

In the above lemma, we substitute all references to x with y, including in the context Γ_2 and the type U. Replacing x with y in the context and the type allows us to replace x with y in all typing and subtyping judgments which were used to derive $\Gamma_1, x : T, \Gamma_2 \vdash t\colon U$ and derive $\Gamma_1, [y/x]\, \Gamma_2 \vdash [y/x]\, t\colon [y/x]\, U$.

4.5 Working from the Bottom Up

Amin et al. [2014] comment that instead of trying to prove type safety proof for an entire DOT calculus from scratch, DOT type safety proofs are easier to develop by starting from

an existing type safety proof and adding features in a bottom-up fashion and fixing the parts of type safety proof which break while keeping most of the proof intact. We followed their advice and slowly evolved κDOT from the simplified proof of type safety Rapoport et al. [2017] for WadlerFest DOT.

WadlerFest DOT has an operational semantics based on evaluation contexts where both values and variables are 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. *Well-typed* is a relation analogous to heap correspondence, but for evaluation contexts.
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 through a static analysis of constructors. To keep the static analysis localized to constructors, we needed constructors to be closed after type erasure. Therefore, constructors need multiple arguments since any references needed by constructors need to be passed as inputs. 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.

4.6 Mechanization and Locally Nameless Representation

The mechanized type safety proof of κDOT uses a locally nameless representation together with cofinite quantification of free variables introduced by binding rules such as (All-I) and (Let). The equivalence between rules using cofinite quantification and the usual rules are discussed by Aydemir et al. [2008]. In this representation, we represent abstract variables with de Bruijn indices and locations by named variables. The binding rules under cofinite quantification are shown in Fig. 4.2, Fig. 4.3, and Fig. 4.4. In the rules shown, L represents a finite set of variables.

$$\frac{\forall y',\ y' \notin L \implies \Gamma,\, y' \colon T \vdash [y'/z]\, t \colon [y'/z]\, U}{\Gamma \vdash \lambda\,(z \colon T)\,.t \colon \forall\,(z \colon T)\,U} \tag{All-I}$$

$$\frac{\begin{array}{l}\forall \overrightarrow{y'}, y'_1,\ \overrightarrow{y'}, y'_1 \notin L \implies \Gamma,\, \overrightarrow{y' \colon T},\, y'_1 \colon [\overrightarrow{y', y'_1}/\overrightarrow{z, z_1}]\, U \vdash [\overrightarrow{y', y'_1}/\overrightarrow{z, z_1}]\, d \colon [\overrightarrow{y', y'_1}/\overrightarrow{z, z_1}]\, U \\ \forall \overrightarrow{y'}, y'_1,\ \overrightarrow{y'}, y'_1 \notin L \implies \Gamma,\, \overrightarrow{y' \colon T},\, y'_1 \colon [\overrightarrow{y', y'_1}/\overrightarrow{z, z_1}]\, U \vdash [\overrightarrow{y', y'_1}/\overrightarrow{z, z_1}]\, t \colon [\overrightarrow{y', y'_1}/\overrightarrow{z, z_1}]\, T'\end{array}}{\Gamma \vdash \kappa\left(\overrightarrow{z \colon T}, z_1 \colon U\right)\{d\}\, t \colon K\left(\overrightarrow{z \colon T}, z_1 \colon U\right)} \tag{K-I}$$

Figure 4.2: Mechanized Literal Typing

$$\frac{\begin{array}{c}\forall y' \notin L \qquad \Gamma,\, y' \colon T \vdash [y'/z]\, u \colon U \\ \Gamma \vdash t \colon T\end{array}}{\Gamma \vdash \mathsf{let}\ z = t\ \mathsf{in}\ u \colon U} \tag{Let}$$

$$\frac{\begin{array}{c}\forall y' \notin L \qquad \Gamma,\, y' \colon T \vdash [y'/z]\, u \colon U \\ \Gamma \vdash l \colon T\end{array}}{\Gamma \vdash \mathsf{let}\ z = l\ \mathsf{in}\ u \colon U} \tag{Lit-I}$$

Figure 4.3: Mechanized Typing for Let Bindings

$$\frac{\begin{array}{c}\forall y' \notin L \qquad \Gamma,\, y' \colon T_2 \vdash [y'/z]\, U_1 <: [y'/z]\, U_2 \\ \Gamma \vdash T_2 <: T_1\end{array}}{\Gamma \vdash \lambda\,(z \colon T_1)\,.U_1 \colon \forall\,(z \colon T_2)\,U_2} \tag{All-<:-All}$$

Figure 4.4: Mechanized Subtyping

In our mechanized Coq proof we use the TLC library provided by Arthur Chargueraud which provides many primitives for working with locally nameless representations.

For type safety we prove the weakening, narrowing, substitution, and renaming lemmas and then build infrastructure to prove canonical forms for inert contexts. We then proceed to prove the progress and preservation lemmas.

4.6.1 Weakening, Narrowing, Substitution and Renaming

We now state the weakening, narrowing, substitution, and renaming lemmas for κDOT.

Lemma 4.6.1 (Weakening). *If* $\Gamma_1, \Gamma_2 \vdash t : T$ *and* $x \notin \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$*, then* $\Gamma_1, x : T, \Gamma_2 \vdash t : T$*.*

The Weakening lemma tells us that extra information in typing contexts does no harm. We prove weakening by a straightforward mutual induction on subtyping and the typing relations for terms, definitions, and literals.

Lemma 4.6.2 (Narrowing). *If* $\Gamma_1, x : T, \Gamma_2 \vdash t : U$ *and* $\Gamma_1 \vdash T' <: T$ *then* $\Gamma_1, x : T', \Gamma_2 \vdash t : U$*.*

The Narrowing lemma tells us that more precise information in typing contexts does no harm. Similar to the weakening lemma, we prove narrowing by a straightforward mutual induction on subtyping and the typing relations for terms, definitions, and literals.

Lemma 4.6.3 (Substitution Lemma). *If* $\Gamma_1, x : T, \Gamma_2 \vdash t : U$ *and* $\Gamma_1 \vdash y : [y/x]\, T$ *for a location* y*, then* $\Gamma_1, [y/x]\, \Gamma_2 \vdash [y/x]\, t : [y/x]\, U$*.*

We prove substitution by a mutual induction on subtyping and the typing relations for terms, definitions, and literals.

We then combine weakening and substitution to prove the renaming lemmas. Renaming says that replacing an abstract variable with a location of the same type preserves typing.

Lemma 4.6.4 (Renaming for Let Bindings). *If* $\forall y' \notin L$*,* Γ*,* $y' : T \vdash [y'/z]\, t : U$ *and for a location* y*,* $\Gamma \vdash y : T$*, then* $\Gamma \vdash [y/z]\, t : U$*.*

The above says that reducing a let binding via the (Let-Push) rule preserves configuration typing.

Lemma 4.6.5 (Renaming for Function Applications). *If* $\forall y' \notin L$*,* Γ*,* $y' : T \vdash [y'/z]\, t : [y'/z]\, U$ *and for a location* y*,* $\Gamma \vdash y : T$*, then* $\Gamma \vdash [y/z]\, t : [y/z]\, U$*.*

The above says that reducing a function application via the (Application) rule preserves term typing.

Lemma 4.6.6 (Renaming for Constructor Calls). *If*

- $\forall \overrightarrow{y', y_1'} \notin L$, Γ, $\overrightarrow{y' : T}$, $y_1' : [\overrightarrow{y', y_1'}/\overrightarrow{z, z_1}]\, U \vdash [\overrightarrow{y', y_1'}/\overrightarrow{z, z_1}]\, t : [\overrightarrow{y', y_1'}/\overrightarrow{z, z_1}]\, T'$,
- *for locations* $\vec{y}$, $\Gamma \vdash \overrightarrow{y : T}$, *and*
- $y_1 \notin \mathsf{dom}(\Gamma)$

then $\Gamma, x_1 : \mu(z_1 : [\vec{x}/\vec{z}]\, U) \vdash [\overrightarrow{y, y_1}/\overrightarrow{z, z_1}]\, t : [\overrightarrow{y, y_1}/\overrightarrow{z, z_1}]\, T'$.

