Dependent Object Types

THÈSE N^{O} 7156 (2016)

PRÉSENTÉE LE 12 DÉCEMBRE 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Nada AMIN

acceptée sur proposition du jury:

Prof. R. Guerraoui, président du jury
Prof. M. Odersky, directeur de thèse
Prof. A. Ahmed, rapporteuse
Prof. Ph. Wadler, rapporteur
Prof. V. Kuncak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

À l'Auvergnat

Acknowledgements

I thank Martin Odersky for supervising my graduate studies.
I thank Viktor Kuncak, Rachid Guerraoui, Amal Ahmed and Philip Wadler for agreeing to be part of my thesis jury.
I met Amal at the Oregon Programming Language Summer School in 2012, and she opened my eyes to a world of proofs beyond the canon.
I met Phil at POPL in Rome, where we escaped from the last session and ended up in the Zoo instead of the City. During that initial debacle, we bashed our heads against quotations of all kinds. At the end, we briefly talked about DOT and he mentioned already that it could be interesting if we could encode other features and calculi into it.
I thank Adriaan Moors, Samuel Grütter, Sandro Stucki and specially Tiark Rompf for joint work on this project.
I am grateful to Adriaan for warning me about "the DOT curse", so that I entered the project with a "Nutcracker" mindset happy to have a tough one to train on. This mindset proved essential through the ups and downs.
Samuel and Sandro offered interesting perspectives on the work. Samuel came with the fresh eyes of an undergraduate, and he also shares a wavelength with Martin, which helps percolate results up. Sandro offered the Parisian perspective of inter-deriving against pure type systems.
Tiark and I were officemates at EPFL, and he dragged me into the rabbit hole of generative programming as much as I dragged him into the rabbit hole of DOT. It's fair to say that he also dragged me out of "the DOT curse" on several occasions.
I thank the whole team at LAMP.
Many thanks to Sébastien Doeraene for editing the Résumé, and to Chantal van Vlaanderen for proofreading it – any remaining misplaced commas or shells are my own.
I thank Fabien Salvi and Danielle Chamberlain, Muriel Videlier and Natascha Fontana for ensuring the LAMP lab runs smoothly. Thanks to Nicolas Stucki for his help with the dead tree printing.
The initial design of DOT is due to Martin Odersky. Geoffrey Washburn, Adriaan Moors, Donna Malayeri, Samuel Grütter, Sandro Stucki and Tiark Rompf have contributed to its development.
For insightful discussions on this work, I thank Amal Ahmed, Jonathan Aldrich, Kim Bruce, William Byrd, Derek Dreyer, Joshua Dunfield, Sebastian Erdweg, Erik Ernst, Matthias Felleisen, François Garillot, Ronald Garcia, Paolo Giarrusso, Scott Kilpatrick, Grzegorz Kossakowski, Alexander Kuklev, Viktor Kuncak, Ondřej Lhoták, Fengyun Liu, Guillaume Martres, Dmitry Petrashko, Alex Potanin, Jon Pretty, Didier Rémy, Julien Richard-Foy, Lukas Rytz, Miles Sabin,

Acknowledgements

Ilya Sergey, Jeremy Siek, Josh Suereth, Ross Tate, André Videla, Eelco Visser, Philip Wadler and Jason Zaugg.
My graduate studies were funded by the European Research Council (ERC) under grant 587327 DOPPLER.
...

Genève, le 26 juillet 2016 N/A

Preface

What do you get if you boil Scala on a slow flame and wait until all incidental features evaporate and only the most concentrated essence remains? After doing this for 8 years we believe we have the answer: it's DOT, the calculus of dependent object types, that underlies Scala [1].

Lausanne, le 3 février 2016 Martin Odersky

[1] http://www.scala-lang.org/blog/2016/02/03/essence-of-scala.html

Abstract

A scalable programming language is one in which the same concepts can describe small as well as large parts. Towards this goal, Scala unifies concepts from object and module systems. In particular, objects can contain type members, which can be selected as types, called path-dependent types. Focusing on path-dependent types, we develop a type-theoretic foundation for Scala: the calculus of Dependent Object Types (DOT).

We derive DOT from System $F_{<:}$, in small steps: (1) in Translucent $F_{<:>}$, we add a lower bound to each type variable, in addition to its usual upper bound, (2) in System D, we turn each type variable into a regular term variable containing a type, (3) for a full subtyping lattice, we add intersection and union types, (4) for objects, we consolidate all values into records, (5) for objects that close over a self, we introduce a recursive type, binding a self term variable, (6) for recursive types, we first extend the theory in typing and then also in subtyping. Through this bottom-up exploration, we discover a sound, uniform yet powerful design for DOT.

We devise strategies and techniques for proving soundness that scale through this iterative step-by-step process: (1) "pushback" of subtyping transitivity or subsumption, to concisely capture inversion of subtyping or typing, (2) distinction between concrete vs. abstract context variables, to resolve tension between preservation of types vs. preservation of type abstractions, (3) and, specifically for big-step semantics, a type that closes over an environment, to relate context-dependent types across closures. While ultimately, we have developed sound models of DOT in both big-step and small-step operational semantics, historically, the shift to big-step semantics has been helpful in focusing the requirements. In particular, by developing a novel big-step soundness proof for System $F_{<:}$, calculi like System $D_{<:}$ emerge as straightforward generalizations, almost like removing artificial restrictions. Interesting in their own right, our type soundness techniques for definitional interpreters extend to mutable references without use of co-induction.

The DOT calculus finally grounds languages like Scala in firm theory. The DOT calculus helps in finding bugs in Scala, and in understanding feature interaction better as well as requirements. The DOT calculus serves as a good basis for future work which studies extensions or encodings on top of the core, bridging the gap from DOT to Dotty / Scala.

Keywords: Type System, Calculus, Scala

Résumé

Un langage de programmation est dit scalable lorsque les mêmes concepts peuvent décrire des parties aussi bien à petite que grande échelle. Tendant vers ce but, le langage Scala unifie des concepts de la programmation orientée objet et des systèmes de modules. En particulier, les objets peuvent contenir comme membres des types, qui peuvent être sélectionnés comme tels, appelés types chemin-dépendants. En portant notre attention sur les types chemin-dépendants, nous développons les fondements d'une théorie des types pour Scala : un système formel appelé Dependent Object Types (DOT).

Nous dérivons DOT par incréments à partir de System $F_{<:}$: (1) avec le système $F_{<:>}$ translucide, nous ajoutons une borne inférieure à chaque variable de type, en plus de l'usuelle borne supérieure, (2) avec le système D, nous transformons chaque variable de type en une variable de terme ordinaire contenant un type, (3) pour obtenir un treillis de sous-typage total, nous ajoutons des types intersection et union, (4) pour les objets, nous regroupons les valeurs dans des enregistrements, (5) pour les objets dotés d'une fermeture sur eux-mêmes, nous introduisons un type récursif, qui introduit au sein de l'objet une variable de terme représentant l'objet lui-même, (6) pour les types récursifs, nous étendons d'abord la théorie dans le cas sans sous-typage et ensuite avec sous-typage. À travers cette exploration de bas en haut, nous découvrons un design de DOT qui est logiquement correct, uniforme et toutefois puissant.

Nous élaborons des stratégies et techniques pour prouver la correction logique au travers de ce processus pas-à-pas : (1) le repoussement de la transitivité du sous-typage pour capturer de manière concise l'inversion du sous-typage ou du typage, (2) la distinction entre variables de contexte abstraites et concrètes afin de résoudre la tension entre préservation de types et préservation des abstractions de types, et (3) spécifiquement pour la sémantique naturelle à grands pas, un type qui fournit une fermeture sur un environnement pour mettre en relation les types dépendants du contexte à travers les fermetures. Bien que, finalement, nous ayons développé des systèmes formels de DOT à la fois à petits et grands pas, historiquement, la perspective de la sémantique naturelle à grands pas fut bénéfique pour identifier les exigences requises. En particulier, en développant une nouvelle preuve de correction logique pour le Système $F_{<:}$ avec une sémantique à grands pas, les systèmes comme $D_{<:}$ émergent tout naturellement par généralisation, comme si des restrictions artificielles étaient levées. Intéressantes en soi, nos techniques de preuve de correction logique pour les interpréteurs définitionnels s'étendent aux références muables sans recours à la co-induction.

Le DOT-calcul place finalement les langages tels que Scala sur une solide base théorique. Il aide à mettre en évidence certains bugs dans Scala, et à mieux comprendre les interactions

entre fonctionnalités du langage, ainsi que les exigences formelles. Le DOT-calcul fournit un bon cadre pour étudier à l'avenir des extensions et encodages sur base de ce calcul formel, rapprochant la théorie de DOT de la pratique de Dotty / Scala.

Mots-clés : Système de types, Calcul formel, Scala…

Contents

List of Figures

List of Tables

1 Introduction

Scala combines functional programming and object-oriented programming. It aims to be *scalable* in that small as well as large parts can be expressed with the same building blocks. Towards this goal, Scala unifies concepts from object and module systems. In particular, objects can contain type members, which can be selected as types, called path-dependent types. Focusing on path-dependent types, we develop, and prove sound, a type-theoretic foundation for Scala: the calculus of Dependent Object Types (DOT).

1.1 Previous Work on Formalizing Scala

Previous attempts to model Scala focused on the whole language, leading to complex models. The *ν*Obj calculus [Odersky et al., 2003] has been the first attempt at a theory of names – however, the model is complex because it strives to encompass all features of Scala. Featherweight Scala [Cremet et al., 2006] has focused on algorithmic decidability. Scalina [Moors et al., 2008] has developed a theory of kind soundness – kind as opposed to type.

In this thesis, we focus on a purified model, with path-dependent types at the center. Instead of full paths, i.e. chains of immutable field selections, we focus on just variables.

In contrast to other typed object languages – like the canonical Featherweight Java [Igarashi et al., 2001] – Dependent Object Types does not have a global class table. Nominality emerges from type ascription and path-dependent types. As we will elaborate, assigning a less precise type member gives rise to translucency, a flexible form of nominality.

In contrast to *ν*Obj, DOT makes no attempt at formalizing the linearization of traits of Scala. Instead, DOT relies on commutative operators for the least upper bound and the greatest lower bound in the subtyping lattice, the syntactic union and intersection types – as is standard in classical type theory [Dunfield, 2014].

1.2 Contributions

We make the following contributions:

- Chapter 2 presents the motivation of this work, giving examples of Scala and DOT counterparts.
- Chapter 3 presents a step-by-step development from System $F_{<:}$ to DOT in a functional small-step store-less style.
- Chapter 4 presents a small-step store-full DOT, which is best suited for efficiency, as it avoids the need for a syntactic comparison of values.
- Chapter 5 presents a new "mental" toolbox to reason about programming languages. We argue for an alternative methodology that complements Wright and Felleisen [1994], using big-step instead of small-step operational semantics while maintaining the strong results of the syntactic approaches.

While each of Chapters 3, 4, 5 cover roughly the same final model, there are differences, and the formal equivalence has been left for future work. In a nutshell, the store-less model is most flexible for encodings, because values can partake in paths, which simplifies encoding both generative and applicative ML functors – generative by fresh identifiers through ascription and applicative by using values in paths. We have taken for granted that the big-step and small-step store-full versions are inter-derivable, once again leaving this as future work from this thesis, though Chapter 5 already sketches an inter-derivability argument for System $F_{<:}$.

Chapters 2, 3, and 5 are independent, while Chapter 4 builds on Chapter 2, Section 2.3.

1.3 Prior Publications

Prior publications of our team include Amin et al. [2012, 2014], Rompf and Amin [2015], Amin et al. [2016], Rompf and Amin [2016], Amin and Rompf [2017].

- Chapter 2 is based partially on Amin et al. [2014], Rompf and Amin [2016].
- Chapter 3 is based partially on Amin et al. [2016].
- Chapter 4 is based partially on Rompf and Amin [2016].
- Chapter 5 is based partially on Amin and Rompf [2017].

2 Motivating Examples

This thesis develops a sound design for Dependent Object Types (DOT).

As we will see, DOT is a small calculus, yet its type system is expressive while uniform. DOT models:

- first-class modules, because objects can have type members,
- path-dependent types, because these type members can be selected as types,
- translucency and variance, because type members are lower- as well as upper- bounded,
- recursive types, because an object is a record where "this" can be used in both enclosing terms and types,
- dependent function types, because the result type may contain type selections on type members of the parameter,
- nominality as type ascription, because a type member with tighter bounds is a subtype of the same type member with wider (e.g. more abstract) bounds,
- full subtyping lattice, because intersection and union types are always well-formed.

2.1 Example of Scala-bility

Scala is a functional language that expresses central aspects of modules as first-class terms and types. It identifies modules with *objects* and signatures with *traits*.

