A PATH TO DOT

MARIANNA RAPOPORT

Formalizing Scala with Dependent Object Types

PhD Thesis

Department of Computer Science
Faculty of Mathematics
University of Waterloo

September 2019 –

DECLARATION

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the 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.

Waterloo, Canada, September 2019

Marianna Rapoport

STATEMENT OF CONTRIBUTIONS

This thesis is a result of my collaboration with Ondřej Lhoták, Ifaz Kabir, Paul He, Lu Wang, and Yaoyu Zhao. The work in the thesis draws heavily on the material presented in the following publications:

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

Rapoport, Marianna and Ondřej Lhoták (2017). "Mutable WadlerFest DOT." In: *FTfJP 2017*, 7:1–7:6.

Rapoport, Marianna and Ondřej Lhoták (2019). "A Path To DOT: Formalizing Fully Path-Dependent Types." In: *PACMPL* 1.OOPSLA.

ABSTRACT

The goal of my thesis is to enable formal reasoning about the Scala programming language. To that end I present a *core calculus* that formalizes Scala's

- *essential features* in a
- *type-safe* way and is
- *easy to extend* with more features.

I build on the Dependent Object Types (DOT) calculus that formalizes *path-dependent types*. My contributions are

- a *generalization* of DOT with types that depend on *paths of arbitrary length*,
- a *simple, extensible type-safety proof* for DOT, and
- an *extension* of DOT with *mutable references*.

The simple proof makes designing smaller extensions such as mutation straightforward, and larger extensions, such as full support for paths, approachable. Adding fully path-dependent types to DOT allows us to model the key feature of Scala's type and module system.

The calculi and proofs presented in this thesis are fully mechanized in Coq.

ACKNOWLEDGMENTS

To appear.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ANF	Administrative Normal Form
DOT	Dependent Object Types calculus by Amin, Grütter, et al., (2016)

PROLOGUE

1 INTRODUCTION

Dependent Object Types (DOT) is intended to be a core calculus for modelling Scala. Its distinguishing feature is *path-dependent types* that refer to fields in objects that hold types rather than values. DOT was designed to serve as an extensible core calculus that could guide the design of future versions of Scala and help us understand the interactions of path-dependent types with other features. The goal of this thesis is to bridge the gap between DOT and Scala by making DOT more expressive and easier to work with.

The first shortcoming of DOT that this thesis addresses is its intricate soundness proof which makes seemingly simple extensions to the calculus complex and unpredictable. I propose a simple, modular, and extensible type-safety proof for DOT. I show how using the simple proof, extensions to the calculus such as mutable references become straightforward.

Second, this thesis presents pDOT, a generalization of DOT that makes it more expressive. DOT is designed to formalize Scala's module system that is based on path-dependent types, but the calculus actually lacks the ability to express a variety of valid Scala paths. As a result, we might be overlooking soundness issues in Scala that are caused by fully *path*-dependent types and can model only a restricted subset of type dependencies that are possible in Scala. In this thesis, I use the simple proof to generalize DOT to support path-dependent types on paths of arbitrary length, as well as *singleton types* to track path equality. I show that naive approaches to add paths to DOT make it inherently unsound, and present the necessary conditions for such a calculus to be sound. I discuss the key changes necessary to adapt the techniques of the DOT soundness proofs so that they can be applied to pDOT.

Path support in DOT allows us to express fully path- (as opposed to variable-) dependent types. Encoding path-dependent types, in turn, allows us to formalize the type dependencies that can occur in Scala, to have a sound theoretical foundation for its type and module system, and realize DOT's full potential for formalizing Scala-like calculi.

1.1 THE SCALA PROGRAMMING LANGUAGE

Scala is a complex programming language with a large number of features that allows one to program in a variety of styles. Higher-order functions, pattern matching, higher-kinded types, type inference, and immutable collections, among other features, make it possible to pro-

gram in a purely functional, immutable style that is reminiscent of Haskell's. Support for classes, traits, and inheritance allow one to also program in a traditionally object-oriented style, using class hierarchies and mutable state; and given Scala's JVM-interoperability, one can essentially write "Java programs" in Scala. Scala supports multiple inheritance through mixin composition; it has both nominal and structural typing. It has rich support for macros; implicit parameters, implicit conversions, and implict return types; it has abstract type members and path-dependent types.

Which of all these features are essential if we want to formally reason about Scala? What features should be part of a core calculus for the language? According to the Dependent Object Types calculus, the central feature of Scala is *path-dependent types*. To understand why, and to arrive at the language features that are necessary to formalize Scala's path-dependent types, let us look at the design goals of Scala.

One of the design goals of the Scala programming language is to unify modules and objects, so that the same language constructs can be used to define the overall structure of a program as well as its implementation detail (Odersky and Zenger, 2005b). Scala achieves modularity in several ways:

1. Being *object-oriented* allows Scala to benefit from inheritance and aggregation. This allows one to easily extend programs with new data entities without modifying or duplicating existing code, and to easily reuse of large pieces of code.

2. Being *functional* allows Scala to use functions as the "glue" that composes smaller parts of a program together (Hughes, 1989). Scala functions can operate on algebraic data types (case classes); deconstruct them through pattern matching; and use higher-order functions to abstract over parts of a computation. By enabling easy, typesafe extensions of datatypes both with new *data variants* and *processors* Scala provides a solution to the well-known *expression problem* (Odersky and Zenger, 2005a).

3. Finally, Scala *unifies modules and objects*, so that the same language constructs can be used to specify the overall structure of a program as well as its implementation details. The unification of the module and term languages is witnessed by the following comparison with the ML module system:

$$\begin{aligned} \text{object} &\leftrightarrow \text{module} \\ \text{class} &\leftrightarrow \text{functor} \\ \text{interface} &\leftrightarrow \text{signature} \end{aligned}$$

1.1.1 *Modularity Through Abstraction*

To be composable, modules need to support abstraction. Functional programming abstracts over values and types using value and type

parameters. Object-oriented programming abstracts over class components by keeping them unimplemented. Scala provides abstraction both through parametrization and aggregation.

In most object-oriented languages, the only kind of class components that are allowed to be abstracted over by remaining unimplimented are *methods*. In Scala, abstraction over object members is also allowed for *values* and *types*.

abstract type members

Types that are members of classes and objects are called *abstract type members* (we will sometimes refer to them just as type members). An abstract type member is a type that is declared inside a class[1] but whose definition can be left unspecified. We can refine the declaration of abstract type members through *type bounds*. If the type member A of an object p should be a subtype of T, we can declare A with an upper bound: { type A <: T }. To make A a supertype of U, we declare it with a lower bound: { type A >: U }. To refer to A, we use the *path-dependent type* p.A — a type that depends on the object whose *path* is $p = x.a_1.\cdots.a_n$ where x is the path's receiver, and a_i are its fields.

type bounds

path-dependent types

1.1.2 *Modularity Through Composition*

To compose modules, Scala uses *mixin composition*, a form of multiple inheritance that uses a carefully designed *linearization* principle to avoid dispatch ambiguity. To abstract away from the rules of class linearization, we can focus on a simpler, commutative way of composing types called *type intersection*, which is not supported by Scala 2 (current Scala), but is already a feature of Dotty, the compiler for Scala 3 (Documentation, 2018a). A type intersection T ∧ U is a type that has the members of both T and U.

type intersection

"Union" might seem like a better name than "intersection". However, consider a class Dog that extends Animal. The set of Dogs is a subset of all Animals. Therefore, to denote the members that are both in Dog and Animal we would use intersection.

Intersection allows us to express *type refinement*; for example, intersecting the types { type A <: T } and { type A >: U } yields { type A >: U <: T }, a type declaration that has a lower and upper bound at the same time. In fact, { type A <: T } and { type A >: U } are just syntactic sugar for { type A >: Nothing <: T } and { type A >: U <: Any }, where Any and Nothing are the top and bottom of Scala's subtyping lattice.

The combination of abstract type members, path-dependent types, and type intersection yields a surprisingly expressive type system. These concepts alone can encode a variety of features that are common in Scala and other programming languages:

- *parametric polymorphism* (generics):
  - a *parametrized class* class List[T] can be encoded using type members as class List { type T }, whereas *type application* List[Int] can be represented using refinement (type intersection): List ∧ { type T = Int };[1]

[1] *More complicated representations such as higher-kinded types require more complex encodings (Odersky, Martres, and Petrashko, 2016).*

1 The following text applies to both classes and *traits*, which are similar to Java's interfaces but allow methods to have implementations.

  - *type parameters* for polymorphic functions such as `def id[T](x: T) = x` can be encoded using type members as follows: `def id(y: { type T })(x: y.T) = x`

- *co-* and *contravariance* can be often encoded by translating declaration-site variance into use-site variance; for example, a covariant list

  `class List[+T] { def tail: List[T] }`

  can be expressed as

  `class List { type T; def tail: List ∧ { type T <: this.T } }`

  (note the path-dependent type `this.T` which references the instance of `T` that belongs to `tail`'s enclosing `List` object) (Amin, Grütter, et al., 2016);

- *existential polymorphism*: type members can encode a subset of use cases for existential types, as shown by Norris, (2015);

- *family polymorphism* (Ernst, 2001): we can distinguish path-dependent types that belong to different objects ("families") at compile time; for example, given two objects `tree` and `dag` of type `Graph` and a method `edge(g: Graph, n1: g.Node, n2: g.Node)`, a method call `edge(tree, treeNode, dagNode)` that is invoked on nodes belonging to different graphs will result in a type error;

- *subtyping hierarchies* between *recursively defined classes*: the general problem is to preserve the recursion between two classes A_2 and B_2 that inherit from two mutually recursive classes A and B respectively; as shown by Bruce, Odersky, and Wadler, (1998), such relationships require the use of additional types that are cumbersome and impractical to express using generics, leading to a quadratic increase in the number of necessary type parameters; however, it is easy to express this type of recursion/subtyping relationships using type members.

Abstract type members are thus a powerful feature that can encode a variety of useful programming concepts; but could they be too powerful? For example, how safe is it to allow type members to be declared with arbitrary lower and upper bounds? What if a type is declared with incompatible bounds, such as `{ type A >: Any <: Nothing }` or `{ type A >: Int <: String }` — should the type system try to detect such "bad bounds" and ban "nonsensical" type declarations? Furthermore, we reference an abstract type member through a path-dependent type `p.A`, which depends on the runtime value of `p`. What happens if `p` does not stop evaluating? Finally, will something break if we couple type members with other features? Which feature combinations are safe and which are not (Amin, Grütter, et al., 2016; Amin, Moors, and Odersky, 2012; Amin, Rompf, and Odersky, 2014; Moors, Piessens,

and Odersky, 2008; Odersky, Cremet, et al., 2003; Rompf and Amin, 2015, 2016a)?

To answer these questions we need to be able to formally reason about Scala's type system. That is where Dependent Object Types come in.

1.2 THE DOT CALCULUS

The Dependent Object Types (DOT) calculus is a sound formalization of "the essence of Scala": abstract type members with arbitrary type bounds, path-dependent types, and type intersection. Rompf and Amin, (2015) and Amin, Grütter, et al., (2016) presented the first DOT soundness proofs based on a big- and small-step semantics after a decade-long search for a type-safe core calculus for Scala. Table 1.1 summarizes this search. The decreasing number of shaded boxes as we go from left to right illustrates that in order to be type-safe, the formalizations had to lose more and more Scala features — sound versions of DOT handle hardly more than the essential features of path-dependent types (displayed on black/dark gray background). The table also shows that the first two type-safe calculi, νObj and μDOT, did not handle arbitrary intersections of type declarations since they either exclude intersections or lower bounds, which further highlights the difficulty of formalizing all the type-member-related features.

This thesis focuses on the small-step version of DOT by Amin, Grütter, et al., (2016).

Discovering a sound DOT has already had the following practical benefits.

1. The formalization of type members enables a better understanding of the features that type members can translate to. For example, because DOT can encode Java's wildcards (which are used to express variance in type parameters), DOT provides the first formalization of a type system that supports wildcards (Amin, Grütter, et al., 2016).

   formalization of Java's wildcards

2. DOT has helped reveal feature combinations in Scala and Java that lead to unsoundness. Amin and Tate, (2016) showed how, using wildcards and null in Java, and type members and null in Scala, we can convert between arbitrary types without casting. The authors report that their examples are inspired by the insights gained from the DOT safety proof.

   discovery of new source of Scala and Java unsoundness

3. To provide a sound theoretical foundation for Scala, insights gained from DOT are guiding the design of the Dotty compiler:
   - Dotty aims to translate type parameters into type members and type application into type intersection because these features are formalized in DOT.

   design guidelines for Dotty

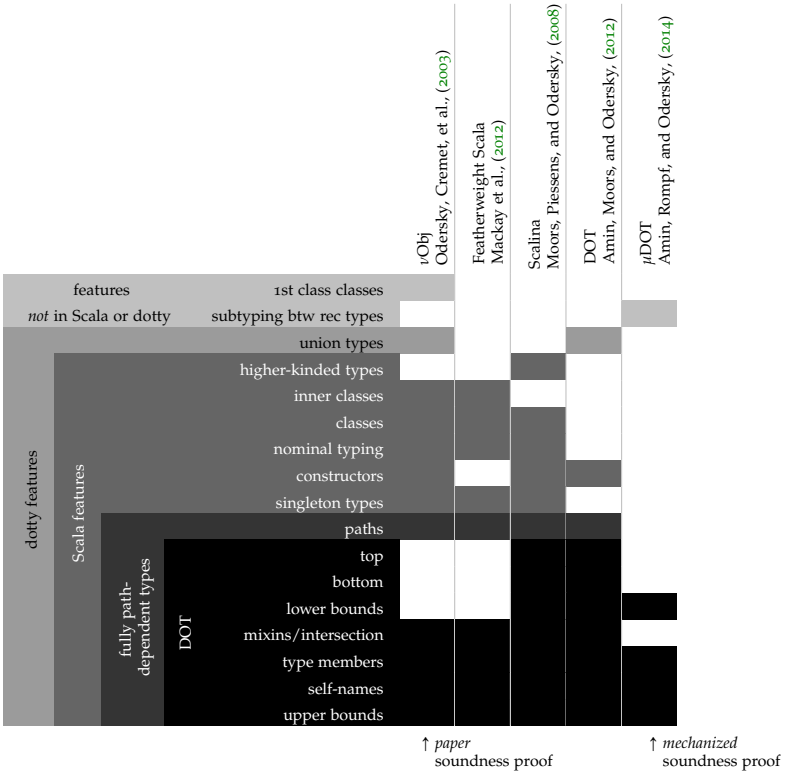

Table 1.1: Scala calculi that precede DOT

  - Intersection has been added to Dotty to faithfully mimic the DOT calculus and ensure soundness (Documentation, 2018a).
  - Examples of how DOT influences decisions for representing higher-kinded types in the Dotty compiler are documented by Odersky, Martres, and Petrashko, (2016).
  - Exposing unsoundness due to null is one of the motivations behind the ongoing Dotty work to discourage the use of null by forcing programmers to explicitly declare nullability through union types (Nieto, 2018).

ruling out nonsensical dependent types can lead to dead end

4. Studying DOT has shown that trying to detect and rule out bad bounds during type-checking is impractical (Amin, Grütter, et al., 2016; Amin, Rompf, and Odersky, 2014). Instead, it is the task of the soundness proof to determine that the types of values that will appear at runtime will eventually be *narrowed* to meaningful types with "good bounds". This realization has guided the decisions of how to approach bug reports such as Issue #1050 (Odersky, 2016).

Given the importance of having a formalization for Scala, we need to be able to extend DOT to bridge the gap between Scala and its core calculus. However, there are some problems.

1.3 LIMITATIONS OF DOT

To come up with a sound DOT, Amin, Grütter, et al., (2016) had to make two crucial decisions. These decisions were groundbreaking in that they enabled the first sound formalization of a difficult concept. But they also turned out to get in the way of making DOT an extensible core calculus.

The first decision relates to the soundness proof. To eliminate bad bounds such as { type A >: String <: Int }, Amin et al. used the insight that if a type is inhabited it cannot have bad bounds since the value that inhabits the type is a proof that the type makes sense. To incorporate reasoning about inhabited types into the proof, whenever the proof depends on the absence of bad bounds, it couples types with values that inhabit those types. Since a large part of the proof requires meaningful information from types, this introduces complicated dependencies and reasoning into the proof. As a result, the proof becomes brittle: when designing extensions it is difficult to predict what will break.

The second decision directly affects the calculus: it is to restrict path-dependent types exclusively to variables. Type selections on variables of the form $x.A$ are DOT's way to underapproximate Scala's type selections on arbitrary *stable* paths $x.a_1 \ldots a_n.A$, which must consist of immutable fields a_i. Restricting paths in path-dependent types to variables limits the expressivity of DOT and does not allow us to express a whole class of recursive dependencies between types. Conversely, generalizing from variables to paths reveals challenging soundness issues, as we will show in Part III of this thesis.

1.4 THIS THESIS

This thesis addresses the issues of DOT's complex soundness proof and incomplete modelling of path-dependent types, which limits DOT's applicability as a reusable core calculus for Scala.

1.4.1 *Part I: A Simple Soundness Proof for DOT*

The first step towards a reusable calculus is to simplify the DOT soundness proof.

Abstract type members allow DOT to express custom subtyping relationships within the calculus. For example, the recursive type

$$\begin{aligned}\mu(a:\ &\{\mathsf{SmallFish}: \bot..\top\} \wedge \\ &\{\mathsf{MediumFish}: a.\mathsf{SmallFish}..a.\mathsf{BigFish}\} \wedge \\ &\{\mathsf{BigFish}: \bot..\top\})\end{aligned}$$

contains three abstract type members. SmallFish and BigFish are fully abstract: they can be anything between $\bot$ and $\top$, whereas MediumFish

must be a supertype of SmallFish and a subtype of BigFish. Since SmallFish <: MediumFish <: BigFish, by transitivity, SmallFish must be a subtype of BigFish, thus allowing us to encode fish size in the type system. Even though the definitions of SmallFish or BigFish say nothing about their size, when we instantiate these abstract type members, the type system will ensure that the declared subtyping relationships hold.

The challenge of the DOT soundness proof is to show that the type system will accept "meaningful" type declarations while ruling out nonsensical types such as {StrangeFish : Piranha..Goldfish}. The proof of Amin, Grütter, et al., (2016) addresses this problem by requiring that all typing contexts be inhabited with values: to show that a type is valid, it must be coupled with a value of that type.

The insight of requiring inhabited typing contexts allowed Amin, Grütter, et al., (2016) to present the first type-safe calculus with path-dependent types. The drawback of this proof is, however, that a significant part of it combines reasoning about types, variables, and values. The proof becomes not just intricate but brittle, making it hard to predict the effects of even the simplest changes.

To make the DOT soundness proof easier to work with, I present a simplified soundness proof with the following contributions:

- A *modular* proof that reasons about types, values, and operational semantics separately.
- The concept of *inert* typing contexts, a syntactic characterization of contexts that rule out any nonsensical subtyping that could be introduced by abstract type members.
- A simple *proof recipe* for deducing properties of terms from their types in full DOT while reasoning only in a restricted, intuitive environment free from the paradoxes caused by abstract type members. Multiple lemmas follow the same recipe, and following the recipe can facilitate the development of new lemmas needed in future extensions for DOT.
- A *Coq formalization* of this DOT soundness proof.

The result is a modular, simple proof of DOT type safety, which enables us to formalize other constituents of Scala, such as classes and inheritance, by a translation to the core features of DOT. The simple soundness proof was developed in collaboration with Ifaz Kabir, Paul He, and Ondřej Lhoták.

1.4.2 *Part II: A DOT With Mutable References*

To illustrate that the developed DOT soundness proof is indeed simple, I present a case study of extending DOT with support for ML-style

mutable references. The extension, called Mutable DOT, adds support for a heap; the ability to create, update, and dereference mutable references; and reference types. Proving Mutable DOT sound involved adding the following to the simple soundness proof:

- a new case to the definition of inert types: any reference type is inert;
- an additional case to the definitions of invertible typing for variables and invertible typing for values;
- two new canonical forms lemmas for values and variables of a reference type; these lemmas follow the proof recipe;
- the mutation-related cases to the proof-recipe lemmas that translate tight into invertible typing, and to the final progress and preservation lemmas.

Extending DOT with mutation reveals two aspects of working with DOT and the simple proof. On the one hand, it shows that if the change to the calculus involves mostly adding new forms of values and types then it can be fairly easy and straightforward to add new features to the calculus. In the case of mutation, the change involves adding a new type of values (locations) and types (reference types). Once we know what exactly we want to extend the calculus with, it is easy to extend the proof.

However, the challenge with DOT is usually to know how to redesign the calculus: to understand and predict what exactly the extended DOT should look like. Even in the case of mutation, a seemingly small and simple change, it was not obvious how to design the mutable store. Specifically, in order to maintain soundness, the mutable store had to map locations to *variables* as opposed to values, as would be more conventional. In the second part of this thesis I present the resulting calculus, show that our design decisions for Mutable DOT are sound, correct, and expressive, and explain in detail how to extend the simple proof to support mutation.

1.4.3 *Part III: A DOT With Fully Path-Dependent Types*

The final step towards a core calculus for Scala is to formalize path-dependent types in their full expressivity. The goal is to allow DOT to have paths of arbitrary length: such paths are admissible in Scala, but not expressible in DOT.

The DOT calculus is defined using Administrative Normal Form (ANF). ANF requires that function applications $x\,y$ and path selections $z.a$ always occur on variables: as we will show in Chapter 2, each of x, y, z must be either bound in a let expression or lambda abstraction, or be the self-variable of an object. Most of the time, this does not

limit expressivity, because we can chain sequences of let bindings to obtain the complex expressions that we need. The exception to that is path-dependent types.

Consider the following Scala object:

```
val o = {
  val a = new { type A = Any }
  type B = this.a.A
}
```

The definition of B cannot be expressed in DOT. B refers to type member A, which is nested inside the definition of another field a. Field accesses always constitute paths at least of length two, since we must reference the object (in this case o) that the fields belong to. As we will show in Chapter 17, any attempts at modifying the example to shorten the path to A involve flattening the structure of the whole program which disallows nesting, a fundamental feature of object-oriented programming. To model the relationships between classes that refer to each other from nested packages we thus need to use paths of arbitrary length.

Allowing full paths in DOT poses several challenges:

avoid non-terminating paths

1. As we will show, to maintain soundness and avoid the *bad bounds* issue we must ensure that type selections occur only on normalizing (i.e. terminating, acyclic) paths.

allow recursion

2. At the same time, the calculus should still permit *non-terminating paths* in general, since that is necessary to express recursive functions.

track path equality

3. We must be able to track *equality* between paths that alias the same object, while distinguishing between paths that reference syntactically equal objects.

proof engineering of paths

4. Reasoning about paths of arbitrary length poses additional *proof-engineering* challenges. For example, the presence of paths introduces new possibilities for cyclic typing derivation which makes doing induction on typing difficult. Another problem is the use of paths where DOT uses variables: ANF allowed the original DOT proof to operate on objects and recursive types at their "outermost" level, while the pDOT proof needs to look inside an object's nested field definitions, supporting recursive reasoning about the correspondences between objects and their types at the nested levels.

In the last part of my thesis I present pDOT, a generalization of DOT with paths of arbitrary length. pDOT addresses the above challenges as follows:

separate path lookup from operational semantics

1. To avoid unsoundness, the following considerations guided the design of pDOT:

   - To ensure that paths are terminating we define them to be *irreducible* in the operational semantics.
   - At the same time, we define a *path lookup* operation that retrieves the value that a path refers to in the runtime environment, and show that paths that have the types of values always reference a value of the same type.
   - We restrict how objects are defined and typed at nested levels while still allowing type abstractions that are even more expressive than DOT's.

2. To allow non-terminating paths pDOT supports cyclic references between paths which makes the calculus less restrictive and also easier to define. *allow cyclic paths*

3. To track path aliasing in the type system we use *singleton types*. A singleton type p.type is inhabited with only one value: p. If a path p has a singleton type q.type it signals to the type system that p and q are aliases. The use of singleton types makes pDOT also the first calculus that formalizes that Scala feature. *singleton types track path equality*

4. The pDOT type-safety proof handles arbitrary paths in places where DOT uses variables using various approaches. For example, the proof recipe is stratified into additional stages that allow it to avoid additional sources for cyclic typing derivations. The proof is presented in detail in Chapter 21. *proof is less "simple"*

1.4.4 *Contributions*

In summary, this thesis makes the following contributions:

1. A soundness proof that is *modular* because it decouples the reasoning of types and values from each other and *simple* because it presents a simple recipe that one can follow whenever they need to make sense of types. *Part I: simple DOT proof*

2. A case study of adding mutable references to DOT that provides an extension of DOT with a fundamental Scala feature and shows how easy it is to extend the simple proof. *Part II: DOT with mutation*

3. The pDOT calculus, a generalization of DOT that lifts the type-selection-on-variables restriction and supports paths of arbitrary length. pDOT fully formalizes path-dependent types; it can express the whole variety of class dependencies that are possible in Scala, and allows us to unleash the full expressive power of path-dependent types. *Part III: DOT with full paths*

2

BACKGROUND: THE DOT CALCULUS

The development in this thesis follows the variant of the DOT calculus defined by Amin, Grütter, et al., (2016).

2.1 DOT ABSTRACT SYNTAX

We begin by describing the abstract syntax of the calculus. DOT defines two forms of *values* v:

- A *lambda abstraction* $\lambda(x\colon T)\,t$ is a function with parameter x of type T and a body consisting of the term t.
- An *object* of type T with definitions d is denoted as $\nu(x\colon T)d$. The body of the object consists of the definitions d, which are a collection of field and type member definitions, connected through the intersection operator. The field definition $\{a = t\}$ assigns a term t to a field labeled a, and the type definition $\{A = U\}$ defines the type label A as an alias for the type U. The object also explicitly declares a recursive *self*, or "this", variable x. As a result, both T and d can refer to x.

A DOT *term* t is a variable x, value v, field selection $x.a$, function application $x\,y$, or let binding $\textbf{let}\ x = t\ \textbf{in}\ u$. To keep the syntax simple, the DOT calculus uses administrative normal form (ANF); as a result, field selection and function application can involve only variables, not arbitrary terms.

A DOT *type* T can be one of the following:

- A *dependent function* type $\forall(x\colon S)\,T$ is the type of a function with a parameter x of type S, and with the return type T, which can refer to the parameter x.
- A *recursive type* $\mu\,(x\colon T)$ declares an object type T which can refer to its self-variable x.
- A *field declaration* $\{a\colon T\}$ states that the field labeled a has type T.
- A *type declaration* $\{A\colon S..T\}$ specifies that an abstract type member A is a subtype of T and a supertype of S.
- A *type projection* $x.A$ is the type assigned to the type member labelled A of the object x (ANF allows type projection only on variables).
- An *intersection* type $S \wedge T$ is the most general subtype of both S and T.

x, y, z	variable
a, b, c	term label
A, B, C	type label
t, u	term
v	value
d	definition
S, T, U	type

Metavariables used throughout the thesis

$$
\begin{aligned}
t, u :=\ & \\
& x \\
& x.a \\
& v \\
& x\,y \\
& \textbf{let}\ x = t\ \textbf{in}\ u \\
v :=\ & \\
& \nu(x\colon T)d \\
& \lambda(x\colon T)\,t \\
d :=\ & \\
& \{a = t\} \\
& \{A = T\} \\
& d \wedge d' \\
S, T, U :=\ & \\
& \top \\
& \bot \\
& \{a\colon T\} \\
& \{A\colon S..T\} \\
& x.A \\
& S \wedge T \\
& \mu\,(x\colon T) \\
& \forall(x\colon S)\,T
\end{aligned}
$$

Figure 2.1: Abstract syntax of DOT

– The *bottom* type $\bot$ and the *top* type $\top$ correspond to the bottom and top of the subtyping lattice, and are analogous to Scala's `Nothing` and `Any`.

2.2 DOT OPERATIONAL SEMANTICS

the store γ maps variables to values:

$$\gamma ::= \varnothing \mid \gamma,\, x \mapsto v$$

The DOT reduction relation operates on terms whose free variables are bound in a runtime value environment, or *store*, γ that maps variables to values.

Variables and values are considered normal form, i. e. they are irreducible. In particular, objects $\nu(x\colon T)d$ are values, and the fields of an object are not evaluated until those fields are selected. Because in DOT fields are thus lazily evaluated, they are similar to Scala's `lazy val`s which declare immutable, lazily evaluated values.[1] The reduction rules use ANF to ensure that before a term is used in a function application or field selection it is reduced to a value that is assigned to a variable through let bindings.

[1] *Evaluating fields strictly would require DOT to introduce a field initialization order which would complicate the calculus. DOT deliberately leaves initialization as an open question. For a DOT with constructors and strict fields, see Kabir and Lhoták, (2018).*

$$\frac{\gamma(x) = \nu(x\colon T) \ldots \{a = t\} \ldots}{\gamma \mid x.a \longmapsto \gamma \mid t} \quad \text{(Proj)}$$

$$\frac{\gamma(x) = \lambda(z\colon T)\, t}{\gamma \mid x\, y \longmapsto \gamma \mid t\,[y/z]} \quad \text{(Apply)}$$

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

$$\gamma \mid \textbf{let } x = v \textbf{ in } t \longmapsto \gamma,\, x \mapsto v \mid t \quad \text{(Let-Value)}$$

$$\frac{\gamma \mid t \longmapsto \gamma' \mid t'}{\gamma \mid \textbf{let } x = t \textbf{ in } u \longmapsto \gamma' \mid \textbf{let } x = t' \textbf{ in } u} \quad \text{(Ctx)}$$

Figure 2.2: DOT reduction rules

We denote the reflexive, transitive closure of $\longmapsto$ as $\longmapsto^*$.

2.3 DOT TYPING RULES

The DOT typing rules are presented in Figure 2.3. The rules All-I and {}-I give types to values. An object $\nu(x\colon T)d$ has the recursive type $\mu\,(x\colon T)$, where the types T must match, and T must summarize the definitions d following the definition typing rules in Figure 2.3. Note that due to Def-Typ, each of the type declarations in an object must have equal lower and upper bounds (i.e. an object $\nu(x\colon \{A\colon S..U\})\,\{A = T\}$ is only well-typed if $S = U = T$). The rules Var, All-E, Fld-E, Let give types to the other four forms of terms, and are unsurprising. The recursion introduction Rec-I, recursion

Term typing $\Gamma \vdash t : T$

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

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

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

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

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \text{ (Fld-E)} \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \text{ (Sub)}$$

Definition typing $\Gamma \vdash d : T$

$$\frac{\Gamma \vdash t : U}{\Gamma \vdash \{a = t\} : \{a : U\}} \text{ (Def-Trm)} \qquad \frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \quad \mathsf{dom}(d_1), \mathsf{dom}(d_2) \text{ disjoint}}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \text{ (AndDef-I)}$$

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

Subtyping $\Gamma \vdash T <: U$

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

$$\Gamma \vdash \bot <: T \text{ (Bot)}$$

$$\Gamma \vdash T <: T \text{ (Refl)}$$

$$\Gamma \vdash T \wedge U <: T \text{ (And}_1\text{-<:)} \qquad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \text{ (Typ-<:-Typ)}$$

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

$$\frac{\Gamma \vdash S <: T \quad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \text{ (<:-And)} \qquad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)\, T_1 <: \forall(x : S_2)\, T_2} \text{ (All-<:-All)}$$

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

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

Figure 2.3: DOT typing and subtyping rules (Amin, Grütter, et al., 2016)

elimination Rec-E, and intersection introduction And-I rules apply only to variables, but the subsumption rule Sub applies to all terms.

The subtyping rules establish the top and bottom of the subtyping lattice (Top, Bot), define reflexivity and transitivity (Refl, Trans), and basic subtyping rules for intersection types (And_1-<:, And_2-<:, <:-And). As is commonplace, dependent functions are covariant in the return type and contravariant in the parameter type All-<:-All. Field typing is covariant by the rule Fld-<:-Fld, whereas type member declarations are contravariant in the lower bound and covariant in the upper bound via Typ-<:-Typ. The most interesting rules that distinguish DOT are <:-Sel and Sel-<:, which introduce an object-dependent type $x.A$ and define subtyping between it and its bounds. As we will see, these rules are responsible for much of the complexity of the safety proof.

2.4 EXAMPLE

This section presents an example DOT program to give the reader a better intuition for the DOT calculus. This example will come up later when we talk about mutation in Part II.

Suppose we want to keep track of fish that live in aquariums. In Scala, we could write:

```
object AquariumModule {
  trait Aquarium {
    type Fish
    val fish: List[Fish]
  }

  def addFish(a: Aquarium)(f: a.Fish) =
    new Aquarium {
      type Fish = a.Fish
      val fish = a.fish :+ f
    }

  val piranhas = new Aquarium {
    type Fish = Piranha
    val fish = List.empty[Piranha]
  }

  val goldfish = new Aquarium {
    type Fish = Goldfish
    val fish = List.empty[Goldfish]
  }
}
```

This program lets us add a fish gf to the goldfish aquarium:

```
val gf: Goldfish = ...
addFish(goldfish, gf)
```

but it will result in a type error when trying to add gf to the piranha aquarium:

```
addFish(piranhas, gf) // expected: piranhas.Fish, actual: goldfish.Fish
```

The reason the goldfish is protected from the piranhas is that the type Fish is *path dependent*, i.e. specific to the run-time Aquarium object that the fish belongs to. This allows the addFish method to guarantee at compile time that an aquarium a accepts only fish of type a.Fish.

As a first attempt to define Aquarium in DOT, we can make it an *intersection* of two types:

$$\{\ \mathsf{Aquarium} = \{\mathsf{Fish}\colon \bot..\top\} \wedge \{\,\mathsf{fish}\colon\ \mathsf{List}\,\}\ \}$$

The first type, $\{\mathsf{Fish}\colon \bot..\top\}$, declares a type member Fish with lower bound $\bot$ (Nothing) and upper bound $\top$ (Any). The second type, $\{\mathsf{fish}\colon \mathsf{List}\}$, is a field declaration of type List that represents the list of fish in the aquarium. The type List is assumed to be defined in a library and contains a type member A for list elements.

A problem with the current Aquarium implementation is that it does not say that the type of elements in the fish list should be Fish. More specifically, the list elements should have the Fish type of the Aquarium *runtime object* to which the list belongs. To let the Aquarium type refer to its own runtime object a, we make Aquarium a *recursive type*:

$$\begin{array}{l}\{\ \mathsf{Aquarium} = \mu(\mathsf{a}\colon \{\mathsf{Fish}\colon\ \bot..\top\} \wedge \\ \qquad\qquad\qquad \{\,\mathsf{fish}\colon\ \mathsf{List}\ \wedge\ \{\mathsf{A}\colon \mathsf{a}.\mathsf{Fish}\,..\,\mathsf{a}.\mathsf{Fish}\}\}\}\end{array}$$

Here, to express that the type Fish should belong to the object a, we use the *type selection* a.Fish. The type a.Fish is then used as a refinement of List's element type A. In this way, the list can contain only the fish that are allowed in the aquarium a.

We can now define addFish as a function that takes an aquarium a and a fish f of type a.Fish, and creates a new aquarium a2:

$$\begin{array}{l}\{\ \mathsf{addFish} = \lambda(\mathsf{a}\colon \mathsf{aq}.\mathsf{Aquarium}).\lambda(\mathsf{f}\colon\ \mathsf{a}.\mathsf{Fish}).\\ \quad \nu(\mathsf{a2}\colon\ \mathsf{aq}.\mathsf{Aquarium} \wedge \{\ \mathsf{Fish}\colon\ \mathsf{a}.\mathsf{Fish}\,..\,\mathsf{a}.\mathsf{Fish}\ \})\ \{\\ \quad\quad \mathsf{Fish}\ =\ \mathsf{a}.\mathsf{Fish}\\ \quad\quad \mathsf{fish}\ =\ \ldots\ \}\}\end{array}$$

The construct $\nu(x\colon T)d$ creates a new object of type T with a self-variable x and definitions d. In this case, the definitions are used to initialize the Fish type and fish list of the new aquarium. The Fish type is assigned a.Fish. The new fish list needs to append the fish f to the old a.fish list.

To be able to add an element to a list, we need access to an append method, which we will get from List. Suppose that the List type is defined in a collections library. It can be defined as a recursive type $\mu(\mathsf{list}\colon \ldots)$ that declares an element type A and an append function. The append function takes a parameter a of the element type list.A and returns a List of elements that are subtypes of a.A:

$$\mathbf{let}\ \ \mathsf{collections}\ =\ \nu(\mathsf{col}\colon$$

```
  {List: μ( list : ({A: ⊥..⊤} ∧
        {append: ∀(a:  list .A)(col. List  ∧ {A: ⊥.. a.A}))}) ...
in  ...
```

With an append method on Lists, we can fully implement the addFish method. The field a2.fish should be defined as a.fish.append(f). However, since DOT uses ANF, before performing any operations on terms, we have to bind the terms to variables:

```
fish = let oldFish = a.fish          in
       let append  = oldFish.append in
       append f
```

For better readability, we introduce the following DOT abbreviations (similar ones are used by Amin, Grütter, et al., (2016)):

$$\begin{aligned}
\{A\} &\equiv \{A\colon \bot..\top\} & t\,u &\equiv \textbf{let } x = t \textbf{ in}\\
\{A\colon T\} &\equiv \{A\colon T..T\} & &\quad\ \textbf{let } y = u \textbf{ in } x\,y\\
\{A <\colon T\} &\equiv \{A\colon \bot..T\} & t.L &\equiv \textbf{let } x = t \textbf{ in } x.L\\
\{D_1; D_2\} &\equiv \{D_1\} \wedge \{D_2\} & \nu(x)d\colon T &\equiv \nu(x\colon T)d
\end{aligned}$$

where D_1, D_2 are declarations or definitions of either fields or types, and L is a label of a type or field.

With those abbreviations, the full aquarium program example looks as follows:

```
let  collections  = ν(col) {  ...  }:
  {List: μ( list : (A; append: ∀(a:  list .A)(col. List ; A <: a.A))}
in ν(aq) {
  Aquarium = μ(a: {Fish;  fish : { collections . List ; A: a.Fish}});
  addFish = λ(a: aq.Aquarium).λ(f: a.Fish).
    ν(a2) {
      Fish = a.Fish
      fish = a. fish .append f
    }: {aq.Aquarium; Fish: a.Fish}
}: {Aquarium: μ(a: {Fish;  fish : { collections . List ; A: a.Fish}});
    addFish: ∀(a: aq.Aquarium)∀(f: a.Fish)
                {aq.Aquarium; Fish: a.Fish}}
```

Part I

A SIMPLE SOUNDNESS PROOF FOR DOT

3

INTRODUCTION

Scala's type system is notoriously complex. It allows one to define abstract types as members of objects. To become less abstract and more interesting, these type members can be lower- and upper-bounded by other types. The type bounds can be arbitrary: there is no rule or restriction on what the relationship between a type member and its bounds should be. Furthermore, a type member A's type bounds can change when A's enclosing class is composed with other types, for example, through mixin composition. The type bounds can also depend on other objects. This includes A's own enclosing object, which allows A's bounds to recursively reference A itself.

If the role of a type-soundness proof is to ensure that we can safely rely on types to give us meaningful information about the values in a program then it is unsurprising that a soundness proof for Scala's type system is hard. After all, it is the task of that proof to show that a type system with the above features is not too expressive for its own good: that it does not let us express anything we want, that it will not allow arbitrary subtyping relationships between types, and that instead it will provide us with meaningful type information about our programs.

This is why 2016 was an exciting year for those who desire a formalism to understand and reason about the unique features of Scala's type system. Mechanized soundness results were published for the DOT calculus (Amin, Grütter, et al., 2016; Amin and Rompf, 2017; Rompf and Amin, 2016b). These proofs were the culmination of an elusive search that spanned more than ten years. The chief subtleties and paradoxes inherent in DOT and the Scala type system, which made the proof so challenging, were documented along the way (Amin, Moors, and Odersky, 2012; Amin, Rompf, and Odersky, 2014).

Since the DOT calculus exhibits such subtle and counterintuitive behaviour, and since the proofs are the result of such a long effort, it is to be expected that the proofs must be complicated. The calculus is dependently typed, so it is not surprising that the lemmas that make up the proofs reason about tricky relationships between types and values. In some contexts, the type system admits typings that seem just plain wrong, and give no hope for soundness, so it seems necessary to have lemmas that reason simultaneously about the intricate properties of values, types, and the environments that they inhabit.

A core calculus needs to be easy to extend. Some extensions of DOT are necessary even just to model essential Scala features. As a prominent example, types in Scala may depend on paths (e. g. $x.a_1.\cdots.a_n.A$)

we address the problem of paths in Part III

but types in the existing DOT calculi can depend only directly on variables ($x.A$). Path-dependent types are needed to model essential features such as classes and traits (as members nested in objects and packages) and the famous cake pattern (Odersky and Zenger, 2005b). Another important Scala feature to be studied in DOT are implicit parameters. Moreover, language modifications and extensions are the raison d'être of a core calculus. DOT enables designers to experiment with exciting new features that can be added to Scala, to tweak them and reason about their properties before attempting to integrate them in the compiler with the complexity of the full Scala language.

The complexity of the proof is a hindrance to such extension and experimentation. Over the past ten years, DOT has been designed and re-designed to be just right, so that the brilliant lemmas that ensure its soundness hold and can be proven. When the DOT calculus is disrupted by a modification, it is difficult to predict which parts of the proof will be affected. Experimenting with modifications to DOT is difficult because each tweak requires many lemmas to be re-proven.

The goal of the first part of this thesis is a soundness proof that is simpler, more modular, and more intuitive. Such a proof separates the concepts of types, values, and operational semantics, and reasons about one concept at a time. Then, if a language extension modifies only one concept, such as typing, the necessary changes are localized to the parts of the proof that deal with types. We also aim to isolate most of the reasoning in a simpler system that is immune to the paradox of *bad bounds*, the key challenge that plagued the long search for a soundness proof. In this system, our reasoning can rely on intuitive notions from familiar object calculi without dependent object types (Pierce, 2002). The results of this reasoning are lifted to the full DOT type system by a single, simple theorem.

the bad-bounds issue is described in Chapter 4

The main focus of our proof is on types. Dependent object types are the one feature that distinguishes DOT, so we aim to decouple that one feature, which mainly affects the static type system, from other concerns. We focus on proving the properties that one expects of types, and deliberately keep the proof independent of other aspects, such as operational semantics and runtime values, which are similar in DOT as in other object calculi. Of course, a soundness proof must eventually speak about execution and values, but once we have the necessary theory to reason about types, these other concerns can be handled separately, at the end of the proof, using standard proof techniques. Our final soundness theorem is stated for the small-step operational semantics given by Amin, Grütter, et al., (2016), but that is only the final conclusion; the theory that we develop about dependent object types would be equally applicable in a proof for a big-step operational semantics.

see Chapter 5 for the simple DOT proof

The power of DOT is also its curse. DOT empowers a program to define a domain-specific type system with a custom subtyping

lattice inside the existing Scala type system. This power has been used to encode in plain Scala expressive type systems that would otherwise require new languages to be designed. But this power also enables typing contexts that make no sense, in which types cannot be trusted and thus become meaningless. For example, a program could define typing contexts in which an object, which is not a function, nevertheless has a function type. Since such "crazy" contexts are possible, a soundness proof needs to consider them (but prove that they are harmless during execution).