Lemma 4.6.7 (Renaming Definitions for Constructor Calls). *If*

- $\forall \overrightarrow{x, x_1} \notin L$, Γ, $\overrightarrow{x : T}$, $x_1 : [\overrightarrow{x, x_1}/\overrightarrow{z, z_1}]\, U \vdash [\overrightarrow{x, x_1}/\overrightarrow{z, z_1}]\, d : [\overrightarrow{x, x_1}/\overrightarrow{z, z_1}]\, U$,
- *for locations* $\vec{y}$, $\Gamma \vdash \overrightarrow{y : T}$, *and*
- $y_1 \notin \mathsf{dom}(\Gamma)$

then $\Gamma, x_1 : \mu(z_1 : [\vec{x}/\vec{z}]\, U) \vdash [\overrightarrow{y, y_1}/\overrightarrow{z, z_1}]\, d : [\overrightarrow{y, y_1}/\overrightarrow{z, z_1}]\, U$.

The above say that reducing a constructor application via the (New) rule preserves configuration typing.

The renaming lemmas all have a similar form and so do their proofs. We show the proof of Lemma 4.6.5.

Proof of Lemma 4.6.5. We pick a named variable y' fresh enough so that $y' \notin L$, $[y/y']\, \Gamma = \Gamma$, $[y/z]\, t = [y/y']\, [y'/z]\, t$, and $[y/z]\, U = [y/y']\, [y'/z]\, U$.

Then from the premise, since $y' \notin L$, Γ, $y' : T \vdash [y'/z]\, t : [y'/z]\, U$.

Now, by the substitution lemma, $\Gamma \vdash [y/y']\, [y'/z]\, t : [y/y']\, [y'/z]\, U$.

Since y' is fresh enough, this is the same as $\Gamma \vdash [y/z]\, t : [y/z]\, U$. □

4.6.2 Inverting Variable Typing

In κDOT, progress and preservation is proved by an inversion of configuration typing followed by an induction on the typing for the focus of execution. For many of the cases, we need to relate the type of a variable to its type in the typing context; for progress we need this to prove canonical forms lemmas and for preservation to apply renaming lemmas. However, these relationships are not available to us directly through an inversion of the

typing judgment. In this section we discuss the function application and field read cases of progress and preservation to illustrate the relationships we need between the type of a variable and its type in the context. We end the section by discussing why inverting variable typing is difficult in κDOT.

Preservation for Function Applications

Suppose we want to prove type preservation for $\langle y\, y_1;\, s;\, \Sigma\rangle \longmapsto \langle [y_1/z]\, t;\, s;\, \Sigma\rangle$, where $y = \lambda\,(z\colon T')\,.t \in \Sigma$ and $\Gamma \vdash y\colon \forall\,(z\colon T)\,U$, $\Gamma \vdash y_1\colon T$, and $\Gamma \vdash y\, y_1\colon\ [y_1/z]\, U$. By inversion of heap correspondence, we get the following:

- $\Gamma\,(y) = \forall\,(z\colon T')\,U'$ for some U'.
- $\forall y' \notin L'$, $\Gamma \vdash [y'/z]\, t\colon\ [y'/z]\, U'$ for some finite set L'.

To apply the renaming lemmas for preservation, we need the following:

- $\forall y' \notin L$, Γ, $y'\colon T \vdash [y'/z]\, U' <\colon [y'/z]\, U$ for some L.
- $\Gamma \vdash T <\colon T'$.

However, a direct inversion of $\Gamma \vdash y\colon \forall\,(z\colon T)\,U$ gives us the following cases.

- $\Gamma\,(y) = \forall\,(z\colon T)\,U$ (Var)
- $\forall\,(z\colon T)\,U = [y/z']\, S$ and $\Gamma \vdash y\colon \mu\,(z'\colon S)$ (Rec-E)
- $\Gamma \vdash S <\colon \forall\,(z\colon T)\,U$ and $\Gamma \vdash y\colon S$ (Sub)

In the (Rec-E) case, it is not helpful to further invert $\Gamma \vdash y\colon \mu\,(z'\colon S)$ since among the generated cases there will be another (Rec-E) case. Similarly, in the (Sub) case, inverting $\Gamma \vdash y\colon S$ will generate another (Sub) case among others. A direct inversion here is not helpful since we can keep building chains of inversions.

For the (Sub) case, inverting the subtyping judgment $\Gamma \vdash S <\colon \forall\,(z\colon T)\,U$ is not helpful since it produces many cases where we can again build chains of inversions. For example, in the (Trans) rule we would have $\Gamma \vdash S' <\colon S$ and $\Gamma \vdash S <\colon \forall\,(z\colon T)\,U$, where we can keep building an inversion chain by inverting $\Gamma \vdash S' <\colon S$.

Progress for Function Applications

Suppose we want to prove type progress for $\langle y\, y_1;\, s;\, \Sigma\rangle$, $\Gamma \vdash y\colon \forall\,(z\colon T)\,U$, $\Gamma \vdash y_1\colon T$, and $\Gamma \vdash y\, y_1\colon\ [y_1/z]\, U$. We need to invert the typing relation on y to get the type $\Gamma\,(y)$. We

need a canonical forms lemma to show that $\Gamma(y) = \forall(z\colon T')\,U'$ for some T', U' and $\forall y' \notin L$, $\Gamma,\, y'\colon T \vdash [y'/z]\,U' <\colon [y'/z]\,U$ for some L.

As above, trying to invert $\Gamma \vdash y\colon \forall(z\colon T)\,U$ to relate $\forall(z\colon T)\,U$ and $\Gamma(y)$ causes the same issues.

Preservation for Field Reads

We run into similar issues when we try to prove type preservation for $\langle y.a;\, s;\, \Sigma\rangle \longmapsto \langle t;\, s;\, \Sigma\rangle$, where $y = \nu(z\colon S)\ldots\{a = t\}\ldots \in \Sigma$ and $\Gamma \vdash y\colon \{a\colon T..U\}$ and $\Gamma \vdash y.a\colon U$. By inversion on heap correspondence, we get $\Gamma \vdash y\colon \{a\colon T'..T'\}$ and $\Gamma \vdash y.a\colon T'$ for some T'. For preservation, we need $\Gamma \vdash T' <\colon U$.

Similar to the function application case, trying to invert $\Gamma \vdash y\colon \{a\colon T..U\}$ to relate $\{a\colon T..U\}$ and $\{a\colon T'..T'\}$ results in chains of inversion.

Issues with Inverting Variable Typing

For all of the above cases, we need to relate the type of a variable to the type that is bound in the context. This is not directly available to us through inversion because typing derivations can be arbitrarily long in κDOT. For instance, we get chains of recursion elimination because we can have chains of recursion introduction forming a large type $\mu(z_1\colon \mu(z_2\colon \ldots \mu(z_n\colon T)))$ which are eliminated by chains of recursion elimination. Similarly, we can have chains of intersection introductions which are eliminated by chains of the ($\textsc{And}_1$-<:), ($\textsc{And}_2$-<:), and (Sub) rules.

The usual approach to proving inversion properties when a direct inversion is not possible is by induction on the typing derivation. However, not only can typing judgments in κDOT be arbitrarily long, the typing rules also allow cycling between pairs of introduction and elimination rules. For instance, if we have $\Gamma \vdash x\colon T$, then $\Gamma \vdash x\colon \mu(z\colon T)$ via the (Rec-I) rule, and the (Rec-E) rule allows us to derive the original $\Gamma \vdash x\colon T$. These cycles impede induction proofs, since the proposition $\Gamma \vdash x\colon T$ is justified by $\Gamma \vdash x\colon \mu(z\colon T)$, which is in turn justified by the original proposition $\Gamma \vdash x\colon T$. Similarly, we can cycle between intersection introductions from the (And-I) rule and intersection elimination from the ($\textsc{And}_1$-<:), ($\textsc{And}_2$-<:), and (Sub) rules.

In general, inverting variable typing in κDOT is difficult because of the following reasons:

- Subtyping can happen through path-dependent types, and a direct inversion of typing does not take into account the lack of bad bounds in our context.
- Variable typing can go through repeated and arbitrarily long cycles of intersection introduction and elimination.
- Variable typing can go through repeated and arbitrarily cycles of recursion elimination and introduction.

For our type safety proof, we need to invert typing for variables with function types, field declarations, and constructor types. Instead of dealing with the different cases for individually, we define additional typing relations and build proof infrastructure for inverting variable typing.