For example, in Scala, we can define a signature for a heap module in the style of Okasaki [1999]:

```
1 trait HeapModule {
2   type Heap
3   type Elem
```

```
  def ord: Ordering[Elem]
  def empty: Heap
  def isEmpty(h: Heap): Boolean
  def insert(x: Elem, h: Heap): Heap
  def findMin(h: Heap): Elem
  def deleteMin(h: Heap): Heap
  def merge(h1: Heap, h2: Heap): Heap
}
```

A heap module encapsulates method members on heap operations, as well as *type members* for the type of the heap and the type of the elements. We can implement such a heap module, giving the previously abstract type `Heap` a concrete representation as a hierarchical list of ranked nodes:

```
trait BinomialHeapModule extends HeapModule {
  type Rank = Int
  case class Node(x: Elem, r: Rank, c: List[Node])
  override type Heap = List[Node]
  // ... implementation ...
}
```

Given two instances of such a heap module, *path-dependent types* enable us to refer to the contained type members as types, ensuring that we do not merge heaps stemming from distinct modules.

```
val m1: HeapModule = ...
val m2: HeapModule = ...
▶ m1.merge(m1.empty, m2.empty)
// error: type mismatch;
// found   : m2.Heap
// required: m1.Heap
// m1.merge(m1.empty, m2.empty)
//                    ^
```

2.2 In the Zoo

We give an example of path-dependent types in Scala, inspired by Igarashi and Pierce [2002].

2.2.1 Nominality through Abstract Type Members

An animal eats food of a certain type that depends on the animal. We model an animal with a `trait Animal` that has an abstract type member `type Food`. The variable `a` denotes the self (i.e. `this`) object, in the scope defining the trait `Animal`. So we can refer to the abstract type member `type Food` as a type through a type selection: `a.Food` – `a.Food` is a *path-dependent type*, in general, a chain, starting with an immutable variable, of immutable field selections, ending with a type selection. Notice that the type selection `a.Food` can appear both covariantly, as in the method `def gets`, where it is the return type, and contravariantly, as in the method `def eats`, where it is a parameter type.

```
trait Animal { a =>
  type Food
  def eats(food: a.Food): Unit = {}
  def gets: a.Food
}
```

A cow is an animal that eats grass. A lion is an animal that eats meat. We can model these two animals by refining the `trait Animal`.

```
trait Grass
trait Meat
trait Cow extends Animal with Meat {
  type Food = Grass
  def gets = new Grass {}
}
trait Lion extends Animal {
  type Food = Meat
  def gets = new Meat {}
}
```

Now, let's have `leo` the Lion eat `milka` the Cow. This is possible, because `Cow` is a subtype of `Meat`:

```
val leo = new Lion {}
val milka = new Cow {}
leo.eats(milka)
```

The lion `leo` can also eat whatever it gets:

```
leo.eats(leo.gets)
```

On the other hand, if we have `lambda` the (unknown) Animal, then we cannot feed it `milka` the Cow. After all, `lambda` the Animal might well be a Cow – or even, `milka` itself:

```
val lambda: Animal = milka
lambda.eats(milka) // type mismatch -- found: Cow -- required: lambda.Food
```

Still, `lambda` can eat whatever it gets:

```
lambda.eats(lambda.gets)
```

The path-dependent type arising from the type member `Food` drives this example. In the trait `Animal`, the type `Food` is a fully abstract type member – with a lower bound of bottom (the uninhabited type) and an upper bound of top (the type of all values). In the traits `Cow` and `Lion`, we refine the type member `Food` – this is allowed, as long as the bounds in the subtype are not wider than the bounds in the supertype. On the one hand, the Scala type system admits that `leo` the Lion eats `milka` the Cow, because `leo.Food`, appearing in a contravariant position, is lower-bounded by the type `Meat` and the type `Cow` is a subtype of the type `Meat`. On the other hand, the Scala type system refuses that `lambda` the unknown Animal eats `milka` the Cow – because `lambda.Food`, being fully abstract, is lower-bounded by bottom – and of course, the type `Cow` is not a subtype of bottom. Still, `lambda` the (unknown) Animal can eat whatever it gets, because that has type `lambda.Food` – through subtyping

reflexivity, we get nominality as an emergent property of type members and path-dependent types. That is, we get the ability to refer to the *name* `lambda.Food` without having to know its *structure*.

2.2.2 Composition and Refinement through Subtyping Lattice

Note that even though we have used traits and inheritance in this example, these mechanisms are not essential here for the subtyping relations. Later, we show a variant that uses only path-dependent types and achieves type abstraction through ascription (up-cast) instead of inheritance.

To keep the core calculus simple, we deliberately choose not to model inheritance and mixin composition. Still, we want the core calculus to be rich enough so that we could express inheritance and mixing composition by translating them to the core. For this requirement, we find it important to also explore a calculus with a complete subtyping lattice, such that meets and joins are defined for all types. With such a core calculus, programming patterns involving greatest lower bounds and least upper bounds would be naturally handled. For example, two animals might want to share a bite:

```
def share(a1: Animal)(a2: Animal)(bite: a1.Food with a2.Food) {
  a1.eats(bite)
  a2.eats(bite)
}
```

Two lions, `leo` and `simba`, can indeed share a bite:

```
val simba = new Lion {}
share(leo)(simba)(leo.gets) // ok
```

However, `lambda` the (unknown) Animal cannot share a bite with `leo` the Lion:

```
share(leo)(lambda)(leo.gets) // error: type mismatch
                             // found   : Meat
                             // required: leo.Food with lambda.Food
```

Even without a full subtyping lattice, we will see that just adding unrestricted refinements breaks some intuitive properties that we expect of subtyping, such as transitivity and environment narrowing. The Scala compiler has ad-hoc restrictions to prevent these issues on an implementation level, but nobody has a proof that these tweaks are sufficient.

In this work, we are not concerned with traits, classes, or implementation inheritance at all, so we present a version that does not use these features in example 2.1. This restricted subset of Scala maps directly to DOT, and even μDOT [Amin et al., 2014]. Each trait maps to the type it represents and each concrete trait in addition to a constructor that builds objects of that type. We express purely nominal subtyping relations through the presence of certain type members (e.g. `IsMeat`). Thus, we use type members to represent *all* nominal types, not only some.

In Scala, function literals of type `A⇒B` are represented as objects of a trait `Function1[A,B]`

```
type Meat = {
  type IsMeat = Any
}
type Grass = {
  type IsGrass = Any
}
type Animal = { a =>
  type Food
  def eats(food: a.Food): Unit
  def gets: a.Food
}
type Cow = {
  type IsMeat = Any
  type Food = Grass
  def eats(food: Grass): Unit
  def gets: Grass
}
type Lion = {
  type Food = Meat
  def eats(food: Meat): Unit
  def gets: Meat
}

def newMeat  = new {
  type IsMeat = Any
}
def newGrass = new {
  type IsGrass = Any
}
def newCow   = new {
  type IsMeat = Any
  type Food = Grass
  def eats(food: Grass) = ()
  def gets = newGrass
}
def newLion = new {
  type Food = Meat
  def eats(food: Meat) = ()
  def gets = newMeat
}
val milka = newCow
val leo = newLion
leo.eats(milka)
```

Table 2.1 – Zoo example expressed in Scala subset that maps to DOT

that implement a single method `def apply(x:A):B`. This desugaring carries over directly to DOT.

2.2.3 Type Refinement

Subtyping allows us to treat a type as a less precise one. Scala provides a dual mechanism that enables us to create a more precise type by *refining* an existing one.

We could use refinement in the definition of `Cow`:

```
type Cow = Animal { a =>
  type Food <: Grass
  def gets: Grass
}
```

Here, we are expressing `Cow` as a more precise version of `Animal`.

This example can easily be expressed without refinement, because the definition of `Animal` is concrete. If the type `Animal` were more abstract, for example only upper-bounded by instead of aliasing its defining record, then the definition of `Cow` expressed without refinement would be lacking, as it would not capture the intention that `Cow` is a subtype of `Animal`.

When refining an abstract type, we speak of "open refinement". As an aside, if the type `Cow` was an open refinement of an abstract type `Animal`, then it would also be difficult to create a new `Cow` from scratch, since one would not know how to conform to its `Animal` nature.

We can think of refinement as a special case of intersection, between a type and a record type which defines additional members or refines already defined members. One difference is that a refinement may refer to members of the type being refined, as if the "self" variable closes over both types of the intersection, while one usually expects each type of an intersection to be well-formed independently.

For example, we can express the type `Hoarder` as a refinement of the type `Animal`, with an additional member that refers to the abstract type member `Food` of `Animal`:

```
type Hoarder = Animal { a =>
  def stash(food: a.Food): Unit
}
```

To express the type `Hoarder` by desugaring the refinement into an intersection type, we need a "self" variable (here `a`) to close over the intersection type, in order to refer to the abstract type member `Food` of `Animal`:

```
type Hoarder = { a =>
  Animal ∧ {
    def stash(food: a.Food): Unit
  }
}
```

2.2.4 Emulating Nominal Constructors

For the type `Animal` to be nominal instead of structural, we can upcast its defining package. However, if the type member `Animal` has an abstract lower bound, then the package is responsible for providing "constructor" methods to create a nominal `Animal` out of structural pieces. Such emulated constructors can be either sealing or open, allowing nominal subclasses, e.g. a nominal `Cow` is a nominal `Animal` too. In the sketch, we again make no attempt to model inheritance, just the subtyping relations.

```
trait AnimalPackage {
  type Animal <: AnimalU
  type AnimalU = { a =>
    type Food
    def eats(food: a.Food): Unit
    def gets: a.Food
  }
  def newAnimal(a: AnimalU): Animal
  def newSubAnimal[T](a: AnimalU ∧ T): Animal ∧ T
}
val p: AnimalPackage = new AnimalPackage { p =>
  type Animal = AnimalU
  override def newAnimal(a: AnimalU): Animal = a
  override def newSubAnimal[T](a: AnimalU ∧ T): Animal ∧ T = a
}
val lambda: p.Animal = p.newAnimal(new { a =>
  type Food = Grass
  def eats(food: a.Food): Unit = {}
```

```
19    def gets: a.Food = newGrass
20  })
21  trait CowPackage { pc =>
22    type Cow <: p.Animal ∧ pc.CowDelta
23    type CowDelta = { type IsMeat = Any; type Food = Grass }
24    type CowU = p.AnimalU ∧ pc.CowDelta
25    def newCow(c: CowU): Cow
26    def newSubCow[T](c: CowU ∧ T): Cow ∧ T
27  }
28  val pc: CowPackage = new CowPackage { pc =>
29    type Cow = p.Animal ∧ pc.CowDelta
30    def newCow(c: CowU): Cow = p.newSubAnimal[pc.CowDelta](c)
31    def newSubCow[T](c: CowU ∧ T): Cow ∧ T = p.newSubAnimal[pc.CowDelta ∧ T
        ](c)
32  }
33  val milka: pc.Cow = pc.newCow(new { a =>
34    type IsMeat = Any
35    type Food = Grass
36    def eats(food: a.Food): Unit = {}
37    def gets: a.Food = newGrass
38  })
```

2.3 Types in Scala and DOT

Scala is a large language that combines features from functional programming, object-oriented programming and module systems. Scala unifies many of these features (e.g. objects, functions, and modules) [Odersky and Rompf, 2014] but still contains a large set of distinct kinds of types. These can be broadly classified [Odersky, 2013] into *modular* types:

```
1                      named type: scala.collection.BitSet
2                   compound type: Channel with Logged
3                    refined type: Channel { def close(): Unit }
```

And *functional* types :

```
1               parameterized type: List[String]
2                 existential type: List[T] forSome { type T }
3               higher-kinded type: List
```

While this variety of types enables programming styles appealing to programmers with different backgrounds (Java, ML, Haskell, ...), not all of them are essential. Further unification and an economy of concepts can be achieved by reducing functional to modular types as follows:

```
1            class List[Elem] {} ⇝ class List { type Elem }
2                   List[String] ⇝ List { type Elem = String }
3     List[T] forSome { type T } ⇝ List
4                           List ⇝ List
```

This unification is the main thrust of the calculi of the DOT family. A further thrust is to replace Scala's compound types `A with B` with proper intersection types `A ∧ B`. It is worth noting that the correspondence between higher-kinded types and existential types suggested above

is not exact, in particular with respect to well-kindedness constraints and partial application to type arguments. Before presenting DOT in a formal setting in Section 4.1, we introduce the main ideas with some high-level programming examples.