Besides the general pursuit of modularity, the simplicity of our new proof depends on two main ingredients.

inertness is defined in Section 5.2

The first ingredient is *inert types* and *inert typing contexts*. The essential property of an inert type is that if all variables have inert types, then no unexpected subtyping relationships are possible, so types can be trusted, and none of the paradoxes are possible. An important part of the soundness proof is to ensure that a term cannot evaluate until the types of all its free variables have been narrowed to inert types.

We define inertness as a concise, easily testable syntactic property of a type. The definition consists of only two non-recursive inference rules, so it can be easily inverted when it occurs in a proof. By contrast, existing DOT proofs achieve similar goals using properties that characterize types by the existence of values with specific relationships to those types. The benefit of our inertness property is that it involves only a type, not any values, and it is defined directly, not via existential quantification of some corresponding value.

tight typing is discussed in Section 5.3.

The second ingredient is tight typing, a small restriction of the DOT typing rules with major consequences. We did not invent tight typing; it appears as a technical definition in the proof of Amin, Grütter, et al., (2016). Our contribution is to identify and demonstrate just how useful and important tight typing is to a simple proof. Amin, Grütter, et al., (2016) use tight typing in a collection of technical lemmas mixed with reasoning about other concerns, such as general typing (the full typing rules of DOT) and correspondences between values and types. In our proof, however, tight typing takes centre stage; it is the main actor that enables intuition and simplicity.

Tight typing neutralizes the two DOT type rules that enable a program to define custom subtyping relationships. Tight typing immunizes the calculus: even if a typing context contains a type that is not inert, tight typing prevents it from doing any harm. The paradoxes that make it challenging to work with DOT disappear under tight typing. Without those two typing rules, the calculus behaves very differently, like object calculi without dependent object types, and our reasoning can rely on familiar properties that we are used to from these calculi.

Of course, DOT with tight typing is not at all the real DOT: it lacks the power to create customized type systems, and it is uninteresting;

Theorem 6 ($\vdash$ to $\vdash_\#$) is in Section 5.3

it is just another calculus with predictable behaviour. One simple theorem bridges the gap by showing that in inert contexts, tight typing has all the power of general typing. Therefore, all the reasoning that we do in the intuitive environment of tight typing applies to the full power of DOT. Even the proof of the theorem itself reasons entirely with tight typing, without having to deal with the paradoxes of general DOT typing, and without having to reason about relationships between types and values.

Combining these two ingredients, we contribute a unified general recipe that can be used whenever a proof about DOT needs to deduce information about a term from its type. Many of our lemmas follow this recipe. The first step of the recipe, which should be the first step of any reasoning about types in DOT, is to drop down from general typing to tight typing using the above theorem. The purpose of the remaining steps is to make inductive reasoning as easy and systematic as possible.

CONTRIBUTIONS

This part of the thesis presents a simplified and extensible soundness proof for the DOT calculus. It contributes the following:

Chapter 6 describes how to extend the simple proof

- A *modular* and *extensible* proof that reasons about types, values, and operational semantics separately.
- The concept of *inert* typing contexts, a syntactic characterization of contexts that rule out any non-sensical subtyping that could be introduced by abstract type members.
- A simple *proof recipe* for deducing properties of terms from their types in full DOT while reasoning only in a restricted, intuitive environment free from the paradoxes caused by abstract type members. Multiple lemmas follow the same recipe, and following the recipe can facilitate the development of new lemmas needed in future extensions for DOT.

Coq proof: git.io/simple-dot-proof

- A *Coq formalization* of the simple DOT soundness proof.

4

BAD BOUNDS

The type selection subtyping rules <:-Sel and Sel-<: enable users to define a type system with a custom subtyping lattice. If a program defines a function $\lambda(x\colon \{A\colon S..U\})\, t$, then t is typed in a context in which S is considered a subtype of U, because $S <: x.A <: U$. The soundness proof must ensure that such a user-defined subtyping lattice does not cause any harm, i.e., cannot cause a violation of type soundness of the overall calculus.

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

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

Let S be the object type $\{a\colon \top\}$ and U be the function type $\forall(z\colon \top)\, \top$. Then the following is a valid and well-typed DOT term:

$$\lambda(x\colon \{A\colon S..U\})\ \textbf{let}\ y = v(y\colon S)\{a = y.a\}\ \textbf{in}\ y\ y$$

$S = \{a\colon \top\}$
$U = \forall(z\colon \top)\, \top$

How is this possible? The inner term $y\ y$ is a function application applying y to itself, but y is bound by the let to an object, not a function. How can y appear in a function application when it is not a function? This is possible because y has the object type S, and in the body of the lambda, we have the subtyping chain $S <: x.A <: U$. The declaration of the lambda asserts that $x.A$ is a supertype of S and a subtype of U, and therefore introduces the new custom subtyping relationship $S <: U$. Inside the body of the lambda, the object type S is a subtype of the function type U, so since the object y has type S, it also has the function type U. The function application of object y to itself is therefore well-typed in this context.

This is crazy, the reader may be thinking. Indeed, in an environment in which subtyping can be arbitrarily redefined, types cannot be trusted. In particular, we cannot conclude from the fact that y has the function type S that it is indeed a function; actually, it is an object. The seemingly obvious fix is to require S to be a subtype of U when the parameter x of the lambda is declared to have type $\{A\colon S..U\}$. But as we will discuss in Chapter 7, this seemingly obvious fix does not work, and the struggle to try to make it work has caused much of the difficulty in the ten-year struggle for a DOT soundness proof.

How can DOT be sound then, when it is so crazy? After all, the function application $y\ y$ is well-typed but its evaluation gets stuck, because y is not a function, so how can DOT be sound? The key is that the DOT semantics is call-by-value. In order to invoke the body of the lambda, one must provide an argument value to pass for the parameter x. This value must contain a type assigned to A that is both a supertype of U and a subtype of S. If no such type exists, then no such argument value can exist, so the lambda cannot be called, so its body containing the crazy application $y\ y$ cannot ever be executed.

Therefore, this term is not a counterexample to the soundness of the DOT type system.

Why should DOT have such a strange feature? The ability to define a custom subtyping lattice turns out to be very useful. For example, we can define the term:

$$\lambda(x\colon \{A\colon \bot..\top\} \wedge \{B\colon x.A..x.C\} \wedge \{C\colon \bot..\top\})\, t$$

In the body t of this lambda, we can make use of unspecified opaque types A, B, and C, making use of only the condition that $A <: B <: C$. We can use this feature to define arbitrary type systems within the language. For example, Scalas and Yoshida, (2016) have implemented session types, a feature that usually requires a custom-designed language, inside plain Scala. As another example, Osvald et al., (2016) used this ability to define a lattice of lifetimes within the Scala type system for categorizing values that cannot outlive different stack frames. Even the well-known Scala cake pattern (Odersky and Zenger, 2005b) is built using this feature.

To reconcile a custom subtyping lattice with a sound language, we only need to force the programmer to provide evidence that the custom lattice does not violate any familiar assumptions (e.g., it does not make object types subtypes of function types). This evidence takes the form of an argument value that must be passed to the lambda before the body that uses the custom type lattice can be allowed to execute. This value must be an object that provides existing types that satisfy the specified custom subtyping constraints. In our example, this is easy: it suffices to pass the same type, such as $\top$, for all three type parameters, since $\top <: \top <: \top$. However, the types are opaque: when checking the body of the lambda, the type checker cannot use the fact that $A = B = C = \top$; the body must type-check even under only the assumptions that $A <: B <: C$.

Since DOT programs can exhibit unexpected subtyping lattices in some contexts, and since this is unavoidable, an essential feature of a soundness proof is to clearly distinguish contexts in which types can be trusted, because any custom subtyping relationships have been justified by actual type arguments, from contexts in which types cannot be trusted, because they could have been derived from arbitrary unjustified custom subtyping relationships. In Section 5.2, we will formally define this property that types can be trusted, and define a simple syntactic characterization of *inert* typing contexts that guarantee this property. In earlier DOT soundness proofs, the trusted types property was not precisely defined, and typing contexts in which there are no bad bounds were defined more indirectly, not in terms of the types themselves, but in terms of the existence of values having those types.

5 THE SIMPLE DOT PROOF

This chapter describes how inert contexts and three auxiliary typing judgments (tight, invertible, and precise typing) yield a simple proof recipe that can be used in the DOT soundness proof whenever one needs to make sense of types.

5.1 OVERVIEW

We will first outline the general recipe that we use to reason throughout the proof about the meaning of a type. The details of each step will be discussed in the following subsections. We present the overview on an example proof of Lemma 13, which will be introduced in Section 5.5, but the specific example is unimportant; most of the reasoning throughout the proof follows the same steps, through the same typing relations, in the same order, using the same reasoning techniques.

Usually, we know that some term has some type (e.g. $\Gamma \vdash x\colon \{a\colon T\}$), and we seek to interpret what the type tells us about the term, and to determine how the type of the term was derived. In this example, we seek more detailed information about x, for example that the typing context Γ assigns it an object type $\Gamma(x) = \mu\,(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots)$, or the shape of the value that it will hold at run time (e.g. an object $\nu(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots)(\cdots \wedge \{a = t'\} \wedge \cdots)$).

Each such derivation follows the same sequence of steps (although sometimes only a subsequence of the steps is necessary):

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash x\colon \{a\colon T\}}{\Gamma \vdash_{\#} x\colon \{a\colon T\}}\ \textsc{Theorem 6} (\vdash \text{ to } \vdash_{\#})}{\Gamma \vdash_{!} x\colon \{a\colon T\}}\ \textsc{Theorem 10} (\vdash_{\#} \text{ to } \vdash_{!})}{\Gamma \vdash_{!} x\colon \{a\colon T'\} \qquad \Gamma \vdash T' <\colon T}\ \textsc{Induction on } \vdash_{!}}{\Gamma(x) = \mu\,(x\colon \cdots \wedge \{a\colon T'\} \wedge \cdots) \qquad \Gamma \vdash T' <\colon T}\ \textsc{Inversion of } \vdash_{!}$$

where we assume that Γ is inert. Although there are four steps, each individual step is quite simple. More importantly, each step is modular, independent of the other steps, and the proof techniques at each step are either directly reusable (theorems) or easily adaptable (induction) to proofs of properties other than this specific lemma.

The derivation starts with general typing ($\Gamma \vdash x\colon \{a\colon T\}$), the typing relation of the DOT calculus. The key property that makes reasoning possible is that the typing context Γ is inert. Inert contexts will be defined in Section 5.2. Inertness ensures that customized subtyping in

general typing $\Gamma \vdash t\colon T$

inert contexts

the program does not introduce unexpected subtyping relationships. If the context were not inert, any type could have been customized to have arbitrary subtypes and be inhabited by arbitrary terms, so it would be impossible to draw any conclusions about a term from its type.

tight typing $\Gamma \vdash_{\#} t : T$

Knowing that the typing context is inert, we apply Theorem 6 ($\vdash$ to $\vdash_{\#}$) to get a tight typing ($\Gamma \vdash_{\#} x : \{a : T\}$), which will be discussed in Section 5.3. A tight typing is immune to any unexpected subtyping relationships that the program may have defined, so our reasoning can now rely on familiar intuitions about what types ought to mean about their terms.

invertible typing $\Gamma \vdash_{¡} t : T$

However, the tight typing rules are not amenable to inductive proofs. Theorem 10 ($\vdash_{\#}$ to $\vdash_{¡}$) gives invertible typing ($\Gamma \vdash_{¡} x : \{a : T\}$), which is specifically designed to make inductive reasoning as easy as possible. Invertible typing will be discussed in Section 5.4.

precise typing $\Gamma \vdash_{!} t : T$

By induction on invertible typing, we obtain a property of all of the precise types $\Gamma \vdash_{!} x : \{a : T'\}$ that could have caused x to have the general type $\{a : T\}$. Informally, the precise typing means that the type $\Gamma(x)$ given to x by the typing context is an object type containing a field a of type T'. We will present precise typing in Section 5.3. Precise typing is also amenable to straightforward induction proofs, so we can use one to obtain $\Gamma(x)$.

5.2 INERT TYPING CONTEXTS

Recall the function $\lambda(x : \{A : S..U\})\, t$ that we discussed in Chapter 4. If the function appears in a context Γ, its body is type checked in an extended context $\Gamma, x : \{A : S..U\}$. The extended context adds a new subtyping relationship $\Gamma, x : \{A : S..U\} \vdash S <: U$ that might not have held in the original context Γ. In particular, the extended context could introduce a subtyping relationship that does not make sense, such as $\forall(x : S)\, T <: \mu\,(x : U)$, or $\top <: \bot$. To control such unpredictable contexts, we define the notion of inert typing contexts and inert types. Inert types are defined through *record types* – records whose type members have equal bounds.

record $\{a : T\}$

record $\{A : U..U\}$

DEFINITION 1 (Record Types). *A* record type *is an intersection of types each of which is either a field declaration* $\{a : T\}$ *or a tight type declaration* $\{A : U..U\}$.

$$\frac{\text{record } T \quad \text{record } U}{\text{record } T \wedge U}$$

inert $\forall(x : T)\, U$

DEFINITION 2 (Inert Types). *A* type U *is* inert *if it is either a function type or a recursive type* $\mu\,(x : T)$ *where* T *is a record type.*

$$\frac{\text{record } T}{\text{inert } \mu\,(x : T)}$$

DEFINITION 3 (Inert contexts). *A* typing context Γ *is* inert *if the type* $\Gamma(x)$ *that it assigns to each variable* x *is inert.*

An inert typing context has the following useful property.

PROPERTY 4 (Inert Context Guarantee). *Let* Γ *be any inert typing context,* t *be a closed term and* U *be a closed type. If* $\Gamma \vdash t : U$*, then* $\vdash t : U$*.*

The significance of this property is that in an inert typing context, a term t does not have any "unexpected" types that it would not have in an empty typing context. For example, we can be sure that in an inert typing context, a function value will not have an object (recursive) type, and an object will not have a function type. Though we do not directly apply the property in the proof, it is useful for intuitive reasoning about typing and subtyping in inert typing contexts.

Every value has an inert type (as long as the value is well formed, i.e., as long as it has any type at all). This is because the two base typing rules for values, ALL-I and {}-I, and the definition typing rules that they depend on, always assign an inert type to the value. The converse is not true: not every inert type is inhabited by a value. For example, we cannot construct a value of type $\lambda(x : \top)\, \bot$.

Returning to the example, suppose now that the function is invoked with some value v bound to a variable y:

$$\textbf{let } y = v \textbf{ in } (\lambda(x : \{A : S..U\})\, t)\ y.$$

Recall that the body t is typed with the assumption that $S <: U$. Type checking the overall term ensures that the argument y provides *evidence* for that assumption. Specifically, the value v has an inert type, so y has an inert type. The typing rule for function application requires subtyping between the argument and parameter types, so the type of y must have a member $\{A : T..T\}$ with $S <: T$ and $T <: U$. (The bounds T of the type member must be tight because the type is inert.) The type T that y provides is evidence that justifies the assumption $S <: U$ under which the body t of the function was type checked. During execution, when the function is called, all occurrences of x in the body t will be replaced by y before evaluation of the body begins. In general, the semantics ensures that before it begins evaluating a term (such as t), the term has a type in a context in which all non-inert types (such as the type of x) have been narrowed to inert types (such as the type of y).

5.3 TIGHT TYPING

Although inert contexts provide the assurance of Property 4 (Inert Context Guarantee), in our proofs, we often need to reason even in contexts that are not inert. Moreover, even when we know that a context is inert, it would be difficult to express the important consequences of the inert context in every proof that deals with the general DOT typing and subtyping rules.

Tight typing (Amin, Grütter, et al., 2016) is a slight restriction of general typing that can bridge the gap between the unpredictability

Tight term typing
$\Gamma \vdash_\# t : T$

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

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

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

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

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

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

$$\frac{\Gamma, x : T; d \vdash T :}{\Gamma \vdash_\# \nu(x : T) d : \mu(x : T)} \text{ ({}-I-\#)}$$

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

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

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

$$\frac{\Gamma \vdash_! x : \{A : T..T\}}{\Gamma \vdash_\# T <: x.A} \text{ (<:-Sel-\#)}$$

Tight subtyping
$\Gamma \vdash_\# T <: U$

$$\Gamma \vdash_\# T <: \top \text{ (Top-\#)}$$

$$\Gamma \vdash_\# \bot <: T \text{ (Bot-\#)}$$

$$\Gamma \vdash_\# T <: T \text{ (Refl-\#)}$$

$$\frac{\Gamma \vdash_! x : \{A : T..T\}}{\Gamma \vdash_\# x.A <: T} \text{ (Sel-<:-\#)}$$

$$\frac{\Gamma \vdash_\# T <: U}{\Gamma \vdash_\# \{a : T\} <: \{a : U\}} \text{ (Fld-<:-Fld-\#)}$$

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

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

$$\Gamma \vdash_\# T \wedge U <: T \text{ (And}_1\text{-<:-\#)}$$

$$\Gamma \vdash_\# T \wedge U <: U \text{ (And}_2\text{-<:-\#)}$$

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

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

Figure 5.1: Tight Typing Rules (Amin, Grütter, et al., 2016)

$$\frac{\Gamma(x) = T}{\Gamma \vdash_! x : T}\ (\textsc{Var}\text{-}!) \qquad \frac{\Gamma \vdash_! x : \mu\,(z : T)}{\Gamma \vdash_! x : T\,[x/z]}\ (\textsc{Rec-E}\text{-}!) \qquad \frac{\Gamma \vdash_! x : T \wedge U}{\Gamma \vdash_! x : T}\ (\textsc{And}_1\text{-E}\text{-}!) \qquad \frac{\Gamma \vdash_! x : T \wedge U}{\Gamma \vdash_! x : U}\ (\textsc{And}_2\text{-E}\text{-}!)$$

Precise variable typing $\Gamma \vdash_! x : T$

$$\frac{\Gamma,\, x : T \vdash t : U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash_! \lambda(x : T)\, t : \forall(x : T)\, U}\ (\textsc{All-I}\text{-}!) \qquad \frac{\Gamma,\, x : T;\, d \vdash T :}{\Gamma \vdash_! \nu(x : T)d : \mu\,(x : T)}\ (\{\}\text{-I}\text{-}!)$$

Precise value typing $\Gamma \vdash_! v : T$

Figure 5.2: Simplified Precise Typing Rules based on (Amin, Grütter, et al., 2016)

of the general DOT typing rules in arbitrary typing contexts and the predictable assurances of Property 4 in inert typing contexts. The tight typing rules are presented in Figure 5.1. They are almost the same as the general DOT typing rules, except that the <:-Sel-# and Sel-<:-# rules have the restricted premise $\Gamma \vdash_! x : \{A : T..T\}$, so they can be applied only when the bounds T of the type member A are tight. Precise typing, denoted $\vdash_!$, is defined in Figure 5.2. The precise type of a variable x is the type $\Gamma(x)$ given to it by the typing context Γ, possibly decomposed using the elimination rules, so that if $\Gamma(x)$ is an object type such as $\mu\,(x : \cdots \wedge \{A : T..T\} \wedge \cdots)$, then x also has just the type member $\{A : T..T\}$ as a precise type. For values, precise typing applies only the base case rules All-I and {}-I from general typing. In premises of rules that extend the typing context (All-I-#, Let-#, {}-I-#), tight typing reverts to general typing in the extended context.

We observe two useful properties of tight typing that together combine to make it especially convenient for reasoning about DOT typing. The first property is that tight typing extends the benefits of Property 4 (Inert Context Guarantee) to *all* typing contexts, not only inert ones:

PROPERTY 5 (Tight Typing Guarantee). *Let* Γ *be any typing context,* t *be a closed term and* U *be a closed type. If* $\Gamma \vdash_\# t : U$*, then* $\vdash_\# t : U$ *and* $\vdash t : U$*.*

The general typing rules that enable DOT programs to define new user-defined subtyping relationships, <:-Sel and Sel-<:, are restricted in tight typing to <:-Sel-# and Sel-<:-#, which allow only to give an alias to an existing type, but not to introduce new subtyping between existing types.

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

$$\frac{\Gamma \vdash_! x : \{A : T..T\}}{\Gamma \vdash_\# T <: x.A <: T}\ (<\text{:-Sel-\#}, \textsc{Sel}\text{-}<\text{:-\#})$$

Property 5 makes reasoning in tight typing easy: we never have to worry about unexpected custom subtyping relationships being introduced by the program, and we do not need to reason about whether we are in an inert typing context, because tight typing gives the guarantee in all contexts.

Although tight typing satisfies the desirable intuitive Property 5, it is not DOT. In particular, tight typing does not, in general, enable a program to use a custom-defined subtyping lattice that is the key feature of dependent object types. We would like the best of both worlds: to allow DOT programs to enjoy the full power of general typing, yet to reason about our proofs with the intuitive tight typing. For this, we need the second property of tight typing.

The second important property of tight typing is that in an inert typing context, tight typing is equivalent to general DOT typing:

THEOREM 6 ($\vdash$ to $\vdash_\#$). *If* Γ *is an inert context, then* $\Gamma \vdash t : T$ *implies* $\Gamma \vdash_\# t : T$, *and* $\Gamma \vdash S <: U$ *implies* $\Gamma \vdash_\# S <: U$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash t : T}{\Gamma \vdash_\# t : T} \ (\vdash \text{ TO } \vdash_\#)$$

We delay giving the proof of the theorem until after some discussion.

The theorem statements in side notes as a quick overview or refresher and might exclude some details, such as existential quantifiers.

These two properties motivate and justify our recommendation that tight typing should be at the core of all reasoning about the meaning of types in DOT. Tight typing is predictable, like the type systems of familiar calculi without dependent object types, yet in an inert typing context, it has the same power as general DOT typing. Therefore, every proof with a premise involving general typing and an inert typing context should immediately apply Theorem 6 ($\vdash$ to $\vdash_\#$) to drop down into the intuitive environment of tight typing for the rest of the reasoning.

What if we do not have an inert context as a premise, and therefore cannot apply Theorem 6? In that case, we should not reason about the meanings of types at all. As we saw in Chapter 4, in such a context, a term could be given an arbitrary type by custom subtyping rules. Therefore, we cannot deduce anything about a term from its type, and it would be futile to try.

In summary, inert contexts, tight typing, and Theorem 6 that justifies reasoning in tight typing should be the cornerstones of any reasoning about the meaning of types in the DOT calculus.

How shall we prove Theorem 6, then? It is tempting to prove the theorem by trying to compare various properties of the tight and general typing *relations*, the closures of the tight and general typing *rules*. This approach was taken in the proof of Amin, Grütter, et al., (2016) for a related theorem (with the same conclusion but different premises). The typing relations are very different from each other (general typing is much more powerful), but the rules that give rise to them are quite similar. It is much easier, therefore, to instead show that the *rules* are equivalent in an inert context. The only rules in general typing missing from tight typing are the <:-SEL and SEL-<: rules. Our goal is therefore to replace these rules with a lemma:

LEMMA 7 (SEL-<:-# Replacement). *If* Γ *is an inert context, then if* $\Gamma \vdash_\# x : \{A : S..U\}$, *then* $\Gamma \vdash_\# S <: x.A$ *and* $\Gamma \vdash_\# x.A <: U$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_\# x : \{A : S..U\}}{\Gamma \vdash_\# S <: x.A <: U} \ (\text{SEL-<:-\# REPLACEMENT})$$

One nice property of this lemma is that it is stated entirely in terms of tight typing. Thus, to prove it, we can ignore the unpredictable

world of general typing, and work exclusively in the intuitive world of tight typing.

But how can we prove it? We would like to apply the <:-Sel-# and Sel-<:-# rules. Their premises are $\Gamma \vdash_! x : \{A : T..T\}$. Therefore, we need to *invert* tight typing, to show the following:

LEMMA 8 (Sel-<:-# Premise). *If* Γ *is an inert context, then if* $\Gamma \vdash_\# x : \{A : S..U\}$, *then there exists a type* T *such that* $\Gamma \vdash_! x : \{A : T..T\}$, $\Gamma \vdash_\# S <: T$, *and* $\Gamma \vdash_\# T <: U$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_\# x : \{A : S..U\}}{\Gamma \vdash_! x : \{A : T..T\}}\ \text{(Sel-<:-\# Premise)}$$

We will discuss how to invert tight typing to prove this lemma in Section 5.4.

Using Lemma 8, proving Lemma 7 (Sel-<:-# Replacement) is easy:

Proof of Lemma 7. Apply Lemma 8, then <:-Sel-# and Sel-<:-#, to get $\Gamma \vdash_\# S <: T <: x.A <: T <: U$. The result follows by Trans-#. □

Using Lemma 7, proving Theorem 6 ($\vdash$ to $\vdash_\#$) is now also quite easy.

Proof of Theorem 6. The proof is by mutual induction on the tight typing and subtyping derivations of $\Gamma \vdash t : T$ and $\Gamma \vdash S <: U$. In general, for each rule of general typing, we invoke the corresponding rule of tight typing. The premises of the tight typing rules differ from those of the general typing rules in that they require tight typing in rules that do not extend the context. Since the unextended context is inert, the general premise implies the tight premise by the induction hypothesis. Premises that do extend the context use general typing, so nothing needs to be proven for them. The exception is the <:-Sel and Sel-<: rules. Lemma 7 is an exact replacement for these rules, so we just apply it. Despite the long explanation, the proof in Coq is only two lines long. □

5.4 INVERSION OF TIGHT TYPING

Although reasoning with tight typing is intuitive because it obeys Property 5 (Tight Typing Guarantee), we often need to invert the tight typing rules to prove properties such as Lemma 8, which we used in the proof of Lemma 7. More generally, we need to prove that if $\Gamma \vdash_\# x : T$, where T is of a certain form, then $\Gamma(x) = U$, and there is a certain relationship between T and U.

The obvious approach to proving such inversion properties is by induction on the derivation of the tight typing. This usually fails, however, because of cycles in the tight typing rules. Each language construct typically has both an introduction and an elimination rule, and the two form a cycle. For example, if $\Gamma \vdash_\# x : T$, then $\Gamma \vdash_\# x : \mu(x : T)$ by Rec-I-#, so again $\Gamma \vdash_\# x : T$ by Rec-E-#. Such cycles block inductive proofs because a proposition $\Gamma \vdash_\# x : T$ is justified by

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

$\Gamma \vdash_\# x : \mu\,(x : T)$, which in turn is justified by the original proposition $\Gamma \vdash_\# x : T$. The solution is to define a set of acyclic, invertible rules on which induction is easy, and to prove that the invertible rules induce the same typing relation as the cyclic tight typing rules.

The construction of the invertible typing rules is simplified by two restrictions:

1. We only ever need to invert typing rules in inert typing contexts.
2. We only ever need to invert typings of variables and values, not of arbitrary terms.

In the invertible rules, we can thus exclude rules that cannot apply to variables or values, and rules that cannot apply to inert types or to types derived from inert types.

It remains to decide, when facing a cycle of two rules that introduce and eliminate a given language construct, which one of the two rules to remove and which one to keep in the acyclic, invertible rule set. In general, because a construct can be introduced an unbounded number of times in tight typing, we must keep the introduction rule. For example, if x has type T, then x also has type $\mu\,(y : \mu\,(y' : \mu\,(y'' : T)))$, and the invertible rules must generate this type. On the other hand, the base case of the typing rules for variables, the rule Var-#, gives each variable x the type $\Gamma(x)$, which in an inert context is an inert type, and can therefore be a recursive type containing an intersection type. Since the tight typing rules eliminate the recursion and the intersection, the invertible rules must also eliminate them. It seems that we have reached a contradiction: the invertible rules must have both introduction and elimination rules for recursive and intersection types.

The solution is to split the invertible rules into two phases. The first phase of rules contains all the elimination rules. After all necessary eliminations have been performed, a second phase containing only introduction rules can then perform all necessary introductions. By splitting the rules into two phases, we ensure that no derivation can cycle between introductions and eliminations, so the rules are invertible. It turns out that we already have rules for the first phase: the precise typing rules introduced in Section 5.3 already contain all of the elimination rules that apply to variables and values, and eliminate from the type of a variable all constructs that can appear in an inert type.[2] To construct the invertible introduction rules, we propose the following recipe:

[2] *Note that even the general DOT typing rules remove recursive and intersection types only from the types of variables, not values.*

1. Start with the tight typing rules.
2. Inline the subsumption rule (inline the subtyping rules into the typing rules). This simplifies the construction, so we define only one relation instead of two separate typing and subtyping relations.

Invertible variable typing $\Gamma \vdash_{\mathsf{i}} x : T$

$$\frac{\Gamma \vdash_! x : T}{\Gamma \vdash_{\mathsf{i}} x : T}\ (\textsc{Var}\text{-}\mathsf{i}) \qquad \frac{\Gamma \vdash_{\mathsf{i}} x : \forall(z : S)\, T \qquad \Gamma \vdash_\# S' <: S \qquad \Gamma, y : S' \vdash T <: T'}{\Gamma \vdash_{\mathsf{i}} x : \forall(z : S')\, T'}\ (\textsc{All}\text{-}{<:}\text{-}\textsc{All}\text{-}\mathsf{i})$$

$$\frac{\Gamma \vdash_{\mathsf{i}} x : \{a : T\} \qquad \Gamma \vdash_\# T <: U}{\Gamma \vdash_{\mathsf{i}} x : \{a : U\}}\ (\textsc{Fld}\text{-}{<:}\text{-}\textsc{Fld}\text{-}\mathsf{i}) \qquad \frac{\Gamma \vdash_{\mathsf{i}} x : T \qquad \Gamma \vdash_{\mathsf{i}} x : U}{\Gamma \vdash_{\mathsf{i}} x : T \wedge U}\ (\textsc{And}\text{-}\textsc{I}\text{-}\mathsf{i})$$

$$\frac{\Gamma \vdash_{\mathsf{i}} x : \{A : T..U\} \qquad \Gamma \vdash_\# T' <: T \qquad \Gamma \vdash_\# U <: U'}{\Gamma \vdash_{\mathsf{i}} x : \{A : T'..U'\}}\ (\textsc{Typ}\text{-}{<:}\text{-}\textsc{Typ}\text{-}\mathsf{i}) \qquad \frac{\Gamma \vdash_{\mathsf{i}} x : S \qquad \Gamma \vdash_! y : \{A : S..S\}}{\Gamma \vdash_{\mathsf{i}} x : y.A}\ (\textsc{Sel}\text{-}\mathsf{i})$$

$$\frac{\Gamma \vdash_{\mathsf{i}} x : T}{\Gamma \vdash_{\mathsf{i}} x : \mu\,(x : T)}\ (\textsc{Rec}\text{-}\textsc{I}\text{-}\mathsf{i}) \qquad \frac{\Gamma \vdash_{\mathsf{i}} x : T}{\Gamma \vdash_{\mathsf{i}} x : \top}\ (\textsc{Top}\text{-}\mathsf{i})$$

Invertible value typing $\Gamma \vdash_{\mathsf{i}} v : T$

$$\frac{\Gamma \vdash_! v : T}{\Gamma \vdash_{\mathsf{i}} v : T}\ (\textsc{Val}\text{-}\mathsf{i}_v) \qquad \frac{\Gamma \vdash_{\mathsf{i}} v : T \qquad \Gamma \vdash_{\mathsf{i}} v : U}{\Gamma \vdash_{\mathsf{i}} v : T \wedge U}\ (\textsc{And}\text{-}\textsc{I}\text{-}\mathsf{i}_v)$$

$$\frac{\Gamma \vdash_{\mathsf{i}} v : S \qquad \Gamma \vdash_! y : \{A : S..S\}}{\Gamma \vdash_{\mathsf{i}} v : y.A}\ (\textsc{Sel}\text{-}\mathsf{i}_v)$$

$$\frac{\Gamma \vdash_{\mathsf{i}} v : \forall(z : S)\, T \qquad \Gamma \vdash_\# S' <: S \qquad \Gamma, y : S' \vdash T <: T'}{\Gamma \vdash_{\mathsf{i}} v : \forall(z : S')\, T'}\ (\textsc{All}\text{-}{<:}\text{-}\textsc{All}\text{-}\mathsf{i}_v) \qquad \frac{\Gamma \vdash_{\mathsf{i}} v : T}{\Gamma \vdash_{\mathsf{i}} v : \top}\ (\textsc{Top}\text{-}\mathsf{i}_v)$$

Figure 5.3: Invertible typing rules

3. Specialize the terms in all rules to variables and values, and remove all rules that cannot apply to variables or values.

4. Remove all elimination rules.

5. Remove all rules that cannot apply in an inert context. Specifically, this means the Bot-# rule, because it has $\Gamma \vdash_\# x : \bot$ as a premise, but this typing cannot be derived by any of the other remaining rules starting from an inert type given to a variable by the Var-# rule or to a value by the All-I-# and {}-I-# rules.

By applying this recipe to the tight typing rules, we arrive at the invertible typing rules shown in Figure 5.3. We must now prove that the typing relation induced by the invertible typing rules is equal to the typing relation induced by the tight typing rules (restricted to inert

contexts and to variables and values). To do this, we need to first show that invertible typing is closed under tight subtyping.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_! x : T \qquad \Gamma \vdash_\# T <: U}{\Gamma \vdash_! x : U} \text{ (Invertible <: Closure)}$$

LEMMA 9 (Invertible <: Closure). *If* Γ *is inert,* $\Gamma \vdash_! x : T$*, and* $\Gamma \vdash_\# T <: U$ *then* $\Gamma \vdash_! t : U$*.*

We can now prove that tight typing implies invertible typing:

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_\# t : T}{\Gamma \vdash_! t : T} \text{ ($\vdash_\#$ to $\vdash_!$)}$$

THEOREM 10 ($\vdash_\#$ to $\vdash_!$). *If* Γ *is an inert context and* $\Gamma \vdash_\# x : T$ *, then* $\Gamma \vdash_! x : T$*.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash_\# x : T$. Although we said that induction on tight typing usually fails because the rules have cycles, in this specific case, the induction is quite straightforward because invertible typing is part of the induction hypothesis. The inductive cases for elimination rules, which would usually lead to cycles in the induction, are all discharged using the invertible typing in the induction hypothesis. □

The soundness proof also has versions of the above two lemmas for values (instead of variables) but we omit them here because they are similar.

With this theorem, inversion proofs such as the proof of Lemma 8 (Sel-<:-# Premise) become easy inductions on the invertible typing rules:

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_\# x : \{A : S..U\}}{\Gamma \vdash_! x : \{A : T..T\}} \text{ (Sel-<:-\# Premise)}$$

Proof of Lemma 8.

$$\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash_\# x : \{A : S..U\}}{\text{inert } \Gamma \qquad \Gamma \vdash_! x : \{A : S..U\}}\ \text{Theorem 10 ($\vdash_\#$ to $\vdash_!$)} \qquad \text{inert } \Gamma \qquad \Gamma \vdash_! x : \{A : T..T\} \qquad \Gamma \vdash_\# S <: x.T <: U}{\Gamma \vdash_\# S <: U}\ \text{Induction on } \vdash_{\#\#}$$

□

We will see more lemmas that follow the same proof strategy in the next section.

5.5 CANONICAL FORMS LEMMAS

In general, soundness proofs require canonical-forms lemmas that show that if a value has a given type, then it is a particular form of value. Following our theme of a modular proof that deals with one concept at a time, we do most of our work at the level of types, following the same general recipe.

Because the DOT syntax enforces ANF, before a value can be used for anything interesting, it must first be assigned to a variable through a let expression. Suppose a variable x is bound to a value v by **let** $x = v$ **in** t and the variable x is used somewhere inside t. From the type U of the use of x, we would like to deduce the form of the value v.

We proceed in two steps. First, from a type U such that $\Gamma' \vdash x\colon U$, where Γ' is the typing context used to type the use of x occurring inside t, we follow the proof recipe to deduce the type $\Gamma'(x)$ given to x by the typing context. The typing context Γ' is constructed by the premises of the LET typing rule, which extends an existing typing context Γ to the typing context Γ' by adding a binding $(x : T)$. Here, T is some type such that $\Gamma \vdash v\colon T$. Therefore, $\Gamma'(x)$ is this T, and we have, in general, that $\Gamma \vdash v\colon \Gamma'(x)$ and thus also $\Gamma' \vdash v\colon \Gamma'(x)$.

For the second step, we know $\Gamma' \vdash v\colon T$, where the type T has been identified by the first step, and we wish to deduce the precise type of v, and thence invert the precise value typing rules to obtain the form of v.

The following lemmas instantiate these two steps, first for dependent function types, and then for field member types.

LEMMA 11 ($\forall$ to $\Gamma(x)$).

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash z\colon \forall(x\colon T)\, U}{\Gamma(z) = \forall(x\colon T')\, U' \qquad \Gamma \vdash T <\colon T' \qquad \Gamma,\, x\colon T \vdash U' <\colon U}$$

LEMMA 12 ($\forall$ to λ).

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash v\colon \forall(x\colon T)\, U}{v = \lambda(x\colon T')\, t \qquad \Gamma \vdash T <\colon T' \qquad \Gamma,\, x\colon T \vdash t\colon U}$$

LEMMA 13 (μ to $\Gamma(x)$).

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash x\colon \{a\colon T\}}{\Gamma(x) = \mu\left(x\colon \cdots \wedge \left\{a\colon T'\right\} \wedge \cdots\right) \qquad \Gamma \vdash T' <\colon T}$$

LEMMA 14 (μ to ν).

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash v\colon \mu\,(x\colon S) \qquad S = \cdots \wedge \{a\colon T\} \wedge \cdots}{v = \nu(x\colon S)(\cdots \wedge \{a = t\} \wedge \cdots) \qquad \Gamma \vdash t\colon T}$$

The proofs of all of the lemmas follow the same general proof recipe that we introduced for Lemma 13 in Section 5.1.

Proof of Lemma 12 ($\forall$ to λ).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash v : \forall(x : T)\, U}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} v : \forall(x : T)\, U}\ \textsc{Theorem 6} (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} v : \forall(x : T)\, U}\ \textsc{Theorem 10} (\vdash_{\#} \text{ to } \vdash_{!})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} v : \forall(x : T')\, U' \qquad \Gamma \vdash T <: T' \qquad \Gamma, x : T' \vdash U' <: U}\ \textsc{Induction on } \vdash_{\#\#}}{v = \lambda(x : T')\, t \qquad \Gamma, x : T' \vdash t : U' \qquad \Gamma \vdash T <: T' \qquad \Gamma, x : T' \vdash U' <: U}\ \textsc{Inversion of All-I-!}}{v = \lambda(x : T')\, t \qquad \Gamma, x : T \vdash t : U' \qquad \Gamma \vdash T <: T' \qquad \Gamma, x : T \vdash U' <: U}\ \textsc{Narrowing}}{v = \lambda(x : T')\, t \qquad \Gamma, x : T \vdash t : U \qquad \Gamma \vdash T <: T'}\ \textsc{Sub}$$

□

Proof of Lemma 11 ($\forall$ to $\Gamma(x)$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash x : \forall(y : T)\, U}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} x : \forall(y : T)\, U}\ \textsc{Theorem 6} (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} x : \forall(y : T)\, U}\ \textsc{Theorem 10} (\vdash_{\#} \text{ to } \vdash_{!})}{\Gamma \vdash_{!} x : \forall(y : T')\, U' \qquad \Gamma \vdash T <: T' \qquad \Gamma, y : T' \vdash U' <: U}\ \textsc{Induction on } \vdash_{!}}{\Gamma \vdash_{!} x : \forall(y : T')\, U' \qquad \Gamma \vdash T <: T' \qquad \Gamma, y : T \vdash U' <: U}\ \textsc{Narrowing}}{\Gamma(x) = \forall(y : T')\, U' \qquad \Gamma \vdash T <: T' \qquad \Gamma, y : T \vdash U' <: U}\ \textsc{Induction on } \vdash_{!}$$

□

Proof of Lemma 13 (μ to $\Gamma(x)$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash x : \{a : T\}}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} x : \{a : T\}}\ \textsc{Theorem 6} (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} x : \{a : T\}}\ \textsc{Theorem 10} (\vdash_{\#} \text{ to } \vdash_{!})}{\Gamma \vdash_{!} x : \{a : T'\} \qquad \Gamma \vdash T' <: T}\ \textsc{Induction on } \vdash_{\#\#}}{\Gamma(x) = \mu(x : \cdots \wedge \{a : T'\} \wedge \cdots) \qquad \Gamma \vdash T' <: T}\ \textsc{Induction on } \vdash_{!}$$

□

Proof of Lemma 14 (μ to ν).

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{inert } \Gamma \qquad \Gamma \vdash v : \mu(x : S) \qquad S = \cdots \wedge \{a : T\} \wedge \cdots}{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} v : \mu(x : S) \qquad S = \cdots \wedge \{a : T\} \wedge \cdots}\ \textsc{Theorem 6} (\vdash \text{ to } \vdash_{\#})}{\text{inert } \Gamma \qquad \Gamma \vdash_{!} v : \mu(x : S) \qquad S = \cdots \wedge \{a : T\} \wedge \cdots}\ \textsc{Theorem 10} (\vdash_{\#} \text{ to } \vdash_{!})}{\text{inert } \Gamma \qquad v = \nu(x : S)(\cdots \wedge \{a = t\} \wedge \cdots) \qquad \Gamma \vdash t : T}\ \textsc{Induction on } \vdash_{\#\#}$$

$\textsc{Inversion of } \{\}\textsc{-I-!}$

□

Since the return type of a dependent function type depends on the parameter type, this proof and the proof of Lemma 11 ($\forall$ to $\Gamma(x)$) rely on a standard narrowing property, which states that making a typing

context more precise by substituting one of the types by its subtype preserves the typing and subtyping relations.

LEMMA 15 (Narrowing). *Suppose* $\Gamma(x) = T$ *and* $\Gamma[x\colon T'] \vdash T' <\colon T$. *Then* $\Gamma \vdash t\colon U$ *implies* $\Gamma[x\colon T'] \vdash t\colon U$, *and* $\Gamma \vdash S <\colon U$ *implies* $\Gamma[x\colon T'] \vdash S <\colon U$.

$$\frac{\Gamma, x\colon T \vdash t\colon U \qquad \Gamma, x\colon T' \vdash T' <\colon T}{\Gamma, x\colon T' \vdash t\colon U}\ (\textsc{Narrowing})$$

Narrowing is proved for DOT by Amin, Grütter, et al., (2016). The proof is standard, with no issues specific to DOT, by induction on the typing and subtyping rules.