4.7 Infrastructure for Inverting Typing

To prove type safety for κDOT and relate the types of variables to the types bound in the context, we introduce three additional typing relations: precise flow of variable types, tight typing and subtyping, and invertible typing for variables.

A precise flow from U to T is denoted as $\Gamma \vdash_! x\colon U \gg T$. It says that $\Gamma(x) = U$ and T is derived from U by only applying rules for recursion elimination and intersection elimination. If $\Gamma \vdash_! x\colon \Gamma(x) \gg T$ is a precise flow in an inert context, if T is a function or constructor type, then $\Gamma(x) = T$, and if T is a declaration then $\Gamma(x) = \mu(z\colon S)$ for a record type S containing T.

A tight typing judgment $\Gamma \vdash_\# t\colon T$ says that t was typed using tight subtyping. A tight subtyping judgment is the same as subtyping except that (<:-Sel) and (Sel-<:) rules are modified so that only tight bounds from precise typing; i.e if $\Gamma \vdash_\# x\colon \{A\colon T\}\, T$ then $\Gamma \vdash_\# T <: x.A$ and $\Gamma \vdash_\# x.A <: T$. In an inert context general typing is equivalent to tight typing since record types have tight bounds.

Invertible typing for variables uses only introduction rules. It uses precise flow to give variables an initial type and then introduction rules give variables further types. This removes cycles of introduction and elimination rules from our typing judgments since all elimination rules are isolated to a single use of precise flow. In inert contexts, tight typing for variables implies invertible typing.

$$\frac{\Gamma(x) = U}{\Gamma \vdash_! x \colon U \gg U} \text{ (Var-!)} \qquad \frac{\Gamma \vdash_! x \colon U \gg S \wedge T}{\Gamma \vdash_! x \colon U \gg S} \text{ (And}_1\text{-E-!)}$$

$$\frac{\Gamma \vdash_! x \colon U \gg \mu\,(z \colon T)}{\Gamma \vdash_! x \colon U \gg\ [x/z]\,T} \text{ (Rec-E-!)} \qquad \frac{\Gamma \vdash_! x \colon U \gg S \wedge T}{\Gamma \vdash_! x \colon U \gg T} \text{ (And}_2\text{-E-!)}$$

Figure 4.5: Precise Flow of Variable Typing

Lastly, to relate variable typing to the type bound in the context, we go through the following "proof recipe":

- Since we are in an inert context, general typing is the same as tight typing.
- For variables, tight typing implies invertible typing.
- Inverting invertible typing and precise flow relate types of variables to the type bound in the context.

4.7.1 Precise Flow

The precise flow relation is shown in Fig. 4.5. $\Gamma \vdash_! x \colon U \gg T$ reads as the type of x flows from U to T, where T is derived from U by eliminating recursion and intersections and $\Gamma\,(x) = U$.

The precise flow relation provides a single reusable syntax for proving inversion lemmas about elimination rules and inert contexts. For example, we can prove the following lemmas by inverting precise flow. These lemmas tell us that recursion and intersection elimination rules do not apply to constructor and function types.

Lemma 4.7.1. *If* $\Gamma \vdash_! x \colon \forall\,(z \colon T)\,U \gg S$ *then* $S = \forall\,(z \colon T)\,U$

Lemma 4.7.2. *If* $\Gamma \vdash_! x \colon K\left(\overrightarrow{z \colon T}, z_1 \colon U\right) \gg S$ *then* $S = K\left(\overrightarrow{z \colon T}, z_1 \colon U\right)$.

For the types of objects we can prove the following lemma about recursion and intersection elimination.

Lemma 4.7.3. *If* $\Gamma \vdash_! x \colon \mu\,(z \colon T) \gg S$ *for a record type* T*, then* $S = \mu\,(z \colon T)$ *or* S *is a record type.*

For inert contexts, we can prove the following inversion lemmas.

Lemma 4.7.4. *If* Γ *is an inert context and* $\Gamma \vdash_! x\colon U \gg \forall\,(z\colon S)\,T$*, then* $U = \forall\,(z\colon S)\,T$*.*

Lemma 4.7.5. *If* Γ *is an inert context and* $\Gamma \vdash_! x\colon U \gg K\left(\overrightarrow{z\colon T}, z_1\colon U\right)$*, then* $U = K\left(\overrightarrow{z\colon T}, z_1\colon U\right)$*.*

Lemma 4.7.6. *If* Γ *is an inert context and* $\Gamma \vdash_! x\colon U \gg S$ *for a record type* S*, then* $U = \mu\,(z\colon T)$ *for a record type* T*.*

Lemma 4.7.7. *If* Γ *is an inert context and* $\Gamma \vdash_! x\colon \mu\,(z\colon T) \gg S$ *for a declaration* S*, then* $[x/z]\,T$ *contains the declaration* S*.*

The above lemmas say that in an inert context, function and constructor types only flow from themselves and record types and declarations flow from recursive object types.

4.7.2 Tight Typing

Types in inert contexts only have tight bounds. To reason about typing without bad bounds, we introduce tight typing and tight subtyping which isolate the possible use of bad bounds to function and constructor literals, where parameters may have bad bounds, and let bindings. The tight typing relation is shown in Fig. 4.6 and Fig. 4.7.

We use general typing for literals so that the body of functions and constructors can be typed using bad bounds from parameters. However, since execution can never create objects with bad bounds, literals with bad bounds are never called during execution.

While tight typing does not use any bad bounds for typing terms, the type produced by tight typing may have bad bounds. For example, the following use of the ({}-E-#) rule allows $x.a$ to have a type with bad bounds, although the type of x itself has tight bounds. The bounds for $\{A\colon \top..\bot\}$ are bad because it allows for $\top <: \bot$ which collapses the entire subtyping lattice.

$$\frac{\Gamma \vdash_\# x\colon \{a\colon \{A\colon \top..\bot\}..\{A\colon \top..\bot\}\}}{\Gamma \vdash_\# x.a\colon \{A\colon \top..\bot\}}\ (\{\}\text{-E-}\#)$$

To type $\mathsf{let}\ x = t\ \mathsf{in}\ u$, the (Let-#) rule uses tight typing to type t but general typing to type u since the type of t might introduce bad bounds.

Other than the above uses of general typing, tight typing is free from the use of bad bounds. Because we use precise flow with tight bounds in the (<:-Sel-#) and (Sel-<:-#)

$$\frac{\Gamma(x) = T}{\Gamma \vdash_{\#} x : T} \quad (\textsc{Var}\text{-}\#)$$

$$\frac{\Gamma \vdash_{\#} x : \{a : T..U\}}{\Gamma \vdash_{\#} x.a : U} \quad (\{\}\text{-E-}\#)$$

$$\frac{\Gamma \vdash_{\#} x : \{a : T..U\} \qquad \Gamma \vdash_{\#} x_1 : T}{\Gamma \vdash_{\#} x.a \coloneqq x_1 : U} \quad (\coloneqq\text{-I-}\#)$$

$$\frac{\Gamma \vdash_{\#} x : T}{\Gamma \vdash_{\#} x : \mu(x : T)} \quad (\textsc{Rec}\text{-I-}\#)$$

$$\frac{\Gamma \vdash_{\#} x : \mu(z : T)}{\Gamma \vdash_{\#} x : [x/z]\,T} \quad (\textsc{Rec}\text{-E-}\#)$$

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

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

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

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

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

$$\frac{\Gamma \vdash_{\#} t : T \qquad \Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{\#} t : U} \quad (\textsc{Sub}\text{-}\#)$$

Figure 4.6: Tight Term Typing in κDOT

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

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

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

$$\frac{\Gamma \vdash_{!} x \colon U \gg \{A \colon T..T\}}{\Gamma \vdash_{\#} T <: x.A} \; (<:\text{-}\textsc{Sel}\text{-}\#) \qquad \frac{\Gamma \vdash_{!} x \colon U \gg \{A \colon T..T\}}{\Gamma \vdash_{\#} x.A <: T} \; (\textsc{Sel}\text{-}<:\text{-}\#)$$

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

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

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

$$\frac{\Gamma \vdash_{\#} S <: T \qquad \Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{\#} S <: U} \quad (\textsc{Trans}\text{-}\#)$$

Figure 4.7: Tight Subtyping in κDOT

rules for subtyping between types and type selections, bad bounds are never introduced into the tight subtyping lattice. In an inert context, all bounds are tight and thus general typing and tight typing are equivalent.