Objects and First Class Modules In Scala and in DOT, every piece of data is conceptually an object and every operation a method call. This is in contrast to functional languages in the ML family that are stratified into a core language and a module system. Below[1] is an implementation of a functional list data structure:

```
val listModule = new { m =>
  type List = { this =>
    type Elem
    def head(): this.Elem
    def tail(): m.List ∧ { type Elem <: this.Elem }
  }
  def nil() = new { this =>
    type Elem = Bot
    def head() = error()
    def tail() = error()
  }
  def cons[T](hd: T)(tl: m.List ∧ { type Elem <: T }) =
    new { this =>
    type Elem = T
    def head() = hd
    def tail() = tl
  }
}
```

The actual `List` type is defined inside a container `listModule`, which we can think of as a first-class module. In an extended DOT calculus `error` may signify an 'acceptable' runtime error or exception that aborts the current execution and transfers control to an exception handler. In the case that we study, without such facilities, we can model the abortive behavior of `error` as a non-terminating function, for example `def error(): Bot = error()`.

Nominality through Ascription In most other settings (e.g. object-oriented subclassing, ML module sealing), nominality is enforced when objects are declared or constructed. Here we can just assign a more abstract type to our list module:

```
type ListAPI = { m =>
  type List <: { this =>
    type Elem
    def head(): this.Elem
    def tail(): m.List ∧ { type Elem <: this.Elem }
  }
  def nil(): List ∧ { type Elem = Bot }
  def cons[T]: T =>
    m.List ∧ { type Elem <: T } =>
```

[1] Code listings of this chapter are in Scala augmented with intersection and union types.

```
10        m.List ∧ { type Elem <: T }
11  }
```

Types `List` and `Elem` are abstract, and thus exhibit nominal as opposed to structural behavior. Since modules are just objects, it is perfectly possible to pick different implementations of an abstract type based on runtime parameters.

```
1  val mapImpl = if (size < 100) ListMap else HashMap
```

Polymorphic Methods In the code above, we have still used the functional notation `cons[T](...)` for parametric methods. We can desugar the type parameter T into a proper method parameter x with a modular type, and at the same time desugar the multi-argument function into nested anonymous functions:

```
1  def cons(x: { type A }) = ((hd: x.A) => ... )
```

References to T are replaced by a path-dependent type $x.A$. We can further desugar the anonymous functions into objects with a single `apply` method:

```
1  def cons(x: { type A }) = new {
2    def apply(hd: x.A) = new {
3      def apply(tl: m.List ∧ { type Elem <: x.A }) =
4        new { this =>
5        type Elem = x.A
6        def head() = hd
7        def tail() = tl
8      }}}
```

More generally, with these desugarings, we can encode the polymorphic λ-calculus System F [Girard, 1972, Reynolds, 1974] and its extension with subtyping, System $F_{<:}$ [Cardelli et al., 1994], by defining a pointwise mapping over types and terms:

$$X \rightsquigarrow \texttt{x.A}$$
$$\forall(X <: S)T \rightsquigarrow \texttt{\{ def apply(x: \{ type A <: S \}): T \}}$$
$$\Lambda(X <: S)t \rightsquigarrow \texttt{new \{ def apply(x: \{ type A <: S \})= t \}}$$
$$t[U] \rightsquigarrow \texttt{t.apply(\{ type A = U \})}$$

Apart from the translation of type variables into term variables with type members, another difference is that $F_{<:}$ supports only upper bounded type variables, while Scala and DOT allow both lower and upper bounds. This gives rise to very fine grained choices of translucency, i.e. how much implementation details are revealed for a partially abstract type, and also allows a precise modelling of covariance and contravariance. We will see an example of a lower bounded type member `{ type Config >: T}` below.

It is easy to see that while path-dependent types can encode System F, the reverse is not likely true. For example, the following function is expressible in Scala and DOT but does not have a System F equivalent: $\lambda(x:\{\texttt{ type A }\}).x$

Path-Dependent Types Let us consider another example to illustrate path-dependent types: a system of services, each with a specific type of configuration object. Here is the abstract interface:

```
type Service {
  type Config
  def default: Config
  def init(data: Config): Unit
}
```

We now create a system consisting of a database and an authentication service, each with their respective configuration types:

```
type DBConfig { def getDB: String }
type AuthConfig { def getAuth: String }
val dbs = new Service {
  type Config = DBConfig
  def default = new DBConfig { ... }
  def init(d:Config) = ... d.getDB ...
}
val auths = new Service {
  type Config = AuthConfig
  def default = new AuthConfig { ... }
  def init(d:Config) = ... d.getAuth ...
}
```

We can initialize `dbs` with a new `DBConfig`, and `auths` with a new `AuthConfig`, but not vice versa. This is because each object has its own specific `Config` type member and thus, `dbs.Config` and `auths.Config` are distinct *path-dependent* types. Likewise, if we have a service `lam: Service` without further refinement of its `Config` member, we can still call `lam.init(lam.default)` but we cannot create a `lam.Config` value directly, because `Config` is an abstract type in `Service`.

Intersection and Union Types At the end of the day, we want only one centralized configuration for our system, and we can create one by assigning an intersection type:

```
val globalConf: DBConfig ∧ AuthConfig = new {
  def getDB = "myDB"
  def getAuth = "myAuth"
}
```

Since `globalConf` corresponds to both `DBConfig` and `AuthConfig`, we can use it to initialize both services:

```
dbs.init(globalConf)
auths.init(globalConf)
```

But we would like to abstract even more.

With the `List` definition presented earlier, we can build a list of services (using `::` as syntactic sugar for `cons`):

```
val services = auths::dbs::nil()
```

We define an initialization function for a whole list of services:

```
def initAll[T](xs:List[Service { type Config >: T }])(c: T) =
  xs.foreach(_ init c)
```

Which we can then use as:

```
initAll(services)(globalConf)
```

How do the types play out here? The definition of `List` and `cons` makes the type member `Elem` covariant. Thus, the type of `auths::dbs::nil()` corresponds to

```
List ∧ {
  type Elem = Service ∧ {
    type Config: (DBConfig ∧ AuthConfig) .. (DBConfig ∨ AuthConfig)
  }
}
```

The notation `type T: S..U` denotes that type member `T` is lower bounded by `S` and upper bounded by `U`. Here, `Config` is lower bounded by the greatest lower bound (the intersection, $\wedge$) and upper bounded by the least upper bound (the union, $\vee$) of the types `DBConfig` and `AuthConfig`. Since the `Config` member is lower bounded by `DBConfig ∧ AuthConfig`, passing an object of that type to `init` is legal.

Records and Refinements as Intersections Subtyping allows us to treat a type as a less precise one. Scala provides a dual mechanism that enables us to create a more precise type by *refining* an existing one.

```
type PersistentService = Service { a =>
  def persist(config: a.Config)
}
```

To express the type `PersistentService` by desugaring the refinement into an intersection type, we need a "self" variable (here `a`) to close over the intersection type, in order to refer to the abstract type member `Config` of `Service`:

```
type PersistentService = { a => Service ∧ {
  def persist(config: a.Config)
}}
```

Our variant of DOT uses intersection types also to model the type of records with multiple type, value, or method members:

```
{ def foo(x: S1): U1    ;   def bar(y: S2): U2 } ⇝
{ def foo(x: S1): U1 } ∧{ def bar(y: S2): U2 }
```

With this encoding of records, we benefit again from an economy of concepts.

Soundness? Our aim for DOT is a strong type soundness result of the form "well-typed programs don't go wrong", i.e. to show the total absence of runtime errors. Clearly we cannot expect the same strong result to hold for Scala, as Scala includes unsafe features like `null` values (which inhabit any object type) and unsafe casts. Especially `null` values make a strong soundness result impossible as they violate the key *canonical forms* property: a value that has a given type might not actually be a proper object of this type at runtime, but it might instead be `null`.

A more realistic notion of soundness needs to include the possibility of *benign* runtime failures. A desirable guarantee in this setting is that modules that do not introduce `null` values cannot be *blamed* for `NullPointerExceptions`, and modules that do not perform unsafe casts cannot be blamed for `ClassCastExceptions`.

In this light, it is important to note that Scala has some fundamental soundness bugs related to path-dependent types, even with respect to these weaker notions of soundness. Some manifestations of these bugs have been known for a long time (the Scala issue tracker lists unfixed soundness problems dating back at least to 2008 [Washburn, 2008]), but they have been properly understood only recently, during the course of this work. We discuss the general issue next, with a particular example related to `null` values [Amin and Tate, 2016].

The key soundness challenge with path-dependent types is to avoid "bad bounds", i.e. type members like `type T: S..U` where `S` is *not* a subtype of `U`, e.g. `type T: String..Int`. If an object `x` were allowed to have such a type member, we could use it to safely treat a `String` first as type `x.T`, and then as `Int`, but of course this would be unsound: any attempt to treat a `String` as an `Int` will lead to a cast exception at runtime!

The formalization effort behind DOT has shown that preventing such "bad bounds" is harder than previously thought (see Section 4.2.3), and that the mechanisms implemented in the Scala compiler are not sufficient. Introducing type `Bot` in simpler systems with path-dependent types is already difficult [Amin et al., 2014], and it is especially important that `Bot` is uninhabited. More precisely, it is necessary to enforce constraints on type members and their bounds at object creation time (see Section 4.2.5). The canonical forms property ensures that an object that is supposed to have a type member `T: S..U` was indeed created with type member `T` set to a type *inbetween* `S` and `U` – thus providing constructive evidence that `S <: U`. However, `null` does behave roughly like a `Bot` value, incompatible with canonical forms: `null` is assignable to any other type, but it does not have any members on which constraints could be enforced. Thus it is clear that we can create a problem with type selections `x.T` when `x` is `null`. The Scala compiler has ad-hoc checks to prevent a direct use of `null`, but those are easy to thwart via indirection as shown in the following example, which fails with a runtime cast exception:

```
1 trait A { type L >: Any}
```

```
def id1(a: A, x: Any): a.L = x
val p: A { type L <: Nothing } = null
def id2(x: Any): Nothing = id1(p, x)
id2("boom!")
```

The problem is that `id2` is a "safe" cast from `Any` to `Nothing`, which we cannot support in a sound language. The subtyping lattice collapses through a `null` path with bad bounds. The issue is described in more detail by Amin and Tate [2016], who show that – remarkably – the same problem also exists in Java, even though Java does not have path-dependent types.

Since this example uses an explicit `null`, one might be tempted to dismiss the issue by saying "`null` is unsound anyways". But what is disturbing is that the result is not a `NullPointerException` but a `ClassCastException`, even though no "unsafe" casts are used. Also, no pointer is dereferenced, so while a `NullPointerException` could be more easily rationalized (e.g. as a "type-level" null deferencing), it would still be unexpected, even for weak notions of soundness.

The use of `null` is also not crucial, as we show in Section 4.5.1. The issue is deeper, and the take-away is that paths always need to refer to proper objects, which are guaranteed to have "good bounds" by construction. As we discuss in Section 4.2, attempting to incorporate "good bounds" into well-formedness of types is not sufficient, as "good bounds" are not preserved under narrowing. Null values, unevaluated `lazy vals`, mutable variables, as well as arbitrary (non-terminating) term dependencies need to be excluded from paths. The core DOT calculus presented in this work does not include these features, but we discuss potential avenues of extension in Section 4.5.2.

Our work on DOT helped identify and clarify these soundness issues, and identify potential fixes. In the case of `null`, there are no easy fixes: the DOT effort has underlined the necessity for future versions of Scala and similar languages to consider nullability explicitly in the type system. For a detailed discussion, we refer the reader to bug reports in Scala and Dotty, notably issue 50 of Dotty [2].

[2] https://github.com/lampepfl/dotty/issues/50

3 From F to DOT Small-Step Store-Less

In this chapter, we derive DOT in small steps:

- (1) We start with System $F_{<:}$.
- (2) In Translucent $F_{<:>}$, we add a lower bound to each type variable, in addition to its usual upper bound.
- (3) In System D, we turn each type variable into a regular term variable containing a type.
- (4) For a full subtyping lattice, we add intersection and union types.
- (5) For DOT, we add recursive objects and types.
  - For objects, we consolidate all values into records.
  - For objects that close over a self, we introduce a recursive type, binding a self term variable.
  - For recursive types, we first extend the theory in typing and then also in subtyping.

Through this bottom-up exploration, we discover a sound, uniform yet powerful design for DOT.

In this chapter, we sketch the steps to give an overall feel for the derivation; while in Chapter 4, we focus on the final step, for Dependent Object Types.

3.1 Background: $F_{<:}$

The TAPL textbook – *Types and Programming Languages* [Pierce, 2002] – has more background information on STLC (Chapter 9), Subtyping (Chapter 15), System F (Chapter 23) and System $F_{<:}$ (Chapter 26).

In the remaining of this section, we highlight the formal definitions and properties of System $F_{<:}$, with an eye towards future extensions. Our starting point is the soundness proof given in the TAPL textbook.