Given the above lemmas we can infer the precise type of a variable given its general type, and the shape of a value given the value's type. What we need to do now is to establish a connection between variables and values so that we can prove the canonical-forms lemmas. The first step is to define a correspondence relation between value and type environments: since evaluation happens on pairs $\gamma \mid t$ of stores and terms we need to make sure that the typing environment in which we type a term corresponds to the term's runtime configuration γ.

DEFINITION 16 (Well-formedness). *A store* $\gamma = \overline{(x_i, v_i)}$ *is* well-formed *with respect to a typing context* $\Gamma = \overline{(x_i, T_i)}$, *denoted* $\gamma\colon \Gamma$, *if* $\Gamma \vdash v_i\colon T_i$ *for all* $i = 1, \ldots, n$.

$$\frac{}{\varnothing\colon\varnothing} \qquad \frac{\gamma\colon\Gamma \quad \Gamma \vdash v\colon T}{\gamma, x \mapsto v\colon \Gamma, x\colon T}$$

The following lemma states that any variable that has a type in the environment maps to a value of the same type in the store.

LEMMA 17 (Corresponding Types). *Suppose that a store* γ *is well-formed with respect to an envrionment* Γ, *and that* Γ *assigns type* T *to a variable* x. *Then there exists a value* v *that is assigned to* x *by the store* γ, *and* v *has the same type* T *as* x.

$$\frac{\gamma\colon\Gamma \qquad \Gamma(x) = T}{\gamma(x) = v \ \wedge\ \Gamma \vdash v\colon T}\ (\textsc{Corresp. Types})$$

The final canonical-forms lemmas presented below immediately follow from Lemma 17 and Lemma 11 to Lemma 14.

LEMMA 18 (Canonical Forms for Objects). *Suppose that a store* γ *is well-formed with respect to an inert environment* Γ, *and that* $\Gamma \vdash x\colon \{a\colon T\}$. *Then the store* γ *assigns the object* $\nu(x\colon U) \cdots \wedge \{a = t\} \wedge \ldots$ *to* x, *where* t *has type* T.

$$\frac{\gamma\colon\Gamma \qquad \text{inert } \Gamma \qquad \Gamma \vdash x\colon \{a\colon T\}}{\gamma(x) = \nu(x) \ldots \{a = t\} \ldots \qquad \Gamma \vdash t\colon T}\ (\text{Can. Forms for } \nu)$$

LEMMA 19 (Canonical Forms for Functions). *Suppose that a store* γ *is well-formed with respect to an inert environment* Γ, *and that* $\Gamma \vdash x\colon \forall(x\colon T)\, U$. *Then the store* γ *assigns the object* $\lambda(x\colon T')\, t$ *to* x, *where* $\Gamma, x\colon T \vdash t\colon U$ *and* $\Gamma \vdash T <\colon T'$.

$$\frac{\gamma\colon\Gamma \qquad \text{inert } \Gamma \qquad \Gamma \vdash x\colon \forall(x\colon T)\, U}{\gamma(x) = \lambda(x\colon T')\, t \qquad \Gamma, x\colon T \vdash t\colon U \qquad \Gamma \vdash T <\colon T'}\ (\text{Can. Forms for } \lambda)$$

5.6 PROGRESS, PRESERVATION, AND SOUNDNESS

To express the final soundness theorems we need to introduce one additional lemma, the notion of a well-formed store, and a definition of normal forms.

The following lemma states that all well-typed values have an inert precise type.

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash_! v : T' \quad \text{inert } T' \quad \Gamma \vdash T' <: T}\ (\textsc{Value Typing})$$

LEMMA 20 (Value Typing). *If $\Gamma \vdash v : T$, then there exists an inert type T' such that $\Gamma \vdash_! v : T'$ and $\Gamma \vdash T' <: T$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash v : T$, and is short because only three typing rules apply to values: All-I, {}-I, and Sub. In the first two cases, the precise type of v coincides with the general type. The subsumption case is handled by using the induction hypothesis and transitivity of subtyping. Furthermore, the precise-typing judgment for values and definition-typing rules ensure that any precise type of a value is inert. □

$x \nrightarrow \qquad v \nrightarrow$

DEFINITION 21 (Normal Form). *A term t is in* normal form, *denoted $t \nrightarrow$, if t is either a variable or a value.*

With these results, we can prove the standard progress theorem that every typable term is a normal form or reduces to some other term.

$$\frac{\gamma : \Gamma \quad \text{inert } \Gamma \quad \Gamma \vdash t : T}{\gamma \mid t \longmapsto \gamma' \mid t' \ \vee\ t \nrightarrow}\ (\textsc{Progress})$$

THEOREM 22 (Progress). *Let γ be a store and Γ an inert typing environment such that $\gamma : \Gamma$. If $\Gamma \vdash t : T$ then t is in normal form or there exists a term t' and a store γ' such that $\gamma \mid t \longmapsto \gamma' \mid t'$.*

Proof. We proceed by induction on the derivation of $\Gamma \vdash t : T$, in each case finding a reduction rule that applies. The interesting cases are All-E and {}-E.

In the premises of All-E, variable x has type $\forall(z : S)\, T$. Lemma 11 ($\forall$ to $\Gamma(x)$) tells us that Γ binds x to a compatible function type. $\gamma : \Gamma$ ensures that γ binds x to some value v, and that v has a compatible function type. Finally, Lemma 12 ($\forall$ to λ) tells us the v is a lambda, so the Apply reduction rule can be applied.

The {}-E case is similar, but using Lemma 13 (μ to $\Gamma(x)$) and Lemma 14 (μ to ν) instead of Lemmas 11 and 12, respectively, and the Project reduction rule instead of Apply. □

We can now prove the standard preservation theorem.

$$\frac{\gamma : \Gamma \quad \text{inert } \Gamma \quad \Gamma \vdash t : T \quad \gamma \mid t \longmapsto \gamma' \mid t'}{\Gamma' \vdash t' : T \quad \gamma' : \Gamma' \quad \text{inert } \Gamma'}\ (\textsc{Preservation})$$

THEOREM 23 (Preservation). *Let γ be a store and Γ an inert typing environment such that $\gamma : \Gamma$. If $\Gamma \vdash t : T$ and $\gamma \mid t \longmapsto \gamma' \mid t'$ then there exists an inert context Γ' such that $\gamma' : \Gamma'$ and $\Gamma' \vdash u : T$.*

Proof. The proof proceeds by induction on the typing derivation. The interesting cases are All-E, Fld-E, and Let. In the first two cases we apply Lemmas 11 to 14 to obtain values with the necessary types like in the proof of Theorem 22 (Progress).

In the Let case, $t = \textbf{let}\ x = u\ \textbf{in}\ u'$, where $\Gamma \vdash u : U$ and $\Gamma, x : U \vdash u' : T$ for some type U. We proceed by a case analysis on the shape of u. Here, the interesting case is when u is a value v. Since $\Gamma \vdash v : U$, by Lemma 20 (Value Typing), there exists an inert type U' such that $\Gamma \vdash_! v : U'$ and $\Gamma \vdash U' <: U$. We choose as our inert context $\Gamma' =$

$\Gamma, x\colon U'$. The term $\gamma \mid \textbf{let}\ x = v\ \textbf{in}\ u'$ reduces to $\gamma, x \mapsto v \mid u'$, and we have $\gamma, x \mapsto v\colon \Gamma, x\colon U'$ as needed. Finally, we have to show that $\Gamma, x\colon U' \vdash u'\colon T$, which follows $\Gamma, x\colon U \vdash u'\colon T$ and $\Gamma \vdash U' <: U$ by Lemma 15 (Narrowing). □

Using progress and preservation it is easy to prove the final DOT type soundness theorem.

DEFINITION 24 (Divergence). *A term t* diverges*, denoted $t \Uparrow$, if there exists an infinite reduction sequence*

$$\varnothing \mid t \longmapsto \gamma_1 \mid t_1 \longmapsto \ldots \longmapsto \gamma_n \mid t_n \longmapsto \ldots$$

THEOREM 25 (DOT Type Soundness). *If $\vdash t\colon T$ then either t diverges ($t \Uparrow$) or t reduces to a normal form t', i.e. $\varnothing \mid t \longmapsto^* \gamma \mid t'$, $t' \not\to$, and $\Gamma \vdash t'\colon T$ for some Γ such that $\gamma\colon \Gamma$.*

$$\frac{\vdash t\colon T}{(\varnothing \mid t \longmapsto^* \gamma \mid t' \wedge t' \not\to) \vee t \Uparrow}\ \text{(DOT Soundness)}$$

5.7 PROOF STRUCTURE AND EXTENSIONS

This section summarizes the structure of the proof and discusses how the proof is affected by changes and extensions of the DOT calculus.

5.7.1 *Proof Structure*

The dependencies between the main lemmas in the proof are summarized in the diagram in Figure 5.4. The gray nodes and solid lines denote the lemmas in the simple proof for DOT. The white boxes and dotted lines correspond to changes needed to prove soundness of an example extension of the calculus that will be described in Chapter 6.

The final progress and preservation theorems depend on the four applications of the proof recipe to prove canonical forms for values and variables of object and function type (written in the figure as $\forall$ to λ, μ to ν, $\forall$ to $\Gamma(x)$, and μ to $\Gamma(x)$). Each application of the proof recipe uses Theorem 6 ($\vdash$ to $\vdash_\#$) and Theorem 10 ($\vdash_\#$ to $\vdash_!$) to convert general typing to tight typing and then to invertible typing. Theorem 6 depends on Lemma 7 (Sel-<:-# Replacement) and Lemma 8 (Sel-<:-# Premise). Theorem 10 depends on a subtyping closure helper lemma. After using the theorems to obtain invertible typing, we invert the invertible typing in each of the four cases (see "invertible to precise" level in the dependency graph) to obtain either the type $\Gamma(x)$ assigned to a variable x by the typing context Γ or the form of the value v with the given type. The four light large boxes in the figure indicate the canonical-forms lemmas, and the three phases of the proof recipe (conversion of general to tight typing, tight to invertible typing, and inversion of invertible typing).

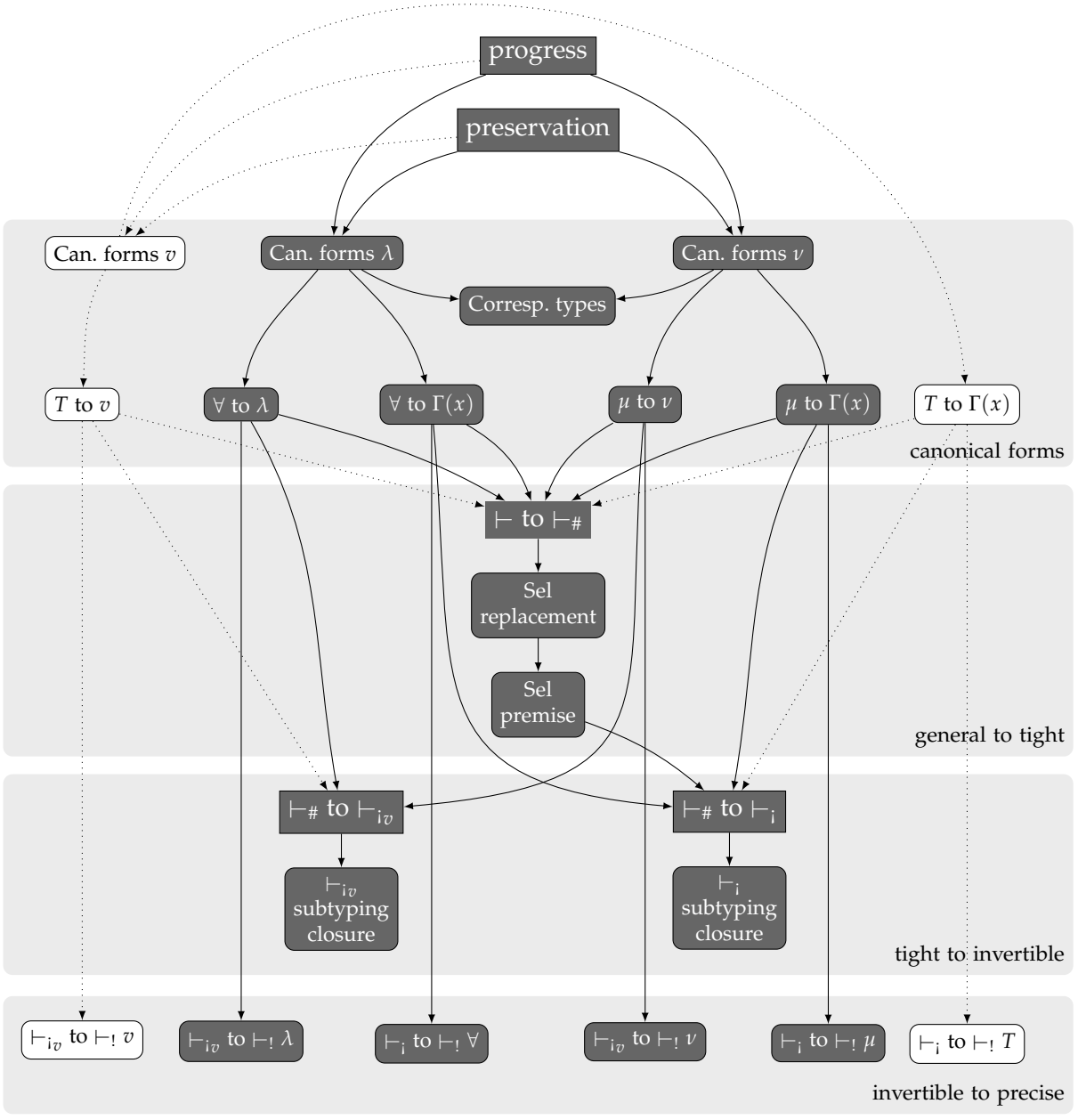

Figure 5.4: Dependencies between main lemmas in the proof. Gray nodes denote existing lemmas. White nodes denote lemmas that would need to be added if DOT were extended with a new type T and a new value v.

6 MODIFICATIONS OF THE CALCULUS

The most common expected extensions of a calculus are the addition of new forms of values and terms, of new forms of types and typing rules, and changes to the evaluation rules. Most extensions will change multiple aspects (e.g., add a new form of value and an associated type), but we discuss each change individually. In Part II, we will present a specific example of an extension that makes all of these kinds of changes.

The only part of our proof that deals with values are the two pairs of canonical forms lemmas in Section 5.5 and the final progress and preservation theorems. A new form of value will require an additional pair of canonical-forms lemmas. The lemma can follow the general recipe: it will apply Theorem 6 ($\vdash$ to $\vdash_{\#}$) and Theorem 10 ($\vdash_{\#}$ to $\vdash_{\text{¡}}$). It does not need to reason with the general DOT typing rules, but only to invert the invertible typing obtained from Theorem 10. This last inversion step should be easy, because invertible typing is designed to be easily invertible. The addition of a new form of value is illustrated in the dependence graph by the two white nodes on the left side of the graph.

The only part of the proof that deals with terms in general are the final progress and preservation theorems. The only non-trivial change required when adding a new term is that if new reduction rules are added for the new term to the operational semantics, cases for the new reduction rules need to be added to the progress and preservation theorems. This is illustrated in the dependence graph by the dotted outlines of the nodes representing those two theorems.

Adding a new form of type is a more significant change. Given general typing rules for the new type, we must incorporate the changes into the tight, invertible, and precise typing rules. Tight typing differs from general typing only in its handling of abstract type members and type projections, so changes unrelated to those features can be incorporated directly into tight typing. A change involving abstract type members or type projections requires corresponding modifications to tight typing. Property 5 (Tight Typing Guarantee) gives a modular specification to guide the design of such modifications. Specifically, we know that as long as the modified tight typing rules satisfy the property and we can prove Theorem 6 ($\vdash$ to $\vdash_{\#}$), then the proof recipe and the rest of the whole soundness proof will continue to hold without requiring non-trivial changes. To incorporate the modifications into invertible and precise typing, it suffices to follow the general recipe outlined in Section 5.4. Specifically, we must classify the new tight

typing rules as either introducing or eliminating a syntactic construct, and then add them to either invertible or precise typing, respectively. Adding new typing rules requires adding the corresponding cases to the proofs of Theorem 6 ($\vdash$ to $\vdash_{\#}$) and Theorem 10 ($\vdash_{\#}$ to $\vdash_{\mathrm{i}}$). In the proof of Theorem 6 ($\vdash$ to $\vdash_{\#}$), all cases except <:-Sel and Sel-<: are so simple that Coq discharges them automatically, so we do not add a dotted outline to the node to indicate new cases in the proof.

A change to the evaluation rules of the calculus does not affect any of the reasoning in Chapter 5 since this chapter is independent of any particular evaluation semantics. Only the final progress and preservation theorems are affected.

The next part of this thesis presents a specific extension of DOT by showing how to add mutable references to the calculus.

7

THE STRUGGLE FOR "GOOD" BOUNDS

A recurring theme in previous work on DOT has been the struggle to enforce "good" bounds. A type member declaration $\{A\colon S..U\}$ is considered to have "good" bounds if $S <: U$. If all type members could be forced to maintain "good" bounds, it would prevent an object of type $\mu\,(x\colon \{A\colon S..U\})$ from introducing a new, possibly non-sensical subtyping relationship $S <: U$ from $S <: x.A <: U$ and transitivity. Many of the challenges along the way to defining a sound DOT calculus arose from the negative interaction between "good" bounds and other properties, such as narrowing and transitivity. For example, although both $\{A\colon \bot..\bot\}$ and $\{A\colon \top..\top\}$ have "good" bounds, the narrowed type $\{A\colon \bot..\bot\} \wedge \{A\colon \top..\top\}$ causes trouble: in the function

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

the body t is type-checked in a typing context in which $\top <: x.A <: \bot$.

Not only do "good" bounds interact poorly with other desirable properties, but even defining precisely what "good" bounds are is surprisingly elusive. Informally, bounds are "good" if $S <: U$. But in what typing context should this subtyping relationship hold? In deciding whether the type $\mu\,(x\colon \{A\colon S..U\})$ should be allowable, it seems appropriate to respect the recursion implied by μ and use a context that includes x; that is, to require that $\Gamma,\, x\colon \{A\colon S..U\} \vdash S <: U$. But this statement is always true regardless of the types S and U because it is self-justifying:

$$\Gamma,\, x\colon \{A\colon S..U\} \vdash S <: x.A <: U.$$

If we decide instead to exclude the self-reference x from the context used to decide whether $S <: U$, we exclude many desirable types from the definition of "good" bounds. For example, we consider "bad" the type

$$\mu\,(x\colon \{A\colon \bot..\top\} \wedge \{B\colon x.A..x.C\} \wedge \{C\colon \bot..\top\})$$

that innocently defines three type members with $A <: B <: C$, because $x.A$ cannot be a subtype of $x.C$ without x in the context. We also consider "bad" the following type that defines two type members $A <: B$ constrained to be function types:

$$\mu\,(x\colon \{A\colon \bot..\forall(y\colon \bot)\,\top\} \wedge \{B\colon x.A..\forall(y\colon \bot)\,\top\})\,.$$

Again, $x.A$ cannot be a subtype of $\forall(y\colon \bot)\,\top$ without x in the context. Finally, such a definition of "good" bounds restricts the applicability

of type aliases: the following type defines A and B as aliases for $\top$ and $\bot$, respectively, but cannot use these aliases in the bounds of C because $x.B \not<: x.A$ in a context without x:

$$\mu\,(x\colon \{A\colon \top..\top\} \wedge \{B\colon \bot..\bot\} \wedge \{C\colon x.B..x.A\})$$

Although it would be possible to come up with some definition of "good" bounds that handles these specific examples, the definition of what was intended to be an obvious and intuitive concept would become very complicated, and other more sophisticated counterexamples would probably continue to exist. Thus, it appears that trying to enforce "good" bounds, and even trying to define what "good" bounds are, is a dead end.

By contrast, inert types obey a purely syntactic property that is easily defined and checked, without requiring a subtyping judgment in some typing context that would have to be specified. The property provided by an inert typing context can be stated precisely and formally (Property 4 (Inert Context Guarantee)).

8

RELATED WORK

The work presented in this part of the thesis is based on a previously published publication of the simple DOT soundness proof (Rapoport, Kabir, et al., 2017).

8.1 DOT SOUNDNESS PROOFS

The work most closely related to ours is Amin, Grütter, et al., (2016), which defines and proves sound the variant of the DOT calculus for which we have developed our alternative soundness proof. That work also defines tight typing, though it does not use it as pervasively as our proof does.

A central notion of that proof is store correspondence, a relationship between typing contexts and stores of runtime values. A typing context Γ corresponds to a store s if for every variable x, $\Gamma \vdash_! s(x) : \Gamma(x)$. Typing and subtyping in a context Γ that corresponds to some store s have similar predictable behaviour as they do in an inert context. Part of the proof consists of lemmas that relate internal details of values in stores with internal details of types in corresponding contexts. By contrast, the property of inert contexts is independent of values, so our proof does not depend on such lemmas.

Another central notion is "possible types": if a typing context Γ corresponds to some store s, and s assigns to variable x the value v, then the possible types of the triple (Γ, x, v) include all types T such that $\Gamma \vdash x : T$. Possible types serve a similar purpose as our invertible typing rules, to facilitate induction proofs. Unlike invertible typing, possible types depend on the runtime value v of x. The possible types lemma relates general typing in a context with a corresponding store to possible types. It serves a similar purpose as our Theorem 10 ($\vdash_\#$ to $\vdash_¡$) (which relates tight to invertible typing), but its proof is more complicated, because it depends on sublemmas that relate types to values in the context corresponding to the store, and on general typing.

Amin, Grütter, et al., (2016) also prove a similar result as Theorem 6 ($\vdash$ to $\vdash_\#$): the general to tight lemma states that in a context Γ for which there exists some corresponding runtime store s, general typing implies tight typing. We prove Theorem 10 ($\vdash_\#$ to $\vdash_¡$) first, which makes proving Theorem 6 ($\vdash$ to $\vdash_\#$) easy. The proof of Amin, Grütter, et al., (2016) does the analogous steps in the opposite order: it proves the general to tight lemma first, and the possible types lemma afterwards, using the general to tight lemma in its proof. The proof of the general to tight lemma is thus complicated because it cannot make use

of possible types. Another complication is that the proof of the general to tight lemma, like the proof of the possible types lemma, depends on sublemmas that relate types to values in the context corresponding to the store.

Rompf and Amin, (2016b) define a variant of the DOT calculus with additional features, most significantly subtyping between recursive types. This adds significant complexity to the proof: Lemmas 6 to 11 are needed only because of this feature. However, subtyping between Scala's types can be already modelled by subtyping between type members in DOT. Scala has nominal subtyping between classes and traits that are explicitly declared to be subtypes using an `extends` clause. A class or trait declaration in Scala corresponds in DOT to a type member definition in some package x that gives a label A to a recursive type. The recursive type is used to define the members of the class, and the recursion is necessary so that members of the class can refer to the object of the class `this`. A subclass B of A can be declared as the type member definition $B = x.A \wedge \mu(z\colon x.A \wedge T)$, where the type T describes the additional members that B adds to A. Then $x.B$ is a subtype of $x.A$, and given a variable of type $x.B$, it is possible to access both members that were declared in A and members that were added in B. This DOT encoding models the Scala subtyping between classes A and B without requiring subtyping between recursive types, and it can be expressed in the DOT of Amin, Grütter, et al., (2016).

Unlike Amin, Grütter, et al., (2016) and our proof, the proof of Rompf and Amin, (2016b) does not use tight typing, the typing relation that neutralizes the two type rules that enable a DOT program to introduce non-sensical subtyping relationships in a custom type system. Instead, the proof uses "precise subtyping", a restriction of general subtyping to relationships whose derivation does not end in the transitivity rule.

8.2 HISTORY OF SCALA CALCULI

Odersky, Cremet, et al., (2003) introduce νObj, a calculus to formalize Scala's path-dependent types. νObj includes abstract type members, classes, compound (non-commutative) mixin composition, and singleton types, among other features. However, the calculus lacks several essential Scala features, such as the ability to define custom lower bounds for type members, and has no top and bottom types. Additionally, νObj, unlike Scala, has classes as first-class values. νObj comes with a type soundness proof. The paper also shows that type checking for νObj is undecidable. Cremet et al., (2006) propose Featherweight Scala, which is similar to νObj, but without classes as first-class values. The paper shows that type inference in Featherweight Scala is decidable, but does not prove type safety. Scalina, introduced by

Moors, Piessens, and Odersky, (2008), presents a formalization for higher-kinded types in Scala, but also without a soundness proof.

Amin, Moors, and Odersky, (2012) present the first DOT. DOT has fewer syntax-level features than νObj: there are no classes, mixins, or inheritance. However, some of the previously missing crucial Scala features are now present. The calculus allows refinement of abstract type members through commutative intersections, combining nominal with structural typing. Type members can have custom lower and upper bounds, and the type system contains a bottom and top type. The paper comes without a type safety proof, but it explains the challenges and provides counterexamples to preservation. The paper shows how the environment narrowing property makes proving soundness complicated: replacing a type in the context with a more precise version can impose a new subtyping relationship, which could disagree with the existing ones.

Amin, Rompf, and Odersky, (2014) have the first mechanized soundness proof for μDOT, a simplified calculus that excludes refinements, intersections, and the bottom and top types, and uses big-step semantics. The paper proposes the idea to circumvent bad bounds by reasoning about types that correspond to runtime values.

Amin, Grütter, et al., (2016) and Rompf and Amin, (2016b) build on this store correspondence idea, to establish the first mechanized soundness proofs for DOT calculi with support for type intersection and refinement, and top and bottom types. The two calculi and soundness proofs were discussed in the previous section.

8.3 OTHER RELATED CALCULI

Path-dependent types were first introduced in the context of *family polymorphism* by Ernst, (2001). In family polymorphism, groups of types can form *families* that correspond to a specific object. Two types from the same class are considered incompatible if the types are associated with different runtime objects.

Family polymorphism is the foundation of *virtual classes*, which were introduced in the Beta programming language (Madsen and Møller-Pedersen, 1989) and further developed in gbeta (Ernst, 1999). Virtual classes are nested classes that can be extended or redefined (overridden), and are dynamically resolved through late binding. Family polymorphism allows for a fine-grained distinction between classes that have the same static path, yet belong to different runtime objects and can thus have different implementations.

Virtual classes were first formalized and proved type safe in the *vc* calculus (Ernst, Ostermann, and Cook, 2006). vc is a class-based, nominally-typed calculus with a big-step semantics. To create path-based types, the keyword out is used to refer to an enclosing object. With its support for classes, inheritance, and mutation of variables,

vc is more complex than DOT, whose purpose is to serve as a simple core calculus for Scala. Additionally, Scala has no support for virtual classes: the language does not allow class overriding, and its classes are resolved statically at compile time.

Tribe by Clarke et al., (2007) is a simpler, more general calculus inspired by vc. One of the main distinctions to vc is that variables, and not just enclosing objects (out), can be used as paths for path-dependent types. This makes the calculus more general, as it can express subtyping relationships between classes with arbitrary absolute paths. Tribe comes with a type-safety proof, which is based on a small-step semantics. Expanding paths to allow variables brings Tribe closer to DOT. However, the complexity of the type system, resulting from modelling classes and inheritance, and the modelling of virtual classes, which are not present in Scala, leaves DOT more suitable as a core calculus for Scala.

Amin and Rompf, (2017) offer a survey of mechanized soundness proofs for big-step, DOT-like calculi using definitional interpreters. The paper explores a family of calculi ranging from System F to System $D_{<:>}$ and general proof techniques that can be applied to this entire family. The paper discusses similarities and differences between System $D_{<:>}$ and DOT.

8.4 TYPE CHECKING DECIDABILITY

Odersky, Cremet, et al., (2003) proved that there exists no algorithm that can decide if a typing judgment in the νObj calculus is well-typed. To carry out the proof, νObj includes first-class classes, which are not present in Scala. The proof works via a reduction from System $F_{<:}$, which was shown to be undecidable (Pierce, 1992a). To come up with a decidable type system for Scala, Cremet et al., (2006) developed the *Featherweight Scala* calculus and proved its type system decidable; however, the formalization was not proved sound and excluded unique lower and upper bounds for the subtyping lattice and lower bounds for type members. By formalizing a large subset of Scala, νObj and Featherweight Scala significantly differ from DOT-like calculi whose purpose is to serve as small extensible core calculi.

The DOT calculus is widely conjectured to have undecidable type-checking because it includes the features of $F_{<:}$, for which typechecking is undecidable (Pierce, 1992b). Rompf and Amin, (2016b) give a mapping from $F_{<:}$ to $D_{<:}$, a simpler calculus than DOT, and prove that if the $F_{<:}$ term is typeable then so is the $D_{<:}$ term. However, the mapping does not preserve typeability in the only-if direction which is required to prove undecidability of typing: Hu and Lhoták, (2019) present an example of a typeble $D_{<:}$ term whose translation to $F_{<:}$ does not have a type. Hu and Lhoták then present an alternative

mapping between $F_{<:}$ and $D_{<:}$ and prove undecidability of $D_{<:}$ in Agda.

To investigate what it would take to make typechecking a DOT-like calculus algorithmic, Nieto, (2017) presents a decidable type-checking algorithm for a subset of $D_{<:}$ with restricted subtyping rules. The paper discusses how bad bounds complicate algorithmic typing of DOT-like calculi, and how the Scala compiler avoids this issue by dropping subtyping transitivity altogether. Hu and Lhoták, (2019) define *kernel* $D_{<:}$, a calculus equivalent to Nieto's subset of $D_{<:}$ and formalize its decidability proof in Coq. The paper also presents a mechanized proof for a version of *kernel* $D_{<:}$ that allows comparing the parameter types of function types for equality, adding significant expressivity to the calculus.

8.5 SYNTACTIC VS. SEMANTIC PROOFS

The DOT papers that establish type safety based on a small-step operational semantics, including this work, use the *syntactic* approach to proving type safety of Wright and Felleisen, (1994).[3] In a syntactic proof, we develop a set of inductive type rules and prove *progress* and *preservation* theorems for our operational semantics. Progress says that any well-typed, closed term is either a value or can take a step. Preservation says that reduction preserves the types of terms. Together, progress and preservation imply type safety: well-typed programs do not get stuck.

[3] *Big-step DOT proofs also use a syntactic approach but based on definitional interpreters (Reynolds, 1998).*

The syntax-based approach treats types as syntactic constructs without directly expressing their meaning. By contrast, *semantic* approaches to type soundness are based on establishing a set-theoretic semantics for types, where types are explicitly defined in terms of the values they represent (Appel and Felty, 2000). Instead of using type rules as axioms, a semantic proof allows us to prove type rules as derived lemmas. This leads to a system in which we only need to trust the operational semantics and type definitions, instead of relying on the meaningfulness of the type system, whose definition is often larger and more difficult to reason about. In addition, semantic proofs tend to be more reusable and are easier to scale (Bell et al., 2008). Unfortunately, encoding types as sets of values proves often difficult and involves more sophisticated mathematics.

Creating a semantic type-soundness proof for DOT remains an open problem. Wang and Rompf, (2017) propose a proof of strong normalization using for $D_{<:}$, a restricted version of DOT. The paper presents the first semantic encoding of a subset of DOT's types, and the normalization proof is a semantic one that is based on logical relations. At the moment, Paolo Giarrusso is actively working on a semantic type soundness proof for the full DOT calculus in Iris that is based on logical relations (Giarrusso et al., 2019).

9

SUMMARY

DOT (Amin, Grütter, et al., 2016) is the result of a long effort to develop a core calculus for Scala. Now that there is a sound version of the calculus, we would like to extend it with other Scala features, such as classes, mixin composition, side effects, implicit parameters, etc. DOT can be also used as a platform for developing new language features and for fixing Scala's soundness issues (Amin and Tate, 2016). But these applications are hindered by the complexity of the existing soundness proofs, which interleave reasoning about variables, types, and runtime values, and their complex interactions.

We have presented a simplified soundness proof for the DOT calculus, formalized in Coq. The proof separates the reasoning about types, typing contexts, and values from each other. The proof depends on the insight of inert typing contexts, a syntactic characterization of contexts that rule out any non-sensical subtyping that could be introduced by abstract type members. The central lemmas of the proof follow a general proof recipe for deducing properties of terms from their types in full DOT while reasoning only in a restricted, intuitive environment free from the paradoxes caused by abstract type members. The same recipe can be followed to prove similar lemmas when the calculus is modified or extended. The result is a simple, modular proof that is well suited for developing extensions.

Part II

CASE STUDY: MUTABLE DOT

10 INTRODUCTION

DOT models the key components of the Scala type system such as type members, path-dependent types, and subtyping. However, the calculus is still lacking some fundamental Scala features, one of which is mutation.

Without mutation, it is difficult to model mutable variables and fields, or to reason about side effects in general. Interestingly, mutation is even necessary to model a sound class initialization order for immutable fields, which are mutated once when they are initialized (Kabir and Lhoták, 2018). Formalizing Scala initialization order would require reasoning about overwriting of class members that were initialized with null, which is not directly possible in DOT.

This chapter presents the Mutable DOT calculus, an extension to DOT with typed mutable references. To that end, we augment the calculus with a mutable heap and the possibility to create, update, and dereference mutable memory cells, or locations. To model mutable variables (vars), one can create a heap location and store immutable variables (vals) in it (immutable variables are already modelled in DOT). For example, a Scala object

```
object O {
    val x = 1
    var y = 2
}
```

can be represented in mutable-DOT pseudocode as follows:

```
new {this: {x: Int} ∧ {y: Ref Int}}
           {x = 1} ∧ {y = ref 2 Int}
```

An unusual characteristic of our heap implementation is that it maps locations to variables instead of values. This design choice is induced by DOT's type system, which disallows subtyping between recursive types. We show how, as a result, storing values on the heap would significantly limit the expressiveness of our calculus, and explain the correctness of storing variables on the heap.

CONTRIBUTIONS

This part of the thesis presents the following contributions:

- an operational semantics and type system for *Mutable DOT*, an extension of the DOT calculus with mutable references; *see Chapter 11*
- a *mechanized type safety proof* in Coq, in the form of an extension of the original DOT proof; *Coq proof: https://git.io/fjlq6, see Chapter 12*

see Chapter 13 – a *discussion* of the Mutable DOT design choices and examples.

11 THE MUTABLE DOT CALCULUS

In this chapter we present the Mutable DOT calculus as an extension of DOT.

11.1 MUTABLE DOT ABSTRACT SYNTAX

To support mutation, we augment the DOT syntax with *references* that point to *mutable memory cells*, or *locations*, as shown in Figure 11.1.

Locations are a new kind of value that is added to the syntax, and are denoted as l. The syntax comes with three new terms to support the following reference operations:

- ref $x\ T$ *creates* a new reference of type T in the heap and initializes it with the variable x. Section 13.3 explains why reference expressions need to contain a declared type T.
- $!x$ *reads* the contents of a reference x.
- $x := y$ *updates* the contents of a reference x with the variable y.

The operations that create, read, and update references operate on variables, not arbitrary terms, in order to preserve ANF.

Newly-created references become *locations*, or memory addresses, denoted as l. Locations are stored in the *heap*, denoted as σ.

The heap is a *map* from locations to *variables*. This differs from the common definition of a heap which maps locations to values. We discuss the motivation for this design choice in Section 13.1. In order to preserve the commonly expected intuitive behaviour of a heap, we must be sure that while a variable is in the heap, it does not go out of scope or change its value. We show this in Section 13.2.

Updating a heap σ that contains a mapping $l \mapsto x$ with a new mapping $l \mapsto y$ overwrites x with y:

$$(\sigma[l \to x])(l') = \begin{cases} x & \text{if } l = l' \\ \sigma(l') & \text{otherwise.} \end{cases}$$

Locations are typed with the reference type Ref T. The underlying type T indicates that the location stores variables of type T.

$t, u :=$
- x
- v
- $x.a$
- $x\ y$
- **let** $x = t$ **in** u
- ref $x\ T$
- $!x$
- $x := y$

$v :=$
- $v(x\colon T)d$
- $\lambda(x\colon T)\, t$
- l

$d :=$
- $\{a = t\}$
- $\{A = T\}$
- $d \wedge d'$

$S, T, U :=$
- $\top$
- $\bot$
- $\{a\colon T\}$
- $\{A\colon S..T\}$
- $x.A$
- $S \wedge T$
- $\mu\,(x\colon T)$
- $\forall(x\colon S)\, T$
- Ref T

Figure 11.1: Abstract syntax of Mutable DOT (cf. DOT syntax in Figure 2.1)

$$\frac{\gamma(x) = v(x\colon T)\ldots\{a = t\}\ldots}{\sigma \mid \gamma \mid x.a \longmapsto \sigma \mid \gamma \mid t} \quad \text{(Project)}$$

$$\frac{\gamma(x) = \lambda(z\colon T)\, t}{\sigma \mid \gamma \mid x\, y \longmapsto \sigma \mid \gamma \mid t\,[y/z]} \quad \text{(Apply)}$$

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

$$\sigma \mid \gamma \mid \textbf{let } x = v \textbf{ in } t \longmapsto \sigma \mid \gamma, x \mapsto v \mid t \quad \text{(Let-Value)}$$

$$\frac{\sigma \mid \gamma \mid t \longmapsto \sigma' \mid \gamma' \mid t'}{\sigma \mid \gamma \mid \textbf{let } x = t \textbf{ in } u \longmapsto \sigma' \mid \gamma' \mid \textbf{let } x = t' \textbf{ in } u} \quad \text{(Ctx)}$$

$$\frac{l \notin \mathsf{dom}(\sigma)}{\sigma \mid \gamma \mid \mathsf{ref}\, x\, T \longmapsto \sigma[l \mapsto x] \mid \gamma \mid l} \quad \text{(Ref)}$$

$$\frac{\gamma(x) = l}{\sigma \mid \gamma \mid x := y \longmapsto \sigma[l \mapsto x] \mid \gamma \mid y} \quad \text{(Heap)}$$

$$\frac{\gamma(x) = l \qquad \sigma(l) = y}{\sigma \mid \gamma \mid !x \longmapsto \sigma \mid \gamma \mid y} \quad \text{(Deref)}$$

Figure 11.2: Mutable DOT operational semantics

To write concise Mutable DOT programs, we extend the abbreviations from Section 2.4 with the following rules:

$$\begin{aligned}
\mathsf{ref}\, t\, T &\equiv \textbf{let } x = t \textbf{ in } \mathsf{ref}\, x\, T \\
t := u &\equiv \textbf{let } x = t \textbf{ in let } y = u \textbf{ in } x := y \\
!t &\equiv \textbf{let } x = t \textbf{ in } !x \\
t;\, u &\equiv \textbf{let } x = t \textbf{ in } u
\end{aligned}$$

11.2 MUTABLE DOT OPERATIONAL SEMANTICS

the heap σ maps locations to variables:

$$\begin{aligned}
\sigma &:= \varnothing \\
&\mid \sigma[l \mapsto x]
\end{aligned}$$

Since the meaning of a Mutable DOT term depends on the heap contents, we represent a program state as a triple $\sigma \mid \gamma \mid t$, denoting a term t that can point to memory contents in the heap σ and whose variables are stored in the store γ.

The new reduction semantics is shown in Figure 11.2:

- A newly created reference $\mathsf{ref}\, x\, T$ reduces to a fresh location with an updated heap that maps l to x (Ref).
- Dereferencing a variable using $!x$ is possible if x is bound to a location l by a let expression. If so, $!x$ reduces to $\sigma(l)$, the variable stored at location l (Deref).

– Similarly, if x is bound to l by a let, then the assignment operation $x := y$ updates the heap at location l with the variable y (HEAP).

Programs written in the Mutable DOT calculus generally do not contain explicit location values in the original program text. Locations are included as values in the Mutable DOT syntax only because terms such as ref x T will evaluate to fresh locations during reduction.

The remaining rules are the DOT evaluation rules, with the only change that they pass along a heap.

11.3 MUTABLE DOT TYPING RULES

The Mutable DOT typing rules, depicted in Figure 11.3, depend on a *heap typing* Σ in addition to a type environment Γ. A heap typing maps locations to the types of the variables that they store. The heap typing spares us the need to re-typecheck locations and allows to typecheck cyclic references (Pierce, 2002).

As an example, the following Mutable DOT program cannot be easily typechecked without an explicit heap typing (using only the runtime heap and the type environment):

$$
\begin{array}{lllll}
 & \mathsf{let} & \mathsf{id} & = \lambda(x\colon \top)\, x & \mathsf{in} \\
p = & \mathsf{let} & r & = \mathsf{ref}\ \mathsf{id}\ (\top \to \top) & \mathsf{in} \\
 & \mathsf{let} & \mathsf{id}' & = \lambda(x\colon \top)\, (!r)\, x & \mathsf{in} \\
 & r := \mathsf{id}' & & &
\end{array}
$$

Starting with an empty heap, after two reduction steps we get

$$\varnothing \mid p \longmapsto^* \{l \to \mathsf{id}'\} \mid p',$$

where

$$
\begin{array}{lllll}
 & \mathsf{let} & \mathsf{id} & = \lambda(x\colon \top)\, x & \mathsf{in} \\
p' = & \mathsf{let} & r & = l & \mathsf{in} \\
 & \mathsf{let} & \mathsf{id}' & = \lambda(x\colon \top)\, (!r)\, x & \mathsf{in} \\
 & \mathsf{id}' & & &
\end{array}
$$

We would see by looking into the heap that to typecheck the location l, we needed to typecheck id'. id' depends on r, which in turn refers to the location l, creating a cyclic dependency.

We therefore augment our typing rules with a heap typing, allowing us to typecheck each location once and for all, at the time of a reference creation. In the example, we would know that l is mapped to $(\top \to \top)$ from the let-binding of r and remember this typing in Σ. To express that a term t has type T under the type environment Γ and heap typing Σ, we write $\Gamma, \Sigma \vdash t\colon T$.