Theorem 4.7.8 ($\vdash$ to $\vdash_{\#}$). *If Γ is an inert context, then $\Gamma \vdash t : T$ implies $\Gamma \vdash_{\#} t : T$, and $\Gamma \vdash S <: U$ implies $\Gamma \vdash_{\#} S <: U$.*

4.7.3 Invertible Typing

The invertible typing relation is shown in Fig. 4.8. Under invertible typing, variables are first given precise types using precise flow. The (Fld-<:-Fld-##), (Typ-<:-Typ-##), and (All-<:-All-##) rules give less precise types to variables through tight subtyping. The rest of the rules introduce type selections, the top type, intersection types, and recursive types.

Invertible typing is essentially an inlining of tight subtyping into tight typing for variables. So for inert contexts, the following theorem holds.

Theorem 4.7.9 ($\vdash_{\#}$ to $\vdash_{\#\#}$). *If Γ is an inert context, then $\Gamma \vdash_{\#} x : T$ implies $\Gamma \vdash_{\#\#} x : T$.*

Invertible typing does not use any elimination rules, in particular type selections, top types, intersection types, and recursive types are never eliminated. This means that the only way invertible typing introduces functions, constructors, and declarations is through precise flow and then possibly making them less precise through the (Fld-<:-Fld-##), (Typ-<:-Typ-##), and (All-<:-All-##) rules. This allows us to prove inversion lemmas such as the following by induction.

Lemma 4.7.10 (Inversion for Function Typing). *If Γ is inert and $\Gamma \vdash_{\#\#} x : \forall\,(z : S_2)\,T_2$, then $\Gamma\,(x) = \forall\,(z : S_1)\,T_1$, $\Gamma \vdash_{\#} S_2 <: S_1$, and $\Gamma,\, x : S_2 \vdash T_1 <: T_2$.*

Proof. We prove the lemma by induction on invertible typing. There are only two cases that apply. For the (Var-##) case, inverting precise typing and the fact that $\Gamma\,(x)$ is inert tells us that $\Gamma\,(x) = \Gamma \vdash_{\#\#} x : \forall\,(z : S_2)\,T_2$. The subtyping results follow from reflexivity of subtyping. We prove the (All-<:-All-##) case by applying the induction hypothesis together with transitivity of tight and general subtyping. □

In the mechanized proof, the (All-<:-All-##) rule uses cofinite quantification, so the mechanized version of the above lemma is as follows.

$$\frac{\Gamma \vdash_! x\colon U \gg T}{\Gamma \vdash_{\#\#} x\colon T} \qquad (\textsc{Var}\text{-}\#\#)$$

$$\frac{\Gamma \vdash_{\#\#} x\colon T}{\Gamma \vdash_{\#\#} x\colon \mu\,(x\colon T)} \qquad (\textsc{Rec-I}\text{-}\#\#)$$

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

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

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

$$\frac{\Gamma \vdash_{\#\#} x\colon T \qquad \Gamma \vdash_{\#\#} x\colon U}{\Gamma \vdash_{\#\#} x\colon T \wedge U} \qquad (\textsc{And-I}\text{-}\#\#)$$

$$\frac{\Gamma \vdash_{\#\#} x\colon T \qquad \Gamma \vdash_! y\colon U \gg \{A\colon T..T\}}{\Gamma \vdash_{\#\#} x\colon y.A} \qquad (<\text{:-}\textsc{Sel}\text{-}\#\#)$$

$$\frac{\Gamma \vdash_{\#\#} x\colon T}{\Gamma \vdash_{\#\#} x\colon \top} \qquad (\textsc{Top}\text{-}\#\#)$$

Figure 4.8: Invertible Variable Typing

Lemma 4.7.11 (Mechanized Inversion for Function Typing). *If* Γ *is inert and* $\Gamma \vdash_{\#\#} x \colon \forall\,(z \colon S_2)\,T_2$*, then* $\Gamma\,(x) = \forall\,(z \colon S_1)\,T_1$*,* $\Gamma \vdash_{\#} S_2 <: S_1$*, and there exists a finite set* L *such that for all* $x \notin L$*,* $\Gamma,\, x \colon S_2 \vdash [x/z]\,T_1 <: [x/z]\,T_2$*.*

The above lemma is used for the function application cases for progress and preservation. We discuss applying the above lemma together with the theorems relating general typing, tight typing, and invertible typing in Section 4.7.4.

4.7.4 Applying the Proof Recipe

We now return to the function application cases for progress and preservation and show how the proof recipe is applied. The cases for constructor calls, field reads, and field writes are very similar.

Preservation for Function Applications

We need to show $\Gamma \vdash [y_1/z]\,t \colon\ [y_1/z]\,U$ given the following:

1. Γ inert,
2. $y = \lambda\,(z \colon T')\,.t \in \Sigma$,
3. $\Gamma \vdash y \colon \forall\,(z \colon T)\,U$,
4. $\Gamma \vdash y_1 \colon T$,
5. $\Gamma \vdash y\,y_1 \colon\ [y_1/z]\,U$,
6. $\Gamma\,(y) = \forall\,(z \colon T')\,U'$ for some U', and
7. $\forall y' \notin L',\ \Gamma,\, y' \colon T' \vdash [y'/z]\,t \colon\ [y'/z]\,U'$ for some finite set L'.

Proof. Since Γ is inert and $\Gamma \vdash y \colon \forall\,(z \colon T)\,U$, we have $\Gamma \vdash_{\#} y \colon \forall\,(z \colon T)\,U$ by Theorem 4.7.8 ($\vdash$ to $\vdash_{\#}$). Now, by applying Theorem 4.7.9 ($\vdash_{\#}$ to $\vdash_{\#\#}$), we have $\Gamma \vdash_{\#\#} y \colon \forall\,(z \colon T)\,U$. By Lemma 4.7.11 (Mechanized Inversion) and since $\Gamma\,(y) = \forall\,(z \colon T')\,U'$, we have $\Gamma \vdash_{\#} T <: T'$ and for some finite set L, $\forall y' \notin L$, $\Gamma,\, y' \colon T \vdash [y'/z]\,U' <: [y'/z]\,U$.

Since, for any y', $y' \notin (L' \cup L)$ implies $y' \notin L'$ and $y' \notin L$, we have

- $\forall y' \notin (L' \cup L),\ \Gamma,\, y' \colon T' \vdash [y'/z]\,t \colon\ [y'/z]\,U'$, and
- $\forall y' \notin (L' \cup L),\ \Gamma,\, y' \colon T \vdash [y'/z]\,U' <: [y'/z]\,U$.

Since the tight subtyping rules are just more restrictive versions of the general subtyping rules, $\Gamma \vdash_{\#} T <: T'$ implies $\Gamma \vdash T <: T'$.

Thus, by subsumption and narrowing, we have $\forall y' \notin (L' \cup L),\ \Gamma,\, y' \colon T \vdash [y'/z]\,t \colon\ [y'/z]\,U$. Lastly, by renaming (Lemma 4.6.5) we have the result $\Gamma \vdash [y_1/z]\,t \colon\ [y_1/z]\,U$. □

Progress for Function Applications

We need to show that the (Application) rule applies to the configuration $\langle y\, y_1;\ s;\ \Sigma\rangle$ given the following:

1. Γ inert and Γ corresponds to Σ,
2. $\Gamma \vdash y \colon \forall\,(z \colon T)\, U$,
3. $\Gamma \vdash y_1 \colon T$, and
4. $\Gamma \vdash y\, y_1 \colon [y_1/z]\, U$.

Proof. Since Γ is inert and $\Gamma \vdash y \colon \forall\,(z \colon T)\, U$, we have $\Gamma \vdash_{\#} y \colon \forall\,(z \colon T)\, U$ by Theorem 4.7.8 ($\vdash$ to $\vdash_{\#}$). Now, by applying Theorem 4.7.9 ($\vdash_{\#}$ to $\vdash_{\#\#}$), we have $\Gamma \vdash_{\#\#} y \colon \forall\,(z \colon T)\, U$. By Lemma 4.7.11 (Mechanized Inversion), we have $\Gamma\,(y) = \forall\,(z \colon T')\, U'$ for some T', U'.