3.1.1 Motivation

System $F_{<:}$ combines System F and subtyping.

System F extends the simply-typed lambda-calculus (STLC) with polymorphism by adding universal quantification over types. Like STLC, System F has function type $T \to T$ as type, and term abstraction $\lambda x : T.t$ and application $t\ t$ as terms. Formally, the function type is introduced by term abstraction and eliminated by term application. On top of STLC, System F adds type variable X and universal type $\forall X.T$ as types, and type abstraction $\Lambda X.t$ and type application $t\ [T]$ as terms. Formally, the universal type is introduced by type abstraction and eliminated by type application.

For the motivating examples, we use records and numbers like `{a=0,b=0}` of type `{a:Nat,b:Nat}`. We do this so that it feels more tangible that things can go wrong. In the formalism, we only introduce records at the very end when we switch from syntactic lambas to objects.

Below, the expression `twice [Nat]` is a type application: it instantiates the type variable parameter `X` of type abstraction `twice` with the primitive type `Nat`. So the term `twice [Nat]` has type `(Nat → Nat) → Nat → Nat`, and so the term `(twice [Nat]) succ` is a valid term application, where the term `succ` is the primitive successor function of type `Nat → Nat`.

(System F + naturals)

```
1   id = ΛX.λx:X.x
2 ▸ id : ∀X.X→X
3   twice = ΛX.λf:X→X.λx:X.f (f x)
4 ▸ twice : ∀X.(X→X)→X→X
5   plus2 : twice [Nat] succ
6 ▸ plus2 : Nat → Nat
7   id2 = ΛX. twice [X] id
8 ▸ id2 : ∀X.X→X
9   twice' = λf:(∀X.X→X).ΛX.twice X (f [X])
10 ▸ twice' = (∀X.X→X)→X
11  id2' = twice' id
12 ▸ id2' : ∀X.X→X
```

Subtyping is a relation on types that enables *subsumption*: relaxing typing up to subtyping.

For example, if a function takes a record type `{a:Nat}` as parameter, it should be safe to pass it a record term `{a=0,b=0}`. Even though the argument type `{a:Nat,b:Nat}` is not identical to the parameter type `{a:Nat}`, the argument type is a subtype of the parameter type: `{a:Nat,b:Nat} <: {a:Nat}`.

In System $F_{<:}$, polymorphism and subtyping interact. Compared to System F, universal quantification is generalized to upper-bounded quantification. System $F_{<:}$ has a top type $\top$, which is a supertype of any type. So, with an upper bound of top $\top$, one can recover universal quantification from upper-bounded quantification. In addition, upper-bounded quantification makes it possible to constrain polymorphism, exploiting the constraints afforded by subtyping (e.g. `succ(x.a)`) and the precision afforded by polymorphism (e.g. `.orig_x.b`).

(System $F_{<:}$ + records + naturals)

```
1   id = ΛX<:⊤.λx:X.x
2 ▸ id : ∀X<:⊤.X→X
3   p = ΛX<:{a:Nat}.λx:X.{orig_x=x, s=succ(x.a)};
4 ▸ p : ∀X<:{a:Nat}.X → {orig_x:X, s:Nat}
5   n = (p [{a:Nat,b:Nat}] {a=0,b=0}).orig_x.b
6 ▸ n: Nat
```

3.1.2 Formal Design

The formal model of System $F_{<:}$ is presented in Figure 3.1.

Note that we use the "full" (as opposed to "kernel") variant of System $F_{<:}$, where rule (S-ALL) is contravariant (as opposed to constant) in the type variable parameter. Though undecidable, the "full" variant is more natural in analogy to the rule (S-ARROW), and more fitted to our extensions, since, starting with $D_{<:}$ (Section 3.3), we unify type and term abstractions, and hence also rules (S-ALL) and (S-ARROW).

Note also that the presentation is non-algorithmic. Typing has an explicit rule for subsumption, rule (T-SUB), while subtyping has explicit rules for transitivity and reflexivity, rules (S-TRANS) and (S-REFL). An alternative, taken in the POPLMark Challenge [Aydemir et al., 2005], is to not define these properties axiomatically, but instead prove that they are admissible, relaxing the model as needed. For subtyping, it is possible to show that the rules (S-TRANS) and (S-REFL) are admissible, if the model relaxes rule (S-TVAR) to rule (SA-TRANS-TVAR) and additionally defines reflexivity on type variables only, rule (SA-REFL-TVAR):

$$\frac{X <: U \in \Gamma \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T} \text{(SA-TRANS-TVAR)}$$

$$\Gamma \vdash X <: X \text{(SA-REFL-TVAR)}$$

For type safety of System $F_{<:}$, either approach to transitivity, axiomatic vs. admissible, succeeds. The axiomatic approach has the advantage that we can use transitivity to prove any other property (e.g. narrowing), but the disadvantage that we need more complex arguments in

$F_{<:}$

Syntax

$t ::=$	**terms**:	$S, T, U ::=$	**types**:
x	variable	X	type variable
$\lambda x : T.t$	abstraction	$\top$	top type
$t\ t$	application	$T \to T$	function type
$\Lambda X <: T.t$	type abstraction	$\forall X <: T.T$	univeral type
$t\ [T]$	type application	$\Gamma ::=$	**contexts**:
$v ::=$	**values**:	$\emptyset$	empty context
$\lambda x : T.t$	abstraction value	$\Gamma, x : T$	term variable binding
$\Lambda X <: T.t$	type abstraction value	$\Gamma, X <: T$	type variable binding

Evaluation $\boxed{t \longrightarrow t'}$

$$(\lambda x : T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \text{ (E-AppAbs)} \qquad (\Lambda X <: T_{11}.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \text{ (E-TAppTAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \text{ (E-App1)} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \text{ (E-App2)} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \text{ (E-TApp)}$$

Subtyping $\boxed{\Gamma \vdash S <: T}$

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

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

$$\Gamma \vdash S <: \top \text{ (S-Top)}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \text{ (S-TVar)}$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ T_1 <: T_2}{\Gamma \vdash S_1 \to T_1 <: S_2 \to T_2} \text{ (S-Arrow)}$$

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

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12}\ ,\ t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \text{ (T-App)}$$

$$\frac{\Gamma, X <: T_1 \vdash t_2 : T_2}{\Gamma \vdash \Lambda X <: T_1.t_2 : \forall X <: T_1.T_2} \text{ (T-TAbs)}$$

$$\frac{\Gamma \vdash t_1 : \forall X <: T_{11}.T_{12}\ ,\ T_2 <: T_{11}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \text{ (T-TApp)}$$

$$\frac{\Gamma \vdash t : S\ ,\ S <: T}{\Gamma \vdash t : T} \text{ (T-Sub)}$$

Figure 3.1 – $F_{<:}$

inversion lemmas. The admissible approach has the advantage that inversion lemmas are simpler, but the disadvantage that we need more work upfront in showing that transitivity is admissible. In particular, there is a mutual dependency in proving transitivity and narrowing. For System $F_{<:}$, it is possible to set up a mutual induction because transitivity only requires narrowing for strictly smaller middle types.

In this thesis, we use the axiomatic approach, because we are not concerned with decidability. In any case, in later extensions, the line between axiomatic vs. admissible becomes blurry, because it is not always straightforward to find a relaxed model that admits transitivity without outright defining it. So we develop principled strategies for inversions, in order to scale to extensions.

3.1.3 Safety

Lemma 1 (Weakening)**.** A judgement still holds in an extended, well-formed context.

1. If $\Gamma \vdash t : T$ and $\Gamma, x : U$ is well formed, then $\Gamma, x : U \vdash t : T$
2. If $\Gamma \vdash t : T$ and $\Gamma, X <: U$ is well formed, then $\Gamma, X <: U \vdash t : T$
3. If $\Gamma \vdash S <: T$ and $\Gamma, x : U$ is well formed, then $\Gamma, x : U \vdash S <: T$.
4. If $\Gamma \vdash S <: T$ and $\Gamma, X <: U$ is well formed, then $\Gamma, X <: U \vdash S <: T$.

Lemma 2 (Narrowing)**.** A judgement still holds in a context where a type variable is narrowed, i.e. by tightening its upper-bound.

1. If $\Gamma, X <: Q, \Delta \vdash S <: T$ and $\Gamma \vdash P <: Q$, then $\Gamma, X <: P, \Delta \vdash S <: T$.
2. If $\Gamma, X <: Q, \Delta \vdash t : T$ and $\Gamma \vdash P <: Q$, then $\Gamma, X <: P, \Delta \vdash t : T$.

Lemma 3 (Substitution preserves typing)**.** If $\Gamma, x : Q, \Delta \vdash t : T$ and $\Gamma \vdash q : Q$, then $\Gamma, \Delta \vdash [x \mapsto q]t : T$.

Lemma 4 (Type substitution preserves subtyping)**.** If $\Gamma, X <: Q, \Delta \vdash S <: T$ and $\Gamma \vdash P <: Q$, then $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]S <: [X \mapsto P]T$.

Lemma 5 (Type substitution preserves typing)**.** If $\Gamma, X <: Q, \Delta \vdash t : T$ and $\Gamma \vdash P <: Q$, then $\Gamma, [X \mapsto P]\Delta \vdash [X \mapsto P]t : [X \mapsto P]T$.

Lemma 6 (Inversion of Subtyping)**.** For any subtyping derivation $\Gamma \vdash S <: U$, transitivity can be pushed back so that the topmost rule is not the transitivity rule (S-TRANS), assuming rule (SA-TRANS-TVAR) instead of rule (S-TVAR).

In TAPL, inversion of subtyping is defined by enumerating all cases for $S <: U$ left to right, and then right to left. Martin Odersky suggested instead the more principled approach of transitivity pushback, where we transform the model with built-in (axiomatic) transitivity

into the relaxed model without topmost transitivity. In later developments, we do not need to prove the models equivalent by eliminating built-in transitivity entirely, instead it suffices to show that transitivity can be eliminated at the top of the derivation tree – pushed back into deeper derivations that do not affect the structural argument of inversion.

Lemma 7 (Inversion of Value Typing)**.**

1. If $\Gamma \vdash \lambda x : S_1 . s_2 : T$ and $\Gamma \vdash T <: U_1 \to U_2$, then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that $\Gamma, x : S_1 \vdash s_2 : S_2$ and $\Gamma \vdash S_2 <: U_2$.
2. If $\Gamma \vdash \lambda x <: S_1 . s_2 : T$ and $\Gamma \vdash T <: \forall X <: U_1 . U_2$, then $\Gamma \vdash U_1 <: S_1$ and there is some S_2 such that $\Gamma, x <: S_1 \vdash s_2 : S_2$ and $\Gamma, X <: U_1 \vdash S_2 <: U_2$.

Theorem 1 (Preservation)**.** If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

Lemma 8 (Canonical Forms)**.**

1. If v is a closed value of type $T_1 \to T_2$, then v has the form $\lambda x : S_1 . t_2$.
2. If v is a closed value of type $\forall X <: T_1 . T_2$, then v has the form $\Lambda X <: S_1 . t_2$.

Theorem 2 (Progress)**.** If t is a closed well-typed term, then either t is a value or else there is some t' with $t \longrightarrow t'$.

Theorem 3 (Type-Safety)**.** If t is a closed well-typed term, $\emptyset \vdash t : T$, then either t is a value or else there is some t' with $t \longrightarrow t'$ and $\emptyset \vdash t' : T$.

3.1.4 Perspective

The soundness of $F_{<:}$ balances transitivity, narrowing and inversion of subtyping: narrowing needs transitivity, inversion of subtyping needs to eliminate transitivity. Because type variables are only upper-bounded, they can never be in the middle of a subtyping chain between two structural types. Indeed, the only way we can get $S <: X <: U$, is if $S = X$ through (S-REFL), and so $S <: U$ follows immediately. Already adding a bottom type $\bot$ makes the argument a bit more involved, but it can still be teased out [Pierce, 1997]. In the next section, we will not only add a bottom type $\bot$ but also lower bounds, enabling type variables to partake in arbitrary subtyping constraints.

3.2 Translucency: $F_{<:>}$

Translucent $F_{<:>}$ (pronounced "Fsubsup") extends System $F_{<:}$ so that quantification is both lower- and upper- bounded. It also adds a bottom type $\bot$, so that upper-bounded and universal quantification can both be recovered.

The word "translucent" is borrowed from the literature on ML module systems [Harper and Lillibridge, 1994], suggesting a fine-grained notion of bound, between fully transparent and fully opaque.