Tight term typing
$\Gamma, \Sigma \vdash t : T$

$$\frac{\Gamma(x) = T}{\Gamma, \Sigma \vdash x : T} \text{ (Var)} \qquad \frac{\Gamma, \Sigma \vdash x : T}{\Gamma, \Sigma \vdash x : \mu\,(x : T)} \text{ (Rec-I)}$$

$$\frac{\Sigma(l) = T}{\Gamma, \Sigma \vdash l : \mathsf{Ref}\, T} \text{ (Loc)} \qquad \frac{\Gamma, \Sigma \vdash x : \mu\,(x : T)}{\Gamma, \Sigma \vdash x : T} \text{ (Rec-E)}$$

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

$$\frac{\Gamma, \Sigma \vdash x : \forall(z : S)\, T \qquad \Gamma, \Sigma \vdash y : S}{\Gamma, \Sigma \vdash x\, y : T\,[y/z]} \text{ (All-E)} \qquad \frac{\Gamma, \Sigma \vdash t : T \qquad \Gamma, \Sigma \vdash T <: U}{\Gamma, \Sigma \vdash t : U} \text{ (Sub)}$$

$$\frac{\Gamma, x : T, \Sigma \vdash d : T}{\Gamma, \Sigma \vdash \nu(x : T) d : \mu\,(x : T)} \text{ ($\{\}$-I)} \qquad \frac{\Gamma, \Sigma \vdash x : T}{\Gamma, \Sigma \vdash \mathsf{ref}\, x\, T : \mathsf{Ref}\, T} \text{ (Ref-I)}$$

$$\frac{\Gamma, \Sigma \vdash x : \{a : T\}}{\Gamma, \Sigma \vdash x.a : T} \text{ ($\{\}$-E)} \qquad \frac{\Gamma, \Sigma \vdash x : \mathsf{Ref}\, T}{\Gamma, \Sigma \vdash !x : T} \text{ (Ref-E)}$$

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

Definition typing
$\Gamma, \Sigma \vdash d : T$

$$\frac{\Gamma, \Sigma \vdash t : T}{\Gamma, \Sigma \vdash \{a = t\} : \{a : T\}} \text{ (Fld-I)} \qquad \frac{\Gamma, \Sigma \vdash d_1 : T_1 \qquad \Gamma, \Sigma \vdash d_1 : T_2 \qquad \mathsf{dom}(d_1), \mathsf{dom}(d_2) \text{ disjoint}}{\Gamma, \Sigma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \text{ (AndDef-I)}$$

$$\Gamma, \Sigma \vdash \{A = T\} : \{A : T..T\} \text{ (Typ-I)}$$

Figure 11.3: Mutable DOT typing rules

The typing rules for Mutable DOT are shown in Figure 11.3. The DOT rules are intact except that all typing derivations carry a heap typing. The new rules related to mutable references are as follows:

- We typecheck locations by looking them up in the heap typing. If, according to Σ, a location l stores a variable of type T, then l has type $\mathsf{Ref}\ T$ (Loc).
- A newly created reference $\mathsf{ref}\ x\ T$ can be initialized with the variable x if x has type T. In particular, if x's precise type U is a subtype of T, then x has type T by Sub, so we can still create a $\mathsf{ref}\ x\ T$ (Ref-I).
- Conversely, dereferencing a variable of a reference type $\mathsf{Ref}\ T$ yields the type T (Ref-E).
- Finally, if x is a reference of type $\mathsf{Ref}\ T$, we are allowed to store a variable y into it if y has type T. To avoid the need to add a Unit type to the type system, we define an assignment $x := y$ to reduce to y, so the type of the assignment is T (Asgn).

11.4 SUBTYPING RULES

The subtyping rules of Mutable DOT include an added heap typing, and a subtyping rule for references. The rules are shown in Figure 11.4.

Subtyping between reference types is invariant: usually, $\mathsf{Ref}\ T <: \mathsf{Ref}\ U$ if and only if $T = U$. Invariance is required because reference types need to be (i) covariant for reading, or dereferencing, and (ii) contravariant for writing, or assignment.

However, in DOT, co- and contra-variance between types does not imply type equality: the calculus contains examples of types that are not equal, yet are equivalent with respect to subtyping. For example, for any types T and U, $T \wedge U <: U \wedge T <: T \wedge U$. Yet, $T \wedge U \neq U \wedge T$. Therefore, subtyping between reference types requires both covariance and contravariance:

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

$$\Gamma, \Sigma \vdash T <: \top \quad \text{(Top)}$$

$$\Gamma, \Sigma \vdash \bot <: T \quad \text{(Bot)}$$

$$\Gamma, \Sigma \vdash T <: T \quad \text{(Refl)}$$

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

$$\Gamma, \Sigma \vdash T \wedge U <: T \ (\text{And}_1\text{-}<:)$$

$$\Gamma, \Sigma \vdash T \wedge U <: U \ (\text{And}_2\text{-}<:)$$

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

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

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

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

$$\frac{\Gamma, \Sigma \vdash T <: U}{\Gamma, \Sigma \vdash \{a\colon T\} <: \{a\colon U\}} \quad (\text{Fld}\text{-}<:\text{-Fld})$$

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

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

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

Figure 11.4: Mutable DOT subtyping rules

12

TYPE SAFETY

In this section, we outline the soundness proof of Mutable DOT as an extension of the simple DOT soundness proof presented in Part I.

The type-safety proof of Mutable DOT involved adding the following definitions to the proof recipe (see Section 5.1):

1. a new case to the definition of *inert types*: any reference type Ref T is inert; *cf. inert types in Definition 2*

2. *tight typing* extends the original tight typing definition with the new typing rules of Mutable DOT (exact versions of Ref-I, Ref-E, Loc, Asgn, and Ref-Sub): *cf. tight typing in Figure 5.1*

$$\frac{\Gamma, \Sigma \vdash_\# x : T}{\Gamma, \Sigma \vdash_\# \mathsf{ref}\ x\ T : \mathsf{Ref}\ T}\ (\textsc{Ref-I-\#}) \qquad \frac{\Gamma, \Sigma \vdash_\# x : \mathsf{Ref}\ T}{\Gamma, \Sigma \vdash_\# !x : T}\ (\textsc{Ref-E-\#})$$

$$\frac{\Sigma(x) = T}{\Gamma, \Sigma \vdash_\# l : \mathsf{Ref}\ T}\ (\textsc{Loc-\#}) \qquad \frac{\Gamma, \Sigma \vdash_\# x : \mathsf{Ref}\ T \qquad \Gamma, \Sigma \vdash_\# y : T}{\Gamma, \Sigma \vdash_\# x := y : T}\ (\textsc{Asgn-\#})$$

$$\frac{\Gamma, \Sigma \vdash_\# T <: U \qquad \Gamma \vdash_\# U <: T}{\Gamma, \Sigma \vdash_\# \mathsf{Ref}\ T <: \mathsf{Ref}\ U}\ (\textsc{Ref-Sub-\#})$$

3. *invertible typing* extends the invertible rules with an additional case that inlines the Ref-Sub subtyping rule: *cf. invertible typing in Figure 5.3*

$$\frac{\Gamma, \Sigma \vdash_¡ x : \mathsf{Ref}\ T \qquad \Gamma \vdash_\# T <: U \qquad \Gamma \vdash_\# U <: T}{\Gamma, \Sigma \vdash_¡ x : \mathsf{Ref}\ U}\ (\textsc{Ref-¡})$$

4. *precise typing* extends the precise value-typing rules with a new case for locations: *cf. precise typing in Figure 5.2*

$$\frac{\Sigma(x) = T}{\Gamma, \Sigma \vdash_! l : \mathsf{Ref}\ T}\ (\textsc{Loc-!})$$

Since the typing relation depends on a heap typing, the *well-formedness* relation also needs to include Σ. *cf. well-foremedness in Definition 16*

DEFINITION 26 (Well-formed Environments). *A store* $\gamma = \overline{(x_i, v_i)}$ *is well-formed with respect to a type environment* $\Gamma = \overline{(x_i, T_i)}$ *and heap typing* Σ, *written* $\gamma : \Gamma, \Sigma$, *if for each* i, $\Gamma, \Sigma \vdash_! v_i : T$.

Additionally, we need to define well-formedness for heaps with respect to heap typings:

DEFINITION 27 (Well-Typed Heap). *A heap* $\sigma = \overline{l_i \mapsto x_i}$ *is* well-typed *with respect to an environment* Γ *and heap typing* $\Sigma = \overline{l_i \mapsto T_i}$*, written* $\Gamma, \Sigma \vdash \sigma$*, if for each* i*,* $\Gamma, \Sigma \vdash x_i : T_i$*.*

We can now present the central lemmas required to prove the Mutable DOT soundness theorems.

The helper lemmas for canonical forms (Lemma 11 ($\forall$ to $\Gamma(x)$), Lemma 12 ($\forall$ to λ), Lemma 13 (μ to $\Gamma(x)$), Lemma 14 (μ to ν), and Lemma 17 (Corresp. Types)) are left unchanged. However, we need two additional lemmas for variables and values that have reference types.

LEMMA 28 (Ref T to $\Gamma(x)$).

$$\frac{\text{inert } \Gamma \qquad \Gamma, \Sigma \vdash x : \mathsf{Ref}\ T}{\Gamma(x) = \mathsf{Ref}\ U \qquad \Gamma, \Sigma \vdash \mathsf{Ref}\ U <: \mathsf{Ref}\ T \qquad \Gamma, \Sigma \vdash \mathsf{Ref}\ T <: \mathsf{Ref}\ U}$$

LEMMA 29 (Ref T to l).

$$\frac{\text{inert } \Gamma \qquad \Gamma, \Sigma \vdash x : \mathsf{Ref}\ T}{\Gamma(x) = \mathsf{Ref}\ U \qquad \Gamma, \Sigma \vdash \mathsf{Ref}\ U <: \mathsf{Ref}\ T \qquad \Gamma, \Sigma \vdash \mathsf{Ref}\ T <: \mathsf{Ref}\ U}$$

With that we can prove canonical forms for mutable references.

$$\frac{\gamma : \Gamma \qquad \Gamma, \Sigma \vdash \sigma \qquad \text{inert } \Gamma \qquad \Gamma, \Sigma \vdash x : \mathsf{Ref}\ T}{\gamma(x) = l \qquad \sigma(l) = y \qquad \Gamma, \Sigma \vdash l : \mathsf{Ref}\ T \qquad \Gamma, \Sigma \vdash y : T}\ \text{(Can. Forms for } l\text{)}$$

LEMMA 30 (Canonical Forms for References). *Suppose that a store* γ *is well-formed with respect to an inert environment* Γ *and a heap typing* Σ*, that a heap* σ *is well-typed with respect to* Γ *and* Σ*, and that* $\Gamma \vdash x : \mathsf{Ref}\ T$*. Then the store* γ *assigns a location* l *of type* $\mathsf{Ref}\ T$ *to* x*, and* l *points to a variable* y *of type* T *in the heap.*

Finally, we can present the progress and preservation theorems, as well as the ultimate soundness result.

$$\frac{\gamma : \Gamma \qquad \Gamma, \Sigma \vdash \sigma \qquad \text{inert } \Gamma \qquad \Gamma \vdash t : T \qquad \gamma \mid t \longmapsto \gamma' \mid t'}{\Gamma' \vdash t' : T \qquad \Gamma', \Sigma' \vdash \sigma' \qquad \gamma' : \Gamma' \qquad \text{inert } \Gamma'}\ \text{(PRESERVATION)}$$

$$\frac{\gamma : \Gamma \qquad \Gamma, \Sigma \vdash \sigma \qquad \text{inert } \Gamma \qquad \Gamma \vdash t : T}{\gamma \mid t \longmapsto \gamma' \mid t'}\ \text{(PROGRESS)}$$

THEOREM 31 (Mutable DOT Preservation). *Let* Γ *be an inert typing environment and* Σ *a heap typing such that* $\gamma : \Gamma, \Sigma$ *and* $\Gamma, \Sigma \vdash \sigma$*. If* $\Gamma, \Sigma \vdash t : T$ *and* $\sigma \mid \gamma \mid t \longmapsto \sigma' \mid \gamma' \mid t'$ *for some store* γ' *and heap* σ' *then there exists an inert context* Γ' *and heap typing* Σ' *such that* $\gamma' : \Gamma', \Sigma'$*,* $\Gamma', \Sigma' \vdash \sigma'$*, and* $\Gamma', \Sigma' \vdash t' : T$*.*

THEOREM 32 (Mutable DOT Progress). *Let* γ *be a store,* σ *a heap,* Γ *an inert typing environment, and* Σ *a heap typing such that* $\gamma : \Gamma, \Sigma$ *and* $\Gamma, \Sigma \vdash \sigma$*. If* $\Gamma, \Sigma \vdash t : T$ *then there exists a term* t'*, store* γ'*, and heap* σ' *such that* $\sigma \mid \gamma \mid t \longmapsto \sigma' \mid \gamma' \mid t'$*.*

THEOREM 33 (Mutable DOT Type Soundness). *If* $\varnothing, \Sigma \vdash t : T$ *then either* t *diverges (*$t \Uparrow$*) or* t *reduces to a normal form* t'*, i.e.* $\sigma \mid \varnothing \mid t \longmapsto \sigma' \mid \gamma' \mid t'$*,* $t' \not\to$*, and* $\Gamma, \Sigma' \vdash t' : T$ *for some* Γ *and* Σ' *such that* $\gamma' : \Gamma, \Sigma'$*.*

$$\frac{\varnothing, \Sigma \vdash t : T}{(\sigma \mid \gamma \mid t \longmapsto \sigma' \mid \gamma' \mid t' \wedge t' \not\to) \vee t \Uparrow}\ \text{(Mutable DOT Soundness)}$$

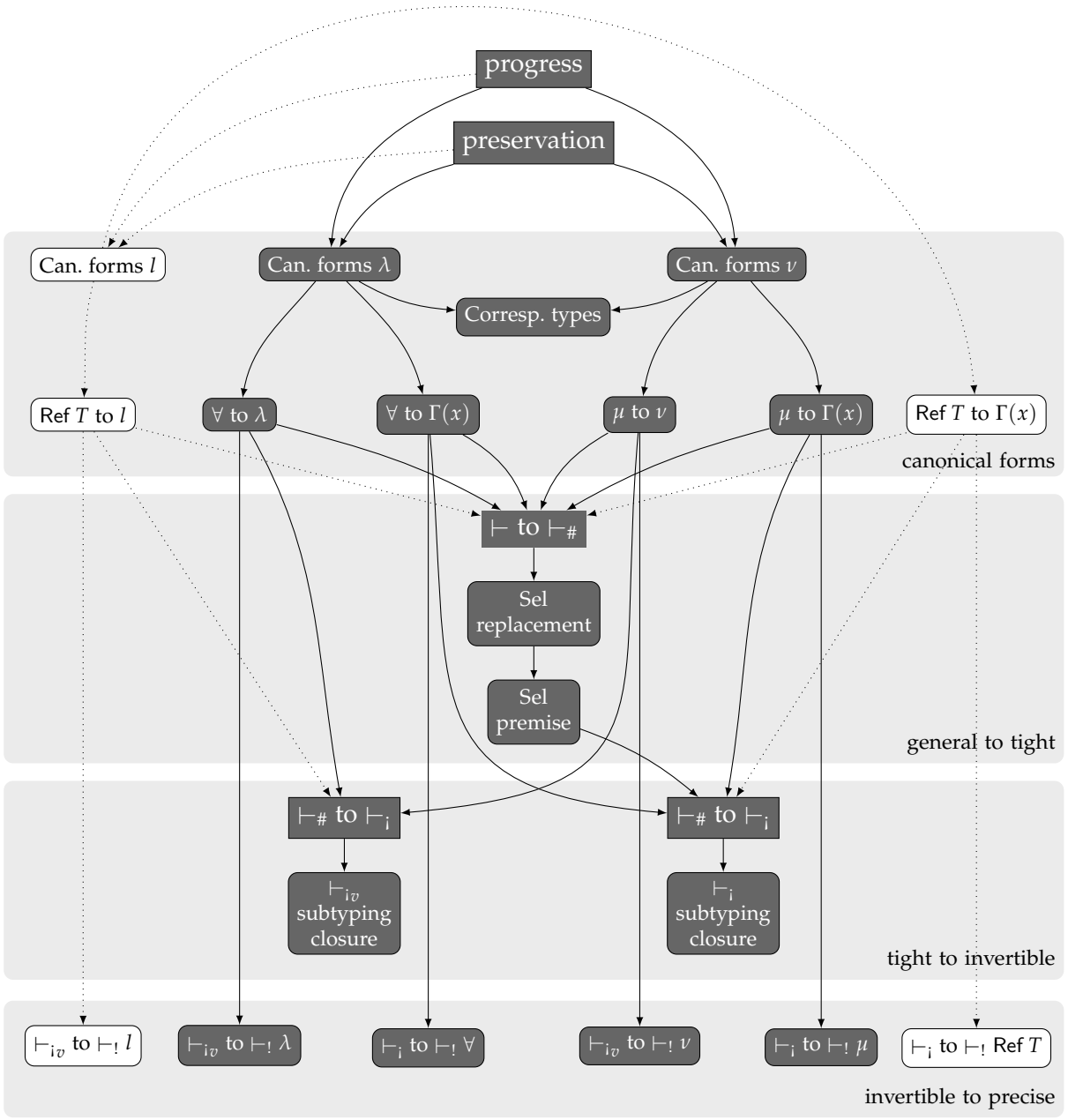

Figure 12.1: An instance of the dependency graph from Figure 5.4 showing the main lemmas in the Mutable DOT proof as an extension of the simple DOT proof (Part I). Gray nodes denote existing lemmas. White nodes denote Mutable DOT specific lemmas

The changes to the lemmas proof are shown using white nodes in the dependence graph in Figure 12.1.

The overall structure of the dependencies between the lemmas did not change. The new canonical forms lemmas followed the proof recipe that we have described in Chapter 5. In the proofs of some lemmas, we had additional new cases to prove, but the structure of the proof of each lemma did not change. In general, we found that the new lemmas and new cases in the existing lemmas were easy to prove.

13 DISCUSSION

In this section, we explain the design choices of Mutable DOT in more detail and discuss possible alternative implementations.

13.1 MOTIVATION FOR A HEAP OF VARIABLES

One unusual aspect of the design of Mutable DOT is that the heap contains variables rather than values. We experimented with alternative designs that contained values, and observed the following undesirable interactions with the existing design of DOT.

A key desirable property is that the heap should be well-typed with respect to a heap typing: $\forall l.\ \Gamma, \Sigma \vdash \sigma(l) : \Sigma(l)$.

Many of the DOT type assignment rules apply only to variables, and not to values. For example, the type $\{a\colon \top\}$ is not inhabited by any value, but a variable can have this type. This is because an object value has a recursive type, and the Rec-E rule that opens a recursive type $\mu\,(x\colon \{a\colon \top\})$ into $\{a\colon \top\}$ applies only to variables, not to values. In particular, in the term

$$\textbf{let}\ x = \nu(y\colon \{a\colon \top\})\{a = t\}\ \textbf{in}\ \mathsf{ref}\ x\ \{a\colon \top\}$$

x has type $\{a\colon \top\}$ but $\nu(y\colon \{a\colon \top\})\{a = t\}$ does not, even though the let binding suggests that the variable and the value should be equal. If memory cells were to contain values, a cell of type $\{a\colon \top\}$ would not make sense, because no values have that type.

We could try to restrict reference types to always store recursive (or function) types. However, this would severly restrict the polymorphism of memory cells, because DOT does not support subtyping between recursive types (subtyping between recursive structural types is not supported by Scala either). In particular, it would be impossible to define a memory cell containing objects with a field a of type $\top$ and possibly additional fields.

The above example let term demonstrates another problem: type preservation. The type system should admit the term $\mathsf{ref}\ x\ \{a\colon \top\}$ because x has type $\{a\colon \top\}$. This term should reduce to a fresh location l of type $\mathsf{Ref}\ \{a\colon \top\}$. But a heap that maps l to $\nu(y\colon \{a\colon \top\})\{a = t\}$ would not be well typed, because the value does not have type $\{a\colon \top\}$.

13.2 CORRECTNESS OF A HEAP OF VARIABLES

Putting variables instead of values in the heap raises a concern: when we write a variable into the heap, we expect that when we read it back, it will still be in scope, and it will still be bound to the same value. For example, in the following program fragment, the variable x gets saved in the heap inside the function f.

$$\begin{array}{lcll} \textbf{let } f & = & \lambda(x\colon \top)\ \mathsf{ref}\ x\ T & \textbf{in} \\ \textbf{let } y & = & v & \textbf{in} \\ \textbf{let } r & = & f\ y & \textbf{in} \\ !r & & & \end{array}$$

Will x go out of scope by the time we read it from the heap?

The reduction sequence for this program is shown in Figure 13.1. Notice that before the body $\mathsf{ref}\ x\ T$ of the function is reduced, the parameter x is first substituted with the argument y, which does not go out of scope.

$$\begin{array}{lcll} \varnothing & \mid & f \mapsto \lambda(x\colon \top)\ \mathsf{ref}\ x\ T,\ y \mapsto v & \mid \quad \textbf{let } r = f\ y \textbf{ in } !r & \longmapsto \\ \varnothing & \mid & f \mapsto \lambda(x\colon \top)\ \mathsf{ref}\ x\ T,\ y \mapsto v & \mid \quad \textbf{let } r = \mathsf{ref}\ x\ T\ [y/x] \textbf{ in } !r & \longmapsto \\ \varnothing & \mid & f \mapsto \lambda(x\colon \top)\ \mathsf{ref}\ x\ T,\ y \mapsto v & \mid \quad \textbf{let } r = \mathsf{ref}\ y\ T \textbf{ in } !r & \longmapsto \\ l \mapsto y & \mid & f \mapsto \lambda(x\colon \top)\ \mathsf{ref}\ x\ T,\ y \mapsto v & \mid \quad \textbf{let } r = l \textbf{ in } !r & \longmapsto \\ l \mapsto y & \mid & f \mapsto \lambda(x\colon \top)\ \mathsf{ref}\ x\ T,\ y \mapsto v,\ r \mapsto l & \mid \quad !r & \longmapsto \\ l \mapsto y & \mid & f \mapsto \lambda(x\colon \top)\ \mathsf{ref}\ x\ T,\ y \mapsto v,\ r \mapsto l & \mid \quad y & \end{array}$$

Figure 13.1: Reduction sequence for example program

More generally, from the store-based reduction semantics, it is immediately obvious that when a variable x is saved in the heap using $\mathsf{ref}\ x\ T$ or $y := x$, the only variables that are in scope are those in the store. There are no function parameters in scope that could go out of scope when the function finishes.

$$\sigma \mid \gamma \mid \textbf{\textit{let}}\ x = v\ \textbf{\textit{in}}\ t \longmapsto \sigma \mid \gamma,\ x \mapsto v \mid t \quad \text{(LET-VALUE)}$$

Moreover, once a variable is in the store, it never goes out of scope, and the value that it is bound to never changes. This is because the only reduction rule that modifies the store is LET-VALUE, and it only adds a new variable binding, but does not affect any existing bindings.

Another natural question is whether a heap of variables limits the expressiveness of the calculus. Since a program contains only a finite number of variables, one might think that the size of the heap is restricted by that number. However, during execution, the reduction rule for function application performs capture-avoiding substitution using alpha renaming, which introduces fresh variables as necessary. Thus, the use of variables in the heap does not impose any restrictions on the number of objects that can be created.

13.3 CREATING REFERENCES

The Mutable DOT reference creation term ref $x\ T$ requires both a type T and an initial variable x. The variable is needed so that a reference cell is always initialized, to avoid the need to add a null value to DOT. If desired, it is possible to model uninitialized memory cells in Mutable DOT by explicitly creating a sentinel null value.

Some other calculi with mutable references (e.g. Types and Programming Languages (Pierce, 2002)) do not require the type T to be given explicitly, but just adopt the precise type of x as the type for the new cell. Such a design does not fit well with subtyping in DOT. In particular, it would prevent the creation of a cell with some general type T initialized with a variable x of a more specific subtype of T.

More seriously, such a design (together with subtyping) would break type preservation. Suppose that $\Gamma, \Sigma \vdash y\colon S$ and $\Gamma, \Sigma \vdash S <\colon T$. Then we could arrive at the following reduction sequence:

$$
\begin{array}{lllll}
\varnothing & | & f \mapsto \lambda(x\colon T)\ \mathsf{ref}\ x,\ y \mapsto v & | & f\ y \longmapsto \\
\varnothing & | & f \mapsto \lambda(x\colon T)\ \mathsf{ref}\ x,\ y \mapsto v & | & \mathsf{ref}\ x\ [y/x] \longmapsto \\
\varnothing & | & f \mapsto \lambda(x\colon T)\ \mathsf{ref}\ x,\ y \mapsto v & | & \mathsf{ref}\ y
\end{array}
$$

The term at the beginning of the reduction sequence has type Ref T, while the term at the end, ref y, has type Ref S. Preservation would require Ref S to be a subtype of Ref T, but this is not the case in general since the only condition that this example imposes on S and T is that $\Gamma, \Sigma \vdash S <\colon T$.

14 RELATED WORK

This part of the thesis shows how to extend the DOT calculus with ML-style mutable references, to serve as a basis for further extensions that involve mutation. An earlier version of the Mutable DOT calculus with a soundness proof that extended the original proof by Amin, Grütter, et al., (2016) appeared in a previous publication (Rapoport and Lhoták, 2017) and a technical report (Rapoport and Lhoták, 2016).

Amin and Rompf, (2017) present mechanized soundness proofs using definitional interpreters for big-step DOT-like calculi. The paper and an earlier technical report (Rompf and Amin, 2016a) mention that mutable references can be added to this class of calculi and present a Coq formalization of System $F_{<:}$ with mutable references. To our knowledge, the formalization does not include a soundness proof for $D_{<:}$ or DOT with mutation. Additionally, the calculi discussed in the papers are based on a big-step semantics, whereas our work focuses on a small-step semantics for DOT.

Kabir and Lhoták, (2018) present κDOT, an extension of DOT with constructors and mutable fields. Unlike DOT, κDOT's objects support strict initialization of fields, which corresponds closer to Scala than DOT's field semantics. κDOT's fields are mutable by default, and they are not explicitly typed as references, unlike in our version of Mutable DOT. Still, κDOT can express read-only fields through setting a field's lower-bound type to $\bot$. The difference between Mutable DOT and κDOT is that the latter focuses on modelling constructors and makes all fields mutable by default, whereas Mutable DOT presents the first and minimal addition to DOT with mutation, and supports both mutable fields and variables.

15

SUMMARY

DOT formalizes the essence of Scala, but it lacks mutation, which is an important feature of object-oriented languages. In this part of the thesis we showed how DOT can be extended to handle mutation in a type-safe way.

We have seen that adding a mutable heap to the semantics of DOT is not straightforward. The lack of subtyping between recursive types leads to situations where variables and values, even though they are bound together, have incompatible types. As a result, if DOT were extended with a conventional heap containing values, it would be impossible for a cell of a given type T to store values of different subtypes of T, thus significantly restricting the kinds of mutable code that could be expressed.

Our key idea was to enable support for mutation in DOT by using a heap that contains variables instead of values. We showed that by using a heap of variables, it is possible to extend DOT with mutable references in a type-safe way. This leads to a formalization of a language with path-dependent types and mutation, and also brings DOT one step closer to encoding the full Scala language.

Although designing Mutable DOT involved non-trivial reasoning about how to maintain preservation while allowing a mutable heap, *proving* type-safety of Mutable DOT involved a straightforward extension of the simple type-safety proof presented in Part I. The extension of DOT with mutation serves as evidence that the simple proof is indeed simple. In the next part, we present an extension of DOT that is more complex and involves significant changes to the proof. Nevertheless, as we will show in Part III, the basic organization and key ideas of the proof remain the same.

Part III

FULLY PATH-DEPENDENT TYPES

16 INTRODUCTION

path-dependent type
composition abstraction

Path-dependent types embody two universal principles of modular programming: abstraction and composition. Abstraction allows us to leave values or types in a program unspecified to keep it generic and reusable. For example, in Scala, we can define trees where the node type remains abstract:

```
trait Tree {
  type Node
  val root: Node
  def add(node: Node): Tree
}
```

If an object `x` has type `Tree`, then the path-dependent type `x.Node` denotes the type of abstract nodes.

Composition is the ability to build our program out of smaller components. For example, if we are interested in a specific kind of tree, say a red-black tree, then we can refine the abstract `Node` type to contain a `Color` type:

```
trait RedBlackTree extends Tree {
  type Node <: { type Color }
}
```

This exemplifies composition in at least two ways: by having `RedBlackTree` extend `Tree` we have *inherited* its members; and by nesting the refined definition of `Node` within `RedBlackTree` we have used *aggregation*. If an object `r` is a `RedBlackTree`, then the path-dependent type `r.root.Color` allows us to traverse the composition and access the `Color` type member.

As described in the previous chapters, the long struggle to formalize path-dependent types recently led to machine-verified soundness proofs for several variants of the DOT calculus. In spite of its apparent simplicity DOT is an expressive calculus that can encode a variety of language features, and the discovery of its soundness proof was a breakthrough for the Scala community.

However, a crucial limitation is that the existing DOT calculi restrict path-dependent types to depend only on variables, not on general paths. That is, they allow the type `x.Node` (path of length 1) but not a longer type such as `r.root.Color` (length 2). We need to lift this restriction in order to faithfully model Scala which does allow general path-dependent types. More importantly, this restriction must be lifted to fulfill the goal of *scalable component abstraction* (Odersky and Zenger, 2005b), in which modules of a program can be arbitrarily nested to form other, larger modules.

Scala:

```
package dotty {
  package core {
    object types {
      class Type
      class TypeRef extends Type {
        val symb:
            core.symbols.Symbol
      }
    }
    object symbols {
      class Symbol {
        val tpe: core.types.Type
      }
    }
  }
}
```

DOT:

```
let dotty = ν(dotty) {
  core = ν(core) {
    types = ν(types) {
      Type = ...
      TypeRef = Type ∧
        { symb: core.symbols.Symbol }
    }
    symbols = ν(symbols) {
      Symbol =
        { tpe: core.types.Type }
    }
  }
in ...
```

Figure 16.1: A simplified excerpt from the Dotty compiler in Scala. This code fragment cannot be expressed in DOT, as shown on the right

The final part of this thesis formalizes and proves sound a generalization of the DOT calculus with path-dependent types of arbitrary length. We call the new path-dependent calculus pDOT. Our Coq-verified proof is built on top of the simple proof presented in Part I.

At this point, two questions naturally arise. Are fully path-dependent types really necessary? That is, do they provide additional expressiveness, or are they just syntactic sugar over variable-dependent types? And if fully path-dependent types are in fact useful, what are the barriers to adding them to DOT?

WHY FULLY PATH-DEPENDENT TYPES ARE NECESSARY

The need for paths of arbitrary length is illustrated by the simplified excerpt from the implementation of the Scala 3 ("Dotty") compiler in Figure 16.1. Type references (TypeRef) are Types that have an underlying class or trait definition (Symbol), while Symbols in the language also have a Type. Additionally, TypeRefs and Symbols are nested in different packages, core.types and core.symbols.

It is impossible to express the above type dependencies in DOT while maintaining the nested program structure, as shown on the right side of Figure 16.1 in DOT syntax (see Chapter 2 for a description of the DOT syntax and type system). To replicate nested Scala modules, DOT uses objects and fields. Unfortunately, we run into problems when typing the symb field because its desired path-dependent type *core*.symbols.Symbol has a path of length two.

We are then tempted to find a workaround. One option is to try to reference Symbol as a path-dependent type of length one: symbols.Symbol instead of *core*.symbols.Symbol. However, this will not do because symbols

is a field, and DOT requires that field accesses happen through the enclosing object (core). Another option is to move the definition of the Symbol type member to the place it is accessed from, to ensure that the path to the type member has length 1:

```
types = v(types) {
  Type = ...;
  Symbol = ...;
  TypeRef = Type ∧ { symb: types.Symbol }
}
```

However, such a transformation would require flattening the nested structure of the program whenever we need to use path-dependent types. This would limit encapsulation and our ability to organize a program according to its logical structure. Yet another approach is to assign the symbols object to a variable that is defined before the dotty object:

```
let symbols = v(symbols) {
  Symbol = { tpe: dotty.core.types.Type }
}
in let dotty = v(dotty) ...
```

This attempt fails as well, as the symbols object can no longer reference the dotty package. For the above example this means that a Symbol cannot have a Type.

see Section 17.1 for a minimal example that illustrates this limitation

This real-world pattern with multiple nested modules and intricate dependencies between them (sometimes even *recursive* dependencies, as in our example), leads to path-dependent types of length greater than one. Because path-dependent types are used in DOT to formalize features like parametric and family polymorphism (Ernst, 2001), covariant specialization (Bruce, Odersky, and Wadler, 1998), and wildcards, among others, a version of DOT with just variable-dependent types can only formalize these features in special cases. Thus, to unleash the full expressive power of DOT we need path-dependent types on paths of arbitrary length.

variable-dependent (limited composition) type (abstraction)

WHY FULLY PATH-DEPENDENT TYPES ARE HARD

The restriction to types dependent on variables rather than paths is not merely cosmetic; it is fundamental. A key challenge in formalizing the DOT calculi is the *bad bounds* problem, discussed in Chapter 4: the occurrence of a type member in a program introduces new subtyping relationships, and these subtyping relationships could undermine type safety in the general case. To maintain type safety, the existing DOT calculi ensure that whenever a type $x.A$ is in scope, any code in the same scope will not execute until x has been assigned some concrete value; the value serves as evidence that type soundness has not been subverted. If we allow a type to depend on a path, rather than a

Section 17.2.1 shows why extending DOT with path can lead to bad bounds

variable, we must extend this property to paths: we must show that whenever a scope admits a given path, that path will always evaluate to some stable value. The challenge of ensuring that the paths of type-selections always evaluate to a value is to rule out the possibility that paths cyclically alias each other, while at the same time keeping the calculus expressive enough to allow recursion. By contrast, the DOT calculus automatically avoids the problem of type selections on non-terminating paths (i.e. paths whose evaluation does not terminate) because in DOT all paths are variables, and variables are considered normal form.

A second challenge of extending DOT with support for general paths is to track *path equality*. Consider the following program:

```
val t1 = new ConcreteTree
val t2 = new ConcreteTree
val t3 = t2
```

A subclass of `Tree` such as `ConcreteTree` (not shown) refines `Node` with a concrete type that implements some representation of nodes. We want the types `t1.Node` and `t2.Node` to be considered distinct even though `t1` and `t2` are both initialized to the same expression. That way, we can distinguish the nodes of different tree instances. On the other hand, notice that the reference `t3` is initialized to be an alias to the same tree instance as `t2`. We therefore want `t2.Node` and `t3.Node` to be considered the same type.

How can the type system tell the difference between `t1.Node` and `t2.Node`, so that the former is considered distinct from `t3.Node`, but the latter is considered the same? Scala uses *singleton types* for this purpose. In Scala, `t3` can be typed with the singleton type `t2.type` which guarantees that it is an alias for the same object as `t2`. The type system treats paths that are provably aliased (as evidenced by singleton types) as interchangeable, so it considers `t2.Node` and `t3.Node` as the same type. We add singleton types to pDOT for two reasons: first, we found singleton types useful for formalizing path-dependent types, and second, enabling singleton types brings DOT closer to Scala.

The contributions are as follows:

Chapter 18 provides an intuition for pDOT's main ideas; Chapter 19 presents the calculus in detail

1. The *pDOT* calculus, a generalization of DOT with *path-dependent types of arbitrary length* that lifts DOT's type-selection-on-variables restriction.

2. The first extension of DOT with *singleton types*, a Scala feature that, in addition to tracking path equality, enables the method chaining pattern and hierarchical organization of components (Odersky and Zenger, 2005b).

Coq proof: https://git.io/dotpaths described in Chapter 21

3. A *Coq-mechanized type soundness proof* of pDOT that is based on the simple soundness proof presented in Part I. Our proof maintains the simple proof's modularity properties which makes it easy to extend pDOT with new features.

4. *Formalized examples* that illustrate the expressive power of pDOT: the compiler example from this section that uses general path-dependent types, a method chaining example that uses singleton types, and a covariant list implementation. *see Chapter 20*

17 CHALLENGES OF ADDING PATHS TO DOT

This chapter shows why the existing DOT calculus cannot encode path-dependent types on paths of arbitrary length and describes how naively extending DOT with support for full paths leads to bad bounds.

17.1 PATH LIMITATIONS IN DOT: A MINIMAL EXAMPLE

Consider the following example DOT object in which a type member B refers to a type member A that is nested inside the definition of a field c:

$$\begin{aligned}&\textbf{let } x = \nu(x)\\ &\qquad \{c = \nu(\_)\,\{A = x.B\}\} \wedge\\ &\qquad \{B = x.c.A\}\ \textbf{in}\ \ldots\end{aligned}$$

In the example, to reference the field c, we must first select the field's enclosing object x through its self variable. As a result, the path to A leads through $x.c$ which is a path of length two. Since DOT does not allow paths of length two, this definition of B cannot be expressed in DOT without flattening the program structure so that all fields and type members become global members of one top-level object.

Since DOT uses ANF, we may try to decompose the path of length two into dereferences of simple variables, but as we will see, this will fail. It does not work to first assign $x.c$ to a local variable y outside of the object x and then use the type $y.A$:

$$\begin{aligned}&\textbf{let } y = x.c && \textbf{in}\\ &\textbf{let } x = \nu(x)\\ &\qquad \{c = \nu(\_)\,\{A = x.B\}\} \wedge\\ &\qquad \{B = y.A\} && \textbf{in}\ \ldots\end{aligned}$$

This program is invalid because at the declaration site of y, x is not yet defined. We could try other ways of let-binding the inner objects to variables before defining the enclosing object, but all such attempts are doomed to failure. A sequence of let bindings imposes a total ordering on the objects and restricts an object to refer only to objects that are defined before it. In the presence of recursive references between the objects, as in this example, no valid ordering of the let bindings is possible.

17.2 CHALLENGES OF ADDING PATHS TO DOT

If restricting path-dependent types exclusively to variables limits the expressivity of DOT then why does the calculus impose such a constraint? Before we explain the soundness issue that makes it difficult to extend DOT with paths we must first review the problem of bad bounds, a key challenge that makes it difficult to ensure soundness of the DOT calculus.

As discussed in Chapter 4, Scala's abstract type members make it possible to define custom subtyping relationships between types. This is a powerful but tricky feature. For example, given any types S and U, consider the function $\lambda(x\colon \{A\colon S..U\})\, t$. In the body of the function, we can use $x.A$ as a placeholder for some type that is a supertype of S and a subtype of U. Some concrete type will be bound to $x.A$ when the function is eventually called with some specific argument. Due to transitivity of subtyping, the constraints on $x.A$ additionally introduce an assumption inside the function body that $S <: U$, because $S <: x.A <: U$ according to the type rules <:-Sel$_{\text{DOT}}$ and Sel-<:$_{\text{DOT}}$:

We add an explicit $_{DOT}$ annotation to DOT rules to distinguish them from pDOT rules

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

However, recall that S and U are arbitrary types, possibly with no existing subtyping relationship. The key to soundness is that although the function body is type-checked under the possibly unsound assumption $S <: U$, the body executes only when the function is called, and calling the function requires an argument that specifies a concrete type T to be bound to $x.A$. This argument type must satisfy the constraints $S <: T <: U$. Thus, the argument type embodies a form of *evidence* that the assumption $S <: U$ which is used to type-check the function body is actually valid.

More generally, given a term t of type $\{A\colon S..U\}$, we can rule out the possibility of bad bounds caused by the use of a dependent type $t.A$ if there exists some object with the same type $\{A\colon S..U\}$. This is because the object must bind the type member A to some concrete type T respecting the subtyping constraints $S <: T$ and $T <: U$, so the object is evidence that $S <: U$.

Existing DOT calculi ensure that whenever some variable x of type T is in scope in some term t, the term reduces only after x has already been assigned a value. The value assigned to x is evidence that T does not have bad bounds. To ensure that any code that uses the type $x.A$ executes only after x has been bound to a value of a compatible type, DOT employs a strict operational semantics. A variable x can be introduced by one of the three binding constructs: **let** $x = t$ **in** u, $\lambda(x\colon T)\, t$, or $\nu(x\colon T)d$. In the first case, x is in scope within u, and the reduction semantics requires that before u can execute, t must

first reduce to a value with the same type as x. In the second case, x is in scope within t, which cannot execute until an argument value is provided for the parameter x. In the third case, the object itself is bound to the self variable x. In summary, the semantics ensures that by the time that evaluation reaches a context with x in scope, x is bound to a value, and therefore x's type does not introduce bad bounds.

17.2.1 *Naive Path Extension Leads to Bad Bounds*

When we extend the type system with types $p.A$ that depend on paths rather than variables, we must take similar precautions to control bad bounds. If a path p has type $\{A\colon S..U\}$ and some normal form n also has this type, then n must be an object that binds to type member A a type T such that $S <: T <: U$.

However, not all syntactic paths in DOT have this property. For example, in an object $v(x)\,\{a = t\}$, where t can be an arbitrary term, t could loop instead of reducing to a normal form of the same type. In that case, there is no guarantee that a value of the type exists, and it would be unsound to allow the path $x.a$ as the prefix of a path-dependent type $x.a.A$.

The following example, in which a function $x.b$ is typed as an object (a record with field c), demonstrates this unsoundness:

$$\begin{aligned} v(x :\ & \{a\colon\ \{C\colon \forall(y\colon \top)\,\top..\{c\colon \top\}\}\} \\ & \{a = x.a\} \end{aligned} \qquad \begin{aligned} & \wedge \{b\colon\ \{c\colon \top\}\}) \\ & \wedge \{b = \lambda(y\colon \top)\,y\} \end{aligned}$$

Here, $x.b$ refers to a function $\lambda(y\colon \top)\,y$ of type $\forall(y\colon \top)\,\top$. If we allowed such a definition, the following would hold:

$$\forall(y\colon \top)\,\top <: x.a.C <: \{c\colon \top\}\,.$$

Then by subsumption, $x.b$, a function, has type $\{c\colon \top\}$ and therefore it must be an object. To avoid this unsoundness, we have to rule out the type selection $x.a.C$ on the non-terminating path $x.a$.

In general, if a path p has a field declaration type $\{a\colon T\}$, then the extended path $p.a$ has type T, but we do not know whether there exists a value of type T because $p.a$ has not yet reduced to a variable. Therefore, the type T could have bad bounds, and we should not allow the path $p.a$ to be used in a path-dependent type $p.a.A$.

The main difficulty we encountered in designing pDOT was to ensure that type selections occur only on terminating paths while ensuring that the calculus still permits non-terminating paths in general, since that is necessary to express recursive functions and maintain Turing completeness of the calculus.

18 MAIN IDEAS

Chapter 19 presents pDOT detail

This chapter outlines the main ideas that have shaped our definition of pDOT. To distinguish the DOT typing rules by Amin, Grütter, et al., (2016) presented in Chapter 2 from the pDOT typing rules we will postfix the DOT typing rules with "$_{DOT}$". For example, Var$_{DOT}$ is a DOT typing rule, but Var is a pDOT typing rule.