Now, for canonical forms we invert heap correspondence to get $y = \lambda\,(z \colon T')\,.t \in \Sigma$.

Thus, the (Application) rule applies and we step to $\langle [y_1/z]\, t;\ s;\ \Sigma\rangle$. □

4.8 Type Safety

We remind the reader that for our operational semantics, type safety states that if we start our abstract machine with a well-typed term, then the machine either runs indefinitely or successfully returns an answer.

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

We prove type safety by proving progress and preservation lemmas for configuration typing with heaps corresponding to inert contexts. Both lemmas are proven by inducting on typing for the focus of execution. Since a well-typed term leads to a well-typed initial configuration, the above type safety theorem follows.

Lemma (Progress). *If Γ is inert and $\Gamma \vdash \langle t;\ s;\ \Sigma\rangle : U$, then either $\langle t;\ s;\ \Sigma\rangle$ is an answer or there exists c' such that $\langle t;\ s;\ \Sigma\rangle \longmapsto c'$.*

Proof. We invert $\Gamma \vdash \langle t;\ s;\ \Sigma\rangle : U$ to get the following:

- Γ corresponds to Σ.
- $\Gamma \vdash t \colon T$ for some T.
- $\Gamma \vdash s \colon T, U$.

We proceed by induction on typing for the focus of execution, $\Gamma \vdash t : T$. The cases are as follows:

(Var), (Rec-I), (Rec-E), (And-I) In these cases the focus of execution is a variable. We either have an empty stack in which case the configuration is an answer, or we can pop a stack frame using either the (Return) rule or the (Let-Loc) rule.

(All-E) This is the function application case covered in the previous section.

(K-E) In this case, we are calling a constructor. Similar to the function application case, we invert variable typing for the constructor variable and retrieve the constructor from the heap to apply the (New) rule with a fresh location.

({}-E) This is the field read case. Similar to the function application case, we invert variable typing on the field declaration to retrieve the field definition from the heap to apply the (Project) rule.

(:=-I) This is the field write case. Similar to the field write case, we invert variable typing on the field declaration to retrieve the field definition on the heap, and then we update this definition using the (Assignment) rule.

(Lit-I) Here the focus of execution is a let binding of a literal and the literal is simply pushed to the heap using the (Let-Lit) rule.

(Let) Here the focus of execution is a let binding of a term and the (Let-Let) rule applies, pushing a new let frame onto the stack.

(Sub) Here the induction hypothesis applies.

□

Lemma (Preservation). *If* Γ *is inert,* $\Gamma \vdash \langle t;\, s;\, \Sigma \rangle : U$*, and* $\langle t;\, s;\, \Sigma \rangle \longmapsto c'$*, then there exists* Γ' *such that the concatenation* $\Gamma +\!\!+ \Gamma'$ *is inert and* $\Gamma +\!\!+ \Gamma' \vdash c' : U$*.*

Proof. Similar to the proof of the progress lemma, we invert $\Gamma \vdash \langle t;\, s;\, \Sigma \rangle : U$ to get the following:

- Γ corresponds to Σ.
- $\Gamma \vdash t : T$ for some T.
- $\Gamma \vdash s : T, U$.

We proceed by induction on the typing for the focus of execution, $\Gamma \vdash t : T$. The cases are as follows:

(Var), (Rec-I), (Rec-E), (And-I) In these cases the focus of execution is a variable. We invert the reduction relation to get that we either pop a return frame via the (Return) rule or pop a let frame via the (Let-Loc) rule. In the (Return) case, inverting stack typing tells us configuration typing is preserved, and in the (Let-Loc) case, we apply the renaming lemma for let bindings.

(All-E) This is the function application case. We invert the reduction relation and show that the type for the focus of execution is preserved as shown in the previous section.

(K-E) In this case, we are calling a constructor. Similar to the function application case, we invert the reduction relation, invert variable typing for the constructor, and apply renaming lemmas for the constructor body and allocated object. An additional step in this case is that we have to show heap correspondence is preserved.

({}-E) This is the field read case, where the focus of execution is $y.a$ and $\Gamma \vdash y : \{a : S..T\}$. We invert the reduction relation, to get that the machine steps to $\langle t;\ s;\ \Sigma\rangle$ and $y = \nu\,(z : \_) \ldots \{a = t\} \ldots \in \Sigma$. Similar to the function application case, we invert variable typing on the field declaration to get $\{a : T'..T'\} \in \Gamma\,(y)$ with $\Gamma \vdash T' <: T$. We invert heap correspondence to get $\Gamma \vdash t : T'$. By subsumption, the type of the focus of execution is preserved.

(:=-I) This is the field write case, where the focus of execution is $y.a := y'$ and $\Gamma \vdash y : \{a : T..U'\}$ and $\Gamma \vdash y' : T$. Similar to the field write case, we invert the reduction relation and variable typing on the field declaration to get $\{a : T'..T'\} \in \Gamma\,(y)$ with $\Gamma \vdash T <: T'$. By subsumption $\Gamma \vdash y' : T'$ and so $\Gamma \vdash \{a = y'\} : \{a : T'..T'\}$. Thus heap correspondence is preserved after the update.

(Lit-I) Here the focus of execution is a let binding of a literal. We show that pushing a literal onto the heap preserves heap correspondence, and apply a renaming lemma for let bindings to show that the type for the focus of execution is preserved.

(Let) Here the focus of execution is a let binding of a term and preservation follows from stack typing.

(Sub) Here the induction hypothesis applies.

□

Chapter 5

Expressive Power of κDOT

In this chapter, we explore the expressive power of κDOT. We will first show that κDOT is at least as expressive as System $F_{<:}$, WadlerFest DOT, and Mutable WadlerFest DOT by showing how their constructs can be encoded in κDOT. Next, we will discuss recursive constructs that can be expressed in DOT calculi. While these recursive constructs can be expressed in both WadlerFest DOT and κDOT there are some programming patterns that cannot be expressed in WadlerFest DOT. We will end the chapter by discussing how memoized lazy fields of Scala (`lazy vals`) can be expressed in κDOT, but not in WadlerFest DOT or Mutable WadlerFest DOT.

5.1 Encoding Other Calculi in κDOT

In this section we will see how κDOT can encode many of the interesting constructs of other calculi, showing that κDOT is at least as expressive as these calculi. We will first show how System $F_{<:}$'s type abstraction and type application can be encoded in κDOT. Next we will show how the object literals of WadlerFest DOT can be encoded in κDOT. Lastly we will show how κDOT can encode the mutable references of Mutable WadlerFest DOT even though it does not have a separate mutable store.

5.1.1 Encoding System $F_{<:}$ in κDOT

System $F_{<:}$ has two constructs that are not directly available in κDOT or other DOT calculi: bounded type abstraction and type application. These can all be encoded in

κDOT, as well as WadlerFest DOT and Mutable WadlerFest DOT, using type members.

A type abstraction of the form $\lambda\,(X <: U)\,.t$, where X is a type variable and U is a type, can be encoded in κDOT as follows:

$$\lambda\,(x\colon\ \{A\colon \bot..U\})\,.t$$

A type application of the form $(x\,T)$ can be encoded in κDOT as:

$$\begin{array}{l}\mathsf{let}\ carryT = \nu\,(z\colon\ \{A\colon T..T\})\,\{A = T\}\ \ \mathsf{in}\\ (x\,carryT)\end{array}$$

In the above κDOT code, we use the WadlerFest DOT object notation as an abbreviation for creating and calling a constructor. WadlerFest DOT and Mutable WadlerFest DOT encode type abstraction and type application in the same way, except they use object literals directly instead of creating and calling a constructor.

5.1.2 Encoding WadlerFest DOT in 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.

Function literals $\lambda\,(z\colon T)\,.t$ of WadlerFest DOT are simply let bound in κDOT as the following.

$$\mathsf{let}\ x = \lambda\,(z\colon T)\,.t\ \mathsf{in}\ x$$

Object literals can be encoded 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 the following.