3.2.1 Motivation

The syntax $X : S..U$ specifies that the type variable X is lower-bounded by type S and upper-bounded by type U. Like in System F, we have a syntactic marker to distinguish between term application $t\ t$ and type application $t\ [T]$. In the example, thanks to the lower bound on the type variable `X`, it is possible to call `f {a=0,b=0}` (on line 1) even though its parameter type is the type variable `X`. Thanks to polymorphism, the `orig` field keeps the more specific type of `f`, i.e. a function type with fewer requirements on the parameter type. Here, this means that the function `pa.orig` can be applied to records that do not have a `b` field.

(Translucent $F_{<:>}$ + records + naturals)

```
1   p = ΛX:{a:Nat,b:Nat}..⊤.λf:X→⊤.{orig=f, r=(f {a=0,b=0})};
2 ▸ p : ∀X:{a:Nat,b:Nat}..⊤.(X→⊤)→{orig:X→⊤, r:⊤}
3   pa = p [{a:Nat}] (λx:{a:Nat}. x.a);
4 ▸ pa : {orig:{a:Nat}→⊤, r:⊤}
```

Of course, we can also combine lower- and upper- bounded quantification, which enables fine-grained generic specifications.

(Translucent $F_{<:>}$ + records + naturals)

```
1   q = ΛY:⊥..{a:Nat}.λy:Y.ΛX:Y..⊤.λf:X→⊤.{orig_y=y,orig_f=f,r=(f y),s=y.a};
2 ▸ q : ∀Y:⊥..{a:Nat}.Y → ∀X:Y..⊤.(X→⊤) → {orig_y:Y,orig_f:X→⊤,r:⊤,s:Nat}
3   p = q [{a:Nat,b:Nat}] {a=0,b=0};
4 ▸ p :∀X:{a:Nat,b:Nat}..⊤.(X→⊤)→{orig_y:{a:Nat,b:Nat},orig_f:X→⊤,r:⊤,s:Nat}
5   pa = p [{a:Nat}] (λx:{a:Nat}. x.a);
6 ▸ pa : {orig_y:{a:Nat,b:Nat}, orig_f:{a:Nat}→⊤,r:⊤,s:Nat}
```

An abstract type can be both lower- and upper- bounded, which enables fine-grained choices of translucency.

(Translucent $F_{<:>}$ + records + naturals)

```
1   p = ΛX:{a:Nat,b:Nat}..{a:Nat}.λf:X→X.(f {a=0,b=0}).a
2 ▸ p : ∀X:{a:Nat,b:Nat}..{a:Nat}.(X → X) → Nat
3   n = p [{a:Nat}] (λx:{a:Nat}. {a=succ(x.a)})
4 ▸ n : Nat
```

Thanks to the lower bound on `X`, we can apply the function `f` to the record `{a=0,b=0}`, because even though we don't know the exact type of `X`, we know that `{a=0,b=0}` satisfies type `X`. Thanks to the upper bound on `X`, we can select the field `a` on the result of `X`, because we know that the type `X` has at least such a field.

For universal types, translucency means that we can constrain polymorphism, and exploit those constraints in the implementation. For existential types, translucency means that we can choose how much implementation details to reveal.

Already, subtyping is user-extensible. For example, on can organize type parameters first by universal quantification, and then by specifying additional constraints as "witness" type parameters.

(Translucent $F_{<:>}$ – lenient checking of bounds)

```
1  ΛA:⊥..⊤. ΛB:⊥..⊤. ΛC:⊥..⊤. ΛX1:A..B. ΛX2:B..C. ...
```

In the example, the type parameters `A`, `B`, `C` are universally quantified while the "witness" type parameters `X1`, `X2` adds subtyping constraints: `A <: B <: C`. Within a type abstraction, the subtyping lattice could collapse due to such constraints:

(Translucent $F_{<:>}$ – lenient checking of bounds)

```
1  ΛX:⊤..⊥. ...
2  ΛX:⊥..⊥. ΛY:⊤..X. ...
3  ΛX:⊥..⊥. ΛY:⊤..⊤. ΛZ:Y..X. ...
```

3.2.2 Formal Design

The formal model of Translucent $F_{<:>}$ is presented in Figure 3.2. The syntax changes are summarized in Table 3.1.

The syntax for type variable bounds is changed from $X <: U$ to $X : S..U$, where S is the lower bound and U the upper bound. Upper-bounded quantification à la System $F_{<:}$ can be recovered by encoding $X <: U$ with an opaque lower-bound $X : \bot..U$. Similarly, universal quantification à la System F can be recovered by encoding a universally quantified type variable X with opaque bounds $X : \bot..\top$.

$F_{<:}$	in $F_{<:>}$	Encoding	/	Generalization
$\Lambda X <: U.t$	$\rightsquigarrow$	$\Lambda X : \bot..U.t$	$\rightsquigarrow$	$\Lambda X : S..U.t$
$\forall X <: U.T$	$\rightsquigarrow$	$\forall X : \bot..U.T$	$\rightsquigarrow$	$\forall X : S..U.T$

Table 3.1 – from $F_{<:}$ to $F_{<:>}$

In typing, we ensure that an applied type is in between the lower- and upper- bounds, rule (T-TAPP). The rule in System $F_{<:}$ ensures that the applied type T is a subtype of the upper-bound type U: $T <: U$ for a type abstraction term $t = \Lambda X <: U.T$ applied to a type T: $t\,[T]$. Here, for a type abstraction term $t = \Lambda X : S..U.T$, the rule ensures that the applied type T is a supertype

$F_{<:>}$ – Δ based on $F_{<:}$ (3.1)

Syntax

$t ::=$		**terms**:	
	x	variable	
	$\lambda x : T.t$	abstraction	
	$t\ t$	application	
	$\Lambda X <: S..U\ T.t$	type abstraction	
	$t\ [T]$	type application	
$v ::=$		**values**:	
	$\lambda x : T.t$	abstraction value	
	$\Lambda X <: S..U\ .t$	type abstraction value	

$S, T, U ::=$		**types**:
	X	type variable
	$\top$	top type
	$\bot$	bottom type
	$T \to T$	function type
	$\forall X <: S..U\ .T$	univeral type
$\Gamma ::=$		**contexts**:
	$\emptyset$	empty context
	$\Gamma, x : T$	term variable binding
	$\Gamma, X <: S..U$	type variable binding

Evaluation $\boxed{t \longrightarrow t'}$

$$(\lambda x : T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$(\Lambda X <: S_{11}..U_{11}\ .t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAppTAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{(E-App1)} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ \text{(E-App2)} \qquad \frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]}\ \text{(E-TApp)}$$

Subtyping $\boxed{\Gamma \vdash S <: T}$

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

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

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

$$\Gamma \vdash \bot <: U \quad (\text{S-Bot})$$

$$\frac{X <: S..U \in \Gamma \qquad \Gamma \vdash S <: U}{\Gamma \vdash X <: U}\ \text{(S-TVar)}$$

$$\frac{X <: S..U \in \Gamma \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: X}\ (\text{S-TVarLow})$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ T_1 <: T_2}{\Gamma \vdash S_1 \to T_1 <: S_2 \to T_2}\ \text{(S-Arrow)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 <: U_1\ ,\ S_2 <: U_2 \\ \Gamma \vdash\ S_1 <: S_2\ ,\ U_2 <: U_1 \\ \Gamma, X <: S_2..U_2\ \vdash T_1 <: T_2\end{array}}{\Gamma \vdash \forall X <: S_1..U_1\ .T_1 <: \forall X <: S_2..U_2\ .T_2}\ \text{(S-All)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}\ \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12}\ ,\ t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}\ \text{(T-App)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_1 <: U_1 \\ \Gamma, X <: S_1..U_1\ \vdash t_2 : T_2\end{array}}{\Gamma \vdash \Lambda X <: S_1..U_1\ .t_2 : \forall X <: S_1..U_1\ .T_2}\ \text{(T-TAbs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \forall X <: S_{11}..U_{11}\ .T_{12} \\ \Gamma \vdash\ S_{11} <: T_2\ ,\ T_2 <: U_{11}\end{array}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}}\ \text{(T-TApp)}$$

$$\frac{\Gamma \vdash t : S\ ,\ S <: T}{\Gamma \vdash t : T}\ \text{(T-Sub)}$$

(Δ alternative enforcing "good" bounds in types)

Figure 3.2 – $F_{<:>}$

of the lower-bound S and a subtype of the upper-bound U: $S <: T <: U$.

In subtyping, a type variable $X : S..U$ is naturally a supertype of its lower-bound and a subtype of its upper-bound: $S <: X <: U$. Compared to System $F_{<:}$, that a type variable on the right can partake in subtyping, via rule (S-TVarLow), raises questions: Do we need to ensure "good bounds" ($S <: U$)? What do we do about subtyping transitivity?

Recall that, in System $F_{<:}$, the admissible approach relies on proving narrowing and transitivity together, while the axiomatic approach on inverting subtyping. This setting with variables on both sides of subtyping breaks the mutual induction metric of the admissible approach and the inversion of the axiomatic approach.

In any case, we still have two options, depending on whether we want to enforce "good bounds" – axiomatic vs. admissible is more blurry henceforth.

Enforcing "Good Bounds" The first option is to enforce "good bounds" for universal types in subtyping and typing. Then, we formulate inversion of subtyping as pushing back transitivity, so that it is not the last rule in a derivation. Transitivity is only eliminated from the last derivation step, but is allowed in deeper sub-derivations. This still requires relaxing the model so that subtyping type variables has some built-in slack. Even then, the proof to transform an arbitrary subtyping derivation into one not ending with transitivity requires two passes: the first pass builds a chain from the left type to the right type via a chain of of middle men consisting of zero or more type variables; the second pass stitches the derivation together assuming "good bounds" or reflexivity for the middle men. Because of the setup, type variables that partake in non-reflexive subtyping must have good bounds. In subtyping, "good bounds" are thus enforced in rules (S-TVar), (S-TVarLow) and (S-All).

Tolerating "Bad Bounds" The second option is to remark that we only need inversion of subtyping in a context without type variables, in which case, inversion is easy. So we don't need to enforce "good bounds": the lattice can collapse via a type variable with bad bounds, but this can only happen in a context that is unrealizable, since a type application, in rule (T-TApp), ensures that the instantiated type is between the lower- and upper- bound of the variable. So we can just prove the easy case, and adapt the type safety proof accordingly. This does not affect the final type safety result, but it means proving a weaker form of preservation: for an empty context (like for progress) instead of an arbitary context.

Furthermore, we need to set up inversion of typing values so that in the case for term and type abstractions, we only need to pushback transitivity at the outermost level. For this, we define a directly invertible relation for value typing, "possible types", and show that is closed under subtyping in an empty context. Then, we can show that empty-context value typing implies invertible value typing by using the pushback in the subsumption case.

Once we do this, we can just get away without the transitivity pushback, and just rely on this

invertible relation for value typing, “possible types”.

Though the second option, which only proves preservation in an empty context, seems less satisfying, it is the one that will scale to extensions. Once we add intersection types, it will not be possible to check “good bounds” statically and a priori. Insisting on “good bounds” would break several monotonic properties, including narrowing and full subtyping lattice.

Semantically, the two options are different. The first one – which rigidly enforces “good bounds” a priori – only allows translucent bounds that are admissible in the original subtyping theory. The second one – which does not enforce “good bounds” – gives rise to user-defined subtyping theories under type abstraction. It is only when instantiated, that witnesses for “good bounds” have to be provided.

3.2.3 Example Derived

In the example 3.2, the function `p` produces a result of type `P`, which is consumed as types `A` and `B` by the functions `a` and `b`. The type variable `X` and `Y` provide evidence for the constraints that type `P` can be seen as types `A` and `B`, respectively.

We show part of the proof tree using the lenient rules. The rules checking “good bounds” would not accept this example, because (T-ABS) would not admit the abstraction over the constraints `X` and `Y`.

We show the derivation for giving the term `p u` the type `A`, which explains why the application `a (p u)` succeeds, given that the term `a` expects an argument of type `A` not `P`. After all the abstractions are introduced, the context contains all the parameter declarations.

```
1 ΛA:⊥..⊤. λa:A → ⊤.
2 ΛB:⊥..⊤. λb:B → ⊤.
3 ΛP:⊥..⊤. ΛX:P..A. ΛY:P..B.
4 λp:⊤ → P.
5 λu:⊤. λs:⊤ → ⊤ → ⊤.
6 s (a (p u)) (b (p u))
```

$$\Gamma = \begin{array}{l} A : \bot..\top, a : A \to \top, \\ B : \bot..\top, b : B \to \top, \\ P : \bot..\top, X : P..A., Y : P..B, \\ p : \top \to P, \\ u : \top, s : \top \to \top \to \top \end{array}$$

$$\dfrac{\Gamma \vdash p\ u : P \qquad \dfrac{\dfrac{X : P..A \in \Gamma}{\Gamma \vdash P <: X}\ \text{(S-TVarLow)} \qquad \dfrac{X : P..A \in \Gamma}{\Gamma \vdash X <: A}\ \text{(S-TVar)}}{\Gamma \vdash P <: A}\ \text{(S-Trans)}}{\Gamma \vdash p\ u : A}\ \text{(T-Sub)}$$

Table 3.2 – $F_{<:>}$ example derived

3.2.4 Safety

Following the second option, we deal with "bad bounds" by restricting preservation to an empty context $\Gamma = \emptyset$.