18.1 PATHS INSTEAD OF VARIABLES

To support fully-path-dependent types, our calculus needs to support paths in all places where DOT permitted variables. Consider the following example:

let $x = \nu(y)\,\{a = \nu(z)\,\{B = U\}\}$
in $x.a$

In order to make use of the fact that $U <: x.a.B <: U$, we need a type rule that reasons about path-dependent types. In DOT, this is done through the Sel-<:$_{DOT}$ and <:-Sel$_{DOT}$ rules, as mentioned in Chapter 4. Since we need to select B on a path $x.a$ and not just on a variable x, we need to extend the rules (merged into one here for brevity) to support paths:

$$\frac{\Gamma \vdash x\colon \{A\colon S..T\}}{\Gamma \vdash S <: x.A <: T}\ \text{<:-Sel-<:}_{DOT} \quad \Rightarrow \quad \frac{\Gamma \vdash p\colon \{A\colon S..T\}}{\Gamma \vdash S <: p.A <: T}\ \text{<:-Sel-<:}$$

However, before we can use this rule we need to also generalize the recursion elimination rule Rec-E$_{DOT}$. In the above example, how do we obtain the typing $\Gamma \vdash x.a\colon \{B\colon U..U\}$? The only identifier of the inner object is $x.a$, a path. The type of the path is $\mu\,(z\colon \{B\colon U..U\})$. In order to use the type member B, it is necessary to specialize this recursive type, replacing the recursive self variable z with the path $x.a$. This is necessary because the type U might refer to the self variable z, which is not in scope outside the recursive type. Thus, in order to support path-dependent types, it is necessary to allow recursion elimination on objects identified by paths:

$$\frac{\Gamma \vdash x\colon \mu\,(y\colon T)}{\Gamma \vdash x\colon T\,[x/y]}\ \text{Rec-E}_{DOT} \quad \Rightarrow \quad \frac{\Gamma \vdash p\colon \mu\,(y\colon T)}{\Gamma \vdash p\colon T\,[p/y]}\ \text{Rec-E}$$

By similar reasoning, we need to generalize all DOT variable-typing rules to path-typing rules. As we show later, we also have to generalize DOT's ANF syntax to use paths wherever DOT uses variables, so all

the DOT reduction rules that operate on variables are generalized to paths in pDOT.

18.2 PATHS AS IDENTIFIERS

A key design decision of pDOT is to let *paths represent object identity*. In DOT, object identity is represented by variables, which works out because variables are irreducible. In pDOT, *paths* are *irreducible*, because reducing paths would strip objects of their identity and break preservation.

18.2.1 *Variables are Identifiers in DOT*

In the DOT calculus by Amin, Grütter, et al., (2016), variables do not reduce to values for two reasons:

- *type safety*: making variables irreducible is necessary to maintain preservation, and
- *object identity*: to access the members of objects (which can recursively reference the object itself), objects need to have a name; reducing variables would strip objects of their identity.

If variables in DOT reduced to values, then in the previous example program, [4] x would reduce to

[4] **let** $x = v(y)$ $\{a = v(z)\,\{B = U\}\}$ **in** $x.a$

$$v = v(y)\,\{a = v(z)\,\{B = U\}\}\,.$$

To maintain type preservation, for any type T such that $\Gamma \vdash x\colon T$, we also must be able to derive $\Gamma \vdash v\colon T$. Since

$$\Gamma \vdash x\colon \mu\,(y\colon\ \{a\colon \mu\,(z\colon\ \{B\colon U..U\})\})\,,$$

by recursion elimination Rec-E$_{DOT}$,

$$\Gamma \vdash x\colon\ \{a\colon \mu\,(z\colon\ \{B\colon U\,[x/y]..U\,[x/y]\})\}\,.$$

Does v also have that type? No!

$$\dfrac{\dfrac{\Gamma \vdash x\colon \mu\,(y\colon\ \{a\colon \mu\,(z\colon\ \{B\colon U..U\})\})}{\Gamma \vdash x\colon\ \{a\colon \mu\,(z\colon\ \{B\colon U\,[x/y]\,..U\,[x/y]\})\}}\ \textsc{Rec-E}_{DOT} \qquad \gamma\colon\Gamma \qquad \dfrac{\gamma(x) = v}{\gamma \mid x \longmapsto \gamma \mid v}\ \textsc{Hypothetical Reduction}}{\Gamma \vdash v\colon\ \{a\colon \mu\,(z\colon\ \{B\colon U\,[x/y]\,..U\,[x/y]\})\}}\ \textsc{preservation}_{DOT}$$

The value v has only the *recursive* type

$$\mu\,(y\colon\ \{a\colon \mu\,(z\colon\ \{B\colon U..U\})\})\,.$$

Since v is no longer connected to any specific name, no recursion elimination is possible on its type. In particular, it does not make sense to give this value the type

$$\{a\colon \mu\,(z\colon\ \{B\colon U\,[x/y]\,..U\,[x/y]\})\}$$

because this type refers to x, but after the reduction, the value is no longer associated with this name.

The example illustrates that in DOT, variables represent the identity of objects. This is necessary in order to access an object's members: object members can reference the object itself, for which the object needs to have a name.

18.2.2 *Paths are Identifiers in pDOT*

In pDOT, *paths* represent the identity of objects and therefore they must be irreducible. Similarly to DOT, reducing paths would lead to unsoundness and strip nested objects of their identity. Making paths irreducible means that in pDOT, we cannot have an analog of DOT's field selection rule Proj_{DOT}.

$$\frac{\gamma(x) = \nu(x\colon T)\ldots\{a = t\}\ldots}{\gamma \mid x.a \longmapsto \gamma \mid t}\ (\textsc{Proj}_{DOT})$$

Consider the field selection $x.a$ from the previous example. What is its type? By recursion elimination,

$$\Gamma \vdash x.a\colon\ \{B\colon U\,[x.a/z]\,..U\,[x.a/z]\}$$

If pDOT had a path-reduction rule Proj analogous to DOT's Proj_{DOT}, then $x.a$ would reduce to $\nu(z)\{B = U\}$. However, that value does not have the type $\{B\colon U\,[x.a/z]\,..U\,[x.a/z]\}$; it only has the recursive type $\mu\,(z\colon\ \{B\colon U..U\})$.

$$\dfrac{\dfrac{\Gamma \vdash x.a\colon \mu\,(z\colon\ \{B\colon U..U\})}{\Gamma \vdash x.a\colon\ \{B\colon U\,[x.a/z]\,..U\,[x.a/z]\}}\ \textsc{Rec-E} \qquad \gamma\colon \Gamma \qquad \dfrac{\gamma(x) = \nu(y)\,\{a = \nu(z)\,\{B = U\}\}}{\gamma \mid x.a \longrightarrow \gamma \mid \nu(z)\,\{B = U\}}\ \textsc{Hypothetical Proj}}{\Gamma \vdash \nu(z)\,\{B = U\}\colon\ \{B\colon U\,[x.a/z]\,..U\,[x.a/z]\}}\ \textsc{preservation}$$

The reduction step from $x.a$ to $\nu(z)\{B = U\}$ caused the object to lose its name. Since the non-recursive type of the term depends on the name, the loss of the name also caused the term to lose its non-recursive type. This reduction step violates type preservation and type soundness.

18.2.3 *Well-Typed Paths Don't Go Wrong*

If pDOT programs can return paths without reducing them to values, could these paths be nonsensical? The type system ensures that they cannot. In particular, we ensure that if a path p has a type then p either identifies some value, and looking up p in the runtime configuration

terminates, or p is a path that cyclically aliases other paths. Additionally, the pDOT safety proof ensures that if a path has a function or object type, then it can be looked up to a value; if p can *only* be typed with a singleton type (or $\top$), then the lookup will loop.

see Chapter 21 for pDOT safety proof

18.3 PATH REPLACEMENT

We introduce a *path replacement* operation for types that contain paths which reference the same object. If a path q is assigned to a path p then q *aliases* p. In the tree example from Chapter 16, `t3` aliases `t2`, but `t1` does not alias `t2`, even though they identify syntactically equal objects.

```
val t1 =
 new ConcreteTree
val t2 =
 new ConcreteTree
val t3 = t2
```

If q is an alias of p we want to ensure that we can use q in the same way as p. For example, any term that has a type $T \to p.A$ should also have the type $T \to q.A$, and vice versa. In pDOT, we achieve this by introducing a subtyping relationship between *equivalent types*: if p and q are aliases, and a type T can be obtained from type U by *replacing* instances of p in U with q then T and U are equivalent. For example, $T \to q.A$ can be obtained from $T \to p.A$ by replacing p with q, and these types are therefore equivalent.

Section 19.2 defines the replacement operation precisely

18.4 SINGLETON TYPES

To keep track of path aliases in the type system we use *singleton types*.

Suppose that a pDOT program assigns the path q to p, and that a type T can be obtained from U by replacing an instance of p with q. How does the type system know that T and U are equivalent? We could try passing information about the whole program throughout the type checker. However, that would make reasoning about types depend on reasoning about values, which would make typechecking more complicated and less modular, as shown in Part I.

Instead, we ensure that the type system keeps track of path aliasing using singleton types, an existing Scala feature. A singleton type of a path p, denoted p.`type`, is a type that is inhabited *only* with the value that is represented by p. In the tree example from Chapter 16, to tell the type system that `t3` aliases `t2`, we ensure that `t3` has the singleton type `t2.`**`type`**. This information is used to allow subtyping between aliased paths, and to allow such paths to be typed with the same types.

Section 19.2 describes how singleton types track path identity

In pDOT, singleton types are an essential feature that is necessary to encode fully path-dependent types. However, this makes pDOT also the first DOT formalization of Scala's singleton types. In Chapter 20, we show a pDOT encoding of an example that motivates this Scala feature.

18.5 DISTINGUISHING FIELDS AND METHODS

Scala distinguishes between fields (vals, immutable fields that are strictly evaluated at the time of object initialization) and methods (defs, which are re-evaluated at each invocation). By contrast, DOT unifies the two in the concept of a term member. Since the distinction affects which paths are legal in Scala, we must make some similar distinction in pDOT. Consider the following Scala program:

```
val x = new {
    val a: { type A } = t_a
    def b: { type B } = t_b
}
val y: x.a.A
val z: x.b.B
```

Scala allows path-dependent types only on *stable* paths (Documentation, 2018b). A val can be a part of a stable path but a def cannot. Therefore, the type selection x.a.A is allowed but x.b.B is not.

DOT unifies the two concepts in one:

$$\textbf{let } x = \nu(x)\,\{a = t_a\} \wedge \{b = t_b\} \qquad \textbf{in} \ldots$$

However, this translation differs from Scala in the order of evaluation. Scala's fields, unlike DOT's, are fully evaluated to values when the object is constructed. Therefore, a more accurate translation of this example would be as follows:

$$\begin{array}{ll} \textbf{let } a' = t_a & \textbf{in} \\ \textbf{let } x = \nu(x)\,\{a = a'\} \wedge \{b = \lambda(\_).\, t_b\} & \textbf{in} \ldots \end{array}$$

This translation highlights the fact that although Scala can initialize x.a to an arbitrary term, that term will be already reduced to a value before evaluation reaches a context that contains x. The reason is that the constructor for x will strictly evaluate all of x's val fields when x is created.

To model the fact that Scala field initializers are fully evaluated when the object is constructed, we require field initializers in pDOT to be values or paths, rather than arbitrary terms. We use the name *stable term* for a value or path.

This raises the question of how to model a Scala method such as b. A method can still be represented by making the delayed evaluation of the body explicit: instead of initializing the field b with the method body itself, we delay the body inside a lambda abstraction. The lambda abstraction, a value, can be assigned to the field b. The body of the lambda abstraction can be an arbitrary term; it is not evaluated during object construction, but later when the method is called and the lambda is applied to some dummy argument.

18.6 PRECISE SELF TYPES

DOT allows powerful type abstractions, but it demands *objects* as proof that the type abstractions make sense. An object assigns actual types to its type members and thus provides concrete evidence for the declared subtyping relationships between abstract type members. To make the connection between the object value and the type system, DOT requires the self type in an object literal to precisely describe the concrete types assigned to type members, and we need to define similar requirements for self types in pDOT.

In the object

$$\nu(x\colon \{A\colon T..T\})\,\{A = T\}$$

DOT requires the self-type to be $\{A\colon T..T\}$ rather than some wider type $\{A\colon S..U\}$. This is not merely a convenience, but it is essential for soundness. Without the requirement, DOT could create and type the object

$$\nu(x\colon \{A\colon \top..\bot\})\,\{A = T\}$$

which introduces the subtyping relationship $\top <: \bot$ and thus makes every type a subtype of every other type. Although we can require the actual assigned type T to respect the bounds (i.e. $\top <: T <: \bot$), such a condition is not sufficient to prohibit this object. The assigned type T and the bounds ($\top$ and $\bot$ in this example) can in general depend on the self variable, and thus the condition makes sense only in a typing context that contains the self variable with its declared self type. But in such a context, we already have the assumption that $\top <: x.A <: \bot$, so it holds that $\top <: T$ (since $\top <: x.A <: \bot <: T$) and similarly $T <: \bot$.

In pDOT, a path-dependent type $p.A$ can refer to type members not only at the top level, but also deep inside the object. Accordingly, we need to extend the precise self type requirement to apply recursively within the object, as follows:

1. An object containing a type member definition $\{A = T\}$ must declare A with tight bounds, using $\{A\colon T..T\}$ in its self type.

2. An object containing a definition $\{a = \nu(x\colon T)d\}$ must declare a with the recursive type $\mu\,(x\colon T)$, using $\{a\colon \mu\,(x\colon T)\}$ in its self type.

3. An object containing a definition $\{a = \lambda(x\colon T)\,U\}$ must declare a with a function type, using $\{a\colon \forall(x\colon S)\,V\}$ in its self type.

4. An object containing a definition $\{a = p\}$ must declare a with the singleton type $p.\mathsf{type}$, using $\{a\colon p.\mathsf{type}\}$ in its self type.

The first requirement is the same as in DOT. The second and third requirements are needed for soundness of paths that select type members from deep within an object. The fourth requirement is needed to

prevent unsoundness in the case of cyclic references. For example, if we were to allow the object

$$\nu(x\colon \{a\colon \{A\colon \top..\bot\}\})\,\{a = x.a\}$$

we would again have $\top <: \bot$. The fourth requirement forces this object to be declared with a precise self type:

$$\nu(x\colon \{a\colon x.a.\mathsf{type}\})\,\{a = x.a\}$$

Now, $x.a$ no longer has the type $\{A\colon \top..\bot\}$, so it no longer collapses the subtyping relation. The precise typing thus ensures that cyclic paths can be only typed with singleton types but not function or object types, and therefore we cannot have type or term selection on cyclic paths.

Although both DOT and pDOT require precision in the self type of an object, the object itself can be typed with a wider type once it is assigned to a variable. For example, in DOT, if we have

$$\textbf{let } x = \nu(x\colon \{A\colon T..T\})\,\{A = T\} \textbf{ in } \ldots$$

then x also has the wider type $\{A\colon \bot..\top\}$. Similarly, in pDOT, if we have

$$\textbf{let } x = \nu(x\colon \{a\colon \mu\,(y\colon \{b\colon \forall(z\colon T)\,U\}) \wedge \{c\colon x.a.b.\mathsf{type}\}\})d \textbf{ in } \ldots$$

then x also has all of the following types:

$$\Gamma \vdash x\colon \{a\colon \{b\colon \forall(z\colon T)\,U\}\}$$
$$\Gamma \vdash x\colon \{a\colon \mu\,(y\colon \{b\colon \top\})\}$$
$$\Gamma \vdash x\colon \{c\colon x.a.b.\mathsf{type}\}$$
$$\Gamma \vdash x\colon \{c\colon \forall(z\colon T)\,U\}$$

In fact, the typings for this object in pDOT are more expressive than in DOT. Because DOT does not open types nested inside of field declarations, DOT cannot assign the first two types to x. In Section 19.2, we show one simple type rule that generalizes pDOT to open and abstract types of term members nested deeply inside an object. In Chapter 20, we encode several examples from previous DOT papers in pDOT and show that the real-world compiler example from Chapter 16 that uses types depending on long paths can be encoded in pDOT as well.

In summary, both DOT and pDOT require the self type in an object literal to precisely describe the values in the literal, but this does not limit the ability to ascribe a more abstract type to the paths that identify the object.

19 FROM DOT TO PDOT

The pDOT calculus generalizes DOT by allowing paths wherever DOT allows variables (except in places where variables are used as binders, such as x in $\lambda(x\colon T)\,t$).

19.1 SYNTAX

Figure 19.1 shows the abstract syntax of pDOT. The signature construct in pDOT is a path, defined to be a variable followed by zero or more field selections (e.g. $x.a.b.c$). pDOT uses paths wherever DOT uses variables. In particular, field selections $x.a$ and function application $x\,y$ are done on paths: $p.a$ and $p\,q$. Most importantly, pDOT also generalizes DOT's types by allowing path-dependent types $p.A$ on paths rather than just on variables. Additionally, as described in Section 18.4, the pDOT calculus formalizes Scala's singleton types. A singleton type p.type is inhabited with only one value: the value that is assigned to the path p. A singleton type thus indicates that a term designates the same object as the path p. Just as a path-dependent type $p.A$ depends on the value of p, a singleton type q.type depends on the value of q. Singleton types are therefore a second form of dependent types in the calculus.

In pDOT, paths and values are considered *stable terms*. As motivated in Section 18.5, to distinguish between fields and methods, term members of object definitions can only be assigned stable terms, while still allowing the same (and even more expressive) type abstractions as in DOT. We use the meta-variable s to denote stable terms.

$$
\begin{array}{rll}
p, q, r & & \text{path} \\
s & & \text{stable term} \\
\\
p, q, r & := & \\
 & & x \\
 & & p.a \\
t, u & := & \\
 & & s \\
 & & p\,q \\
 & & \textbf{let } x = t \textbf{ in } u \\
s & := & \\
 & & p \\
 & & v \\
v & := & \\
 & & \nu(x\colon T)d \\
 & & \lambda(x\colon T)\,t \\
d & := & \\
 & & \{a = s\} \\
 & & \{A = T\} \\
 & & d \wedge d' \\
S, T, U & := & \\
 & & \top \\
 & & \bot \\
 & & \{a\colon T\} \\
 & & \{A\colon S..T\} \\
 & & p.A \\
 & & p.\text{type} \\
 & & S \wedge T \\
 & & \mu\,(x\colon T) \\
 & & \forall(x\colon S)\,T
\end{array}
$$

Figure 19.1: Abstract syntax of pDOT (cf. DOT syntax in Figure 2.1)

19.2 PDOT TYPING RULES

The typing and subtyping rules of pDOT are shown in Figures 19.2 and 19.3.

19.2.1 *From Variables to Paths*

The first thing to notice in the pDOT typing and subtyping rules is that all variable-specific rules, except Var, are generalized to paths, as motivated in Section 18.1. The key rules that make DOT and pDOT interesting are the type-selection rules <:-Sel and Sel-<:. These rules enable us to make use of the type member in a path-dependent type. When a path p has type $\{A\colon S..U\}$, the rules introduce the path-dependent type $p.A$ into the subtyping relation by declaring the

Term typing
$\Gamma \vdash t : T$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{ (Var)} \qquad \frac{\Gamma \vdash p : q.\mathsf{type} \qquad \Gamma \vdash q : T}{\Gamma \vdash p : T} \text{ (Sngl-Trans)}$$

$$\frac{\Gamma, x : T \vdash t : U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash \lambda(x : T)\, t : \forall(x : T)\, U} \text{ (All-I)} \qquad \frac{\Gamma \vdash p : q.\mathsf{type} \qquad \Gamma \vdash q.a}{\Gamma \vdash p.a : q.a.\mathsf{type}} \text{ (Sngl-E)}$$

$$\frac{\Gamma \vdash p : \forall(z : S)\, T \qquad \Gamma \vdash q : S}{\Gamma \vdash p\, q : T\,[q/z]} \text{ (All-E)}$$

$$\frac{x;\, \Gamma, x : T \vdash d : T}{\Gamma \vdash \nu(x : T) d : \mu\,(x : T)} \text{ ({}-I)} \qquad \frac{\Gamma \vdash p : T\,[p/x]}{\Gamma \vdash p : \mu\,(x : T)} \text{ (Rec-I)}$$

$$\frac{\Gamma \vdash p : \{a : T\}}{\Gamma \vdash p.a : T} \text{ (Fld-E)} \qquad \frac{\Gamma \vdash p : \mu\,(x : T)}{\Gamma \vdash p : T\,[p/x]} \text{ (Rec-E)}$$

$$\frac{\Gamma \vdash p.a : T}{\Gamma \vdash p : \{a : T\}} \text{ (Fld-I)} \qquad \frac{\Gamma \vdash p : T \qquad \Gamma \vdash p : U}{\Gamma \vdash p : T \wedge U} \text{ (\&-I)}$$

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

Definition typing
$p;\, \Gamma \vdash d : T$

$$p;\, \Gamma \vdash \{A = T\} : \{A : T..T\} \text{ (Def-Typ)} \qquad \frac{\Gamma \vdash q}{p;\, \Gamma \vdash \{a = q\} : \{a : q.\mathsf{type}\}} \text{ (Def-Path)}$$

$$\frac{p;\, \Gamma \vdash \lambda(x : T)\, t : \forall(x : U)\, V}{p;\, \Gamma \vdash \{a = \lambda(x : T)\, t\} : \{a : \forall(x : U)\, V\}} \text{ (Def-All)}$$

$$\frac{p.a;\, \Gamma \vdash d\,[p.a/y] : T\,[p.a/y] \qquad \mathsf{tight}\, T}{p;\, \Gamma \vdash \{a = \nu(y : T) d\} : \{a : \mu\,(y : T)\}} \text{ (Def-New)}$$

$$\frac{p;\, \Gamma \vdash d_1 : T_1 \qquad p;\, \Gamma \vdash d_1 : T_2 \qquad \mathsf{dom}(d_1),\, \mathsf{dom}(d_2)\ \text{disjoint}}{p;\, \Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \text{ (AndDef-I)}$$

Typeable paths
$\Gamma \vdash p$

$$\frac{\Gamma \vdash p : T}{\Gamma \vdash p} \text{ (Wf)}$$

Tight bounds
$\mathsf{tight}\, T$

$$\mathsf{tight}\, T = \begin{cases} U = V & \text{if } T = \{A : U..V\} \\ \mathsf{tight}\, U & \text{if } T = \mu\,(x : U) \text{ or } T = \{a : U\} \\ \mathsf{tight}\, U \text{ and } \mathsf{tight}\, V & \text{if } T = U \wedge V \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Figure 19.2: pDOT typing rules (cf. DOT typing in Figure 2.3)

$$\Gamma \vdash T <: \top \quad \text{(Top)}$$

$$\Gamma \vdash \bot <: T \quad \text{(Bot)}$$

$$\Gamma \vdash T <: T \quad \text{(Refl)}$$

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

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

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

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

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

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

$$\frac{\Gamma \vdash p : \{A : S..T\}}{\Gamma \vdash S <: p.A} \quad \text{(<:-Sel)}$$

$$\frac{\Gamma \vdash p : q.\mathsf{type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: T\,[q/p]} \quad \text{(Sngl}_{pq}\text{-<:)}$$

$$\frac{\Gamma \vdash p : q.\mathsf{type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: T\,[p/q]} \quad \text{(Sngl}_{qp}\text{-<:)}$$

$$\frac{\Gamma \vdash p : \{A : S..T\}}{\Gamma \vdash p.A <: T} \quad \text{(Sel-<:)}$$

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

Figure 19.3: pDOT subtyping rules (cf. DOT subtyping in Figure 2.3)

subtyping constraints $S <: p.A$ and $p.A <: U$. Thanks to these two rules, pDOT supports fully-path-dependent types.

19.2.2 *Object Typing*

Similarly to the DOT calculus, the {}-I rule gives an object $\nu(x : T)d$ with declared type T which may depend on the self variable x the recursive type $\mu\,(x : T)$. The rule also checks that the definitions d of the object actually do have type T under the assumption that the self variable has this type. The object's definitions d are checked by the Definition typing rules. As discussed in Section 18.6, the rules assign a precise self type for objects, ensuring that paths are declared with singleton types, functions with function types, and objects with object types. For objects, the $\mathsf{tight}\ T$ condition ensures that all type members that can be reached by traversing T's fields have equal bounds, while still allowing arbitrary bounds in function types.[5]

[5] *see Section 21.1 for details*

A difference with DOT is that pDOT's definition-typing judgment keeps track of the path that represents an object's identity. When we typecheck an outermost object that is not nested in any other object, we use the {}-I rule. The rule introduces x as the identity for the object and registers this fact in the definition-typing judgment. To typecheck an object that is assigned to a field a of another object p we use the Def-New rule. This rule typechecks the object's body assuming the object identity $p.a$ and replaces the self-variable of the object with that path. The definition-typing judgment keeps track of the path to the

definition's enclosing object starting from the root of the program. This way the type system knows what identities to assign to nested objects. For example, when typechecking the object assigned to $x.a$ in the expression

$$\textbf{let } x = \nu(x)\,\{a = \nu(y)\,\{b = y.b\}\} \textbf{ in } \ldots$$

we need to replace y with the path $x.a$:

$$\dfrac{\dfrac{\dfrac{\Gamma,\, x\colon \{a\colon \mu\,(y\colon \{b\colon y.b.\mathsf{type}\})\} \vdash x.a}{x.a;\, \Gamma,\, x\colon \{a\colon \mu\,(y\colon \{b\colon y.b.\mathsf{type}\})\} \vdash \{b = x.a.b\} : \{b\colon x.a.b.\mathsf{type}\}}\ \textsc{Def-Path} \quad \mathsf{tight}\ \{b\colon y.b.\mathsf{type}\}}{x;\, \Gamma,\, x\colon \{a\colon \mu\,(y\colon \{b\colon y.b.\mathsf{type}\})\} \vdash \{a = \nu(y)\,\{b = y.b\}\} : \{a\colon \mu\,(y\colon \{b\colon y.b.\mathsf{type}\})\}}\ \textsc{Def-New}}{\Gamma \vdash \nu(x)\,\{a = \nu(y)\,\{b = y.b\}\} : \mu\,(x\colon \{a\colon \mu\,(y\colon \{b\colon y.b.\mathsf{type}\})\})}\ \{\}\text{-I}$$

An alternative design of the Def-New rule can be to introduce a fresh variable y into the context (similarly to the {}-I rule). However, we would have to assign y the type $x.a.\mathsf{type}$ to register the fact that these two paths identify the same object. We decided to simplify the rule by immediately replacing the nested object's self variable with the outer path to avoid the indirection of an additional singleton type.

19.2.3 *Path Alias Typing*

In pDOT, singleton-type related typing and subtyping rules are responsible for the handling of aliased paths and equivalent types.

SINGLETON TYPE CREATION How does a path p obtain a singleton type? A singleton type indicates that in the initial program, a prefix of p (which could be all of p) is assigned a path q. For example, in the program

$$\textbf{let } x = \nu(x\colon \{a\colon x.\mathsf{type}\} \wedge \{b\colon S\})\,\{a = x\} \wedge \{b = s\} \textbf{ in } \ldots$$

the path $x.a$ should have the type $x.\mathsf{type}$ because $x.a$ is assigned the path x. The singleton type for $x.a$ can be obtained as follows. Suppose that in the typing context of the **let** body, x is mapped to the type of its object, $\mu\,(x\colon \{a\colon x.\mathsf{type}\} \wedge \{b\colon S\})$. Through applying recursion elimination (Rec-E), field selection (Fld-E), and finally subsumption (Sub) with the intersection subtyping rule $\textsc{And}_1$-<:, we will obtain that $\Gamma \vdash x.a\colon x.\mathsf{type}$.

In the above example, $x.a$ aliases x, so anything that we can do with x we should be able to do with $x.a$. Since x has a field b and we can create a path $x.b$, we want to also be able to create a path $x.a.b$. Moreover, we want to treat $x.a.b$ as an alias for $x.b$. This is done through the Sngl-E rule: it says that if p aliases q, and $q.a$ has a type (denoted with $\Gamma \vdash q.a$), then $p.a$ aliases $q.a$. This rule allows us to conclude that $\Gamma \vdash x.a.b\colon x.b.\mathsf{type}$.

$$\dfrac{\Gamma \vdash q.a \qquad \Gamma \vdash p\colon q.\mathsf{type}}{\Gamma \vdash p.a\colon q.a.\mathsf{type}}\ (\textsc{Sngl-E})$$

SINGLETON TYPE PROPAGATION In the above example we established that the path $x.a.b$ is an alias for $x.b$. Therefore, we want to be able to type $x.a.b$ with any type with which we can type $x.b$. The SNGL-TRANS rule allows us to do just that: if p is an alias for q, then we can type p with any type with which we can type q. Using that rule, we can establish that $\Gamma \vdash x.a.b : S$ because $\Gamma \vdash x.b : S$.

$$\frac{\Gamma \vdash q : T \qquad \Gamma \vdash p : q.\mathsf{type}}{\Gamma \vdash p : T}\ (\text{SNGL-TRANS})$$

EQUIVALENT TYPES As described in Section 18.3, we call two types equivalent if they are equal up to path aliases. We need to ensure that equivalent types are equivalent by subtyping, i.e. that they are subtypes of each other. For example, suppose that $\Gamma \vdash p : q.\mathsf{type}$, and the path r refers to an object $\nu(x)\,\{a = p\} \wedge \{b = p\}$. Then we want to be able to type r with all of the following types:

$$\Gamma \vdash r : \{a : p.\mathsf{type}\} \wedge \{b : p.\mathsf{type}\} \qquad \Gamma \vdash r : \{a : p.\mathsf{type}\} \wedge \{b : q.\mathsf{type}\}$$
$$\Gamma \vdash r : \{a : q.\mathsf{type}\} \wedge \{b : q.\mathsf{type}\} \qquad \Gamma \vdash r : \{a : q.\mathsf{type}\} \wedge \{b : p.\mathsf{type}\}$$

The pDOT subtyping rules SNGL_{pq}-$<:$ and SNGL_{qp}-$<:$ allow us to assign these types to r by establishing subtyping between equivalent types. Specifically, if we know that $\Gamma \vdash p : q.\mathsf{type}$ then the rules allow us to replace any occurrence of p in a type T with q, and vice versa, while maintaining subtyping relationships in both directions.

$$\frac{\Gamma \vdash p : q.\mathsf{type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: T\,[q/p]}\ (\text{SNGL}_{pq}\text{-}<:)$$

$$\frac{\Gamma \vdash p : q.\mathsf{type} \qquad \Gamma \vdash q}{\Gamma \vdash T <: T\,[p/q]}\ (\text{SNGL}_{qp}\text{-}<:)$$

We express that two types are equivalent using the *replacement operation*. The operation is similar to the substitution operation, except that we replace paths with paths instead of variables with terms, and we replace only one path at a time rather than all of its occurrences. The statement $T\,[q/p] = U$ denotes that the type T contains one or more paths that start with p, e.g. $p.\overline{b_1}, \ldots, p.\overline{b_n}$, and that exactly one of these occurrences $p.\overline{b_i}$ is replaced with $q.\overline{b_i}$, yielding the type U. Note that it is not specified exactly in which occurrence of the above paths the prefix p is replaced with q. The precise definition of the replacement operation is presented in Figure 19.4.

Given the path r from the above example, we can choose whether to replace the first or second occurrence of p with q; for example, we can derive

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash r : \{a : p.\mathsf{type}\} \wedge \{b : p.\mathsf{type}\}}\ \text{REC-E} \qquad \dfrac{\Gamma \vdash p : q.\mathsf{type} \qquad \dfrac{\ldots}{\{a : p.\mathsf{type}\} \wedge \{b : p.\mathsf{type}\}\,[q/p] = \{a : p.\mathsf{type}\} \wedge \{b : q.\mathsf{type}\}}\ \text{REPL-AND}_2}{\Gamma \vdash \{a : p.\mathsf{type}\} \wedge \{b : p.\mathsf{type}\} <: \{a : p.\mathsf{type}\} \wedge \{b : q.\mathsf{type}\}}\ \text{SNGL}_{pq}\text{-}<:}{\Gamma \vdash r : \{a : p.\mathsf{type}\} \wedge \{b : q.\mathsf{type}\}}\ \text{SUB}$$

To replace several occurrences of a path with another, we repeatedly apply SNGL_{pq}-$<:$ or SNGL_{qp}-$<:$.

$$p.\overline{b}.A\,[q/p] = q.\overline{b}.A \quad \text{(Repl-Path)}$$

$$p.\overline{b}.\mathsf{type}\,[q/p] = q.\overline{b}.\mathsf{type} \quad \text{(Repl-Sngl)}$$

$$\frac{T_1\,[q/p] = T_2}{(T_1 \wedge U)\,[q/p] = T_2 \wedge U} \quad \text{(Repl-And}_1\text{)}$$

$$\frac{T_1\,[q/p] = T_2}{(U \wedge T_1)\,[q/p] = U \wedge T_2} \quad \text{(Repl-And}_2\text{)}$$

$$\frac{T\,[q/p] = U}{\mu\,(x\colon T)\,[q/p] = \mu\,(x\colon U)} \quad \text{(Repl-Rec)}$$

$$\frac{T_1\,[q/p] = T_2}{(\forall(x\colon T_1)\,U)\,[q/p] = \forall(x\colon T_2)\,U} \quad \text{(Repl-All}_1\text{)}$$

$$\frac{T_1\,[q/p] = T_2}{(\forall(x\colon U)\,T_1)\,[q/p] = \forall(x\colon U)\,T_2} \quad \text{(Repl-All}_2\text{)}$$

$$\frac{T_1\,[q/p] = T_2}{\{a\colon T_1\}\,[q/p] = \{a\colon T_2\}} \quad \text{(Repl-Fld)}$$

$$\frac{T_1\,[q/p] = T_2}{\{A\colon T_1..U\}\,[q/p] = \{A\colon T_2..U\}} \quad \text{(Repl-Typ}_1\text{)}$$

$$\frac{T_1\,[q/p] = T_2}{\{A\colon U..T_1\}\,[q/p] = \{A\colon U..T_2\}} \quad \text{(Repl-Typ}_2\text{)}$$

Figure 19.4: Replacement of a path p in a type by q

19.2.4 *Abstracting Over Field Types*

Finally, we describe one of the most interesting pDOT rules which adds significant expressivity to pDOT.

Consider a function

$$f = \lambda(x\colon \{a\colon T\})\ \ldots$$

and a path p that refers to the object

$$\nu(x\colon \{a\colon q.\mathsf{type}\})\ \{a = q\}$$

where $\Gamma \vdash q\colon T$. Since

$$\Gamma \vdash p\colon \mu\,(x\colon \{a\colon q.\mathsf{type}\})$$

by Rec-E, we have

$$\Gamma \vdash p\colon \{a\colon q.\mathsf{type}\}$$

we assume for simplicity that q does not start with x

Therefore, since $\Gamma \vdash q\colon T$, we would like to be able to pass p into the function f which expects an argument of type $\{a\colon T\}$. Unfortunately, the typing rules so far do not allow us to do that because although q has type T, $q.\mathsf{type}$ is *not* a subtype of T, and therefore $\{a\colon q.\mathsf{type}\}$ is not a subtype of $\{a\colon T\}$.

$$\frac{\Gamma \vdash p.a\colon T}{\Gamma \vdash p\colon \{a\colon T\}} \quad \text{(Fld-I)}$$

The type rule Fld-I allows us to bypass that limitation. If a path p has a record type $\{a\colon T\}$ (and therefore $\Gamma \vdash p.a\colon T$), then the rule lets

us type p with any type $\{a\colon U\}$ as long as $p.a$ can be typed with U. More generally, using this typing rule, we can derive

$$\frac{\Gamma \vdash p.a_1.a_2.\cdots.a_n\colon U}{\Gamma \vdash p\colon \{a_1\colon \{a_2\colon \ldots\{a_n\colon U\}\}\}}$$

For the above example, we can prove that $\Gamma \vdash p\colon \{a\colon T\}$ and pass it into f as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash p\colon \{a\colon q.\mathsf{type}\}}{\Gamma \vdash p.a\colon q.\mathsf{type}}\,\textsc{Fld-E} \qquad \Gamma \vdash q\colon T}{\Gamma \vdash p.a\colon T}\,\textsc{Sngl-Trans}}{\Gamma \vdash p\colon \{a\colon T\}}\,\textsc{Fld-I}$$

The Fld-I rule allows us to eliminate recursion on types that are nested inside fields, which is not possible in DOT. If a DOT function f expects a parameter of type $\{a\colon \mu\,(x\colon T)\}$, then in DOT, we cannot pass a variable y of type $\{a\colon \mu\,(x\colon T \wedge U)\}$ or a variable z of type $\{a\colon T\,[z.a/x]\}$ into f because there is no subtyping between recursive types, and there is no subtyping relationship between $\mu\,(x\colon T)$ and $T\,[z.a/x]$ (and the latter type might not exist in the first place due to the lack of fully-path-dependent types). All of the above is possible in pDOT because both $y.a$ and $z.a$ can be typed with $\mu\,(x\colon T)$, which allows us to use the Fld-I rule and type y and z as $\{a\colon \mu\,(x\colon T)\}$.

19.3 REDUCTION SEMANTICS

The operational semantics of pDOT is presented in Figure 19.5. pDOT's reduction rules mirror the DOT rules with three distinctions:

- *paths everywhere*: wherever DOT uses (as opposed to defines) variables, pDOT uses paths;
- *no* $\textsc{Proj}_{\text{DOT}}$: there is no reduction rule for field projection because in pDOT, paths are normal form (as motivated in Section 18.2.2);
- *path lookup*: pDOT uses the reflexive, transitive closure of the *path lookup* operation $\rightsquigarrow$ that generalizes variable lookup in stores to paths.

The path lookup operation is presented in Figure 19.6. This operation allows us to look up a value that is nested deeply inside an object. If a path is a variable the lookup operation is a straightforward variable lookup (Lookup-Step-Var). If in a store γ, a path p is assigned an object $v(x)\,\{a = s\}$ then $\gamma \vdash p.a \rightsquigarrow s\,[p/x]$ because the self variable x in s gets replaced with p (Lookup-Step-Val). If p is equal to another path q then $\gamma \vdash p.a \rightsquigarrow q.a$ (Lookup-Step-Path).

Finally, we want to be able to follow a sequence of paths in a store: for example, if $\gamma \vdash p \rightsquigarrow q$ and $\gamma \vdash q \rightsquigarrow v$, we want to conclude that

$$\frac{\gamma \vdash p \rightsquigarrow^* \lambda(z\colon T)\, t}{\gamma \mid p\, q \longmapsto \gamma \mid t\,[q/z]} \quad \text{(Apply)}$$

$$\gamma \mid \textbf{let } x = p \textbf{ in } t \longmapsto \gamma \mid t\,[p/x] \quad \text{(Let-Path)}$$

$$\gamma \mid \textbf{let } x = v \textbf{ in } t \longmapsto \gamma,\, x \mapsto v \mid t \quad \text{(Let-Value)}$$

$$\frac{\gamma \mid t \longmapsto \gamma' \mid t'}{\gamma \mid \textbf{let } x = t \textbf{ in } u \longmapsto \gamma' \mid \textbf{let } x = t' \textbf{ in } u} \quad \text{(Ctx)}$$

Figure 19.5: Operational semantics of pDOT

$$\frac{\gamma(x) = v}{\gamma \vdash x \rightsquigarrow v} \quad \text{(Lookup-Step-Var)}$$

$$\frac{\gamma \vdash p \rightsquigarrow \nu(x\colon T) \ldots \{a = s\} \ldots}{\gamma \vdash p.a \rightsquigarrow s\,[p/x]} \quad \text{(Lookup-Step-Val)}$$

$$\frac{\gamma \vdash p \rightsquigarrow q}{\gamma \vdash p.a \rightsquigarrow q.a} \quad \text{(Lookup-Step-Path)}$$

$$\gamma \vdash s \rightsquigarrow^* s \quad \text{(Lookup-Refl)}$$

$$\frac{\gamma \vdash s_1 \rightsquigarrow s_2 \qquad \gamma \vdash s_2 \rightsquigarrow^* s_3}{\gamma \vdash s_1 \rightsquigarrow^* s_3} \quad \text{(Lookup-Trans)}$$

Figure 19.6: Value-environment path lookup

looking up p yields v. This is done through the reflexive, transitive closure $\rightsquigarrow^*$ of the $\rightsquigarrow$ relation (LOOKUP-REFL and LOOKUP-TRANS).

For example, looking up $x.a.c$ in the environment

$$\begin{aligned}\gamma = &(y, \nu(y')\{b = \nu(y'')\{c = \lambda(z\colon \top)\, z\}\}),\\ &(x, \nu(x)\{a = y.b\})\end{aligned}$$

yields $\lambda(z\colon \top)\, z$:

$$\begin{array}{llll}
\gamma(x) = \nu(x)\{a = y.b\} & & \gamma(y) = \nu(y')\{b = \nu(y'')\{c = \lambda(z\colon \top)\, z\}\} & \\
\gamma \vdash x \rightsquigarrow \nu(x)\{a = y.b\} & & \gamma \vdash y \rightsquigarrow \nu(y')\{b = \nu(y'')\{c = \lambda(z\colon \top)\, z\}\} & \\
\gamma \vdash x.a \rightsquigarrow y.b & & \gamma \vdash y.b \rightsquigarrow \nu(y'')\{c = \lambda(z\colon \top)\, z\} & \\
\gamma \vdash x.a.c \rightsquigarrow y.b.c & & \gamma \vdash y.b.c \rightsquigarrow \lambda(z\colon \top)\, z &
\end{array}$$

$$\gamma \vdash x.a.c \rightsquigarrow^* \lambda(z\colon \top)\, z$$

The reduction rule that uses the lookup operation is the function application rule APPLY: to apply p to q we must be able to look up p in the store and obtain a function. Since pDOT permits cycles in paths, does this mean that the lookup operation for this type rule might not terminate? Fortunately, pDOT's type safety ensures that this will not happen. As shown in Section 21.4, if $\Gamma \vdash p\colon \forall(T\colon U)$ then lookup of p eventually terminates and results in a function value. Therefore, a function application $p\, q$ always makes progress.

20 EXAMPLES

In this chapter, we present three pDOT program examples that illustrate different features of the calculus. All of the programs were formalized and typechecked in Coq.

To make the examples easier to read, we simplify the notation for objects $\nu(x\colon U)d$ by removing type annotations where they can be easily inferred, yielding a new notation $\nu(x \Rightarrow d')$ where d' are the definitions d modified as follows:

- a type definition $\{A = T\}$ can be only typed with $\{A\colon T..T\}$, so we will skip type declarations;
- in a definition $\{a = p\}$, the field a is assigned a path and can be only typed with a singleton type; we will therefore skip the type $\{a\colon p.\mathsf{type}\}$;
- in a definition $\{a = \nu(x\colon T)d\}$, a is assigned an object that must be typed with $\mu\,(x\colon T)$; since we can infer T by looking at the object definition, we will skip the typing $\{a\colon \mu\,(x\colon T)\}$;
- we inline the type of abstractions into the field definition (e.g. $\{a\colon \forall(T\colon U)\ = \lambda(x\colon T)\,t\}$).