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

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

5.1.3 Encoding Mutable WadlerFest DOT in κDOT

We now show how we can encode constructs from Mutable WadlerFest DOT in κDOT. Mutable WadlerFest DOT adds a mutable store and references to WadlerFest DOT. In κDOT we can emulate ML-style references without needing a separate mutable store because our heap is already mutable. The encodings are as follows:

$\mathsf{ref}\ x\ T$	$\mathsf{let}\ refT = \kappa\,(z : T, z' \colon \{a \colon T\})\,\{\{a = z\}\}\,z'\ \mathsf{in}$ $\mathsf{new}\ refT\,(x)$
$\mathsf{!x}$	$x.a$
$x := y$	$x.a := y$

A reference creation just creates an object with a single field and stores the variable in the field. A dereference is emulated by a field read and a reference update is emulated by a field assignment.

The subtyping rule between references can also be emulated in κDOT through field subtyping. Below, the (Ref-Sub) rule on the left is the reference subtyping rule from Mutable WadlerFest DOT and the rule on the right is an instance of the (Fld-<:-Fld) rule of κDOT that emulates (Ref-Sub).

$$\frac{\Gamma \vdash T <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash \mathsf{Ref}\ T <: \mathsf{Ref}\ U}\ (\textsc{Ref-Sub}) \qquad \frac{\Gamma \vdash U <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash \{a \colon T..T\} <: \{a \colon U..U\}}\ (\textsc{Fld-<:-Fld})$$

5.2 Recursive Constructs in κDOT

In this section, we discuss recursive constructs in DOT calculi using κDOT as the example calculus. The programs discussed in this section can be expressed in WadlerFest DOT and Mutable WadlerFest DOT and variants of them can be expressed in other DOT calculi.

5.2.1 Recursive Functions and Infinite Loops in κDOT

In κDOT, recursive functions are written in a direct style by putting the function in an object and referring back to the function through the `this` variable of the object.

For example, a simple infinite loop can be written as:

$$\begin{array}{l}\mathsf{let}\ carryA = \nu\,(z\colon\ \{a\colon \forall\,(z\colon \top)\,\bot\})\,\{a = \lambda\,(z_1\colon \top)\,.\mathsf{let}\ z_2 = z.a\ \mathsf{in}\ z_2\,z_1\}\ \ \mathsf{in}\\ \mathsf{let}\ x = carryA.a\ \mathsf{in}\\ x\,carryA\end{array}$$

Running the above program causes the κDOT abstract machine to repeatedly allocate the function $\lambda\,(z_1\colon \top)\,.\mathsf{let}\ z_2 = z.a\ \mathsf{in}\ z_2\,z_1$ onto the heap and push $\mathsf{let}\ z_2 = \square\ \mathsf{in}\ z_2\,z_1$ onto the stack with appropriate substitutions.

We can also define infinite loops using lazy field reads:

$$\Omega = \mathsf{let}\ x = \nu\,(z\colon\ \{a\colon \bot\})\,\{a = z.a\}\ \ \mathsf{in}\ x.a$$

Running the above allocates the object $\nu\,(z\colon\ \{a\colon \bot\})\,\{a = x.a\}$ onto the heap, and repeatedly accesses $x.a$.

$$\begin{array}{rl} & \langle \Omega;\, \varepsilon;\, \cdot \rangle \\ \longmapsto^* & \langle x.a;\, \varepsilon;\, x = \nu\,(z\colon\ \{a\colon \bot\})\,\{a = x.a\} \rangle \\ \longmapsto & \langle x.a;\, \varepsilon;\, x = \nu\,(z\colon\ \{a\colon \bot\})\,\{a = x.a\} \rangle \end{array}$$

Notice how both of the above programs are typed as $\bot$. In Chapter 6, we will use Ω as the notion of null reference in κDOT.

5.2.2 Recursive Types in κDOT

In κDOT, types can refer to themselves through self-references of objects the same way that terms refer to themselves. This means, that the type system is expressive enough to type constructs such as the Y-combinator without using a direct style of function recursion. The following program defines a Y-combinator for functions of the type $\forall\,(z\colon T)\,U \to \forall\,(z\colon T)\,U$ only using type recursion. We added extra type annotations on let bindings for the benefit of the reader.

$$
\begin{array}{l}
\mathsf{let}\ mu = \nu\,(mu\colon\ \{A\colon mu.A \to \forall\,(z\colon T)\,U\})\,\{A = mu.A\}\ \ \mathsf{in} \\
\mathsf{let}\ yT = \lambda\,(f\colon \forall\,(z\colon T)\,U \to \forall\,(z\colon T)\,U)\,. \\
\quad \mathsf{let}\ x_f : mu.A = \lambda\,(x\colon mu.A)\,. \\
\quad\quad \mathsf{let}\ y' : \forall\,(z\colon T)\,U = \lambda\,(y\colon T)\,. \\
\quad\quad\quad \mathsf{let}\ x' : \forall\,(z\colon T)\,U = x\,x\ \mathsf{in} \\
\quad\quad\quad x'\,y \\
\quad\quad \mathsf{in} \\
\quad\quad f\,y' \\
\quad \mathsf{in} \\
\quad x_f\,x_f
\end{array}
$$

Here yT has the type $(\forall\,(z\colon T)\,U \to \forall\,(z\colon T)\,U) \to \forall\,(z\colon T)\,U$. To see that the program implements the Y-combinator, recall that the Y-combinator for the call-by-value lambda calculus is $\lambda\,f.(\lambda\,x.f(\lambda\,y.xxy))(\lambda\,x.f(\lambda\,y.xxy))$.

5.3 Memoized Laziness in κDOT

The lazy val fields in Scala are lazily evaluated, but once evaluated the result is memoized. So when the code in Fig. 5.1 executes, fruits.mangoes is only evaluated in the first read of fruits.mangoes; in the 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. 5.2 shows a naive approach to encoding the field reads of Fig. 5.1 in κDOT.

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

Figure 5.1: `lazy vals` in Scala

$$
\begin{aligned}
&\mathsf{let}\ mangoes2 = fruits.mangoes\ \mathsf{in}\\
&\mathsf{let}\ \_ = fruits.mangoes := mangoes2\ \mathsf{in}\\
&\mathsf{let}\ mangoes2Again = fruits.mangoes\ \mathsf{in}\\
&\mathsf{let}\ \_ = fruits.mangoes := mangoes2
\end{aligned}
$$

Figure 5.2: Manual Memoization in κDOT

$$
\begin{aligned}
&\mathsf{let}\ kFruits =\\
&\quad \kappa\,(z : \{mangoes\colon Mangoes\} \wedge \{getMangoes\colon \top \to Mangoes\})\\
&\qquad \{\{mangoes = \mathsf{new}\ kMangoes\,()\}\\
&\qquad \wedge \{getMangoes = \lambda\,(z_1\colon \top)\,.\mathsf{let}\ x = z.mangoes\ \mathsf{in}\ z.mangoes := x\}\}\\
&\mathsf{in}\\
&\mathsf{let}\ fruits = \mathsf{new}\ kFruits\,()\ \ \mathsf{in}\\
&\mathsf{let}\ mangoes2 = \mathsf{let}\ x = fruits.getMangoes\ \mathsf{in}\ x\,x\ \mathsf{in}\\
&\mathsf{let}\ mangoes2Again = \mathsf{let}\ x = fruits.getMangoes\ \mathsf{in}\ x\,x\ \mathsf{in}
\end{aligned}
$$

Figure 5.3: Memoization via Getter Functions in κDOT

However, since getter and setter types can be different, the above may not be typeable in the context in which the field read occurs. One possible solution is to completely replace lazy semantics in our calculus with a memoized lazy semantics by having the abstract machine push update frames on field reads. We opt for a different approach and ask that memoized lazy fields be decided at constructor definition time, just like in Scala. Memoized lazy fields can then be accessed through getter functions. So the program in Fig. 5.1 can be encoded in κDOT as shown in Fig. 5.3. In Fig. 5.3, calling `fruits.getMangoes` for the first time evaluates the `mangoes` field and subsequent calls return the cached value.

While we have shown how `lazy vals` can be encoded in κDOT, `lazy vals` cannot be encoded in WadlerFest DOT or Mutable WadlerFest DOT. WadlerFest DOT has no notion of mutation and hence it has no notion of memoization. The references in Mutable WadlerFest DOT cannot store terms and can only store references to fully evaluated values, so a similar encoding of `lazy vals` will not work. Thus, in this sense, κDOT is more expressive than WadlerFest DOT and Mutable WadlerFest DOT.