Theorem 4 (Preservation)**.** If $\emptyset \vdash t : T$ and $t \longrightarrow t'$, then $\emptyset \vdash t' : T$.

We define invertible value typing, also known as "possible types": $v :: T$.

Definition 1 (Possible Types)**.** Invertible value typing $v :: T$ is defined by the following rules:

1. $v :: \top$.
2. If $x : S_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_2 <: S_1, T_1 <: T_2$ then $\lambda x : S_1 . t_1 :: S_2 \to T_2$.
3. If $X : S_1..U_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_1 <: S_2, U_2 <: U_1$ and $X : S_2..U_2 \vdash T_1 <: T_2$ then $\Lambda X : S_1..U_1 . t_1 :: \forall X : S_2..U_2 . T_2$.

Lemma 9 (Subtyping Closure aka Widening of Possible Types)**.** If $v :: T$ and $\emptyset \vdash T <: U$ then $v :: U$.

Lemma 10 (Value Typing imples Possible Types)**.** If $\emptyset \vdash v : T$ then $v :: T$.

Now we can prove inversion of value typing and canonical forms via (direct) inversion of possible types.

The mechanized soundness proof is available from http://sound-small-step-fsubsup.namin.net.

3.2.5 Perspective

To sum up, enabling both lower and upper bounds for type variables poses two challenges: (1) dealing with bad bounds, (2) dealing with type variables as middle men in subtyping.

For (1) bad bounds, we can choose to allow them, and only check conformance during type applications. This means that user-defined subtyping theories may be inconsistent (under a type abstraction that is unsatisfiable).

For (2) type variables as middle men, we observe that for type safety we only need to invert subtyping in a context empty of type variables, to circumvent the issue entirely.

3.3 Type in Term, Term in Type: D, $D_{<:}$, $D_{<:>}$

3.3.1 Motivation

System D steps towards DOT by unifying terms and types in abstraction and application. It conflates quantification over terms and types by providing only abstraction over a term (the usual $\lambda x : T.t$), which introduces a dependent function type ($\forall x : T_1.T_2$), where the return type T_2 might depend on the parameter term x. How so? System D provides a new term construct for a term that holds a type, and a new type construct to select this type member from a term as a type.

3.3.2 Formal Design

We can build System D either from System $F_{<:}$, giving rise to System $D_{<:}$ where type members are either aliases (fully transparent) or just upper-bounded, or from System $F_{<:>}$, giving rise to System $D_{<:>}$ in which type members are both lower- and upper- bounded. Here in Figure 3.3, we show the rules for System $D_{<:>}$ because they are more uniform. The syntax changes from $F_{<:>}$ to $D_{<:>}$ are summarized in Table 3.3.

The new term construct is $\{A = T\}$ which creates a term value that holds a type T. Mnemonically, the type is held at a label – for now, a unique label A. This term construct introduces the type $\{A : S..U\}$. Elimination is at the type-level through a *path-dependent type* or *type selection*: given a term "path" p of type $\{A : S..U\}$, the type selection $p.A$ is a type. Semantically, subtyping gives a type selection $p.A$ its meaning: in the lattice, it is in between its lower bound and upper bound, $S <: p.A <: U$.

$F_{<:>}$	in $D_{<:>}$	Encoding	/	Generalization
$\Lambda X : S..U.t$	$\rightsquigarrow$	$\lambda x : \{A : S..U\}.t$	$\rightsquigarrow$	$\lambda x : T'.t \;/\; \{A = T\}$
$\forall X : S..U.T$	$\rightsquigarrow$	$\forall x : \{A : S..U\}.T$	$\rightsquigarrow$	$\forall x : T'.T \;/\; x.A$

Table 3.3 – from $F_{<:>}$ to $D_{<:>}$

The question now is what should we allow as "paths", i.e. what terms should we allow in type selections? One constraint is that substitution must preserve the syntactic and semantic validity of paths. Another constraint is that if we allow non-normal paths, we also have to consider relating "paths" across evaluation. Here, we consider only paths that are normal form, avoiding the second issue altogether.

One option, which fits nicely with the final object-oriented approach of DOT is to use a store for values in order for all paths to be identifiers, referring either to abstract variables bound in a context (i.e. under a lambda) or to a concrete variable bound in a store. Then, "values" actually reduce to fresh store identifiers.

Another option is simply to allow both variables and values (e.g. "good" normal forms) in paths, in which case, the identity of values is structural. To preserve syntactic validity of paths

$D_{<:>}$

Syntax

$t ::=$	**terms**:	$S, T, U ::=$	**types**:
x	variable	$\top$	top type
$\{A = T\}$	type tag	$\bot$	bottom type
$\lambda x : T.t$	abstraction	$\{A : S..U\}$	type of type tag
$t\ t$	application	$\forall x : S.U$	dependent function type
$v ::=$	**values**:	$p.A$	type selection
$\lambda x : T.t$	abstraction value	$\Gamma ::=$	**contexts**:
$\{A = T\}$	type tag value	$\emptyset \mid \Gamma, x : T$	variable bindings
$p ::=$	**paths**:		
x	variable		
v	value		

Evaluation $\boxed{t \longrightarrow t'}$

$$(\lambda x : T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

Subtyping $\boxed{\Gamma \vdash S <: T}$

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

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

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

$$\Gamma \vdash \bot <: U \quad \text{(S-Bot)}$$

$$\frac{\Gamma \vdash p : \{A : S..U\}}{\Gamma \vdash S <: p.A} \quad \text{(S-TSel1)}$$

$$\frac{\Gamma \vdash p : \{A : S..U\}}{\Gamma \vdash p.A <: U} \quad \text{(S-TSel2)}$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ U_1 <: U_2}{\Gamma \vdash \{A : S_1..U_1\} <: \{A : S_2..U_2\}} \quad \text{(S-Typ)}$$

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

Typing $\boxed{\Gamma \vdash t : T}$

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

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

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : \forall x : T_1.T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \forall x_1 : T_{11}.T_{12}\ ,\ t_2 : T_{11} \qquad x_1 \notin fv(T_{12})}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash t_1 : \forall x_1 : T_{11}.T_{12}\ ,\ p_2 : T_{11}}{\Gamma \vdash t_1\ p_2 : [x_1 \mapsto p_2]T_{12}} \quad \text{(T-AppDep)}$$

$$\frac{\Gamma \vdash t : S\ ,\ S <: T}{\Gamma \vdash t : T} \quad \text{(T-Sub)}$$

Figure 3.3 – $D_{<:>}$

under substitution, it also means that the odd type selection $(\lambda x : T.t).A$ should be a valid type, syntactically and semantically, albeit not a very useful one. This option fits nicely with the functional roots of System D, and makes it easier to formally prove an embedding from System $F_{<:}$ to System $D_{<:}$. Thus, we will use this option here and consider store identifiers for the full DOT version in Chapter 4, because it fits better with recursive types and object identities (though even there, it is not necessary).

3.3.3 Safety

For type selections in subtyping, the (S-TSEL1) and (S-TSEL2) rules delegate to typing for the path term:

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

When the path is precisely a type tag value, we can define a "tight" variant:

$$\Gamma \vdash T <: \{A = T\}.A <: T \quad \text{(S-TSEL-TIGHT)}$$

Tight selection is natural for "type tag" values, and it should suffice thanks to subtyping transitivity. However, it complicates substitution, replacing an abstract variable with a concrete value, because for variables, it's natural to allow subsumption on the path, even just to get to the right type shape (a type of "type tag"). Hence, for separation of concerns between substitution and narrowing lemmas, it's easier to work with only non-tight selections at the outset. For "possible types" however, tight selection is preferable in order to prove closure (aka widening) more easily.

Hence, we define both the usual non-tight subtyping <: and also tight subtyping <<. We can easily show that tight subtyping implies non-tight subtyping. We can also easily show tight closure of "possible types". Now, we can reconnect the layers by showing that non-tight subtyping in an empty context implies tight subtyping. As another layer, it's helpful to specify "shallow" and "deep" possible types. The "shallow" possible types does not check ∀-types beyond syntax, and is used for bootstrapping lemmas involving typing in subtyping.

These tweaks in layers bring us back to soundness proof for $F_{<:>}$. As before, we prove subtyping closure of, and value typing implies, and inversions using, "possible types".

Definition 2 (Tight Subtyping)**.** Let tight subtyping $S << U$ be defined as $\emptyset \vdash S <: U$ except using rule (S-TSEL-TIGHT) instead of rule (S-TSEL), whenever in empty context.

Definition 3 (Possible Types)**.** Invertible value typing $v :: T$ is defined by the following rules:

1. $v :: \top$.
2. If $S << T$ and $T << U$ then $\{A = T\} :: \{A : S..U\}$.

3. If $v :: S$ then $v :: \{A = S\}.A$.

4. For "deep" variant: If $x : S_1 \vdash t_1 : T_1$ and $\emptyset \vdash S_2 <: S_1$ and $x : S_2 \vdash T_1 <: T_2$ then $\lambda x : S_1.t :: \forall x : S_2.T_2$.

5. For "shallow" variant", roughly preserve shape: $\lambda x : S_1.t :: \forall x : S_2.T_2$.

Lemma 11 (Tight Subtyping Closure aka Widening of Possible Types)**.** If $v :: T$ and $T << U$ then $v :: U$.

Lemma 12 (Non-Tight Implies Tight)**.** If $\emptyset \vdash S <: U$ then $S << U$. If $\emptyset \vdash v : T$ then "shallow" $v :: T$.

Lemma 13 (Value Typing imples Possible Types)**.** If $\emptyset \vdash v : T$ then "deep" $v :: T$.

The mechanized soundness proof is available from http://sound-small-step-dsubsup.namin.net.

3.3.4 Perspective

From type variables to type members and type selections, the key design options concern well-formedness: when is a path-dependent type syntactically and semantically valid? A key requirement is that substitution must preserve syntactic validity of path-dependent types, so we need an operational semantics where substitution operates only on path-able terms. For soundness, we also need to ensure substituted terms have only type members with "good bounds". A key observation is that pushback is only needed for canonical forms, in fully concrete context.

3.4 Full Subtyping Lattice: $D_\diamond$

For System $D_\diamond$ (pronounced "D diamond"), we add subtyping lattice to System $D_{<:>}$. Once we have intersection types, we can also easily introduce records.

3.4.1 Motivation

A full subtyping lattice is useful for type inference, remedying a current brittleness in Scala.

Scala currently lacks a full subtyping lattice, because greatest lower bounds and least upper bounds do not always exist. Here is such an example in Scala:

```
trait A { type T <: A }
trait B { type T <: B }
trait C extends A with B { type T <: C }
trait D extends A with B { type T <: D }
```

The least upper bound of types `C` and `D` does not have a finite representation because the type member `T` can be refined arbitrarily:

```
  val cond: Boolean
  val o: A with B{type T <: A with B{type T <: A with B/*..*/} =
    if (cond) (new C{}) else (new D{})
```

On the other hand, the type inference engine must arbitrarily settle eagerly on a finite type to represent the least upper bound. This approximation can be a source of inefficiency, brittleness and impredictability:

```
val i = if (cond) (new C{}) else (new D{})  // type inferred
val i1 = (i : A with B)                     // ok
val i2 = (i : A with B{type T <: A with B}) // ok
val i3 = (i : A with B{type T <: A with B{type T <: A with B}})
 // error: type mismatch;
 // found   : A with B{type T <: A with B}
 // required: A with B{type T <: A with B{type T <: A with B}}
```

If the core calculus had classical intersection and union types, then the type inference engine could simply lazily construct the least upper bound of types `C` and `D` as its union `C ∨ D`.

3.4.2 Formal Design

Thanks to our lenient design, which tolerates bad bounds in types, it is straightforward to extend the calculus to a full subtyping lattice, with commutative intersection and union types, which are always syntactically and semantically valid. Thus, the intersection and the union of two types construct their greatest lower bound and least upper bound, respectively.