For readability we will also remove the curly braces around object definitions and replace the $\wedge$ delimiters with semicolons. As an example of our abbreviations, the object

$$\begin{aligned}\nu(x\colon\ &\{A\colon T..T\} \wedge \{a\colon p.\mathsf{type}\} \wedge \{b\colon \mu\,(y\colon U)\} \wedge \{c\colon \forall(z\colon S)\,V\}\\ &\{A = T\} \wedge \{a = p\} \wedge \{b = \nu(y\colon U)d\} \wedge \{c = \lambda(z\colon S')\,t\})\end{aligned}$$

will be encoded as

$$\nu(x \Rightarrow\ A = T;\ a = p;\ b = \nu(y \Rightarrow d');c\colon \forall(z\colon S)\,V = \lambda(z\colon S')\,t)$$

Note that translating the nested object $\nu(y\colon U)d$ yielded a new notation $\nu(y \Rightarrow d')$ where d' stands for the new encoding of d.

20.1 CLASS ENCODINGS

Fully path-dependent types allow pDOT to define encodings for Scala's module system and classes, as we will see in the examples below.

In Scala, declaring a class A(*args*) automatically defines both a type A for the class and a constructor for A with parameters *args*. We will

encode such a Scala class in pDOT as a type member A and a method newA that returns an object of type A:

$$
\begin{array}{l}
\nu(p \Rightarrow \\
\quad A = \mu\,(\textbf{this}\colon \{\mathsf{foo}\colon \forall(\_)\,U\})\,; \\
\quad \mathsf{newA}\colon \forall(x\colon U)\,p.A \\
\qquad = \nu(\textbf{this})\,\{\mathsf{foo} = \lambda(\_).\,x\})
\end{array}
\qquad
\begin{array}{l}
\textbf{package}\ \mathsf{p}\ \{ \\
\quad \textbf{class}\ \mathsf{A}(\mathsf{x}\colon \mathsf{U})\ \{ \\
\quad\quad \textbf{def}\ \mathsf{foo}\colon \mathsf{U} = \mathsf{x} \\
\quad \} \\
\}
\end{array}
$$

To encode subtyping we use type intersection. For example, we can define a class B that extends A as follows:

$$
\begin{array}{l}
\nu(p \Rightarrow \\
\quad B = p.A \wedge \{C\colon \bot..\top\}\,; \\
\quad \mathsf{newB}\colon \forall(x\colon U)\,p.B \\
\qquad\quad = \nu(\textbf{this})\{\mathsf{foo} = \lambda(\_).\,x; \\
\qquad\qquad\quad C = \ldots\})
\end{array}
\qquad
\begin{array}{l}
\textbf{package}\ \mathsf{p}\ \{ \\
\quad \textbf{class}\ \mathsf{B}(\mathsf{x}{:}\mathsf{U})\ \textbf{extends}\ \mathsf{A}(\mathsf{x}{:}\mathsf{U})\ \{ \\
\\
\quad\quad \textbf{type}\ \mathsf{C} \\
\quad \} \\
\}
\end{array}
$$

20.2 LISTS

As an example to illustrate that pDOT supports the type abstractions of DOT we formalize the covariant-list library by Amin, Grütter, et al., (2016) in pDOT, presented in Figure 20.1. The encoding defines List as a data type with an element type A and methods head and tail. The library contains nil and cons fields for creating lists. To soundly formalize the list example, we encode head and tail as *methods* (defs) as opposed to vals by wrapping them in lambda abstractions, as discussed in Section 18.5. This encoding also corresponds to the Scala standard library where head and tail are defs and not vals, and hence one cannot perform a type selection on them.

By contrast, the list example by Amin, Grütter, et al., (2016) encodes head and tail as fields without wrapping their results in functions. For a DOT that supports paths, such an encoding is unsound because it violates the property that paths to objects with type members are acyclic. In particular, since no methods should be invoked on nil, its head and tail methods are defined as non-terminating loops, and nil's element type is instantiated to $\bot$. If we allowed nil.head to have type $\bot$ then since $\bot <: \{A: \top..\bot\}$, we could derive $\top <:$ nil.head.A $<: \bot$.

$$
\begin{aligned}
&\nu(\textbf{sci} \Rightarrow \mathsf{List} = \mu(\textbf{self}: \{\mathsf{A}: \bot..\top\} \wedge \\
&\qquad\qquad \{\mathsf{head}: \forall(\_)\ \textbf{self}.\mathsf{A}\} \wedge \\
&\qquad\qquad \{\mathsf{tail}: \forall(\_)\ (\textbf{sci}.\mathsf{List} \wedge \{\mathsf{A}: \bot..\textbf{self}.\mathsf{A}\})\});\\
&\quad \mathsf{nil}: \forall(x: \{\mathsf{A}: \bot..\top\})\ \textbf{sci}.\mathsf{List} \wedge \{\mathsf{A}: \bot..\bot\} \\
&\qquad = \lambda(x: \{\mathsf{A}: \bot..\top\}) \\
&\qquad\quad \textbf{let}\ \mathsf{result} = \nu(\textbf{self} \Rightarrow \mathsf{A} = \bot; \\
&\qquad\qquad\qquad\qquad \mathsf{head}: \forall(y: \top)\ \textbf{self}.\mathsf{A} = \lambda(y: \top)\ \textbf{self}.\mathsf{head}\, y; \\
&\qquad\qquad\qquad\qquad \mathsf{tail}: \forall(y: \top)\ (\textbf{sci}.\mathsf{List} \wedge \textbf{self}.\mathsf{A}) = \lambda(y: \top)\ \textbf{self}.\mathsf{tail}\, y) \\
&\qquad\quad \textbf{in}\ \mathsf{result}; \\
&\quad \mathsf{cons}: \forall(x: \{\mathsf{A}: \bot..\top\})\ \forall(\mathsf{hd}: x.\mathsf{A})\ \forall(\mathsf{tl}: \textbf{sci}.\mathsf{List} \wedge \{\mathsf{A}: \bot..x.\mathsf{A}\})\ \textbf{sci}.\mathsf{List} \wedge \{\mathsf{A}: \bot..x.\mathsf{A}\} \\
&\qquad = \lambda(x: \{\mathsf{A}: \bot..\top\})\ \lambda(\mathsf{hd}: x.\mathsf{A})\ \lambda(\mathsf{tl}: \textbf{sci}.\mathsf{List} \wedge \{\mathsf{A}: \bot..x.\mathsf{A}\}) \\
&\qquad\quad \textbf{let}\ \mathsf{result} = \nu(\textbf{self} \Rightarrow \mathsf{A} = x.\mathsf{A}; \\
&\qquad\qquad\qquad\qquad \mathsf{head}: \forall(\_)\ \textbf{self}.\mathsf{A} = \lambda\_.\, \mathsf{hd} \\
&\qquad\qquad\qquad\qquad \mathsf{tail}: \forall(\_)\ (\textbf{sci}.\mathsf{List} \wedge \textbf{self}.\mathsf{A}) = \lambda\_.\, \mathsf{tl}) \\
&\qquad\quad \textbf{in}\ \mathsf{result})
\end{aligned}
$$

Figure 20.1: A covariant list library in pDOT

20.3 MUTUALLY RECURSIVE MODULES

The second example, presented in Figure 20.2, illustrates pDOT's ability to use path-dependent types of arbitrary length. It formalizes the compiler example from Chapter 16 in which the nested classes Type and Symbol recursively reference each other.

$$
\begin{aligned}
\nu(\mathbf{dc} \Rightarrow \mathsf{types} = \nu(\mathbf{types} \Rightarrow \mathsf{Type} = \mu\,(\mathbf{this}\colon \{\mathsf{symb}\colon \mathbf{dc}.\mathsf{symbols.Symbol}\})\,;\\
\mathsf{newType}\colon \forall(s\colon \mathbf{dc}.\mathsf{symbols.Symbol})\,\mathbf{types}.\mathsf{Type}\\
= \lambda(s\colon \mathbf{dc}.\mathsf{symbols.Symbol})\\
\mathbf{let}\ \mathsf{result'} = \nu(\mathbf{this} \Rightarrow \mathsf{symb} = s)\ \mathbf{in}\ \mathsf{result'});\\
\mathsf{symbols} = \nu(\mathbf{symbols} \Rightarrow \mathsf{Symbol} = \mu\,(\mathbf{this}\colon \{\mathsf{tpe}\colon \mathbf{dc}.\mathsf{types.Type}\})\,;\\
\mathsf{newSymbol}\colon \forall(t\colon \mathbf{dc}.\mathsf{types.Type})\,\mathbf{symbols}.\mathsf{Symbol}\\
= \lambda(t\colon \mathbf{dc}.\mathsf{types.Type})\\
\mathbf{let}\ \mathsf{result'} = \nu(\mathbf{this} \Rightarrow \mathsf{tpe} = t)\ \mathbf{in}\ \mathsf{result'}))
\end{aligned}
$$

Figure 20.2: Mutually recursive types in a compiler package: fully-path-dependent types

20.4 CHAINING METHODS WITH SINGLETON TYPES

The last example focuses on pDOT's ability to use singleton types as they are motivated by Odersky and Zenger, (2005b). An example from that paper introduces a class C with an incr method that increments a mutable integer field x and returns the object itself (this). A class D extends C and defines an analogous decr method. The example shows how we can invoke a chain of incr and decr methods on an object of type D using singleton types: if C.incr returned an object of type C this would be impossible since C does not have a decr member, so the method's return type is this.type, a singleton type.

Our formalization of the example is displayed in Figure 20.3. Since pDOT does not support mutation, our example excludes the mutation side effect of the original example which is there to make the example more practical.

$$
\begin{array}{l}
\textbf{let}\ \mathsf{pkg} = \nu(p \Rightarrow\ C = \mu\,(\textbf{this}\colon\ \{\mathsf{incr}\colon \textbf{this}.\mathsf{type}\})\,; \\
\qquad\qquad\qquad\quad D = \mu\,(\textbf{this}\colon\ p.C \wedge \{\mathsf{decr}\colon \textbf{this}.\mathsf{type}\})\,; \\
\qquad\qquad\qquad\quad \mathsf{newD}\colon \forall(\_)\ p.D = \lambda\,\_. \\
\qquad\qquad\qquad\qquad\quad \textbf{let}\ \mathsf{result} = \nu(\textbf{this} \Rightarrow \mathsf{incr} = \textbf{this}; \mathsf{decr} = \textbf{this}) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathsf{result}) \\
\textbf{in let}\ d = \mathsf{pkg.newD}\ \_ \\
\textbf{in}\ d.\mathsf{incr.decr}
\end{array}
$$

Figure 20.3: Chaining method calls using singleton types

21 TYPE SAFETY

We implemented the type-safety proof of pDOT in Coq as an extension of the simple DOT soundness proof by Rapoport, Kabir, et al., (2017). Compared to the 2,051 LOC, 124 lemmas and theorems, and 65 inductive or function definitions in the simple DOT proof, the pDOT Coq formalization consists of 7,343 LOC, 429 lemmas and theorems, and 115 inductive or function definitions. This section presents an overview of the key challenges and insights of proving pDOT sound.

The challenges of adapting the DOT soundness proof to pDOT can be classified into three main themes:

- adapting the notion of inert types to pDOT,
- adapting the stratification of typing rules to pDOT, and
- adapting the canonical forms lemma to changes in the operational semantics in pDOT.

21.1 INERT TYPES IN PDOT

The purpose of inertness is to prevent the introduction of a possibly undesirable subtyping relationship between arbitrary types $S <: U$ arising from the existence of a type member that has those types as bounds. If a variable x has type $\{A : S..U\}$, then $S <: x.A$ and $x.A <: U$, so by transitivity, $S <: U$.

As presented in Section 5.2, a DOT type is *inert* if it is a function type or a recursive type $\mu\,(x : T)$ where T is a *record type*. A record type is either a type-member declaration with equal bounds $\{A : U..U\}$ or an arbitrary field declaration $\{a : S\}$. In DOT, this is sufficient to rule out the introduction of new subtyping relationships.

In pDOT, a new subtyping relationship $S <: U$ arises when a *path* p, rather than only a variable x, has a type member $\{A : S..U\}$. Therefore, the inertness condition needs to enforce equal bounds on type members not only at the top level of an object, but recursively in any objects nested deeply within the outermost object. Therefore, a field declaration $\{a : T\}$ is inert only if the field type T is also inert. Moreover, since pDOT adds singleton types to DOT, the definition of a record type is also extended to allow a field to have a singleton type.

$$\frac{\text{inert } T}{\text{record } \{a : T\}}$$

$$\text{record } \{a : p.\mathsf{type}\}$$

$$\text{record } \{A : U..U\}$$

$$\frac{\text{record } T \qquad \text{record } U}{\text{record } T \wedge U}$$

$$\text{inert } \forall(x : T)\,U$$

$$\frac{\text{record } T}{\text{inert } \mu\,(x : T)}$$

(cf. Definition 2)

DEFINITION 34 (Record Types in pDOT). *A* record type *is an intersection of types each of which is either a field declaration* $\{a : T\}$ *where* T *is inert or a singleton type, or a tight type declaration* $\{A : U..U\}$.

DEFINITION 35 (Inert Types in pDOT). *A* type U *is* inert *if it is either a function type or a recursive type* $\mu\,(x\colon T)$ *where* T *is a record type.*

Both the DOT and pDOT preservation lemmas must ensure that reduction preserves inertness of typing contexts.

ASIDE: WHY WE NEED THE tight JUDGMENT This section motivates the need for the tight judgment that occurs in the DEF-NEW definition-typing rule (Figure 19.2). The reader may skip this section as it explains the design of the type rules and not the pDOT proof.

In designing pDOT and its safety proof, we must ensure that reduction preserves inertness: that is, when $\gamma \mid t \longmapsto \gamma' \mid t'$ there is an inert typing environment Γ' that is well-formed with respect to γ' and in which t' has the required type (i. e. we need to be able to prove the analogue of Lemma 20 (Value Typing)). In order to guarantee this, we need to ensure that *all well-typed values can be typed with an inert type*. As we will show now, if we omit the tight judgment from the object-definition typing rule DEF-NEW, there will be well-typed values that do not have a *precise* inert type. The values might still have an inert type under general typing but relying on such reasoning would further complicate the soundness proof.

$$\frac{\mathsf{tight}\ T \qquad p.a; \Gamma \vdash d\,[p.a/y]\colon T\,[p.a/y]}{p; \Gamma \vdash \{a = \nu(y\colon T)d\}\colon \{a\colon \mu\,(y\colon T)\}}\ \text{(DEF-NEW)}$$

The object-definition typing rule DEF-NEW requires that the type T of an object $\nu(x\colon T)d$ be tight, i. e. that all type declarations of the type (except those nested inside function types) have equal bounds. Together with the pDOT definition-typing rules, this ensures that T has the form of a concatenation of record types, and, due to tight, that each type declaration $\{A\colon T..U\}$ of that concatenation has equal bounds $(T = U)$.

Given that every record that declares a type member must be typed with the DEF-TYP rule, which requires tight bounds anyway, why do we need this additional tightness restriction?

If we do not require the tightness restriction in object-definition typing then there will exist well-typed objects $\nu(x\colon T)d$ whose precise recursive types $\mu\,(x\colon T)$ are not inert, which would complicate the soundness proof. Consider the following value:

$$\begin{aligned}\nu(x\colon \{a\colon \mu\,(y\colon\ \{A\colon y.\mathsf{type}..x.a.\mathsf{type}\})\}&\\ \{a = \nu(y\colon \{A\colon y.\mathsf{type}..x.a.\mathsf{type}\})&\\ \{A = y.\mathsf{type}\}\}&\end{aligned}$$

It cannot be typed with an inert type using the {}-I rule. Is this value well-typed? If we remove the tightness condition, then it is:

$$\dfrac{\dfrac{\dfrac{}{x.a;\,\Gamma \vdash \{A = x.a.\mathsf{type}\}\colon \{A\colon x.a.\mathsf{type}..x.a.\mathsf{type}\}}\ \text{DEF-TYP}}{x;\,\Gamma \vdash \left\{a = \nu(y\colon T^{x.a}_{A\,y})\,\{A = y.\mathsf{type}\}\right\}\colon \left\{a\colon \mu\left(y\colon T^{x.a}_{A\,y}\right)\right\}}\ \text{DEF-NEW}}{\vdash \nu(x)\left\{a = \nu(y\colon T^{x.a}_{A\,y})\,\{A = y.\mathsf{type}\}\right\}\colon \mu\left(x\colon\ \left\{a\colon \mu\left(y\colon T^{x.a}_{A\,y}\right)\right\}\right)}\ \{\}\text{-I}$$

where we use the shorthand

$$T^{x.a}_{A\,y} = \{A : y.\mathsf{type}..x.a.\mathsf{type}\}$$

The reason that this works is that the substitution of the self-variable y with $x.a$ in DEF-NEW made the type $T^{x.a}_{A\,y}$ have equal bounds, yielding $\{A : x.a.\mathsf{type}..x.a.\mathsf{type}\}$, and the DEF-TYP typing went through.

To avoid this problem we must either prevent the above value from being well-typed or change the definition of inertness to include the above type. Since the latter would significantly complicate the inertness definition which is supposed to be a simple syntactic property, we chose the former: a check in the definition typing rules that object types have tight bounds.

21.2 PROOF RECIPE FOR PDOT

The DOT proof presented in Part I employs the *proof recipe*, a stratification of the typing rules into multiple typing relations that rule out cycles in a typing derivation, but are provably as expressive as the general typing relation under the condition of an inert typing context. Recall that besides the general typing relation, the proof uses three intermediate relations:

- *tight* typing neutralizes the <:-SEL and SEL-<: rules that could introduce bad bounds,
- *invertible* typing contains introduction rules that create more complex types out of smaller ones, and
- *precise* typing contains elimination rules that decompose a type into its constituents.

see Section 5.3 for tight typing in DOT

see Section 5.4 for invertible typing in DOT

see Figure 5.2 for precise typing in DOT

The language features that pDOT adds to DOT create new ways to introduce cycles in a typing derivation. The stratification of the typing rules needs to be extended to eliminate these new kinds of cycles.

21.2.1 *Overview of Extended Proof Recipe*

The notion of aliased paths is inherently symmetric: if p and q are aliases for the same object, then any term with type $p.A$ also has type $q.A$ and vice versa. This is complicated further because the paths p and q can occur deeply inside some complex type, and whether a term has such a type should be independent of whether the type is expressed in terms of p or q. Another complicating factor is that a prefix of a path is also a path, which may itself be aliased. For example, if p is an alias of q and $q.a$ is an alias of r, then by transitivity, $p.a$ should also be an alias of r.

The pDOT proof eliminates cycles due to aliased paths by breaking the symmetry of path aliasing. When p and q are aliased, either

$\Gamma \vdash p : q.\mathsf{type}$ or $\Gamma \vdash q : p.\mathsf{type}$. The typing rules carefully distinguish these two cases, so that for every pair p, q of aliased paths introduced by a typing declaration, we know whether the aliasing was introduced by the declaration of p or of q. A key lemma then proves that if we have any sequence of aliasing relationships

$$p_0 \sim p_1 \sim \cdots \sim p_n$$

where for each i, either

$$\Gamma \vdash p_i : p_{i+1}.\mathsf{type} \quad \text{or} \quad \Gamma \vdash p_{i+1} : p_i.\mathsf{type}$$

we can reorder the replacements so that the ones of the first type all come first and the ones of the second type all come afterwards. More precisely, we can always find some "middle" path q such that

$$\Gamma \vdash p_0 : q.\mathsf{type} \quad \text{and} \quad \Gamma \vdash p_n : q.\mathsf{type}$$

(in degenerate cases, the middle path q might actually be p_0 or p_n). Therefore, we further stratify the proof recipe into two typing judgments the first of which accounts for the $\textsc{Sngl}_{pq}$-<: rule, and the second for the $\textsc{Sngl}_{qp}$-<: rule. This eliminates cycles in a typing derivation due to aliased paths, but the replacement reordering lemma ensures that it preserves expressiveness.

Another new kind of cycle is introduced by the field elimination rule Fld-E and the field introduction rule Fld-I that is newly added in pDOT. This cycle can be resolved in the same way as other cycles in DOT, by stratifying these rules in two different typing relations.

The final stratification of the pDOT typing rules requires seven typing relations rather than the four required in the soundness proof for DOT. General and tight typing serve the same purpose as in the DOT proof, but pDOT requires three elimination and two introduction typing relations.

Table 21.1 shows which typing rules of pDOT are handled by each of the auxiliary typing relations which are introduced in the next sections.

Relation		Type rules	Inlined subtyping rules (Sub +...)
Precise	$\vdash_!$	Var, Fld-E, Rec-E	$\textsc{And}_1$-<:, $\textsc{And}_2$-<:
	$\vdash_{!!}$	Sngl-E	
	$\vdash_{!!!}$	Sngl-Trans	
Invertible	$\vdash_¡$	&-I	$\textsc{Sngl}_{pq}$-<:, <:-And, Top, All-<:-All, Fld-<:-Fld, Typ-<:-Typ
	$\vdash_{¡¡}$	&-I, Fld-I, Rec-I	$\textsc{Sngl}_{qp}$-<:, <:-And, <:-Sel
Tight	$\vdash_\#$	all	all, tight versions of Sel and Sngl rules
General	$\vdash$	all	all

Table 21.1: Auxiliary typing relations that make up the proof recipe of pDOT

$$\frac{\Gamma, x\colon T \vdash t\colon U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash_! \lambda(x\colon T)\, t\colon \forall(x\colon T)\, U}\ (\textsc{All-I}_!) \qquad \frac{x;\, \Gamma, x\colon T \vdash d\colon T}{\Gamma \vdash_! \nu(x\colon T) d\colon \mu\,(x\colon T)}\ (\{\}\text{-I}_!)$$

Precise typing for values $\Gamma \vdash_! v\colon T$

$$\frac{\Gamma(x) = T}{\Gamma \vdash_! x\colon T}\ (\textsc{Var}_!) \qquad \frac{\Gamma \vdash_! p\colon T \wedge U}{\Gamma \vdash_! p\colon U}\ (\textsc{And}_2\text{-E}_!)$$

Precise-I typing $\Gamma \vdash_! p\colon T$

$$\frac{\Gamma \vdash_! p\colon \mu\,(z\colon T)}{\Gamma \vdash_! p\colon T\,[p/z]}\ (\textsc{Rec-E}_!) \qquad \frac{\Gamma \vdash_! p\colon \{a\colon T\}}{\Gamma \vdash_! p.a\colon T}\ (\textsc{Fld-E}_!)$$

$$\frac{\Gamma \vdash_! p\colon T \wedge U}{\Gamma \vdash_! p\colon T}\ (\textsc{And}_1\text{-E}_!)$$

$$\frac{\Gamma \vdash_! p\colon T}{\Gamma \vdash_{!!} p\colon T}\ (\textsc{Path}_{!!}) \qquad \frac{\Gamma \vdash_{!!} p\colon q.\mathsf{type} \qquad \Gamma \vdash_{!!} q.a}{\Gamma \vdash_{!!} p.a\colon q.a.\mathsf{type}}\ (\textsc{Sngl-E}_{!!})$$

Precise-II typing $\Gamma \vdash_{!!} p\colon T$

$$\frac{\Gamma \vdash_{!!} p\colon T}{\Gamma \vdash_{!!!} p\colon T}\ (\textsc{Path}_{!!!}) \qquad \frac{\Gamma \vdash_{!!} p\colon q.\mathsf{type} \qquad \Gamma \vdash_{!!!} q\colon U}{\Gamma \vdash_{!!!} p\colon U}\ (\textsc{Sngl-Trans}_{!!!})$$

Precise-III typing $\Gamma \vdash_{!!!} p\colon T$

Figure 21.1: Precise typing in pDOT

21.2.2 *Typing Judgments for pDOT's Proof Recipe*

This section introduces the auxiliary typing relations that build up pDOT's proof recipe.

21.2.2.1 *Three Levels of Elimination (Precise Typing) Rules*

The precise typing rules (responsible for type elimination, such as Rec-E) are presented in Figure 21.1. The rules are divided into three stages that closely correspond to store lookup.

store lookup is defined in Figure 19.6

The *precise-I* type represents a path's exact type as assigned by the environment (modulo possible recursion and intersection elimination). If a path is a variable x, precise-I typing is the same as DOT's precise typing. Additionally, precise-I typing can perform field selection (Fld-E$_!$): if x's precise-I type is the recursive type

see Figure 5.2 for precise DOT typing

$$\mu\,(x\colon \cdots \wedge \{a_1\colon \mu\,(y\colon \cdots \wedge \{a_n\colon T_n\} \wedge \dots)\} \wedge \dots)$$

then the precise-I type of $x.a_1 \ldots a_n$ returns the type T_n that is nested deeply in x's type. Precise-I typing mimics the Lookup-Step-Var and Lookup-Step-Val path-lookup rules.

The purpose of *precise-II* typing is to do field selection on paths that have singleton types. If the typing environment assigns a singleton type to a path p then precise-II typing enables field selections on p. In particular, if p has $q.\mathsf{type}$ and $q.a$ is typeable then under precise-II typing $p.a$ has $q.a.\mathsf{type}$ (Sngl-E$_{!!}$). This corresponds to the Lookup-Step-Path path-lookup rule.

Precise-III typing transitively follows a path's aliases through the typing environment. If a path p has a precise-II type $q.\mathsf{type}$ (i. e. p aliases q) then precise-III typing allows p to be typechecked with any of q's precise-III types. Precise-III typing is similar to the transitive closure of value lookup, realized by the Lookup-Trans lookup rule.

Obtaining a path's precise-III type is the ultimate goal of the proof recipe. If p has type T in precise-III typing, we know that either the environment directly assigns p the type T or that p is assigned a singleton type q, and by recursively following path aliases starting with q we eventually arrive at T. More precisely, if $\Gamma \vdash_{!!!} p : T$, then one of the following is true:

1. $\Gamma \vdash_{!} p : T$, i.e. T is the most precise type that Γ assigns to p (modulo possible recursion and intersection elimination), or
2.
   - $p = p'.\overline{b}$ (i.e. p' is a prefix of p) and
   - $\Gamma \vdash_{!} p' : q.\mathsf{type}$ (i.e. $\Gamma \vdash_{!!} p'.\overline{b} : q.\overline{b}.\mathsf{type}$), and either
     - $T = q.\overline{b}.\mathsf{type}$, or
     - $\Gamma \vdash_{!!!} q.\overline{b} : T$.

Precise typing is the only place in the proof recipe that encodes the Sngl-Trans, Sngl-E, and Fld-E typing rules, along with the explicit elimination rules such as Rec-E.

21.2.2.2 *Two Levels of Introduction (Invertible Typing) Rules*

Just like in DOT, type-introduction rules take place in *invertible* typing. Hoewever, pDOT's invertible typing is further split into two typing judgments. The reason is that the path-replacement subtyping rules Sngl$_{pq}$-<: and Sngl$_{qp}$-<: introduce new possibilities for cyclic typing derivations. For example, if $\Gamma \vdash p : q.\mathsf{type}$ then we want to rule out an infinite derivation that changes back and forth between applying the Sngl$_{pq}$-<: and Sngl$_{qp}$-<: rules:

$$\dfrac{\dfrac{\dfrac{\ldots}{\Gamma \vdash r : T} \qquad \dfrac{\Gamma \vdash p : q.\mathsf{type}}{\Gamma \vdash T <: T\,[q/p]}\ \textsc{Sngl}_{pq}\text{-}<:}{\Gamma \vdash r : T\,[q/p]}\ \textsc{Sub} \qquad \dfrac{\Gamma \vdash p : q.\mathsf{type}}{\Gamma \vdash T\,[q/p] <: T}\ \textsc{Sngl}_{qp}\text{-}<:}{\Gamma \vdash r : T}\ \textsc{Sub}$$

This derivation is infinite because it keeps alternating between typing r with T and $T\,[q/p]$.

In order to avoid such infinite derivations we restrict the direction in which path replacements can be performed. Specifically, we stratify invertible typing into two stages: one stage can only apply the SNGL_{pq}-<: replacement, the other can only apply the SNGL_{qp}-<: replacement. Invertible-I typing, denoted $\Gamma \vdash_{¡} p: T$, contains introduction rules and inlined versions of the SNGL_{pq}-<: subtyping rule and is presented in Figure 21.2. Invertible-II typing, denoted $\Gamma \vdash_{¡¡} p: T$, contains inlined versions of the SNGL_{qp}-<: subtyping rule and is shown in Figure 21.3.

21.2.2.3 *Tight Typing Rules*

In DOT, new subtyping relationships are introduced through the SEL-<: $_{DOT}$ and <:-SEL $_{DOT}$ subtyping rules that allow us to use variable-dependent types. The ability to use such dependent types is restricted in tight typing, which is equivalent to general typing but does not admit the introduction of new subtyping relationships: the tight versions of the above rules, SEL-<:-# $_{DOT}$ and <:-SEL-# $_{DOT}$, require tight bounds and precise typing in their premises.

$$\frac{\Gamma \vdash_{!} x: \{A: T..T\}}{\Gamma \vdash_{\#} x.A <: T}\ (\text{SEL-<:-\#}_{DOT})$$

pDOT's tight typing, shown in Figure 21.4, has to generalize DOT's tight typing to paths for which it uses the third level of precise typing introduced in Section 21.2.2.1.

$$\frac{\Gamma \vdash_{!!!} p: \{A: T..T\}}{\Gamma \vdash_{\#} p.A <: T}\ (\text{SEL-<:-\#})$$

Additionally to path-dependent types, pDOT has a second form of dependent types: singleton types. The subtyping rules related to singleton types, SNGL_{pq}-<: and SNGL_{qp}-<:, are handled in the same way as <:-SEL and SEL-<:. We define the tight versions of these rules, SNGL_{pq}-<:-# and SNGL_{qp}-<:-#, with restricted precise-typing premises.

$$\frac{\Gamma \vdash_{!!!} p: q.\mathsf{type}}{\Gamma \vdash_{\#} T <: T\,[q/p]}\ (\text{SNGL}_{pq}\text{-<:-\#})$$

21.2.3 *Proof Recipe Lemmas*

Given a typing $\Gamma \vdash p: T$ in an inert context Γ, the proof recipe works as follows:

1. prove that p's *general* type T is equal to its *tight* type, i. e.

$$\Gamma \vdash_{\#} p: T$$

2. prove that p's *tight* type is equal to its *invertible-*¡¡ type T, i. e.

$$\Gamma \vdash_{¡¡} p: T$$

3. establish a relationship between p's *invertible-II* type T and its *invertible-I* type T', i. e.

$$\Gamma \vdash_{¡} p: T'$$

Invertible-I path typing
$\Gamma \vdash_{\text{¡}} p : T$

$$\frac{\Gamma \vdash_{!!!} p : T}{\Gamma \vdash_{\text{¡}} p : T} \text{(Path-¡)} \qquad \frac{\Gamma \vdash_{\text{¡}} p : T}{\Gamma \vdash_{\text{¡}} p : \top} \text{(Top-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} p : \{a : S\} \qquad \Gamma \vdash_{\#} S <: T}{\Gamma \vdash_{\text{¡}} p : \{a : T\}} \text{(Fld-<:-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} r : \mu\,(x : T) \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{\text{¡}} r : \mu\,(x : T)\,[q/p]} \text{(Sngl-Rec-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} p : \{A : T..U\} \qquad \Gamma \vdash_{\#} T' <: T \qquad \Gamma \vdash_{\#} U <: U'}{\Gamma \vdash_{\text{¡}} p : \{A : T'..U'\}} \text{(Typ-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} r : r'.A \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{\text{¡}} r : r'.A\,[q/p]} \text{(Sngl-Sel-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} p : \forall(x : S)\,T \qquad \Gamma \vdash_{\#} S' <: S \qquad \Gamma, x : S' \vdash T <: T'}{\Gamma \vdash_{\text{¡}} p : \forall(x : S')\,T'} \text{(All-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} r : r'.\mathsf{type} \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{\text{¡}} r : r'.\mathsf{type}\,[q/p]} \text{(Sngl-Sngl-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} p : S \qquad \Gamma \vdash_{\text{¡}} p : T}{\Gamma \vdash_{\text{¡}} p : S \wedge T} \text{(And-¡)}$$

Invertible-I value typing
$\Gamma \vdash_{\text{¡}} v : T$

$$\frac{\Gamma \vdash_{!} v : T}{\Gamma \vdash_{\text{¡}} v : T} \text{(Path-}v\text{-¡)} \qquad \frac{\Gamma \vdash_{\text{¡}} v : T}{\Gamma \vdash_{\text{¡}} v : \top} \text{(Top-}v\text{-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} v : \mu\,(x : T) \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{\text{¡}} v : \mu\,(x : T)\,[q/p]} \text{(Rec-Sngl-}v\text{-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} v : \forall(x : S)\,T \qquad \Gamma \vdash_{\#} S' <: S \qquad \Gamma, x : S' \vdash T <: T'}{\Gamma \vdash v : \forall(x : S')\,T'} \text{(All-}v\text{-¡)}$$

$$\frac{\Gamma \vdash_{\text{¡}} v : S \qquad \Gamma \vdash_{\text{¡}} v : T}{\Gamma \vdash_{\text{¡}} v : S \wedge T} \text{(And-}v\text{-¡)}$$

Figure 21.2: Invertible-I typing in pDOT

Invertible-II path typing
$\Gamma \vdash_{ii} p : T$

$$\frac{\Gamma \vdash_{i} p : T}{\Gamma \vdash_{ii} p : T}\ (\textsc{Path-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} p : S \qquad \Gamma \vdash_{ii} p : T}{\Gamma \vdash_{ii} p : S \wedge T}\ (\textsc{And-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} p : T\,[p/x]}{\Gamma \vdash_{ii} p : \mu\,(x : T)}\ (\textsc{Rec-I-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} p.a : T}{\Gamma \vdash_{ii} p : \{a : T\}}\ (\textsc{Fld-I-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} p : T \qquad \Gamma \vdash_{!} q : \{A : T..T\}}{\Gamma \vdash_{ii} p : q.A}\ (\textsc{Typ-Sel-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} r : \mu\,(x : T) \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{ii} r : \mu\,(x : T)\,[p/q]}\ (\textsc{Sngl-Rec-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} r : r'.A \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{ii} r : r'.A\,[p/q]}\ (\textsc{Sngl-Sel-}_{ii})$$

$$\frac{\Gamma \vdash_{ii} r : r'.\mathsf{type} \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{ii} r : r'.\mathsf{type}\,[p/q]}\ (\textsc{Sngl-Sngl-}_{ii})$$

Invertible-II value typing
$\Gamma \vdash_{ii} v : T$

$$\frac{\Gamma \vdash_{i} v : T}{\Gamma \vdash_{ii} v : T}\ (\textsc{Path-}_{ii}\text{-}v)$$

$$\frac{\Gamma \vdash_{ii} v : S \qquad \Gamma \vdash_{ii} p : T}{\Gamma \vdash_{ii} v : S \wedge T}\ (\textsc{And-}_{ii}\text{-}v)$$

$$\frac{\Gamma \vdash_{ii} v : T\,[p/x]}{\Gamma \vdash_{ii} v : \mu\,(x : T)}\ (\textsc{Rec-I-}_{ii}\text{-}v)$$

$$\frac{\Gamma \vdash_{ii} v : T \qquad \Gamma \vdash_{!} q : \{A : T..T\}}{\Gamma \vdash_{ii} v : q.A}\ (\textsc{Typ-Sel-}_{ii}\text{-}v)$$

$$\frac{\Gamma \vdash_{ii} v : r'.A \qquad \Gamma \vdash_{!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_{ii} v : r'.A\,[p/q]}\ (\textsc{Sngl-Sel-}_{ii}\text{-}v)$$

Figure 21.3: Invertible-II typing in pDOT

Tight term typing
$\Gamma \vdash_\# t : T$

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

$$\frac{\Gamma \vdash_\# p : q.\mathsf{type} \qquad \Gamma \vdash q : T}{\Gamma \vdash_\# p : T} \text{(Sngl-Trans}_\#\text{)}$$

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

$$\frac{\Gamma \vdash_\# p : q.\mathsf{type} \qquad \Gamma \vdash_\# q.a}{\Gamma \vdash_\# p.a : q.a.\mathsf{type}} \text{(Sngl-E}_\#\text{)}$$

$$\frac{\Gamma \vdash_\# p : \forall(z : S)\, T \qquad \Gamma \vdash_\# q : S}{\Gamma \vdash_\# p\, q : T\,[q/z]} \text{(All-E}_\#\text{)}$$

$$\frac{\Gamma \vdash_\# p : T\,[p/x]}{\Gamma \vdash_\# p : \mu(x : T)} \text{(Rec-I}_\#\text{)}$$

$$\frac{x; \Gamma, x : T \vdash d : T}{\Gamma \vdash_\# \nu(x : T) d : \mu(x : T)} \text{({}-I}_\#\text{)}$$

$$\frac{\Gamma \vdash_\# p : \{a : T\}}{\Gamma \vdash_\# p.a : T} \text{(Fld-E}_\#\text{)}$$

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

$$\frac{\Gamma \vdash_\# p.a : T}{\Gamma \vdash_\# p : \{a : T\}} \text{(Fld-I}_\#\text{)}$$

$$\frac{\Gamma \vdash_\# p : T \qquad \Gamma \vdash_\# p : U}{\Gamma \vdash_\# p : T \wedge U} \text{(\&-I}_\#\text{)}$$

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

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

Tight subtyping
$\Gamma \vdash_\# T <: U$

$$\Gamma \vdash_\# T <: \top \text{(Top}_\#\text{)}$$

$$\Gamma \vdash_\# \bot <: T \text{(Bot}_\#\text{)}$$

$$\Gamma \vdash_\# T <: T \text{(Refl}_\#\text{)}$$

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

$$\Gamma \vdash_\# T \wedge U <: T \text{(And}_1\text{-<:}_\#\text{)}$$

$$\Gamma \vdash_\# T \wedge U <: U \text{(And}_2\text{-<:}_\#\text{)}$$

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

$$\frac{\Gamma \vdash_{!!!} p : \{A : S..S\}}{\Gamma \vdash_\# S <: p.A} \text{(<:-Sel}_\#\text{)}$$

$$\frac{\Gamma \vdash_{!!!} p : \{A : S..S\}}{\Gamma \vdash_\# p.A <: S} \text{(Sel-<:}_\#\text{)}$$

$$\frac{\Gamma \vdash_{!!!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_\# T <: T\,[q/p]} \text{(Sngl}_{pq}\text{-<:}_\#\text{)}$$

$$\frac{\Gamma \vdash_{!!!} p : q.\mathsf{type} \qquad \Gamma \vdash_{!!} q}{\Gamma \vdash_\# T <: T\,[p/q]} \text{(Sngl}_{qp}\text{-<:}_\#\text{)}$$

$$\frac{\Gamma \vdash_\# T <: U}{\Gamma \vdash_\# \{a : T\} <: \{a : U\}} \text{(Fld-<:-Fld}_\#\text{)}$$

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

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

Figure 21.4: Tight typing for pDOT

4. establish a relationship between p's *invertible-I* type T' and its *precise-III* type T'', i. e.

$$\Gamma \vdash_{!!!} p : T''$$

This type is the type that is directly assigned to p (or its aliases) by Γ, modulo recursion and intersection elimination.

The lemmas that establish the proof recipe have to be proved in reverse order and this is how we present them below. Note that except for tight typing, each of the proof-recipe relations comes in two versions: for paths and values. We present only the lemmas for paths; the formulations for values are similar and also simpler since there are fewer rules that apply to values than to paths.

21.2.3.1 *From Invertible-I to Precise-III Typing*

The lemmas in this section are specialized to function, record, and singleton types. The proof-recipe lemmas for values are specialized to function and object types but we omit them here because the function-type-related lemmas are the same as for paths, and the reasoning about object types is similar to the reasoning for singleton types.

FUNCTION TYPES To convert a function type from invertible-I typing into precise-III typing, we use the following lemma.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{¡} p : \forall(x : T)\, U}{\Gamma \vdash_{!!!} p : \forall(x : T')\, U' \qquad \Gamma \vdash_{\#} T <: T' \qquad \Gamma, x : T \vdash_{\#} U' <: U} \; (\vdash_{¡} \text{TO} \vdash_{!!!} \forall)$$

LEMMA 36 ($\vdash_{¡}$ to $\vdash_{!!!}$ $\forall$). *If* $\Gamma \vdash_{¡} p : \forall(x : T)\, U$ *and* Γ *is inert then there exist types* T' *and* U' *such that* $\Gamma \vdash_{\#} T <: T'$, $\Gamma, x : T \vdash_{\#} U' <: U$, *and* $\Gamma \vdash_{!!!} p : \forall(x : T')\, U'$.

For the proof recipe, we do not need to further convert p's type into precise-II and -I typings. However, these typing relations are needed to prove the lemmas of the proof recipe.

RECORD TYPES The following lemma converts from an invertible-I type declaration to a precise-III type:

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{¡} p : \{A : S..U\}}{\Gamma \vdash_{!!!} p : \{A : T..T\} \qquad \Gamma \vdash_{\#} S <: T <: U} \; (\vdash_{¡} \text{TO} \vdash_{!!!} \{A\})$$

LEMMA 37 ($\vdash_{¡}$ to $\vdash_{!!!}$ $\{A\}$). *If* $\Gamma \vdash_{¡} p : \{A : S..U\}$ *and* Γ *is inert then there exists a type* T *such that* $\Gamma \vdash_{!!!} p : \{A : T..T\}$, $\Gamma \vdash_{\#} S <: T$ *and* $\Gamma \vdash_{\#} T <: U$.

SINGLETON TYPES If a path has a precise-III singleton type $T_{!!!}$ then invertible-I typing can introduce pq replacements to that type, yielding a type $T_{¡}$. Subsequent invertible-II typing can introduce qp replacements to $T_{¡}$, yielding a type $T_{¡¡}$. To reason about the exact relationship between $T_{!!!}$, $T_{¡}$, and $T_{¡¡}$, we distinguish between pq- and qp-*replacements*, defined as follows.

$$\frac{\Gamma \vdash_{!!} q \qquad \Gamma \vdash_{!} p : q.\mathsf{type}}{\Gamma \vdash T \rightsquigarrow T\,[q/p]}$$

DEFINITION 38. *If* $\Gamma \vdash_{!} p : q.\mathsf{type}$ *then replacing an occurrence of the path* p *with* q *in a type is a* pq-replacement, *and replacing an occurrence of* q *with* p *in a type is a* qp-replacement.