Chapter 6

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, where fields are initialized with lazily evaluated terms.

In this section, we will define a notion of null reference and initialization for κDOT and show how strictly evaluating fields in Scala can be emulated in κDOT. We will show why similar notions do not work for other DOT calculi such as Mutable WadlerFest DOT. We will also show an example of a static analysis that is possible in κDOT.

Next, we will discuss how initialization influences the metatheory of κDOT. We will discuss how bad bounds influence the operation of the abstract machine for κDOT and discuss a conjecture about how an initialization system can be used to enrich the subtyping relation in variants of κDOT.

6.1 Nulls in κDOT

We define Ω to be the notion of null in κDOT. We redefine it here in a more Scala-like style for the readers benefit.

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

Ω has type $\bot$, and executing it causes the abstract machine to allocate the object $\nu\,(z\colon\ \{null\colon \bot\})\,\{null = x.null\}$ onto the heap and loop indefinitely reading $nPkg.null$. Since Ω has type $\bot$, it also has all other types via subtyping.

$$
\begin{aligned}
& \langle \Omega;\ s;\ \Sigma \rangle \\
\longmapsto^{*}\ & \langle nPkg.null;\ s;\ nPkg = \nu\,(z\colon\ \{null\colon \bot\})\,\{null = x.null\} :: \Sigma \rangle \\
\longmapsto\ & \langle nPkg.null;\ s;\ nPkg = \nu\,(z\colon\ \{null\colon \bot\})\,\{null = x.null\} :: \Sigma \rangle \\
\longmapsto\ & \langle nPkg.null;\ s;\ nPkg = \nu\,(z\colon\ \{null\colon \bot\})\,\{null = x.null\} :: \Sigma \rangle \\
\vdots & \qquad\qquad\qquad \vdots
\end{aligned}
$$

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 reference exception is executing a default Ω.

6.2 Fully Evaluated Path-dependent Fields

Amin et al. [2016] comment that compared to simply typed strict records in other calculi, 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". However, this does not hold for fields with path-dependent types. For example, for the Scala program in Fig. 6.1, 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 in a closed term. So we cannot express the program in Fig. 6.1 by pre-evaluating mangoTree.fruit through a let binding.

In WadlerFest DOT, dependent fields must always be lazy. In κDOT however, a field can start off being lazy as part of the default definitions allocated by a constructor but later be fully evaluated through assignment. For example, we can express the program in Fig. 6.1 in κDOT as shown in Fig. 6.2. In Fig. 6.2, 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. In this sense, κDOT is more expressive than WadlerFest DOT since κDOT can express fully evaluated path-dependent fields. In particular, κDOT can emulate strict mutable fields (`vars`) of Scala.

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

Figure 6.1: Initializing a dependently typed object

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

Figure 6.2: Fully Evaluated Fields in κDOT

$$
\begin{array}{l}
\mathsf{let}\ mangoTree = \\
\quad \nu\,(z : \{TreeFruit\colon z.TreeFruit\} \\
\qquad \wedge \{fruit\colon \mathsf{Ref}\ \{A\colon z.TreeFruit\}\}) \\
\qquad TreeFruit = z.TreeFruit \\
\qquad fruit = \mathsf{let}\ x = \Omega\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{ref}\ x\ \{A\colon z.Fruit\} \\
\mathsf{in} \\
\mathsf{let}\ mangoFruit = \\
\quad \nu\,(z_1\colon \{A\colon mango.Fruit\})\,\{A = mango.Fruit\} \\
\mathsf{in} \\
\mathsf{let}\ mangoRef = mangoTree.fruit\ \mathsf{in} \\
mangoRef := mangoFruit
\end{array}
$$

Figure 6.3: Divergent Program in Mutable WadlerFest DOT

6.3 ML-style References Cannot Express Nulls

A similar notion of null references does not directly carry over to calculi with ML-style references which can only contain objects or literals since programs like Ω are neither objects nor literals. The references in the calculi discussed by Amin and Rompf [2017] must always point to an object and the references in Mutable WadlerFest DOT must always point to a literal. So, attempting to assign a default bottom typed program to a reference and updating it later causes the program to diverge.

For example, a naive port of the program in Fig. 6.2 to Mutable WadlerFest DOT is shown in Fig. 6.3. Beyond the problems highlighted in Chapter 1, here executing the field `mangoTree.fruit` just once causes $\mathsf{let}\ x = \Omega$ to get executed and causes the program to diverge.

6.4 Definite Assignment Analysis in κDOT

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. Given

our above notion of nulls, we can define objects to be initialized as follows.

Definition 6.4.1 (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 $\nu\,(x\colon T)\,d$ in Σ and every term declaration in x is bound to a location.*

Definition 6.4.2 (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.*

In κDOT, a simple way to ensure objects produced by a constructor are locally initialized is by using a definite assignment analysis to ensure the constructor assigns variables to all fields via the `this` self reference in a constructor body.

Scala allows assignments of `null` and uninitialized paths to fields, so a definite assignment analysis is not enough to ensure objects are locally initialized since an assignment might have assigned a `null` to a field. However, κDOT only assigns locations to fields and during assignment the location must refer to a heap item, so a simple definite assignment analysis is enough to ensure that the object returned by the constructor is locally initialized. In the κDOT code that follows, *kBad* does not pass a definite assignment analysis since it does not assign to field a and *kGood* and *kLoop* do pass the definite assignment analysis since their bodies always assign to every field. When the body of the *kGood* constructor, the a field is assigned the fully evaluated `this` variable, and therefore produces a locally initialized object.

$$\begin{aligned}
&\mathsf{let}\ kBad = \kappa\,(z\colon\ \{a\colon T\})\,\{\{a = \Omega\}\}\,z\\
&\mathsf{let}\ kGood = \kappa\,(z\colon\ \{a\colon \top\})\,\{\{a = \Omega\}\}\,z.a \coloneqq z\\
&\mathsf{let}\ kLoop = \kappa\,(z\colon\ \{a\colon T\})\,\{\{a = \Omega\}\}\,\mathsf{let}\ x = z.a\ \mathsf{in}\ z.a \coloneqq x
\end{aligned}$$

A definite assignment analysis of the constructor body does not ensure that the constructor will always return an object, only that if it does return an object, it will be locally initialized. In the above κDOT code, although the *kLoop* constructor passes a definite assignment check, the constructor body diverges and therefore the constructor does not ever return an object.

6.5 Bad Bounds and Divergent Programs

As discussed in Section 4.3, bad bounds allow 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. However, κDOT is still type-safe because $f.a$ is never executed.

$$
\begin{array}{l}
\mathsf{let}\ f = \lambda\,(x\colon \top)\,.x\ \mathsf{in} \\
\mathsf{let}\ x = \Omega\ \mathsf{in} \\
f.a
\end{array}
$$

$$
\begin{array}{ll}
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash x\colon \bot & (\textsc{Var}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash \bot <\colon \{A\colon \forall\,(z\colon \top)\,\top..\{a\colon \top\}\} & (\textsc{Bot}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash x\colon \{A\colon \forall\,(z\colon \top)\,\top..\{a\colon \top\}\} & (\textsc{Sub}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash x.A <\colon \{a\colon \top\} & (\textsc{Sel-}<\colon) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash \forall\,(z\colon \top)\,\top <\colon x.A & (<\colon\textsc{-Sel}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash \forall\,(z\colon \top)\,\top <\colon \{a\colon \top\} & (\textsc{Trans}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash f\colon \forall\,(z\colon \top)\,\top & (\textsc{Var}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash f\colon \{a\colon \top\} & (\textsc{Sub}) \\
f : \forall\,(z\colon \top)\,\top, x : \bot \vdash f.a\colon \top & (\{\}\text{-E})
\end{array}
$$

$f.a$ is typed in a context with $\Gamma\,(x) = \bot$ and subtyping allows us to use the bad bounds $\forall\,(x\colon \top)\,\top..\{a\colon \top\}$. Then using the ($<\colon$-Sel), (Sel-$<\colon$), and (Trans) rules we give the function type $\forall\,(x\colon \top)\,\top$ a super type of $\{a\colon \top\}$. Then by subsumption, f has type $\{a\colon \top\}$ and hence we are allowed to read the field a of f.