A strict strategy, which enforces "good" bounds a priori, would not work in the presence of intersection types. Indeed, such a strict strategy would break composition and several monotonicity properties, including narrowing – see Section 3.4.4.

$$\frac{}{\Gamma \vdash \bot <: T}\ (\textsc{Bot}) \qquad \frac{}{\Gamma \vdash T <: \top}\ (\textsc{Top})$$

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T}\ (\textsc{And11}) \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2}\ (\textsc{Or21})$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T}\ (\textsc{And12}) \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2}\ (\textsc{Or22})$$

$$\frac{\Gamma \vdash T <: T_1\ ,\ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2}\ (\textsc{And2}) \qquad \frac{\Gamma \vdash T_1 <: T\ ,\ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T}\ (\textsc{Or1})$$

Figure 3.4 – Full Subtyping Lattice

We can also add introduction rules for intersection in typing. This is useful when certain types are semantically second-class in subtyping. We will see such a situation with recursive types, which we can choose to introduce in typing but not subtyping [Amin et al., 2016].

3.4.3 Safety

Safety is straightforward. Intersection adds one case and union adds two cases, to "possible types".

The mechanized soundness proof is available from http://sound-small-step-d-lattice.namin.net.

3.4.4 False Starts

A false start is losing monotonicity by trying to tame bad bounds.

Previously, we had the option of statically delineating good and bad bounds. With intersection types, the delineation is no longer so clear cut, because we can have components with perfectly good bounds that compose into a component that has bad bounds. Even worse, we cannot rule out bad bounds a priori. By trying to enforce good bounds, we lose several monotonicity properties including narrowing. Such complications defeat the goal of having a full subtyping lattice that exists just by syntactic constructs.

To recap, recall the rules (S-TSEL1) and (S-TSEL2) for subtyping type selections. If we try to "tame" them by enforcing "good" bounds:

$$\frac{\Gamma \vdash p : \{A : S..U\},\ S <: U}{\Gamma \vdash S <: p.A <: U}\ (\text{S-TSEL})$$

We run into trouble. The first is that typing of the path already has a subsumption step, but if we want to ensure "good" bounds, we need to check the precise bounds, not subsumed bounds which may be good, even if the precise bounds are bad. Furthermore, even supposing we could check the bounds precisely, static enforcement does not work because it is not

preserved by narrowing. Tighter assumptions may yield worse results.

We will revisit this issue in Section 4.2.3.

3.4.5 Perspective

Intersection types force our design option here: with respect to bad bounds, we must be lenient, because we can no longer check bad bounds a priori. In general context, (non-axiom) transitivity and narrowing do not hold. For canonical forms, we need to pushback indirection caused by subsumption – but only in concrete context, so distinguishing between abstract and concrete contexts becomes essential.

3.5 From Records to Objects: DOT

For DOT, we add recursive binding in terms and types.

DOT (store-less):

Syntax

$t ::=$	**terms**:	$S, T, U ::=$	**types**:
x	variable	$\top$	top
$\{z \Rightarrow \overline{d}\}$	object	$\bot$	bottom
$t.m(t)$	method invocation	$T \wedge T$	intersection
$d ::=$	**initialization**:	$T \vee T$	union
$L = T$	type member	$L : S..U$	type member
$m(x : T) = t$	method member	$m(x : S) : U$	method member
$v ::=$	**values**:	$p.L$	selection
$\{z \Rightarrow \overline{d}\}$	object	$\{z \Rightarrow T\}$	recursive self
$p ::=$	**paths**:	$\Gamma ::=$	**contexts**:
x	variable	$\emptyset \mid \Gamma, x : T$	variable bindings
v	value		

Figure 3.5 – DOT (store-less): Syntax

DOT (store-less):

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{[z \mapsto \overline{d}]\overline{d} \ni m(x : T_{11}) = t_{12}}{\{z \Rightarrow \overline{d}\}.m(v_2) \longrightarrow [x \mapsto v_2]t_{12}} \quad \text{(E-App)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1.m(t_2) \longrightarrow t_1'.m(t_2)} \quad \text{(E-App1)} \qquad \frac{t_2 \longrightarrow t_2'}{v_1.m(t_2) \longrightarrow v_1.m(t_2')} \quad \text{(E-App2)}$$

Figure 3.6 – DOT (store-less): Small-Step Operational Semantics

3.5.1 Motivation

Objects with a self bring lots of power: F-bounded abstraction and beyond, non-termination, nominality through type abstraction, etc.

For F-bounded quantification, contrast: `X<:{a:Nat, b:X}`
vs.
`x:{z ⇒ X:⊥ .. {a:Nat, b:z.X}}`

Here is a "branding" example, where an identity function returns a path-dependent type:

```
1 { o ⇒ L = ⊤; m(x: ⊤) = x } : { o ⇒ (L: ⊥ .. ⊤) ∧ (m(x: ⊤): o.L) }
```

3.5.2 Formal Design

For objects, we tweak the syntax as shown in Table 3.4. See also Figure 3.5 for the full syntax. We consolidate the values of System D into a new syntatic category of definitions d, and we tie the definitions into an object by closing over a self term variable z: $\{z \Rightarrow \overline{d}\}$. Such an object value introduces a recursive type $\{z \Rightarrow T\}$ over a sequence of member types folded by intersecting. Type-wise, the constructs for types for members, either type or method, carry the label in addition to the type tag or dependent function. The syntax enforces disjointness and good bounds by construction. The introductory typing rule (TOBJ) preserves these guarantees. A recursive type also closes over a self *term*. Once more, in DOT, all quantification is over terms, not types. We type an object *without* subsumption, then close over the recursive type. This is important for soundness, to avoid circular reasoning when ensuring "good" bounds – see Section 3.5.5.

in $D_{<:>}$ value terms	types		in DOT object value term	type
			$\{z \Rightarrow$	$\{z \Rightarrow$
$\lambda x : S.t$	$: \forall x : S.T$		$m(x : S) = t$	$m(x : S) : T$
		$\rightsquigarrow$	$;\ldots$	$: \quad \wedge \ldots$
$\{A = T\}$	$: \{A : S..U\}$		$L = T$	$L : S..U$
			$\}$	$\}$

Table 3.4 – from $D_{<:>}$ to DOT
All value terms consolidate into objects: λ-values become object method members, type tag values become object type members.

We can use a store to keep track of object identities – we will do this in Chapter 4. Here, we just have objects as structural values without a store intermediary. See Figure 3.6 for the operational semantics.

Now, we consider recursive types semantically, generalizing from their introduction for typing objects. We introduce the recursive type binder $\{z \Rightarrow T\}$ for an arbitrary type T.

In typing, we add rules packing (PACK) and unpacking (UNPACK), to introduce and eliminate a recursive type.

Figure 3.7 summarizes two options regarding subtyping with recursive types.

Second-Class Recursive Types Let's first consider the stepping-stone option pursued by pragmatism in prior work [Amin et al., 2016] of omitting recursive types from subtyping, making them second-class types. This option has the big advantage of simplicity: typing can be used without caveats in subtyping type selections. Furthermore, this option is a decent match for Dotty / Scala which already has several restrictions on structural recursive types.

The use of nominal recursion is mostly expressible through unpacking and re-packing. For this, it's useful to add the classical typing introduction rule for intersection types, so that member types can be disassembled (by unpacking and subsumption) and reassembled (by introducing an intersection) semantically during typing.

First-Class Recursive Types The other option is to further enable subtyping comparison with recursive types. We must confront issues of contractiveness. In Chapter 4, we present a full-fledged solution. In particular, Section 4.4.6 sketches a soundness proof based on "possible types". Here, we detail the design, and in the next section, the issue with the proof strategy inherited from System D, and give an idea of the current work-around for contractiveness, though it is bound not to be the final word on the topic.

In subtyping, we can compare two recursive types, but also compare a recursive type with another type without a self. See rules (BIND) and (BIND1). Notice that the typing judgement comes in two variant : and :!. The second variant is used in typing paths during subtyping of type selections. In particular, this variant for subtyping disallows packing. These restrictions are only necessary if recursive types partake in subtyping. See Figure 3.7 for subtyping and Figure 3.8 for typing.

3.5.3 Recursion and Normalization

There are some counterintuitive results related to the spectrum of recursion and normalization. At one end, of course a calculus with unscoped names is not normalizing.

```
1 new { z ⇒
2   def f(u: ⊤): Nat = z.f(u)
3 }.f(u)
```

No surprises here, because the recursion is explicit.

What about F-bounds? We know that System $F_{<:}$ with upper F-bounds is strongly normalizing [Baldan et al., 1999]. Surprisingly, the anti-symmetric System $F_{<:}$ with lower F-bounds is *not* strongly normalizing. Figure 3.5 is a counter-example to normalization in that subset.

So we need to be very careful when conjecturing unless we are using proof techniques with strong by construction guarantees such as Wright and Felleisen [1994]. Combination of features creates unanticipated emergent behavior.

3.5.4 Safety

For the variant without subtyping recursive types, the proof scales straightforwardly. With recursive subtyping however, there are some complications.

For soundness, we need to adjust the design and proof strategy once more. Compared to

```
λz : { A = z.A → ⊤ → ⊤ }
λf : z.A
λx : ⊤.
f f x
```

$\Gamma = \quad z : A = z.A -> \top -> \top$
$\qquad\quad f : z.A$
$\qquad\quad x : \top$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash z.A <: z.A \to \top \to \top}\,\text{(S-VarLow)}}{\Gamma \vdash f : z.A \to \top \to \top}\,\text{(T-Sub)} \qquad \Gamma \vdash f : z.A}{\Gamma \vdash ff : \top \to \top}\,\text{(T-App)}$$

Table 3.5 – $F_{:>}$ with F-Bound anormal example

System $D_{<:>}$, the main issue is that we cannot establish the equivalence of tight and non-tight subtyping in an empty context, because for the (Bind) case, we would need substitution, but then we would forfeit the induction hypothesis.

Our solution involves restrictions and simplifications.

First, we always use tight subtyping: for values – objects, the rules (SSel1) and (SSel2) simply look up the type member in the self-substituted object; for abstract variables in the context, subtyping remains untight via the rules (Sel1) and (Sel2).

Now, the substitution lemma becomes more challenging to prove, as it needs to convert untight selections on the substituted variable into tight selections on the substituting value. Recall that such a conversion from untight to tight during substitution would be challenging in System $D_{<:>}$ as well.

Our solution is to prove widening for invertible value typing or possible types closure mutually with substitution. However, recall that invertible value typing is only defined on empty context, and so this widening needed for the untight-to-tight conversion can only be expresssed in an empty context. For this reason, we only prove substitution if the prefix context is empty, i.e. the variable to be substituted is the first in the context. We still need a suffix context to strengthen the induction hypothesis.

For this reason, we *restrict* the context of typing the variable of a type selection to the prefix context up to and including the variable. This restriction excludes any variable in the context that were added after the variable under consideration. Thanks to subtyping transitivity, this restriction doesn't matter much in practice. Yet, it ensures that during transitivity, the subsumptions underlying a typing on the variable to be substituted is expressed in an empty context – the prefix is empty because the substitution lemma is restricted, and the suffix is empty by design because of the restriction on the context in typing during type selection. The setup is in place for the mutual induction.

This is why we have the restriction on Γ in rules (Sel1) and (Sel2): so that we can use tight

subtyping – on values only – from the outset.

Definition 4 (Restricted Substitution)**.** We restrict substitution so that substituted variable is first in context: $\Gamma' = \emptyset$.
If $\Gamma', x : U, \Gamma \vdash t : T$ and $\Gamma' \vdash v : U$, then $\Gamma', [x \mapsto v]\Gamma \vdash [x \mapsto v]t : [x \mapsto v]T$

Definition 5 (Possible Types)**.** $v :: T$ if value v can possibly have type T – a directly invertible relation defined in Figure 3.9.

Definition 6 (Metric)**.** Let $v ::_m T$ denote a derivation of $v :: T$ with no more than m uses of (V-BIND).

Definition 7 (Widening)**.** Let Widen_m denote the assumption that $v ::_m T$ can be widened:
If $v ::_m T$ and $\emptyset \vdash T <: U$ then $v ::_m U$.

Lemma 14 (Substitution Lemmas assuming Widening)**.**

1. for subtyping –
   If $v ::_m T$ and Widen_m and $x : T, \Gamma \vdash S <: U$, then $[x \mapsto v]\Gamma \vdash [x \mapsto v]S <: [x \mapsto v]U$.
2. for abstract paths –
   If $v ::_m T$ and Widen_m and $x \neq z$ and $x : T, \Gamma \vdash z :_! T$, then $[x \mapsto v]\Gamma \vdash z :_! [x \mapsto v]T$.
3. for turning abstract path into concrete value typing –
   If $v ::_m T$ and Widen_m and $x : T, \Gamma \vdash x :_! U$, then $v ::_m [x \mapsto v]U$.

Lemma 15 (Widening)**.** $\forall m.\text{Widen}_m$.

Lemma 16 (Empty-Context Value Typing implies "Possible Types".)**.** If $\emptyset \vdash v : T$ then $v :: T$.