We denote the fact that U is the result of a pq replacement in T as

$$\Gamma \vdash T \rightsquigarrow U$$

Additionally, if $\Gamma \vdash T \rightsquigarrow U$ and $\Gamma \vdash S \rightsquigarrow U$ we will write

$$\Gamma \vdash T \rightsquigarrow U \leftsquigarrow S$$

Finally, we denote the reflexive, transitive closure of $\rightsquigarrow$ as $\rightsquigarrow^*$.

The following lemma establishes the relationship between a path's invertible-I singleton type and its precise-III type.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{\text{i}} p : q.\mathsf{type}}{\Gamma \vdash_{\text{!!!}} p : q'.\mathsf{type} \qquad \Gamma \vdash q'.\mathsf{type} \rightsquigarrow q.\mathsf{type}}\ (\vdash_{\text{i}} \text{TO } \vdash_{\text{!!!}} \forall)$$

LEMMA 39 ($\vdash_{\text{i}}$ to $\vdash_{\text{!!!}}$ _.type). *If* $\Gamma \vdash_{\text{i}} p : q.\mathsf{type}$ *and* Γ *is inert then there exists a path* q' *such that* $\Gamma \vdash_{\text{!!!}} p : q'.\mathsf{type}$ *and* $\Gamma \vdash q'.\mathsf{type} \rightsquigarrow q.\mathsf{type}$.

21.2.3.2 *From Invertible-II to Invertible-I Typing*

This section presents the conversions from invertible-II to invertible-I typing for paths of function, record, and singleton types.

FUNCTION TYPES Invertible-II typing does not have rules for function types. Therefore, if a path has a function type in invertible-II typing it must have had the same type in invertible-I typing:

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{\text{ii}} p : \forall(x : T)\, U}{\Gamma \vdash_{\text{i}} p : \forall(x : T)\, U}\ (\vdash_{\text{ii}} \text{TO } \vdash_{\text{i}} \forall)$$

LEMMA 40 ($\vdash_{\text{ii}}$ to $\vdash_{\text{i}}$ $\forall$). *If* $\Gamma \vdash_{\text{ii}} p : \forall(x : T)\, U$ *and* Γ *is inert then* $\Gamma \vdash_{\text{i}} p : \forall(x : T)\, U$.

RECORD TYPES Similarly, invertible-II typing does not affect the invertible-I type of a path that is typed with a type declaration:

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{\text{ii}} p : \{A : S..U\}}{\Gamma \vdash_{\text{i}} p : \{A : S..U\}}\ (\vdash_{\text{ii}} \text{TO } \vdash_{\text{i}} \{A\})$$

LEMMA 41 ($\vdash_{\text{ii}}$ to $\vdash_{\text{i}}$ $\{A\}$). *If* $\Gamma \vdash_{\text{ii}} p : \{A : S..U\}$ *and* Γ *is inert then* $\Gamma \vdash_{\text{i}} p : \{A : S..U\}$.

SINGLETON TYPES Invertible-II typing may introduce qp replacements to singleton types. Therefore, if

$$\Gamma \vdash_{\text{ii}} p : q.\mathsf{type}$$

then p must have had a singleton type $q'.\mathsf{type}$ in invertible-I typing:

$$\Gamma \vdash_{\text{i}} p : q'.\mathsf{type}$$

such that $q.\mathsf{type}$ could be obtained from $q'.\mathsf{type}$ through a sequence of qp replacements. Since a qp replacement in a singleton type is the inverse of a pq replacement[6] we can say that $q'.\mathsf{type}$ was obtained from $q.\mathsf{type}$ through a series of pq replacements:

$$\Gamma \vdash q.\mathsf{type} \rightsquigarrow^* q'.\mathsf{type}$$

[6] *note that* $q.\overline{b}.\mathsf{type}\,[p/q]\,[q/p] = q.\overline{b}.\mathsf{type}$

LEMMA 42 ($\vdash_{ii}$ to $\vdash_{i}$ _.type). *If* Γ *is an inert context and* $\Gamma \vdash_{ii} p : q.\mathsf{type}$ *then there exists a path* q' *such that* $\Gamma \vdash_{i} p : q'.\mathsf{type}$ *and* $\Gamma \vdash q.\mathsf{type} \rightsquigarrow^* q'.\mathsf{type}$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{ii} p : q.\mathsf{type}}{\Gamma \vdash_{i} p : q'.\mathsf{type} \qquad \Gamma \vdash q.\mathsf{type} \rightsquigarrow^* q'.\mathsf{type}}\ (\vdash_{ii} \text{ TO } \vdash_{i} \_.\mathsf{type})$$

21.2.3.3 *From Tight to Invertible-II Typing*

Given the tight type of a path, we can convert it into an invertible-II type (cf. Theorem 10 ($\vdash_{\#}$ to $\vdash_{i}$)):

LEMMA 43 ($\vdash_{\#}$ to $\vdash_{ii}$). *If* $\Gamma \vdash_{\#} p : T$ *and* Γ *is inert then* $\Gamma \vdash_{ii} p : T$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{\#} p : T}{\Gamma \vdash_{ii} p : T}\ (\vdash_{\#} \text{ TO } \vdash_{ii})$$

Just as in DOT, to prove this lemma we need to first show that invertible-II typing is closed under tight subtyping:

LEMMA 44 (Invertible-II <: Closure). *If* Γ *is inert,* $\Gamma \vdash_{ii} p : T$, *and* $\Gamma \vdash_{\#} T <: U$ *then* $\Gamma \vdash_{ii} p : U$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{ii} p : T \qquad \Gamma \vdash_{\#} T <: U}{\Gamma \vdash_{ii} p : U}\ (\vdash_{ii} <: \textsc{Closure})$$

The proof of the last lemma is by induction on the tight-subtyping derivation. The $\textsc{Sngl}_{pq}$-<:-# and $\textsc{Sngl}_{qp}$-<:-# induction cases require the proofs of the following *replacement closure* lemmas:

LEMMA 45 (Invertible-II qp-Replacement Closure). *If* Γ *is inert,* $\Gamma \vdash_{ii} r : T$, $\Gamma \vdash_{!!!} p : q.\mathsf{type}$, *q has a precise-II type, and T contains a path q, then* $\Gamma \vdash_{ii} r : T\,[p/q]$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{!!} q \qquad \Gamma \vdash_{!!!} p : q.\mathsf{type} \qquad \Gamma \vdash_{ii} r : T}{\Gamma \vdash_{ii} r : T\,[p/q]}\ (\vdash_{ii} qp \textsc{ Closure})$$

Since invertible-II typing explicitly inlines the $\textsc{Sngl}_{qp}$-<:-# typing rule which performs qp-replacement, the proof of this lemma is straightforward. However, the proof of the analogous lemma for the reverse replacement direction (pq) is more challenging. Before presenting it we first need to prove that invertible-I typing is closed under pq replacement since it explicitly inlines the $\textsc{Sngl}_{pq}$-<:-# rule:

LEMMA 46 (Invertible-I pq-Replacement Closure). *If* Γ *is inert,* $\Gamma \vdash_{i} r : T$, $\Gamma \vdash_{!!!} p : q.\mathsf{type}$, *q has a precise-II type and T contains a path p, then* $\Gamma \vdash_{i} r : T\,[q/p]$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{!!} q \qquad \Gamma \vdash_{!!!} p : q.\mathsf{type} \qquad \Gamma \vdash_{i} r : T}{\Gamma \vdash_{i} r : T\,[q/p]}\ (\vdash_{i} pq \textsc{ Closure})$$

Using this lemma it is possible to prove the pq-replacement closure for invertible-II typing:

LEMMA 47 (Invertible-II pq-Replacement Closure). *If* Γ *is inert,* $\Gamma \vdash_{ii} r : T$, $\Gamma \vdash_{!!!} p : q.\mathsf{type}$, *q has a precise-II type, and T contains a path p, then* $\Gamma \vdash_{ii} r : T\,[q/p]$.

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_{!!} r \qquad \Gamma \vdash_{!!!} p : q.\mathsf{type} \qquad \Gamma \vdash_{ii} r : T}{\Gamma \vdash_{ii} r : T\,[q/p]}\ (\vdash_{ii} pq \textsc{ Closure})$$

The proof is by induction on the invertible-II typing derivation. To illustrate a point of difficulty in the proof consider, for instance, the $\textsc{Sngl-Sel-}_{ii}$ induction case where a path r's type T is derived from a qp-replacement. We want to prove that if we apply a pq-replacement (possibly of other paths p' and q') to T, yielding a type T', then $\Gamma \vdash_{ii} r : T'$. However, there is no invertible-II type rule that allows pq replacements, so we need to show that the same type T' could have been derived if we had applied the pq-replacement *before* the qp-replacement that yielded T. We omit the details of this proof here and refer the interested reader instead to the Coq proof.

$$\frac{\Gamma \vdash_{!!} q \qquad \Gamma \vdash_{ii} r : r'.A \qquad \Gamma \vdash_{!} p : q.\mathsf{type}}{\Gamma \vdash_{ii} r : r'.A\,[p/q]}\ (\textsc{Sngl-Sel-}_{ii})$$

21.2.3.4 *From General to Tight Typing*

The translation from general into tight typing is the start of the proof recipe. Recall that to prove that DOT's general typing implies tight typing in Theorem 6 ($\vdash$ to $\vdash_\#$), we needed to be able to "replace" the restricted precise-typing premise of Sel-<: $_{DOT}$ with a more general tight-typing premise, for which we proved Lemma 7 (Sel-<:-# Replacement) and Lemma 8 (Sel-<:-# Premise).

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_\# p\colon \{A\colon S..U\}}{\Gamma \vdash_{!!!} p\colon \{A\colon T..T\}}\ (\text{Sel-<:-\# Premise})$$

In pDOT, we have to do the same:

LEMMA 48 (Sel-<:-# Premise). *If* Γ *is an inert context, then if* $\Gamma \vdash_\# p\colon \{A\colon S..U\}$*, then there exists a type* T *such that* $\Gamma \vdash_{!!!} p\colon \{A\colon T..T\}$, $\Gamma \vdash_\# S <: T$, *and* $\Gamma \vdash_\# T <: U$.

Proof. The proof of this lemma uses the proof recipe lemmas described in the previous sections. It first converts p's tight-typing type $\{A\colon S..U\}$ to the same invertible-II type using Lemma 43 and to the same invertible-I type using Lemma 41. It then uses Lemma 37 to arrive at a precise-III type $\{A\colon T..T\}$ for p. □

$$\frac{\text{inert } \Gamma \qquad \Gamma \vdash_\# p\colon \{A\colon S..U\}}{\Gamma \vdash_\# S <: p.A <: U}\ (\text{Sel-<:-\# Replacement})$$

LEMMA 49 (Sel-<:-# Replacement). *If* Γ *is an inert context, then if* $\Gamma \vdash_\# p\colon \{A\colon S..U\}$*, then* $\Gamma \vdash_\# S <: p.A$ *and* $\Gamma \vdash_\# p.A <: U$.

Proof. Directly follows from Lemma 48. □

Similarly to strengthening the precise premises of the Sel tight-subtyping rules, we have to strengthen the precise-typing premises to tight-typing premises for the Sngl$_{pq}$-<: and Sngl$_{qp}$-<: rules:

$$\frac{\text{inert } \Gamma \quad \Gamma \vdash_\# q \qquad \Gamma \vdash_\# p\colon q.\mathsf{type}}{\Gamma \vdash_{!!!} p\colon q'.\mathsf{type} \qquad \Gamma \vdash_\# q'.\mathsf{type} <: q.\mathsf{type} \qquad \Gamma \vdash_{!!} q'}\ (\text{Sngl-<: Premise})$$

LEMMA 50 (Sngl-<:-# Premise). *If* Γ *is an inert context, then if* $\Gamma \vdash_\# p\colon q.\mathsf{type}$ *where* $\Gamma \vdash_\# q$*, then there exists a path* q' *such that*

- $\Gamma \vdash_{!!} q'$,
- $\Gamma \vdash_{!!!} p\colon q'.\mathsf{type}$, *and*
- $\Gamma \vdash_\# q'.\mathsf{type} <: q.\mathsf{type}$.

Proof. By Lemma 43 ($\vdash_\#$ to $\vdash_{ii}$) p's tight type is the same as its invertible-II type, i. e.

$$\Gamma \vdash_{ii} p\colon q.\mathsf{type}$$

By Lemma 42 ($\vdash_{ii}$ to $\vdash_i$ _.type) and Lemma 39 ($\vdash_i$ to $\vdash_{!!!}$ _.type),

$$\Gamma \vdash_{!!!} p\colon q'.\mathsf{type}$$

where

$$\Gamma \vdash q.\mathsf{type} \rightsquigarrow^* q''.\mathsf{type} \,{}^*\!\!\leftsquigarrow q'.\mathsf{type}$$

which allows us to prove that $\Gamma \vdash_\# q'.\mathsf{type} <: q''.\mathsf{type} <: q.\mathsf{type}$. Note that the actual proof needs to use stronger versions of Lemma 42 and Lemma 39 in order to prove that q' is typeable; however, we leave these details out to just display the main idea of the proof. □

$$\vdash \varnothing\Diamond \qquad \text{(Tp-Empty)}$$

$$\frac{\vdash \Gamma\Diamond \qquad x \notin \mathsf{fv}(\Gamma) \qquad \forall \bar{b}, q, \dfrac{\Gamma, x\colon T \vdash_! x.\bar{b}\colon q.\mathsf{type}}{\exists U, \Gamma, x\colon T \vdash_{!!} q\colon U}}{\vdash \Gamma, x\colon T\Diamond} \qquad \text{(Tp)}$$

Figure 21.5: Typed-paths environments

The proof of the next lemma immediately follows from Lemma 50.

LEMMA 51 (Sngl-<:-# Replacement). *If Γ is an inert context, then if $\Gamma \vdash_\# p\colon q.\mathsf{type}$ where $\Gamma \vdash_\# q$, then for any type T that contains a path p, $\Gamma \vdash_\# T <: T\,[q/p]$, and for any type U that contains a path q, $\Gamma \vdash_\# U <: U\,[p/q]$.*

$$\frac{\text{inert } \Gamma \quad \Gamma \vdash_\# q \qquad \Gamma \vdash_\# p\colon q.\mathsf{type}}{\Gamma \vdash_\# T <: T\,[q/p] \qquad \Gamma \vdash_\# U <: U\,[p/q]} \qquad \text{(Sngl-<:-\# Replacement)}$$

21.3 TYPED-PATHS ENVIRONMENTS

A pDOT type can depend on paths rather than just variables, and the type makes sense only if the paths within it make sense, i. e. if all paths have a type. To ensure that all paths in a type are typeable, we introduce the notion of *typed-paths environments*. The typed-path property requires that any path that appears in a type should itself also be typeable in the same typing context. Without this property, it would be possible for the typing rules to derive types for ill-formed paths, and there could be paths that have types but do not resolve to any value during program execution. We will need the typed-path property when we formulate the canonical-forms, progress, and preservation lemmas.

The precise definition of typed-path environments, denoted $\vdash \Gamma\Diamond$, is given in Figure 21.5. The Wt rule defines how to extend a typed-path environment Γ with a type T to maintain the typed-paths property. Specifically, it considers all the paths ($x.\bar{b}$) that are introduced by T and, if a path has a singleton type $q.\mathsf{type}$, requires q to be precisely typeable in an environment.

How does the rule retrieve all paths that are introduced by T? For that we just need to consider all precise-I typings of paths that start with x (recall that precise-I typing gives us information about the exact type, modulo recursion and intersection elimination, that an environment assigns to a path). For example, if $T = \mu\,(y\colon \{a\colon y.a.a.\mathsf{type}\})$, there are the following precise-I typings that T introduces:

see Figure 21.1 for precise typing

$$\Gamma, x\colon T \vdash_! x\colon \mu\,(y\colon \{a\colon y.a.a.\mathsf{type}\}) \tag{21.1}$$

$$\Gamma, x\colon T \vdash_! x\colon \{a\colon x.a.a.\mathsf{type}\} \tag{21.2}$$

$$\Gamma, x\colon T \vdash_! x.a\colon x.a.a.\mathsf{type} \tag{21.3}$$

The only typing applicable to our definition is (21.3).

Finally, how does the rule ensure that the paths in the above singleton types are typeable? For that it is not sufficient to require precise-I

typing: precise-I typing only allows us to do field selection on paths that have recursive or record types, whereas we also need to do field selection on paths that have singleton types. In the above example, the path $x.a.a$ has a precise-II type but no precise-I type.

Note that we also want our type system to allow paths that have been accessed through precise-III typing, which transitively follows path aliases in the environment. However, it can be easily shown that every precise-III typeable path also has a precise-II typeable path, which is what we require in the above definition of typed-paths environments.

21.4 CANONICAL FORMS IN PDOT

As described in Section 5.5, the DOT proof depends on canonical forms lemmas that state that if a variable has a function type, then it resolves to a corresponding function at execution time, and if it has a recursive object type, then it resolves to a corresponding object. The change from DOT to pDOT involves several changes to Canonical Forms.

Two changes follow directly from pDOT's operational semantics. The DOT canonical forms lemmas apply to variables. Since the pDOT Apply reduction rule applies to paths rather than variables, the canonical forms lemma is needed for paths. Since paths are normal forms in pDOT and there is no Proj reduction rule for them, on the surface, pDOT needs a canonical forms lemma only for function types but not for object types. However, to reason about a path with a function type, we need to reason about the prefixes of the path, which have an object type. Therefore, the induction hypothesis in the canonical forms lemma for function types must still include canonical forms for object types. Moreover, since pDOT adds singleton types to the type system, the induction hypothesis needs to account for them as well.

A more subtle but important change is that lookup of a path in an execution environment is a recursive operation in pDOT, and therefore its termination cannot be taken for granted. An infinite loop in path lookup would be a hidden violation of progress for function application, since the Apply reduction rule steps only once path lookup has finished finding a value for the path. Therefore, the canonical forms lemma proves that if a path has a function type, then lookup of that path does terminate, and the value with which it terminates is a function of the required type. The intuitive argument for termination requires connecting the execution environment with the typing environment: if direct lookup of path p yields another path q, then the context assigns p the singleton type $q.\mathsf{type}$. But in order for p to have a function type, there cannot be a cycle of paths in the typing context (because a cycle would limit p to have only singleton types),

and therefore there cannot be a cycle in the execution environment. The statement of the canonical forms lemma is:

LEMMA 52 (Canonical Forms for Functions). *Let γ be a store and Γ be an inert, typed-paths environment such that $\gamma : \Gamma$. If $\Gamma \vdash p : \forall(x : T)\, U$ then there exists a type T' and a term t such that*

1. $\gamma \vdash p \rightsquigarrow^* \lambda(x : T')\, t$,
2. $\Gamma \vdash T <: T'$, *and*
3. $\Gamma, x : T \vdash t : U$.

$$\frac{\text{inert } \Gamma \quad \gamma : \Gamma \quad \vdash \Gamma\Diamond \qquad \Gamma \vdash p : \forall(x : T)\, U}{\gamma \vdash p \rightsquigarrow^* \lambda(x : T')\, t \qquad \Gamma \vdash T <: T' \qquad \Gamma, x : T \vdash t : U}\ (\text{Canonical Forms } \forall)$$

This simple statement hides an intricate induction hypothesis and a long, tedious proof, since it needs to reason precisely about function, object, and singleton types and across all seven typing relations in the stratification of typing. We describe the key lemmas that are necessary for this proof below.

21.4.1 *Canonical Forms Proof*

One difficulty in proving Canonical Forms for pDOT is to establish a correspondence between a typing environment and a store. In DOT, if a store γ is well-formed with respect to a typing environment Γ ($\gamma : \Gamma$), then it immediately follows that any variable that has a type in Γ is mapped to a value in γ. We would like the same property for paths: if $\Gamma \vdash p$ then we would like there to exist a *value v of the same type* such that $\gamma \vdash p \rightsquigarrow^* v$. However, there might not be such a value: it is possible that p cyclically references other paths. Fortunately, since we only need a canonical-forms lemma for functions, we can prove a more restricted form of the corresponding-types lemma for function types. We split this lemma into two. The first one states that if a path p has a function type then p looks up to a value in the store.

LEMMA 53 (Corresponding Values for Functions). *Suppose that a store γ is well-formed with respect to an inert, typed-paths environment Γ, and that Γ assigns type $\forall(x : T)\, U$ to a path p, i.e. $\Gamma \vdash_{!!!} p : \forall(x : T)\, U$. Then there exists a value v that is assigned to p by the store γ, i.e. $\gamma \vdash p \rightsquigarrow^* v$.*

$$\frac{\text{inert } \Gamma \qquad \gamma : \Gamma \quad \vdash \Gamma\Diamond \qquad \Gamma \vdash_{!!!} p : \forall(x : T)\, U}{\gamma \vdash p \rightsquigarrow^* v}\ (\text{Corresp. Values } \forall)$$

The purpose of the second lemma is to show that if a path p has a function type and looks up to a value v in the store (as established by the previous lemma) then v has the same type as p. To prove that we state a more general lemma. It says that store lookup preserves function types:

LEMMA 54 (Corresponding Types for Functions). *Suppose that a store γ is well-formed with respect to an inert, typed-paths envrionment Γ, and that Γ assigns type $\forall(x : T)\, U$ to a stable term s (recall that a stable term is a path or a value), i.e. $\Gamma \vdash s : \forall(x : T)\, U$, and s looks up to s' in the store, i.e. $\gamma \vdash s \rightsquigarrow^* s'$. Then s' has the same type as s, i.e. $\Gamma \vdash s' : \forall(x : T)\, U$.*

$$\frac{\text{inert } \Gamma \qquad \gamma : \Gamma \quad \vdash \Gamma\Diamond \qquad \gamma \vdash s \rightsquigarrow^* s' \qquad \Gamma \vdash s : \forall(x : T)\, U}{\Gamma \vdash s' : \forall(x : T)\, U}\ (\text{Corresp. Types } \forall)$$

We present the main lemmas necessary to prove the above two.

The following lemma states that if a path p has a precise-III function type T then p either has type T under precise-II typing, or p aliases another path q that has T under precise-II typing.

LEMMA 55. *If* Γ *is an inert environment and* $\Gamma \vdash_{!!!} p : \forall(x : T)\, U$ *then either*

- $\Gamma \vdash_{!} p : \forall(x : T)\, U$, *or*
- *there exists a path* q *such that* $\Gamma \vdash_{!!!} p : q.\mathsf{type}$ *and* $\Gamma \vdash_{!} q : \forall(x : T)\, U$.

Proof. By induction on the precise-III typing derivation of p. □

This simple lemma states that opening a recursive type in the typing environment does not affect the typing of a term.

LEMMA 56. *If* $\Gamma, x : \mu\,(x : U) \vdash t : T$ *then* $\Gamma, x : U \vdash t : T$.

The following lemma establishes a correspondence between the precise-I type of a path and the term that it looks up to in the store.

LEMMA 57 (Lookup Preservation I). *Let* γ *be a store that is well-formed with respect to an inert, typed-paths envrionment* Γ, *and for some path* p, *stable term* s, *and type* U,

- $\gamma \vdash p \rightsquigarrow s$
- $\Gamma \vdash_{!} p : U$.

Then one of the following is true:

1. s *is a function value of type* U,
2. *there exist types* T, S, W *and definitions* d *such that*
   - s *is an object* $\nu(S : d)$,
   - $\Gamma \vdash_{!} p : \mu\,(y : T)$,
   - $p; \Gamma \vdash d\,[p/y] : S\,[p/y]$, *and*
   - $\Gamma \vdash S \rightsquigarrow^{*} W\, {}^{*}\!\!\leftsquigarrow T$ *for some type* W, *or*
3. *there exist paths* q, r, *and* r' *such that*
   - $t = q$,
   - $U = r.\mathsf{type}$, *and*
   - $\Gamma \vdash q.\mathsf{type} \rightsquigarrow^{*} r'.\mathsf{type}\, {}^{*}\!\!\leftsquigarrow r.\mathsf{type}$

see Definition 16 for the definition of $\gamma : \Gamma$

Proof. The proof is by induction on the derivation of $\gamma : \Gamma$. The empty-environment case cannot happen since a path cannot be typed in an

empty environment. In the inductive case, suppose that $p = x_p.\overline{b}$; then we have

$$\text{inert}\ (\Gamma, x\colon T) \tag{21.4}$$
$$\gamma\colon \Gamma \tag{21.5}$$
$$\vdash \Gamma, x\colon T \tag{21.6}$$
$$\Gamma \vdash v\colon T \tag{21.7}$$
$$\Gamma, x\colon T \vdash_! x_p.\overline{b}\colon U \tag{21.8}$$
$$\gamma, x \mapsto v \vdash x_p.\overline{b} \rightsquigarrow t \tag{21.9}$$

We consider two cases depending on whether x is equal to x_p.

Case 1: $x = x_p$.

We proceed by induction on the lookup-derivation (21.9).

see Figure 19.6 for the lookup definition

Case 1.i: Lookup-Step-Var. In this case, $p = x$, $s = v$. If v is a function then it is easy to show that $T = U$, and we have proved that s is a function value of type U as required.

If v is an object, then since by (21.4), T is inert, T must be a function, object, or singleton type. Applying the the proof recipe to v's typing (21.7) in each case, we can rule out the function and singleton-type cases and obtain that

$$T = \mu\,(y\colon T') \tag{21.10}$$
$$\Gamma \vdash_!\ \nu(y\colon S)d\colon \mu\,(y\colon S) \tag{21.11}$$
$$\Gamma \vdash\ S \rightsquigarrow^*\ U'\ {}^*\!\!\leftsquigarrow\ T' \tag{21.12}$$
$$\Gamma, x\colon \mu\,(y\colon T') \vdash_!\ x\colon \mu\,(y\colon T') \tag{21.13}$$

From here the result can be obtained by inverting (21.11) and using Narrowing (see Lemma 15) and Lemma 56.

Case 1.ii: Lookup-Step-Val. We have $p = x.\overline{b}.a$ and

$$\gamma, x \mapsto v \vdash x.\overline{b} \rightsquigarrow \nu(y\colon T')\,d \tag{21.14}$$
$$\Gamma, x\colon T \vdash_! x.\overline{b}.a\colon U \tag{21.15}$$

where $d = \cdots \wedge \{a = s\} \wedge \ldots$ To apply the induction hypothesis to (21.14) we need to know the precise-I type of $x.\overline{b}$. It is easy to show that (21.15) implies that for some type V,

$$\Gamma, x\colon T \vdash_! x.\overline{b}\colon \mu\,(y\colon V) \tag{21.16}$$

The only applicable case in the result of applying the induction hypothesis to (21.14) is (2) because $x.\overline{b}$'s type must be recursive. This yields

$$x.\overline{b}; \Gamma, x\colon T \vdash\ d\left[x.\overline{b}/y\right] : S\left[x.\overline{b}/y\right] \tag{21.17}$$
$$\Gamma \vdash\ S \rightsquigarrow^*\ W\ {}^*\!\!\leftsquigarrow\ V \tag{21.18}$$

Since the definitions d contain a record $\{a = s\}$ we can infer that

$$V = \cdots \wedge \{a \colon V'\} \wedge \ldots \tag{21.19}$$
$$\Gamma, x \colon T \vdash s \colon V' \tag{21.20}$$

Using (21.19) and (21.18) we can then show that

$$W = \cdots \wedge \{a \colon W'\} \wedge \ldots \qquad \text{and} \qquad \Gamma \vdash V' \rightsquigarrow^* W' \tag{21.21}$$
$$S = \cdots \wedge \{a \colon S'\} \wedge \ldots \qquad \text{and} \qquad \Gamma \vdash S' \rightsquigarrow^* W' \tag{21.22}$$

The remainder of the proof is based on a case analysis on the shapes of s and V' and is straightforward.

Case 1.iii: Lookup-Step-Path. In this case, $p = x.\overline{b}.a$, $s = q.a$, and we have

$$\gamma, x \mapsto v \vdash x.\overline{b} \rightsquigarrow q \tag{21.23}$$
$$\Gamma, x \colon T \vdash_! x.\overline{b}.a \colon U \tag{21.24}$$

To apply the induction hypothesis to (21.23) we proceed as in the Lookup-Step-Val case, obtaining

$$\Gamma, x \colon T \vdash_! x.\overline{b} \colon \mu\,(y \colon V) \tag{21.25}$$

The only applicable case in the result of applying the induction hypothesis to (21.23) is (3) since there, s is a path. However, in that case $x.\overline{b}$'s precise-I type is a singleton type which is incompatible with its recursive type according to (21.25), a contradiction.

Case 2: $x \neq x_p$. We have

$$\Gamma, x \colon T \vdash_! x_p.\overline{b} \colon U \tag{21.26}$$
$$\gamma, x \mapsto v \vdash x_p.\overline{b} \rightsquigarrow s \tag{21.27}$$

Since the receiver x_p of the path $x_p.\overline{b}$ is not equal to x, x_p must be contained in Γ, which means that the whole path $x_p.\overline{b}$ must be typeable in Γ:

$$\Gamma \vdash_! x_p.\overline{b} \colon U \tag{21.28}$$
$$\tag{21.29}$$

By similar reasoning we can look up the path in γ:

$$\gamma \vdash x_p.\overline{b} \rightsquigarrow s \tag{21.30}$$

This allows us to apply the induction hypothesis and obtain the result we need to prove.

□

Lemma 57 establishes a connection between a path's precise-I type and the term it looks up to in the store. The following lemma does the same for a path's precise-II type.

LEMMA 58 (Lookup Preservation II). *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and for some path p, stable term s, and type T,*

- $\gamma \vdash p \rightsquigarrow s$ *and*
- $\Gamma \vdash_{!!} p : T$.

Then one of the following is true:

1. *there exists a type S and term u such that*
   - *s is a function $\lambda(y : S)\, u$,*
   - $\Gamma \vdash s : T$, *and*
   - $\Gamma \vdash_{!} p : T$, *or*
2. *there exist types S and W, definitions d, and a path p such that*
   - *t is an object $\nu(y : S)d$,*
   - $\Gamma \vdash_{!} p : \mu\,(z : U)$,
   - $\Gamma \vdash_{!} p : T$,
   - $p; \Gamma \vdash d\,[p/y] : S\,[p/y]$, *and*
   - $\Gamma \vdash S \rightsquigarrow^* W\, {}^*\!\!\leftsquigarrow U$, *or*

   Note that a path's precise-I typing does not have to be unique since precise-I typing can apply type-elimination rules
3. *there exist paths q, r, and r' such that*
   - *$s = q$,*
   - *$T = r.\mathsf{type}$, and*
   - *$\Gamma \vdash r.\mathsf{type} \rightsquigarrow^* r'.\mathsf{type}\, {}^*\!\!\leftsquigarrow q.\mathsf{type}$.*

Proof. The proof is by induction on the precise-II derivation of p. *see Figure 21.1 for the definition of precise-II typing*

Case 1: PATH$_{!!}$. In this case, we have a precise-I derivation for p, and the result follows from Lemma 57.

Case 2: SNGL-E$_{!!}$. We have $T = q.a.\mathsf{type}$ and

$$\Gamma \vdash_{!!} p : q.\mathsf{type} \tag{21.31}$$
$$\Gamma \vdash_{!!} q.a : U \tag{21.32}$$
$$\gamma \vdash p.a \rightsquigarrow s \tag{21.33}$$

We proceed by induction on the lookup derivation (21.33). The variable-case is immediately ruled since our path ends with a field selection.

Case 2.i: LOOKUP-STEP-VAL. This case can be ruled out by applying the outer induction hypothesis.

Case 2.ii: LOOKUP-STEP-PATH. We have

$$\gamma \vdash p \rightsquigarrow q' \tag{21.34}$$

We apply the outer induction hypothesis, which yields only one possible case (3) where for some path r',

$$\Gamma \vdash q.\mathsf{type} \rightsquigarrow^* r'.\mathsf{type} \,{}^*\!\!\leftsquigarrow q'.\mathsf{type} \tag{21.35}$$

We now have $s = q'.a$ and $T = q.a.\mathsf{type}$. Equations (21.31) and (21.32) imply that $\Gamma \vdash_{!!} p : q.a.\mathsf{type}$, and from (21.35) it follows that

$$\Gamma \vdash q.a.\mathsf{type} \rightsquigarrow^* r'.a.\mathsf{type} \,{}^*\!\!\leftsquigarrow q'.a.\mathsf{type} \tag{21.36}$$

which is what we needed to show.

□

Next, we prove a lemma similar to the above two lemmas (57 and 58) that establishes a correspondence between a path's precise-III typing and the term it looks up in the store. Note that we only prove this lemma for the case where a path's type is inert. The case where a path has a precise-III singleton type is proved separately.

LEMMA 59 (Lookup Preservation III-Inert). *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and for some path p, stable term s, and type T,*

- $\gamma \vdash p \rightsquigarrow s$ *and*
- $\Gamma \vdash_{!!!} p : T$ *where T is* inert.

Then one of the following is true:

- *T is a function type $\forall(x : S)\, U$, and $\Gamma \vdash s : \forall(x : S)\, U$, or*
- *T is a recursive type $\mu\,(x : U)$, and there exist types S and W, definitions d, and a path p such that*
  - *t is an object $\nu(y : S)d$,*
  - *p; $\Gamma \vdash d\,[p/y] : S\,[p/y]$, and*
  - *$\Gamma \vdash S \rightsquigarrow^* W \,{}^*\!\!\leftsquigarrow U$, or*
- *t is a path q, and $\Gamma \vdash_{!!!} q : T$.*

see Figure 21.1 for the definition of precise-III typing

Proof. The proof is by induction on the precise-III typing of p. In each case, we get a precise-II typing for p to which we apply Lemma 58 and consider the three possible cases that the lemma yields. The proof for each case involves low-level reasoning about properties of precise-I, -II, and -III typing. We omit the details of the proof here and refer the reader to the Coq version instead. □

The following lemma states that if a path $x.\overline{b}$ looks up to a path $x.\overline{c}$ with the same receiver x, then the path's precise-II type is $x.\overline{c}.\mathsf{type}$.

LEMMA 60. *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and*

- $\gamma \vdash x.\overline{b} \rightsquigarrow x.\overline{c}.\mathsf{type}$
- $\Gamma \vdash_{!!} x.\overline{b} : T$

Then $T = x.\overline{c}.\mathsf{type}$.

Proof. The proof is by induction on $\gamma : \Gamma$. In the inductive case with the extended environment $\Gamma, y : U$, we distinguish between whether x is equal to y. If it is not, we simply apply the induction hypothesis. If $x = y$ then we finish the proof using Lemma 58. □

The next lemma establishes that if a path has a precise-I type then it can be looked up in the store.

LEMMA 61. *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and p has a precise-I type. Then there exists a stable term s to which p can be looked up in the store, i. e. $\gamma \vdash p \rightsquigarrow s$.*

Proof. Similarly to the proof of Lemma 57, we perform two inductions. The outer induction is on $\gamma : \Gamma$. In the inductive case where we have an extended environment $\Gamma, x : T$, we do a case analysis on whether p's receiver x_p is equal to x. If $x = x_p$ we induct on the precise-I typing of p. The interesting case is the field-selection case FLD-E$_!$ where we have

$$\Gamma, x : T \vdash_! x.\overline{b} : \mu\,(y : U) \tag{21.37}$$
$$\Gamma, x : T \vdash_! x.\overline{b} : \{a : U'\} \tag{21.38}$$

and we need to prove that $x.\overline{b}.a$ looks up to a term. By induction,

$$\gamma, x \mapsto v \vdash x.\overline{b} \rightsquigarrow s' \tag{21.39}$$

Applying Lemma 57 to (21.38) and (21.39) yields only one possible case where for some types S, W, and definitions d,

$$s' = \nu(z : S)d \tag{21.40}$$
$$x.\overline{b}; \Gamma, x : T \vdash d\left[x.\overline{b}/z\right] : S\left[x.\overline{b}/z\right] \tag{21.41}$$
$$\Gamma \vdash S \rightsquigarrow^* W \,{}^*\!\!\leftsquigarrow U \tag{21.42}$$

From (21.37) and (21.38) we can infer that for some type U',

$$U = \cdots \wedge \{a : U'\} \wedge \ldots \tag{21.43}$$

because if a path has a recursive type T and a record type U, it means that U was obtained by performing recursion elimination followed by

intersection elimination on T. This allows us to conclude from (21.42) that for some types S' and W',

$$S = \cdots \wedge \{a \colon S'\} \wedge \ldots \tag{21.44}$$
$$W = \cdots \wedge \{a \colon W'\} \wedge \ldots \tag{21.45}$$
$$\Gamma \vdash S' \rightsquigarrow^* W' \,^*\!\!\leftsquigarrow U' \tag{21.46}$$

because pq- and qp-replacements preserve a type's shape. Finally, we can show using (21.44) and (21.41) that the definitions d must match the type S, and therefore

$$d = \cdots \wedge \{a = s''\} \wedge \ldots \tag{21.47}$$

for some stable term s''. Finally, this allows us to conclude that

$$\gamma \vdash x.\overline{b}.a \rightsquigarrow s''$$

□

The following two lemmas state the same for precise-II and -III typing: if a path has a precise-II and -III type then it can be looked up in the store.

LEMMA 62. *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and p has a precise-II type. Then there exists a stable term s to which p can be looked up in the store, i.e. $\gamma \vdash p \rightsquigarrow s$.*

Proof. The proof is by induction on p's precise-II derivation. The PATH$_{!!}$ case is proved using Lemma 61. In the SNGL-E$_{!!}$ case, we use Lemma 58 and simple properties of precise typing. □

LEMMA 63. *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and p has a precise-III type. Then there exists a stable term s to which p can be looked up in the store, i.e. $\gamma \vdash p \rightsquigarrow s$.*

Proof. The proof is by induction on p's precise-III typing; in both cases it is easily solved using Lemma 62. □

If a path shares the same receiver with its precise-II singleton type q.type (i.e. $\Gamma \vdash_{!!} x.\overline{b} \colon x.\overline{c}.\mathsf{type}$) then the path looks up to q in the store.

LEMMA 64. *Let γ be a store that is well-formed with respect to an inert, typed-paths environment Γ, and $\Gamma \vdash_{!!} x.\overline{b} \colon x.\overline{c}.\mathsf{type}$. Then $\gamma \vdash x.\overline{b} \rightsquigarrow x.\overline{c}$.*

Proof. By Lemma 62, there exists a stable term s such that

$$\gamma \vdash x.\overline{b} \rightsquigarrow s$$

The result can be then proved using Lemma 58. □

If a path shares the same receiver with its precise-III singleton type q.type (i.e. $\Gamma \vdash_{!!!} x.\overline{b} \colon x.\overline{c}.\mathsf{type}$) then the path looks up to q in the store in a finite number of steps.

LEMMA 65. *Let γ be a store that is well-formed with respect to an inert, typed-paths envrionment Γ, and $\Gamma \vdash_{!!!} x.\overline{b} : x.\overline{c}.\mathsf{type}$. Then $\gamma \vdash x.\overline{b} \rightsquigarrow^* x.\overline{c}$.*

Proof. We proceed by induction on the precise-III derivation of p. The PATH$_{!!}$ case follows from Lemma 64. In the SNGL-E$_{!!}$ case, we have for some path q

$$\Gamma \vdash_{!!} x.\overline{b} : q.\mathsf{type} \tag{21.48}$$

$$\Gamma \vdash_{!!!} q : x.\overline{c}.\mathsf{type} \tag{21.49}$$

Suppose that $q = x_q.\overline{b_q}$. We distinguish between two cases depending on whether x is equal to x_q.

Case 1: $x = x_q$. By the induction hypothesis applied to (21.49),

$$\gamma \vdash x.\overline{b_q} \rightsquigarrow^* x.\overline{c} \tag{21.50}$$

By Lemma 64 applied to (21.48),

$$\gamma \vdash x.\overline{b} \rightsquigarrow x.\overline{b_q} \tag{21.51}$$

The result follows from (21.50) and (21.51).

Case 2: $x \neq x_q$. If $x.\overline{b}$'s precise-II type is $x_q.\overline{b_q}.\mathsf{type}$ where $x \neq x_q$ it means that the variable x_q must occur before x in the environment (otherwise the environment would be not be a typed-paths environment since we would not be able to have a type for $x_q.\overline{b_q}$ at the moment when x has this path as its singleton type). However, according to (21.49), $x_q.\overline{b_q}$'s precise type is $x.\overline{c}.\mathsf{type}$, which means that x must be defined before x_q by analogous reasoning. Since each variable occurs only once in the domain of an inert environment, we arrive at a contradiction.

□

The following lemma shows that if $x.\overline{b}$ has precise-III type $y.\overline{c}$, and $x \neq y$, then there must exist a type $x.\overline{b'}.\mathsf{type}$ that is the "last" precise-III type of $x.\overline{b}$ that has x as a receiver.

LEMMA 66. *Let γ be a store that is well-formed with respect to an inert, typed-paths environment Γ, and*

$$\Gamma \vdash_{!!!} x.\overline{b} : y.\overline{c}.\mathsf{type}$$

where $x \neq y$. Then there exist paths $x.\overline{b'}$ and $z.\overline{c'}$ such that $x \neq z$ and

- $\Gamma \vdash_{!!!} x.\overline{b} : x.\overline{b'}.\mathsf{type}$ *or* $x.\overline{b} = x.\overline{b'}$
- $\Gamma \vdash_{!!} x.\overline{b'} : z.\overline{c'}.\mathsf{type}$
- $\Gamma \vdash_{!!!} z.\overline{c'} : y.\overline{c}.\mathsf{type}$ *or* $z.\overline{c'} = y.\overline{c}$

Proof. The proof is by induction on the precise-III typing of $x.\overline{b}$. The interesting case is Sngl-Trans$_{!!!}$. We do a case analysis based on whether y is equal to z. If it is the proof follows immediately. If $y \neq z$ we have

$$\Gamma \vdash_{!!!} z.\overline{c'} : y.\overline{c}.\mathsf{type} \tag{21.52}$$
$$\Gamma \vdash_{!!} x.\overline{b} : z.\overline{c'}.\mathsf{type} \tag{21.53}$$

By the induction hypothesis applied to (21.52), there exist paths $z.\overline{c_1}$ and $z_2.\overline{c_2}$ such that $z \neq z_2$ and

$$\Gamma \vdash_{!!!} z.\overline{c'} : z.\overline{c_1}.\mathsf{type} \quad \text{or} \quad z.\overline{c'} = z.\overline{c_1} \tag{21.54}$$
$$\Gamma \vdash_{!!} z.\overline{c_1} : z_2.\overline{c_2}.\mathsf{type} \tag{21.55}$$
$$\Gamma \vdash_{!!!} z_2.\overline{c_2} : y.\overline{c}.\mathsf{type} \quad \text{or} \quad z_2.\overline{c_2} = y.\overline{c} \tag{21.56}$$

The proof can be then finished by a case analysis on whether x is equal to z. □

In a typed-paths environment, there is a precise-typing relation between equivalent singleton paths

LEMMA 67. *If Γ is an inert, typed-paths environment and $\Gamma \vdash p.\mathsf{type} \rightsquigarrow^* q.\mathsf{type}$ where p is a precise-III typeable path then either $p = q$ or $\Gamma \vdash_{!!!} p : q.\mathsf{type}$.*

Proof. The proof is by induction on the replacement closure relation. We omit the details here. □

We want to be able to prove that if a path p has a precise-III singleton type $r.\mathsf{type}$ then p can be looked up to r. However, we can only prove that if we know that p and r do not participate in an aliasing cycle. To establish that, we use the fact that r's type is a function type which guarantees that eventually the lookup chain results in a value. The following lemma says exactly this: if a path p's precise-III type is $r.\mathsf{type}$ and r has a precise-I function type then p can be looked up to r in the store in a finite number of steps.