When proving type safety for κDOT, we 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 $f.a$.

6.6 Path-Dependent Subtyping via Initialization

Bad bounds are one of the reasons why κDOT and other DOT calculi currently offer 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 fully path-dependent types, consider the type $\mu\,(x\colon \{a\colon \{A\colon T..U\}\})$. 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: $\mathsf{let}\ y = x.a$.

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

$$\begin{array}{l}\mathsf{let}\ k = \kappa\,(x\colon \{a\colon \{A\colon T..U\}\})\,\{\{a = \Omega\}\}\,x\ \mathsf{in}\\ \mathsf{let}\ x = \mathsf{new}\ k\,()\end{array}$$

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. In the rules shown on the left, since we know that x is locally initialized, we know that $x.a$ refers to a fully evaluated object containing a definition $\{A = T'\}$ with $S <: T' <: T$. Thus we should be able to reason that $S :< x.a.A$ and $x.a.A <: T$. Similarly, for the rules on the right, since x is fully initialized, $x \ldots a$ refers to a fully evaluated object with a definition $\{A = T'\}$ with $S <: T' <: T$ and thus we should be able to reason that $S :< x \ldots a.A$ and $x \ldots a.A <: T$. Therefore there are metatheoretic reasons to care about initialization even if we are happy with lazy semantics.

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

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

Chapter 7

Design Choices

In this section, we compare some of our design choices for κDOT to other DOT calculi and Scala.

7.1 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. Being able to define constructors at runtime allows us to keep all the features of WadlerFest DOT objects since WadlerFest DOT allowed objects with different types to be defined at runtime.

Unlike constructors in Scala, constructors in κDOT do not allow constructor parameters to be path-dependent on earlier parameters. However, this is not a strong limitation since nested function parameters can be path-dependent on earlier function parameters and we can always wrap a constructor inside nested functions.

κDOT does not define any subtyping relation between constructor types to keep the calculus simple. Since we were interested in expressing ideas of object initialization in a DOT calculus and κDOT already has subtyping between objects, subtyping between constructor types would complicate the inversions we would need to do without helping us toward any of our goals.

7.2 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. 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 cannot be avoided.

7.3 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.

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

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 for closed terms, the two sides of the (Let-Let)

rule are observationally equivalent and the κDOT abstract machine can optionally use the (Let-Let) rule for closed terms to reduce the number of frames used.

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.

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

7.4 ML-style References in κDOT

We now compare κDOT to other DOT calculi that use ML-style references.

We saw the Mutable WadlerFest DOT calculus of Rapoport and Lhoták [2017] in Chapter 2 and how κDOT can encode all of Mutable WadlerFest DOT. Mutable WadlerFest DOT firstly extends WadlerFest DOT with a separate store of references which point to locations and secondly extends typing judgments with a store typing context. In their type safety proof, they use a separate correspondence between the mutable store and the store typing context. This calculus does not have separate setter and getter types for references and subtyping between references was defined by the (Ref-Sub) invariance rule.

$$\frac{T <: U \qquad U <: T}{\mathsf{Ref}\ T <: \mathsf{Ref}\ U} \tag{Ref-Sub}$$

In κDOT, by keeping setter and getter types separate for fields, we allow a much richer form of subtyping between our notion of references. For example, we have the following type of subtyping which cannot be replicated in Mutable WadlerFest DOT.

$$\frac{T <: U \qquad U \not<: T}{\{a\colon U..U\} <: \{a\colon T..U\}}$$

In particular, if we let $T = \bot$, we can make fields behave as if they are read-only.

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. Furthermore, fields in Scala hold references to objects, not objects themselves, and hence κDOT corresponds to Scala more strongly.

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 hope to extend the calculus in the future with initialization tracking by adding more contexts to typing relations.

Chapter 8

Conclusion

In this thesis we presented κDOT, a calculus with path-dependent types which allows assigning to object fields and uses constructors to build objects. We explored how it relates to previous calculi with path-dependent types and presented soundness results for κDOT.

The WadlerFest DOT calculus was developed to be a simple calculus containing only the parts essential for path-dependent types and a foundation on which features of Scala could be slowly added in a bottom up fashion till all the fundamental features could be proven safe. In that spirit, we added field assignment and constructors to WadlerFest DOT.

DOT calculi also act as a safe space for experimentation where features can be experimented with and their soundness proven before they are added to Scala. The key aim of κDOT was to guide the design of initialization systems for Scala and to be a base in which interactions between object initialization and path-dependent typing could be explored.

κDOT came about as we tried to reason about initialization in Mutable WadlerFest DOT and tried to answer questions such as "What is the analogue of `null` in DOT?". As we tried to answer these questions, a calculus emerged with not only a rich type system, but also a very expressive operational semantics, where we could reason about fully evaluated data structures with path-dependent types.

κDOT opens up a rich area of exploration. We are already planning to create a DOT calculus without lazy fields based on κDOT. We plan to do this by adding an initialization system which guarantees we never read fields containing `null`s. Much more remains to be explored.

We hope that κDOT will be useful to a large group of people. κDOT is a simple calculus that can be useful to Scala programmers for reasoning about fully evaluated path-dependent data structures. We designed mutation in κDOT to closely mimic mutation in Scala, so we hope κDOT is useful for Scala compiler writers to reason about safety of mutation related compiler transformations.

References

Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. ISBN 978-1-4503-4660-3. DOI: 10.1145/3009837. URL http://dl.acm.org/citation.cfm?id=3009866.

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

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 233–249. ACM, 2014. ISBN 978-1-4503-2585-1. DOI: 10.1145/2660193.2660216. URL http://doi.acm.org/10.1145/2660193.2660216.

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. ISBN 978-3-319-30935-4. DOI: 10.1007/978-3-319-30936-1_14. URL http://dx.doi.org/10.1007/978-3-319-30936-1_14.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. DOI: 10.1145/1328438.1328443. URL http://doi.acm.org/10.1145/1328438.1328443.

L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1):4 – 56, 1994. ISSN 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1013. URL http://www.sciencedirect.com/science/article/pii/S0890540184710133.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In Rastislav Královič and Paweł Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, pages 1–23, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37793-1.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925. DOI: 10.1145/503502.503505. URL http://doi.acm.org/10.1145/503502.503505.

Ifaz Kabir and Ondřej Lhoták. kDOT: Scaling DOT with mutation and constructors. In *Proceedings of the 9th ACM SIGPLAN International Scala Symposium*, SCALA '18, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5836-1/18/09. DOI: 10.1145/3241653.3241659. URL http://doi.acm.org/10.1145/3241653.3241659.

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

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, pages 201–224, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45070-2.

B.C. Pierce and B. C. *Types and Programming Languages*. MIT Press, 2002. ISBN 9780262162098. URL https://books.google.ca/books?id=ti6zoAC9Ph8C.

Marianna Rapoport and Ondřej Lhoták. Mutable WadlerFest DOT. In *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs*, FTFJP'17, pages 7:1–7:6, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5098-3. DOI: 10.1145/3103111.3104036. URL http://doi.acm.org/10.1145/3103111.3104036.

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, October 2017. ISSN 2475-1421. DOI: 10.1145/3133870. URL http://doi.acm.org/10.1145/3133870.

Tiark Rompf and Nada Amin. From F to DOT: type soundness proofs with definitional interpreters. *CoRR*, abs/1510.05216v2, 2016a. URL http://arxiv.org/abs/1510.05216v2.

Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 624–641. ACM, 2016b. ISBN 978-1-4503-4444-9. DOI: 10.1145/2983990.2984008. URL http://doi.acm.org/10.1145/2983990.2984008.

AMR Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3):289–360, Nov 1993. ISSN 1573-0557. DOI: 10.1007/BF01019462. URL https://doi.org/10.1007/BF01019462.

Peter Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997. URL http://journals.cambridge.org/action/displayAbstract?aid=44087.

Alexander J. Summers and Peter Mueller. 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, pages 1013–1032, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. DOI: 10.1145/2048066.2048142. URL http://doi.acm.org/10.1145/2048066.2048142.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38 – 94, 1994. ISSN 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093. URL http://www.sciencedirect.com/science/article/pii/S0890540184710935.