LEMMA 68. *Let γ be a store that is well-formed with respect to an inert, typed-paths environment Γ,*

- $\Gamma \vdash_{!!!} p : r.\mathsf{type}$, *and*
- $\Gamma \vdash_{!} r : \forall (x : S)\, T$.

Then $\gamma \vdash p \rightsquigarrow^* r$.

Proof. As we have done before, we start with an induction on $\gamma : \Gamma$. In the inductive case, we have an extended environment $\Gamma, x : T$ and a path $p = x_p.\overline{b}$. We distinguish between whether x is equal to x_p. The

interesting case is when $x = x_p$ (the induction hypothesis takes care of the other case). We have

$$\Gamma, x\colon T \vdash_{!} r\colon \forall(y\colon S)\, V \tag{21.57}$$
$$\Gamma, x\colon T \vdash_{!!!} x.\overline{b}\colon r.\mathsf{type} \tag{21.58}$$

and we need to prove that $\gamma,\ x \mapsto v \vdash x.\overline{b} \rightsquigarrow^* r$.

By Lemma 63, there exists a stable term s such that

$$\gamma,\ x \mapsto v \vdash x.\overline{b} \rightsquigarrow s \tag{21.59}$$

By Lemma 59 applied to (21.58) and (21.59), there exist paths r' and r'' such that $s = r'$ and

$$\Gamma, x\colon T \vdash r.\mathsf{type} \rightsquigarrow^* r''.\mathsf{type} \,{}^*\!\!\leftsquigarrow r'.\mathsf{type} \tag{21.60}$$

Furthermore, because r has a precise-I function type by (21.57), r and all its prefixes cannot have a precise-I or -II singleton type (we do not present the proof here). This means that the only way how the $\Gamma, x\colon T \vdash r.\mathsf{type} \rightsquigarrow^* r''.\mathsf{type}$ could have been obtained in (21.60) is if $r = r''$, i. e.

$$\Gamma, x\colon T \vdash r'.\mathsf{type} \rightsquigarrow^* r.\mathsf{type} \tag{21.61}$$

Suppose now that $r = x_r.\overline{c}$. We proceed by case analysis on whether x is equal to x_r.

Case 1: $x = x_r$. Follows from Lemma 65.

Case 2: $x \neq x_r$. By Lemma 66 applied to (21.58), there exist paths $x.\overline{b'}$ and $z.\overline{c'}$ such that

$$x \neq z \tag{21.62}$$
$$\Gamma, x\colon T \vdash_{!!!} x.\overline{b}\colon x.\overline{b'}.\mathsf{type} \tag{21.63}$$
$$\Gamma, x\colon T \vdash_{!!} x.\overline{b'}\colon z.\overline{c'}.\mathsf{type} \tag{21.64}$$
$$\Gamma, x\colon T \vdash_{!!!} z.\overline{c'}\colon x_r.\overline{c}.\mathsf{type} \tag{21.65}$$

we leave out the cases of Lemma 66 when $x.\overline{b} = x.\overline{b'}$ or $z.\overline{c'} = x_r.\overline{c}$ here because they are simpler

By Lemma 62 applied to (21.64), there exists a stable term s' such that

$$\gamma,\ x \mapsto v \vdash x.\overline{b'} \rightsquigarrow s' \tag{21.66}$$

Applying Lemma 58 to (21.64) and (21.66), after eliminating the impossible cases we obtain that there exist paths q and q' such that $s' = q$ and

$$\Gamma, x\colon T \vdash z.\overline{c'}.\mathsf{type} \rightsquigarrow^* q'.\mathsf{type} \,{}^*\!\!\leftsquigarrow q.\mathsf{type} \tag{21.67}$$

By Lemma 65 applied to (21.63),

$$\gamma,\, x \mapsto v \vdash x.\overline{b} \rightsquigarrow^* x.\overline{b'} \tag{21.68}$$

By (21.68) and (21.66) we now know that

$$\gamma,\, x \mapsto v \vdash x.\overline{b} \rightsquigarrow^* q \tag{21.69}$$

and we need to prove that $\gamma,\, x \mapsto v \vdash x.\overline{b} \rightsquigarrow^* x_r.\overline{c}$. Therefore, it remains to show that $\gamma,\, x \mapsto v \vdash q \rightsquigarrow^* x_r.\overline{c}$. To do that we apply Lemma 67 to the three qp-replacement closures in (21.61) and (21.69), each of which yields a disjunction of equalities or precise-III typing relationships between the involved paths. The proof is based on a case analysis of the resulting eight cases and involves low-level reasoning about the properties of precise typing. The resulting cases are either ruled out due to contradictions or by applying the induction hypothesis.

□

$$\frac{\text{inert } \Gamma \quad \gamma : \Gamma \quad \vdash \Gamma\Diamond \quad \Gamma \vdash_{!!!} p : \forall(x : T)\, U}{\gamma \vdash p \rightsquigarrow^* v}\ (\textsc{Corresp. Values }\forall)$$

We can now sketch the proof of Lemma 53 (Corresp. Values ∀).

Proof of Lemma 53 (Corresp. Values ∀). By Lemma 55 one of the following is true:

Case 1: $\Gamma \vdash_! p : \forall(x : T)\, U$. By Lemma 61, there exists a term s such that $\gamma \vdash p \rightsquigarrow s$. Using Lemma 57 we can infer that either

- s is a function, in which case we are done, or
- p has a recursive or singleton type under precise-I typing which is incompatible with p's precise-I function type, a contradiction.

Case 2: There exists a path q such that $\Gamma \vdash_{!!!} p : q.\mathsf{type}$ and $\Gamma \vdash_! q : \forall(x : T)\, U$. By Lemma 68,

$$\gamma \vdash p \rightsquigarrow^* q,$$

and by Lemma 61,

$$\gamma \vdash q \rightsquigarrow s$$

for some stable term s. According to Lemma 57, s must be a function $\lambda(x : T)\, u$. Therefore, $\gamma \vdash p \rightsquigarrow^* \lambda(x : T)\, u$.

□

$$\frac{\text{inert } \Gamma \quad \gamma : \Gamma \quad \vdash \Gamma\Diamond \quad \gamma \vdash s \rightsquigarrow^* s' \quad \Gamma \vdash s : \forall(x : T)\, U}{\Gamma \vdash s' : \forall(x : T)\, U}\ (\textsc{Corresp. Types }\forall)$$

Finally, we outline the proof of Lemma 54.

Proof of Lemma 54 (Corresp. Types ∀). The proof is by induction on the reflexive, transitive closure of store lookup; the reflexive case is vacu-

ously true. In the inductive case, we have

$$\gamma \vdash p \rightsquigarrow s_1 \tag{21.70}$$
$$\gamma \vdash s_1 \rightsquigarrow^* s' \tag{21.71}$$
$$\Gamma \vdash p : \forall(x : T)\, U \tag{21.72}$$

We would like to apply the induction hypothesis to (21.71), which would conclude the proof; however, for that we need to show that s' has the same type as p.

Applying the proof recipe to (21.72), we obtain that

$$\Gamma \vdash_{!!!} p : \forall(x : T')\, U'$$

where $\Gamma \vdash_{\#} T <: T'$ and $\Gamma, x : T \vdash_{\#} U' <: U$. By Lemma 59 applied to (21.70) and (21.72), s_1 must have the same type as p's precise type:

$$\Gamma \vdash s_1 : \forall(x : T')\, U'$$

Since $\Gamma \vdash \forall(x : T')\, U' <: \forall(x : T)\, U$, by subsumption,

$$\Gamma \vdash s_1 : \forall(x : T)\, U$$

which is what we needed to show. □

21.5 VALUE TYPING

Recall from Section 5.6 that the preservation lemma depends on Lemma 20 (Value Typing) according to which any well-typed value has an inert precise type. In pDOT, the lemma needs to additionally state that the value's precise type maintains the typed-paths property of an environment:

$$\frac{\Gamma \vdash v : T \qquad \text{inert } \Gamma \qquad \vdash \Gamma\Diamond}{\Gamma \vdash_! v : T' \quad \text{inert } T' \quad \vdash \Gamma, x : T'\Diamond \quad \Gamma \vdash T' <: T}\ (\textsc{Value Typing})$$

LEMMA 69 (Value Typing). *If Γ is an inert, typed-paths environment and $\Gamma \vdash v : T$, then there exists an inert type T' such that*

- $\Gamma \vdash_! v : T'$,
- $\Gamma \vdash T' <: T$, *and*
- $\vdash \Gamma, x : T'\Diamond$ *where* $x \notin \mathsf{dom}(\Gamma)$

The proof of pDOT's version of value typing is more complicated than in DOT. We first introduce a *type-lookup* relation and a few auxiliary lemmas.

To prove Value Typing for pDOT we need to be able to look inside of deeply nested field declarations of types. For that we introduce a *type-lookup* relation on a path p and two types T, U, denoted $T \overset{p}{\Rightarrow} U$, which allows us to follow the path p inside of the recursive type T to

$$\varnothing \overset{x}{\Rightarrow} \varnothing \qquad \text{(Tl-Empty)}$$

$$\frac{T \overset{x.\overline{b}}{\Longrightarrow} \mu\,(y\colon \cdots \wedge \{a\colon S\} \wedge \dots)}{T \overset{x.\overline{b}.a}{\Longrightarrow} S\left[x.\overline{b}/y\right]} \qquad \text{(Tl)}$$

Figure 21.6: The type lookup relation

yield the type U. For example, $\mu\,(x\colon \{a\colon \mu\,(y\colon \{b\colon \top\})\}) \overset{x.a.b}{\Longrightarrow} \top$. The definition of type lookup is presented in Figure 21.6.

The following lemma connects precise-I typing of a path $x.b_1.\cdots.b_n$ with type lookup in x's environment type $\Gamma(x)$. Specifically, it gives one access to the environment type that is assigned to b_1 and to b_n.

LEMMA 70. *Let* $\Gamma, x\colon \mu\,(x\colon T)$ *be an inert environment, and*

$$\Gamma, x\colon \mu\,(x\colon T) \vdash_! x.\overline{b}\colon W$$

where $\overline{b} = b_1.\cdots.b_n$ *and* $n > 0$. *Then there exists a type* V *such that*

- $T = \cdots \wedge \{b_1\colon V\} \wedge \cdots$ *and*
- $\mu\,(x\colon T) \overset{x.b_1\cdots.b_{n-1}}{\Longrightarrow} \mu\,(z\colon \cdots \wedge \{b_n\colon W\} \wedge \cdots)$

Proof. By induction on precise-I typing. □

Suppose that the path p corresponds to an object that contains a record d of type $\{b\colon V\}$, and that $V = \cdots \wedge \{a\colon q.\mathsf{type}\} \wedge \cdots$. Then q is well-typed.

LEMMA 71. *Suppose that for an inert type* S,

- $p; \Gamma \vdash d\colon \{b\colon V\}$,
- $S \overset{p}{\Rightarrow} \mu\,(y\colon T)$,
- $T\,[p/y] = \cdots \wedge \{b\colon V\} \wedge \cdots$,
- $S \overset{p.b.\overline{b}}{\Longrightarrow} \mu\,(z\colon U)$, *and*
- $U\left[p.b.\overline{b}/z\right] = \cdots \wedge \{a\colon q.\mathsf{type}\} \wedge \dots.$

Then q *is typeable in* Γ.

Proof. The proof is done by a case analysis on d's typing derivation. It requires reasoning about properties of type lookup which we omit here. □

The same generalizes to the case where a path p refers to an object with multiple declarations d whose record type contains $\{b\colon V\}$.

LEMMA 72. *Suppose that for an inert type* S,

- $p; \Gamma \vdash d\colon \cdots \wedge \{b\colon V\} \wedge \cdots$,

- $S \overset{p}{\Rightarrow} \mu\,(y\colon T)$,
- $T\,[p/y] = \cdots \wedge \{b\colon V\} \wedge \cdots$,
- $S \overset{p.b.\bar{b}}{\Longrightarrow} \mu\,(z\colon U)$, *and*
- $U\,\left[p.b.\bar{b}/z\right] = \cdots \wedge \{a\colon q.\mathsf{type}\} \wedge \ldots.$

Then q is typeable in Γ.

Proof. The proof is by induction on the definition-typing derivation of d and uses Lemma 71. □

We can now outline the proof of Lemma 69 (Value Typing).

Proof of Lemma 69 (Value Typing). The proof is by induction on the typing of v. The interesting case is the object typing rule {}-I (see Figure 19.2). We have $v = \nu(x\colon U)d$, $T = \mu\,(x\colon U)$, and

$$x;\, \Gamma,\, x\colon U \vdash d\colon U \tag{21.73}$$

We need to prove that there exists an inert type T' that is a subtype of $\mu\,(x\colon U)$, such that $\Gamma \vdash_! \nu(x\colon U)d\colon T'$ and such that $\vdash \Gamma,\, x\colon T'\Diamond$. We choose $\mu\,(x\colon U)$ as this type. The challenge is to show that $\vdash \Gamma,\, x\colon \mu\,(x\colon U)\,\Diamond$. That is, we must show that if a path whose receiver is x has a singleton type $q.\mathsf{type}$ then q must be precise-II typeable in the extended environment $\Gamma,\, x\colon T$ (see rule Wt in Figure 21.5). To prove that, suppose that

$$\Gamma,\, x\colon \mu\,(x\colon U) \vdash_! x.\bar{b}\colon q.\mathsf{type} \tag{21.74}$$

We first prove that q has a *general* type in the extended environment. We must show that there exists a type S such that $\Gamma,\, x\colon \mu\,(x\colon U) \vdash_{!!} q\colon S$.

Suppose that $\bar{b} = b_1.\cdots.b_n$. If $n = 0$ then from (21.74) we can infer that $U = q.\mathsf{type}$. However, then $\mu\,(x\colon U)$ is not inert, a contradiction.

Therefore, $n > 0$, and by Lemma 70 applied to (21.74), we have

$$\mu\,(x\colon U) \overset{x.b_1.\cdots.b_{n-1}}{\Longrightarrow} \mu\,(z\colon \cdots \wedge \{b_n\colon q.\mathsf{type}\} \wedge \cdots) \tag{21.75}$$

Then by Lemma 72 applied to (21.74) and (21.75) (we choose $\mu\,(x\colon U)$ as S and x as p, and we use the fact that $\mu\,(x\colon U) \overset{x}{\Rightarrow} \mu\,(x\colon U)$) we obtain that q has a general type. It is then easy to show using the proof recipe that if a path has a general type it also has a precise-II type, which concludes the proof. □

$$\frac{\Gamma \vdash v\colon T \qquad \text{inert } \Gamma \qquad \vdash \Gamma\Diamond}{\Gamma \vdash_! v\colon T'} \quad \text{inert } T' \quad \vdash \Gamma,\, x\colon T'\Diamond \quad \Gamma \vdash T' <\colon T$$
(Value Typing)

21.6 TYPE SOUNDNESS FOR PDOT

To formulate the soundness theorems we update the definition of normal forms for pDOT:

$p \twoheadrightarrow$ $v \twoheadrightarrow$

DEFINITION 73 (Normal Form). *A term t is in* normal form, *denoted $t \twoheadrightarrow$, if t is either a path or a value.*

The two central lemmas of the soundness proof are Progress and Preservation:

$$\frac{\gamma : \Gamma \quad \text{inert } \Gamma \quad \vdash \Gamma\Diamond \quad \Gamma \vdash t : T}{\gamma \mid t \longmapsto \gamma' \mid t' \quad \vee\ t \twoheadrightarrow}\ (\text{PROGRESS})$$

LEMMA 74 (Progress). *Let γ be a store and Γ a typing environment. If all of the following hold:*

- $\gamma : \Gamma$,
- Γ *is inert,*
- Γ *is a typed-paths environment, and*
- $\Gamma \vdash t : T$,

then t is in normal form or there exists a term t' and a store γ' such that $\gamma \mid t \longmapsto \gamma' \mid t'$.

$$\frac{\gamma : \Gamma \quad \text{inert } \Gamma \quad \vdash \Gamma\Diamond \quad \Gamma \vdash t : T \quad \gamma \mid t \longmapsto \gamma' \mid t'}{\Gamma' \vdash t' : T \quad \gamma' : \Gamma' \quad \text{inert } \Gamma' \quad \vdash \Gamma'\Diamond}\ (\text{PRESERVATION})$$

LEMMA 75 (Preservation). *Let γ be a store and Γ a typing environment. If all of the following hold:*

- $\gamma : \Gamma$,
- Γ *is inert,*
- Γ *is a typed-paths environment,*
- $\Gamma \vdash t : T$, *and*
- $\gamma \mid t \longmapsto \gamma' \mid t'$,

then there exists an inert, typed-paths environment Γ' such that $\gamma' : \Gamma'$ and $\Gamma' \vdash t' : T$.

With the canonical forms and value typing lemmas in place, the proofs of the above two lemmas mirror the proofs of the same lemmas of the simple soundness proof for DOT (Section 5.6).

Progress and Preservation allow us to easily prove type safety which ensures that any well-typed pDOT program does not get stuck, i.e. it either diverges or reduces to a normal form (a path or a value):

$$\frac{\vdash t : T}{(\varnothing \mid t \longmapsto^* \gamma \mid t' \ \wedge\ t' \twoheadrightarrow) \ \vee\ t \Uparrow}\ (\text{pDOT SOUNDNESS})$$

THEOREM 76 (pDOT Type Soundness). *If $\vdash t : T$ then either t diverges ($t \Uparrow$), or t reduces to a normal form s, i.e. $\varnothing \mid t \longmapsto^* \gamma \mid s$ and $\Gamma \vdash s : T$ for some Γ such that $\gamma : \Gamma$.*

Since evaluating pDOT programs can result in paths (which are normal form), one might ask whether looking up those paths yields anything meaningful. As mentioned in Section 18.2.3, looking up any well-typed path in the runtime environment results either in a value or an infinite loop. To formulate the final soundness theorem that reasons about both term reduction and path lookup we define the following extended reduction relation $\twoheadrightarrow$:

$$\frac{\gamma \mid t \longmapsto \gamma' \mid t'}{\gamma \mid t \twoheadrightarrow \gamma' \mid t'} \qquad \frac{\gamma \vdash s \rightsquigarrow s'}{\gamma \mid s \twoheadrightarrow \gamma \mid s'}$$

We denote the reflexive, transitive closure of extended reduction as $\twoheadrightarrow^*$.

DEFINITION 77 (Extended Divergence). *A term t* extendedly diverges, *denoted $t \Uparrow\Uparrow$, if there exists an infinite extended-reduction sequence*

$$\varnothing \mid t \twoheadrightarrow \gamma_1 \mid t_1 \twoheadrightarrow \ldots \twoheadrightarrow \gamma_n \mid t_n \twoheadrightarrow \ldots$$

Finally, we state the following extended soundness theorem:

THEOREM 78 (Extended Type Soundness). *If $\vdash t : T$ then either t extendedly diverges ($t \Uparrow\Uparrow$), or t reduces to a* value, *i.e. $\varnothing \mid t \twoheadrightarrow^* \gamma \mid v$.*

$$\frac{\vdash t : T}{\varnothing \mid t \twoheadrightarrow \gamma \mid v \;\vee\; t \Uparrow\Uparrow}$$
(EXTENDED SOUNDNESS)

A diagram with some of the pDOT-related changes to the DOT soundness proof is presented in Figure 21.7.

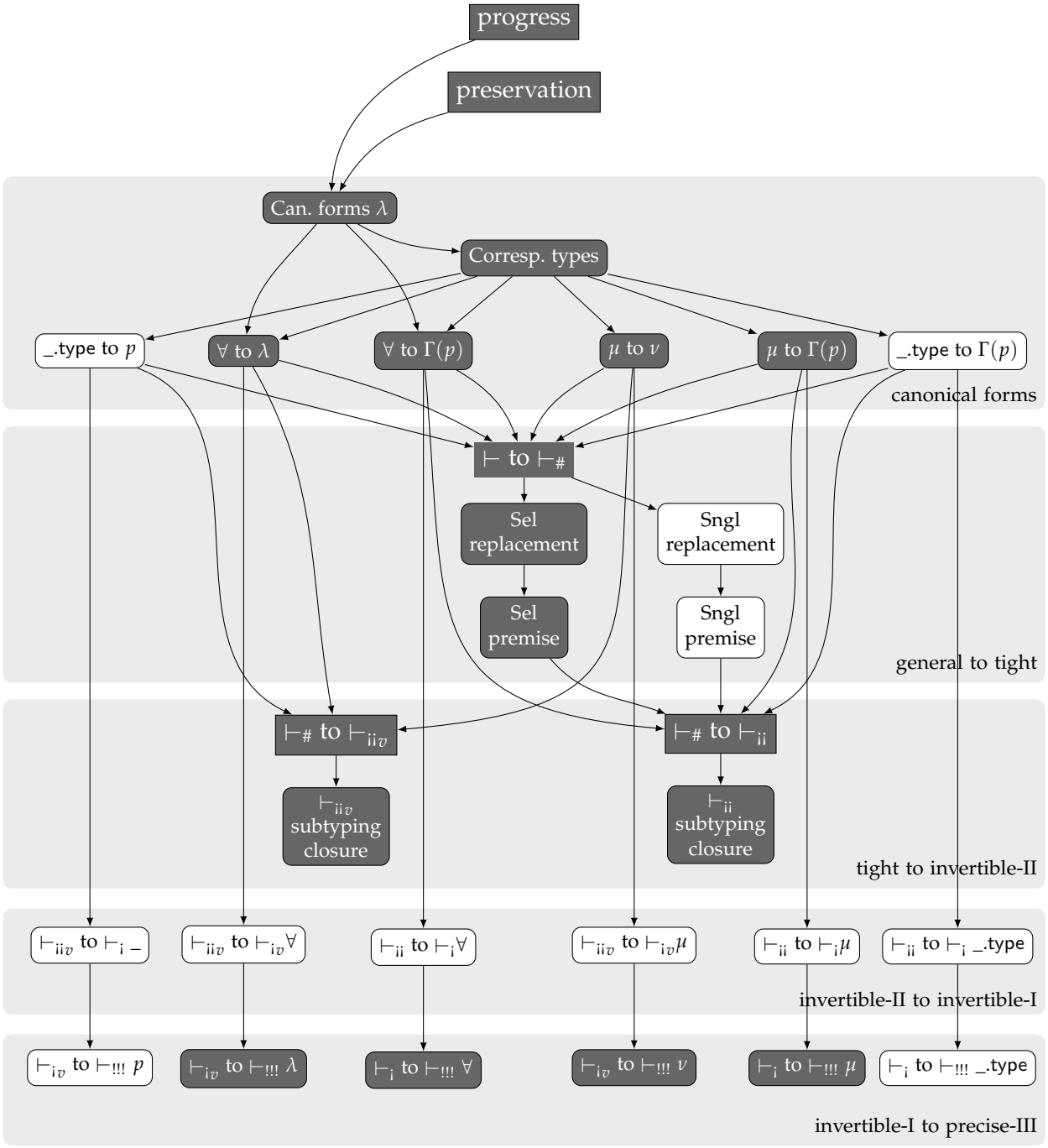

Figure 21.7: An instance of the dependency graph from Figure 5.4 showing the main lemmas in the pDOT proof as an extension of the simple DOT proof (Part I). Gray nodes denote the pDOT lemmas that have similar analogues in the DOT proof. White nodes denote pDOT specific lemmas. We omit the Precise-II and Precise-I related lemmas as well as additional lemmas required to prove Corresponding Types and the conversions between tight and invertible-II, and between invertible-II and invertible-I typing.

22 RELATED WORK

This section reviews the work related to formalizing Scala with support for fully path-dependent types.

22.1 EARLY CLASS-BASED SCALA FORMALIZATIONS

Several predecessors of the DOT calculus support path-dependent types on paths of arbitrary length. The first Scala formalization, νObj (Odersky, Cremet, et al., 2003), is a nominal, class-based calculus with a rich set of language features that formalizes object-dependent type members. Two subsequent calculi, Featherweight Scala (FS_{alg}) (Cremet et al., 2006) and Scalina (Moors, Piessens, and Odersky, 2008), build on νObj to establish Scala formalizations with algorithmic typing and with full support for higher-kinded types. All three calculi support paths of arbitrary length, singleton types, and abstract type members. Whereas FS_{alg} supports type-member selection directly on paths, νObj and Scalina allow selection $T\#A$ on types. A path-dependent type $p.A$ can thus be encoded as a selection on a singleton type: $p.\mathsf{type}\#A$. νObj is the only of the above calculi that comes with a type-safety proof. The proof is non-mechanized.

Both pDOT and these calculi prevent type selections on non-terminating paths. νObj achieves this through a *contraction* requirement that prevents a term on the right-hand side of a definition from referring to the definition's self variable. At the same time, recursive calls can be encoded in νObj by referring to the self variable from a nested class definition. FS_{alg} ensures that paths are normalizing through a cycle detection mechanism that ensures that a field selection can appear only once as part of a path. Scalina avoids type selection $T\#A$ on a non-terminating type T by explicitly requiring T to be of a *concrete* kind, which means that T expands to a structural type R that contains a type member A. Although Scalina allows A to have upper and lower bounds, bad bounds are avoided because A also needs to be immediately initialized with a type U that conforms to A's bounds, which is more restrictive than DOT. In pDOT, it is possible to create cyclic paths but impossible to do a type selection on them because as explained in Section 18.6, cyclic paths that can appear in a concrete execution context cannot be typed with a type-member declaration.

A difference between pDOT and the above calculi is that to ensure type soundness, paths in pDOT are normal form. This is necessary to ensure that each object has a name, as explained in Section 18.2.2. νObj and FS_{alg} achieve type safety in spite of reducing paths by allowing

field selection only on variables. This way, field selections always occur on named objects. Scalina does not require objects to be tied to names. In particular, its field selection rule E_Sel allows a field selection $\mathsf{new}\, T.a$ on an object if T contains a field definition $\{a = s\}$. The selection reduces to $s\,[\mathsf{new}\, T/\mathbf{this}]$, i.e. each occurrence of the self variable is replaced with a copy of $\mathsf{new}\, T$.

A second difference to pDOT is the handling of singleton types. In order to reason about a singleton type $p.\mathsf{type}$, νObj, FS_{alg}, and Scalina use several recursively defined judgments (membership, expansion, and others) that rely on analyzing the shape and well-formedness of the type that p expands to. By contrast, pDOT contains one simple Sngl-Trans rule that allows a path to inherit the type of its alias. On the other hand, pDOT has the shortcoming that singleton typing is not reflexive. Unlike in the above systems and in Scala, pDOT lacks a type axiom $\Gamma \vdash p\colon p.\mathsf{type}$. Such a rule would undermine the anti-symmetry of path aliasing which is essential to the safety proof.

None of the other calculi have to confront the problem of bad bounds. Unlike DOT, pDOT, and Scala, νObj and FS_{alg} do not support lower bounds of type members and have no unique upper and lower bounds on types. Scalina does have top and bottom types and supports bounds through interval kinds, but it avoids bad bounds by requiring types on which selection occurs to be concrete. In addition, it is unknown whether Scalina and FS_{alg} are sound.

Finally, the three type systems are nominal and class based, and include a large set of language features that are present in Scala. DOT is a simpler and smaller calculus that abstracts over many of the design decisions of the above calculi. Since DOT aims to be a base for experimentation with new language features, it serves well as a minimal core calculus for languages with type members, and the goal of pDOT is to generalize DOT to fully path-dependent types.

22.2 DOT-LIKE CALCULI

Amin, Moors, and Odersky, (2012) present the first version of a DOT calculus. It includes type intersection, recursive types, unique top and bottom types, type members with upper and lower bounds, and path-dependent types on paths of arbitrary length. This version of DOT has explicit support for fields (vals) and methods (defs). Fields must be initialized to variables, which prevents the creation of non-terminating paths (since that would require initializing fields to paths), but it also limits expressivity. Specifically, just like in DOT by Amin, Grütter, et al., (2016), path-dependent types cannot refer to nested modules because modules have to be created through methods, and method invocations cannot be part of a path-dependent type. The calculus is not type-safe, as illustrated by multiple counterexamples in the paper.

In particular, this version of DOT does not track path equality which, as explained in the paper, breaks preservation.

To be type-safe, DOT must ensure that path-dependent types are invoked only on terminating paths. A possible strategy to ensure a sound DOT with support for paths is to investigate the conditions under which terms terminate, and to impose these conditions on the paths that participate in type selections. To address these questions, Wang and Rompf, (2017) present a Coq-mechanized semantic proof of strong normalization for the $D_{<:}$ calculus. $D_{<:}$ is a generalization of System $F_{<:}$ with lower- and upper-bounded type tags and variable-dependent types. The paper shows that recursive objects constitute the feature that enables recursion and hence Turing-completeness in DOT. Since $D_{<:}$ lacks recursive objects, it is strongly normalizing. Furthermore, the lack of objects and fields implies that this version of $D_{<:}$ can only express paths that are variables.

Hong, Park, and Ryu, (2018) present πDOT, a strongly normalizing version of a $D_{<:}$ without top and bottom types but with support for paths of arbitrary length. πDOT keeps track of path aliasing through path-equivalence sets, and the paper also mentions the possibility of using singleton types to formalize path equality. Like the calculus by Wang and Rompf, (2017), this version of $D_{<:}$ is strongly normalizing due to the lack of recursive self variables. This guarantees that paths are acyclic. It also ensures that due to the lack of recursion elimination, reducing paths preserves soundness (unlike in pDOT, as explained in Section 18.2.2). πDOT comes with a non-mechanized soundness proof.

By contrast with these two papers, our work proposes a Turing-complete generalization with paths of arbitrary length of the full DOT calculus, which includes objects and type intersections.

22.2.1 *Other Related Languages and Calculi*

Scala's module system shares many similarities with languages of the ML family. Earlier presentations of ML module systems Dreyer, Crary, and Harper, (2003), Harper and Lillibridge, (1994), and Leroy, (1994) allow fine-grained control over type abstractions and code reuse but do not support mutually recursive modules, and separate the language of terms from the language of modules. *MixML* extends the essential features of these type systems with the ability to do both *hierarchical* and *mixin* composition of modules (Rossberg and Dreyer, 2013). The language supports *recursive* modules which can be packaged as *first-class values*. The expressive power of *MixML*'s module system, plus support for decidable type-checking requires a set of constraints on the *linking* (module mixin) operation that restrict recursion between modules, including a total order on *label paths*, and yields a complex type system that closely models actual implementations of ML.

Rossberg, Russo, and Dreyer, (2014) and Rossberg, (2018) address the inherent complexity of ML module systems by presenting encodings of an ML-style module language into System F_ω. The latter paper presents *1ML*, a concise version of ML that fully unifies the language of modules with the language of terms. However, both formalizations exclude recursive modules.

A type system that distinguishes types based on the runtime values of their enclosing objects was first introduced by Ernst, (2001) in the context of *family polymorphism*. Notably, family polymorphism is supported by *virtual classes*, which can be inherited and overriden within different objects and whose concrete implementation is resolved at runtime. Virtual classes are supported in the *Beta* and *gbeta* programming languages Ernst, (1999) and Madsen and Møller-Pedersen, (1989) (but not in Scala in which classes are statically resolved at compile time) and formalized by the *vc* and *Tribe* calculi Clarke et al., (2007) and Ernst, Ostermann, and Cook, (2006). Paths in *vc* are relative to this and consist of a sequence of out keywords, which refer to enclosing objects, and field names. To track path equality, *vc* uses a normalization function that converts paths to a canonical representation, and to rule out cyclic paths it defines a partial order on declared names. *Tribe*'s paths can be both relative or absolute: they can start with a variable, and they can intermix class and object references. The calculus uses singleton types to track path equality and rules out cyclic paths by disallowing cyclic dependencies in its inheritance relation.

A difference between pDOT and all of *vc*, *Tribe*, and the ML formalizations is that pDOT does not impose any orderings on paths, and fully supports recursive references between objects and path-dependent types. In addition, pDOT's ability to define type members with both lower and upper bounds introduces a complex source of unsoundness in the form of *bad bounds* (alas, the cost for its expressiveness is that pDOT's type system is likely not decidable). Yet, by being mostly structurally typed, without having to model initialization and inheritance, pDOT remains general and small. Finally, by contrast to the above, pDOT comes with a mechanized type-safety proof.

22.2.2 *Decidability*

As discussed in Chapter 8, typechecking the DOT calculus is conjectured to be undecidable. The open question of decidability of DOT needs to be resolved before we can consider decidability of pDOT.

We believe that pDOT does not introduce additional sources of undecidability into DOT. One feature of pDOT that might call this into question is singleton types. In particular, Stone and Harper, (2006) study systems of singleton kinds that reason about types with non-trivial reduction rules, yet it remains decidable which types reduce to the same normal form. The singleton types of both Scala and pDOT

are much simpler and less expressive in that only assignment of an object between variables and paths is allowed, but the objects are not arbitrary terms and do not reduce. Thus, the Scala and pDOT singleton types only need to track sequences of assignments. Therefore, although decidability of pDOT is unknown because it is unknown for DOT, the singleton types that we add in pDOT are unlikely to affect decidability because they are significantly less expressive than the singleton types studied by Stone and Harper.

23

CONCLUSION

The DOT calculus was designed as a small core calculus to model Scala's type system with a focus on path-dependent types. However, DOT can only model types that depend on variables, which significantly under-approximates the behaviour of Scala programs. Scala and, more generally, languages with type members need to rely on fully path-dependent types to encode the possible type dependencies in their module systems without restrictions. Until now, it was unclear whether combining the fundamental features of languages with path-dependent types, namely bounded abstract type members, intersections, recursive objects, and paths of arbitrary length is type-safe.

We propose pDOT, a calculus that generalizes DOT with support for paths of arbitrary length. The main insights of pDOT are to represent object identity through paths, to ensure that well-typed paths without cyclic aliasing always represent values, to track path equality with singleton types, and to eliminate type selections on cyclic paths through precise object typing. pDOT allows us to use the full potential of path-dependent types. pDOT comes with a type-safety proof and motivating examples for fully-path dependent types and singleton types that are mechanized in Coq.

BIBLIOGRAPHY

Amin, Nada, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki (2016). "The Essence of Dependent Object Types." In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pp. 249–272.

Amin, Nada, Adriaan Moors, and Martin Odersky (2012). "Dependent Object Types." In: *International Workshop on Foundations of Object-Oriented Languages (FOOL 2012)*.

Amin, Nada and Tiark Rompf (2017). "Type soundness proofs with definitional interpreters." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pp. 666–679. URL: http://dl.acm.org/citation.cfm?id=3009866.

Amin, Nada, Tiark Rompf, and Martin Odersky (2014). "Foundations of path-dependent types." In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pp. 233–249.

Amin, Nada and Ross Tate (2016). "Java and Scala's type systems are unsound: the existential crisis of null pointers." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 838–848.

Appel, Andrew W. and Amy P. Felty (2000). "A Semantic Model of Types and Machine Instructions for Proof-Carrying Code." In: *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 243–253.

Bell, CJ, Robert Dockins, Aquinas Hobor, and David Walker (2008). "Comparing semantic and syntactic methods in mechanized proof frameworks." In: *International Workshop on Proof-Carrying Code (PCC*. Citeseer.

Bruce, Kim B., Martin Odersky, and Philip Wadler (1998). "A Statically Safe Alternative to Virtual Types." In: *ECOOP'98 - Object-Oriented Programming, 12th European Conference*, pp. 523–549.

Clarke, Dave, Sophia Drossopoulou, James Noble, and Tobias Wrigstad (2007). "Tribe: a simple virtual class calculus." In: *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, pp. 121–134.

Cremet, Vincent, Francois Garillot, Sergueï Lenglet, and Martin Odersky (2006). "A Core Calculus for Scala Type Checking." In: *Mathe-*

matical Foundations of Computer Science, 31st International Symposium, Slovakia.

Documentation, Dotty (2018a). *Intersection Types*. URL: https://dotty.epfl.ch/docs/reference/intersection-types.html.

Documentation, Scala (2018b). *Paths*. URL: https://www.scala-lang.org/files/archive/spec/2.11/03-types.html#paths.

Dreyer, Derek, Karl Crary, and Robert Harper (2003). "A type system for higher-order modules." In: *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pp. 236–249.

Ernst, Erik (1999). "gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance." PhD thesis. Department of Computer Science, University of Aarhus, Århus, Denmark.

– (2001). "Family Polymorphism." In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, pp. 303–326.

Ernst, Erik, Klaus Ostermann, and William R. Cook (2006). "A virtual class calculus." In: *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pp. 270–282.

Giarrusso, Paolo, Léo Stefanesco, Amin Timany, and Lars Birkedal (2019). *Towards Semantic Type Soundness for Dependent Object Types and Scala with Logical Relations in Iris*. URL: https://www.dropbox.com/s/ljn89labnf9g5kp/beamer-intro-double-delft-2019-02-13-03.13.pdf.

Harper, Robert and Mark Lillibridge (1994). "A Type-Theoretic Approach to Higher-Order Modules with Sharing." In: *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pp. 123–137.

Hong, Jaemin, Jihyeok Park, and Sukyoung Ryu (2018). "Path Dependent Types with Path-equality." In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. Scala 2018. St. Louis, MO, USA, pp. 35–39. ISBN: 978-1-4503-5836-1.

Hu, Jason and Ondrej Lhoták (2019). "Undecidability of $D_{<:}$ and Its Decidable Fragments." In: *CoRR* abs/1908.05294. URL: http://arxiv.org/abs/1908.05294.

Hughes, John (1989). "Why Functional Programming Matters." In: *Comput. J.* 32.2, pp. 98–107.

Kabir, Ifaz and Ondřej Lhoták (2018). "κDOT: scaling DOT with mutation and constructors." In: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. ACM, pp. 40–50.

Leroy, Xavier (1994). "Manifest Types, Modules, and Separate Compilation." In: *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pp. 109–122.

Mackay, Julian, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Robert Cameron (2012). "Encoding Featherweight Java with assignment and immutability using the Coq proof assistant." In: *FTfJP 2012*, pp. 11–19.

Madsen, Ole Lehrmann and Birger Møller-Pedersen (1989). "Virtual Classes: A Powerful Mechanism in Object-Oriented Programming." In: *Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), New Orleans, Louisiana, USA, October 1-6, 1989, Proceedings*. Pp. 397–406.

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

Nieto, Abel (2017). "Towards algorithmic typing for DOT (short paper)." In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, pp. 2–7.

– (2018). *Scala with explicit nulls*. URL: https://github.com/abeln/dotty/wiki/scala-with-explicit-nulls.

Norris, Rob (2015). *Returning the "Current" Type in Scala*. URL: https://tpolecat.github.io/2015/04/29/f-bounds.html.

Odersky, Martin (2016). *Type projection is unsound #1050*. URL: https://github.com/lampepfl/dotty/issues/1050.

Odersky, Martin, Vincent Cremet, Christine Röckl, and Matthias Zenger (2003). "A Nominal Theory of Objects with Dependent Types." In: *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, pp. 201–224.

Odersky, Martin, Guillaume Martres, and Dmitry Petrashko (2016). "Implementing higher-kinded types in Dotty." In: *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016*, pp. 51–60.

Odersky, Martin and Matthias Zenger (2005a). "Independently extensible solutions to the expression problem." In: *FOOL 2005*. Vol. 12.

– (2005b). "Scalable component abstractions." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pp. 41–57.

Osvald, Leo, Grégory M. Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf (2016). "Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016,*

part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016, pp. 234–251.

Pierce, Benjamin C. (1992a). "Bounded Quantification is Undecidable." In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '92. Albuquerque, New Mexico, USA: ACM, pp. 305–315. ISBN: 0-89791-453-8.

– (1992b). "Bounded Quantification is Undecidable." In: *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, pp. 305–315.

– (2002). *Types and programming languages*. MIT Press. ISBN: 978-0-262-16209-8.

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

Rapoport, Marianna and Ondrej Lhoták (2016). "Mutable WadlerFest DOT." In: *CoRR* abs/1611.07610. arXiv: 1611.07610. URL: http://arxiv.org/abs/1611.07610.

Rapoport, Marianna and Ondřej Lhoták (2017). "Mutable WadlerFest DOT." In: *FTfJP 2017*, 7:1–7:6.

Reynolds, John C (1998). "Definitional interpreters for higher-order programming languages." In: *Higher-order and symbolic computation* 11.4, pp. 363–397.

Rompf, Tiark and Nada Amin (2015). "From F to DOT: Type Soundness Proofs with Definitional Interpreters." In: *CoRR* abs/1510.05216v1. URL: http://arxiv.org/abs/1510.05216v1.

– (2016a). "From F to DOT: Type Soundness Proofs with Definitional Interpreters." In: *CoRR* abs/1510.05216v2. URL: http://arxiv.org/abs/1510.05216v2.

– (2016b). "Type soundness for dependent object types (DOT)." In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 624–641.

Rossberg, Andreas (2018). "1ML — Core and modules united." In: *Journal of Functional Programming* 28, e22.

Rossberg, Andreas and Derek Dreyer (2013). "Mixin' Up the ML Module System." In: *ACM Trans. Program. Lang. Syst.* 35.1, 2:1–2:84.

Rossberg, Andreas, Claudio V. Russo, and Derek Dreyer (2014). "F-ing modules." In: *J. Funct. Program.* 24.5, pp. 529–607.

Scalas, Alceste and Nobuko Yoshida (2016). "Lightweight Session Programming in Scala." In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, 21:1–21:28.

Stone, Christopher A. and Robert Harper (2006). "Extensional equivalence and singleton types." In: *ACM Trans. Comput. Log.* 7.4, pp. 676–722.

Wang, Fei and Tiark Rompf (2017). "Towards Strong Normalization for Dependent Object Types (DOT)." In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, 27:1–27:25.

Wright, Andrew K. and Matthias Felleisen (1994). "A Syntactic Approach to Type Soundness." In: *Inf. Comput.* 115.1, pp. 38–94.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*".

Final Version as of October 9, 2019 ().