The main challenge is to orchestrate the induction metrics. We bootstrap mutual induction between substitution and widening of value typing: Outer induction on number of packing uses, inner induction on size of derivation. Note that in type selections, we use tight typing for values and only need $\Gamma \vdash x :: T$ for type selections involving abstract variables.

The mechanized soundness proof is available from http://sound-small-step-dot-storeless.namin.net.

3.5.5 False Starts

We describe some unsound variations due to circular reasoning.

For soundness, we cannot allow real objects with "bad bounds", e.g. $\top..\bot$, as they could create real – as opposed to hypothetical – collapse of the subtyping lattice. In the calculus, we force good bounds trivially by syntax, since type member are *defined* by aliases though *declared* by bounds. Here are some simple variations that cause trouble due to circular reasoning when trying to enforce good bounds semantically rather than syntactically.

- Adding subsumption to definition type assignment...

$$\frac{\Gamma \vdash d : T \quad \Gamma \vdash T <: U}{\Gamma \vdash d : U} \qquad (\text{DEF-SUB})$$

... would fail to flag this bad typing:

$$\{x \Rightarrow X = \top\} : \{x \Rightarrow X : \top..\bot\}$$

... because the following typing derivation tree would be accepted:

$$\dfrac{\dfrac{}{x : (X : \top..\bot) \vdash (X = \top) : (X : \top..\top)}\,(\text{DTYP}) \quad \dfrac{\dfrac{x : (X : \top..\bot) \vdash \top <: x.X <: \bot}{x : (X : \top..\bot) \vdash \top <: \bot}\,(\text{TRANS})}{x : (X : \top..\bot) \vdash (X : \top..\top) <: (X : \top..\bot)}\,(\text{TYP})}{\dfrac{x : (X : \top..\bot) \vdash (X = \top) : (X : \top..\bot)}{\emptyset \vdash \{x \Rightarrow X = \top\} : \{x \Rightarrow X : \top..\bot\}}\,(\text{TOBJ})}\,(\text{DEF-SUB})$$

- Changing type definition from $\{L = T\}$ to $\{L : S..U\}$ ...

$$\frac{\Gamma \vdash S <: U}{\Gamma \vdash \{L : S..U\} : \{L : S..U\}} \qquad (\text{DTYP'})$$

... would fail to flag this bad typing:

$$\{x \Rightarrow X : \top..\bot\} : \{x \Rightarrow X : \top..\bot\}$$

... because the following typing derivation tree would be accepted:

$$\dfrac{\dfrac{\dfrac{x : (X : \top..\bot) \vdash \top <: x.X <: \bot}{x : (X : \top..\bot) \vdash \top <: \bot}\,(\text{TRANS})}{x : (X : \top..\bot) \vdash (X : \top..\bot) : (X : \top..\bot)}\,(\text{DTYP'})}{\emptyset \vdash \{x \Rightarrow X : \top..\bot\} : \{x \Rightarrow X : \top..\bot\}}\,(\text{TOBJ})$$

3.5.6 Perspective

Recursive subtyping introduces some challenge in proof termination. To ensure good bounds for real objects despite recursive subtyping, we need to be careful when tying the recursive knot to avoid circular reasoning. Furthermore, design must not rely “lazy” or “void” evidence.

For recursive types in term typing, we add object creation, and general introduction and elimination rules through packing and unpacking of the recursive self type.

For recursive types in subtyping: we can compare recursive types (on both sides, or on the left) but this comes at the price of handling typing in subtyping with more care.

We will revisit the proof in more details in Chapter 4, using a variant with explicit object creation and a store to track object identities.

DOT (store-less):

Subtyping $\boxed{\Gamma \vdash S <: U}$

Lattice structure

$$\frac{}{\Gamma \vdash \bot <: T}\ (\textsc{Bot}) \qquad \frac{}{\Gamma \vdash T <: \top}\ (\textsc{Top})$$

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T}\ (\textsc{And11}) \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2}\ (\textsc{Or21})$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T}\ (\textsc{And12}) \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2}\ (\textsc{Or22})$$

$$\frac{\Gamma \vdash T <: T_1\ ,\ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2}\ (\textsc{And2}) \qquad \frac{\Gamma \vdash T_1 <: T\ ,\ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T}\ (\textsc{Or1})$$

Type and method members

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ U_1 <: U_2}{\Gamma \vdash L : S_1..U_1 <: L : S_2..U_2}\ (\textsc{Typ}) \qquad \frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash m(x : S_1) : U_1 <: m(x : S_2) : U_2}\ (\textsc{Fun})$$

Type selections

$$\frac{\Gamma \vdash p : (L : T..\top)}{\Gamma \vdash T <: p.L}\ (\textsc{PSel2}) \qquad \frac{\Gamma \vdash p : (L : \bot..T)}{\Gamma \vdash p.L <: T}\ (\textsc{PSel1})$$

$$\frac{\Gamma_{[x]} \vdash x :! (L : T..\top)}{\Gamma \vdash T <: x.L}\ (\textsc{Sel2}) \qquad \frac{\Gamma_{[x]} \vdash x :! (L : \bot..T)}{\Gamma \vdash x.L <: T}\ (\textsc{Sel1})$$

$$\frac{[z \mapsto \overline{d}]\overline{d} \ni L = T}{\Gamma \vdash T <: \{z \Rightarrow \overline{d}\}.L}\ (\textsc{SSel2}) \qquad \frac{[z \mapsto \overline{d}]\overline{d} \ni L = T}{\Gamma \vdash \{z \Rightarrow \overline{d}\}.L <: T}\ (\textsc{SSel1})$$

Recursive self types

$$\frac{\Gamma, z : T_1 \vdash T_1 <: T_2}{\Gamma \vdash \{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}}\ (\textsc{Bind}) \qquad \frac{\Gamma, z : T_1 \vdash T_1 <: T_2 \quad z \notin fv(T_2)}{\Gamma \vdash \{z \Rightarrow T_1\} <: T_2}\ (\textsc{Bind1})$$

Properties

$$\frac{}{\Gamma \vdash T <: T}\ (\textsc{Refl}) \qquad \frac{\Gamma \vdash T_1 <: T_2\ ,\ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}\ (\textsc{Trans})$$

(Δ / Δ without / with recursive subtyping)

Figure 3.7 – DOT (store-less): Subtyping

DOT (store-less):

Type assignment $\boxed{\Gamma \vdash t :_{(!)} T}$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x :_{(!)} T} \text{ (VAR)} \qquad \frac{\Gamma \vdash t :_{(!)} T_1 \ , \ T_1 <: T_2}{\Gamma \vdash t :_{(!)} T_2} \text{ (SUB)}$$

$$\frac{\Gamma \vdash p : [z \mapsto p]T}{\Gamma \vdash p : \{z \Rightarrow T\}} \text{ (PACK)} \qquad \frac{\Gamma \vdash p :_{(!)} \{z \Rightarrow T\}}{\Gamma \vdash p :_{(!)} [z \mapsto p]T} \text{ (UNPACK)}$$

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2) \ , \ t_2 : T_1 \quad x \notin fv(T_2)}{\Gamma \vdash t.m(t_2) : T_2} \text{ (TAPP)} \qquad \frac{\Gamma \vdash t : (m(x : T_1) : T_2) \ , \ p : T_1}{\Gamma \vdash t.m(p) : [x \mapsto p]T_2} \text{ (TAPPDEP)}$$

$$\frac{\text{(labels disjoint)} \quad \Gamma, x : T_1 \wedge \ldots \wedge T_n \vdash d_i : T_i \qquad \forall i, 1 \leq i \leq n}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : [x \mapsto \{x \Rightarrow d_1 \ldots d_n\}](T_1 \wedge \ldots \wedge T_n)} \text{ (TOBJ)}$$

Member initialization $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma \vdash T <: T}{\Gamma \vdash (L = T) : (L : T..T)} \text{ (DTYP)} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (m(x) = t) : (m(x : T_1) : T_2)} \text{ (DFUN)}$$

Figure 3.8 – DOT (store-less): Typing

DOT (store-less – proof device):

Possible Types $\boxed{v :: T}$

Base Cases

$$v :: \top \text{ (V-TOP)} \qquad \frac{\text{(labels disjoint)} \quad \forall i, 1 \leq i \leq n \quad \emptyset, (x : T_1 \wedge \ldots \wedge T_n) \vdash d_i : T_i \quad \exists j, \ [x \mapsto \{x \Rightarrow \overline{d}\}]d_j = (m(z : S) = t) \quad [x \mapsto \{x \Rightarrow \overline{d}\}]T_j = (m(z : S) : U) \quad \emptyset \vdash S' <: S \qquad \emptyset, (z : S') \vdash U <: U'}{\{x \Rightarrow \overline{d}\} :: (m(x : S') : U')} \text{ (V-FUN)}$$

$$\frac{(L = T) \in \overline{[x \mapsto \{x \Rightarrow \overline{d}\}]d} \quad \emptyset \vdash S <: T \ , \ T <: U}{\{x \Rightarrow \overline{d}\} :: (L : S..U)} \text{ (V-TYP)}$$

Inductive Cases

$$\frac{v :: T \qquad (L = T) \in \overline{[x \mapsto \{x \Rightarrow \overline{d}\}]d}}{v :: (\{x \Rightarrow \overline{d}\}.L)} \text{ (V-SEL)} \qquad \frac{v :: T_1 \qquad v :: T_2}{v :: T_1 \wedge T_2} \text{ (V-AND)}$$

$$\frac{v :: T_1}{v :: T_1 \vee T_2} \text{ (V-OR1)}$$

$$\frac{v :: [x \mapsto v]T}{v :: \{x \Rightarrow T\}} \text{ (V-BIND)} \qquad \frac{v :: T_2}{v :: T_1 \vee T_2} \text{ (V-OR2)}$$

Figure 3.9 – DOT (store-less – proof device): Possible Types

4 Type Soundness for DOT

The main contribution of this chapter is to demonstrate how a rich DOT calculus that includes recursive type refinement and a subtyping lattice with intersection types can still be proved sound. The key insight is that subtyping transitivity only needs to be invertible in code paths *executed at runtime*, with contexts consisting entirely of valid runtime objects, whereas inconsistent subtyping contexts can be permitted for code that is never executed.

This chapter is structured as follows:

- We present a DOT calculus that include recursive type refinement and a subtyping lattice with intersection and union types. The presentation in section 4.1 is formal. For an informal presentation with examples in Scala, see section 2.3.
- We discuss the key properties that make a soundness proof difficult (Section 4.2).
- We present our soundness result. First, we give a small-step operational semantics (Section 4.3). Then we explain the proof strategy and key lemmas in details (Section 4.4).
- We offer some perspective on how foundational type-theoretic work influences practice (Section 4.5).

The mechanized soundness proofs are available from http://sound-small-step-dot.namin.net.

4.1 Formal Model

Figure 4.1 shows the syntax and static semantics of the DOT calculus we study. For readability, we omit well-formedness requirements from the rules, and assume all types to be syntactically well-formed in the given environment. We write T^x when x is free in T. The type assignment syntax is overloaded: we use $\Gamma \vdash t : T$ for the usual term type assignment and $\Gamma \vdash t :_! T$ for a path type assignment used in subtyping of type selections, specifically rules (SEL1) and (SEL2). We use $\Gamma \vdash t :_{(!)} T$ for rules that are defined for both cases, specifically rules (VAR), (VARUNPACK), and (SUB).

The Scala syntax used above maps to the formal notation in a straightforward way:

```
1          { type T = Elem }  ⇝  T:Elem..Elem
2        { type T >: S <: U }  ⇝  T:S..U
3         { def m(x: T) = t }  ⇝  m(x) = t
```

In addition, top-level definitions of `val`s and `type`s desugar to local definitions, by the usual desugaring of let-bindings into applications. Intersection and union types, along with the $\bot$ and $\top$ types, provide a full subtyping lattice.

Note that each kind of object member, i.e. a type member $L = T$ or a method member $m(x) = t$, includes its specific label L or m, respectively. The same holds for the corresponding types. A sensible alternative would be to decouple object membership and the underlying primitive types for methods (having dependent function types instead) and types (having "type tag" types instead that can be dereferenced as types). Because objects are recursive in their self type, we find coupling the labels more intuitive in terms of reasoning.

When typing object creation, rules (TNEW) and (DTYP) ensure syntactically that object values have "good bounds", avoiding soundness issues due to lattice collapsing (which we explain in Section 4.2.3). First, labels are disjoint. Second, type member initialization requires exact aliases, not bounds. Finally, to avoid circular reasoning, the checking of member initialization is done without subsumption on the self type.

In subtyping, members of the same label and kind can be compared via rules (TYP) and (FUN). The type member upper bound, and method result type are covariant while the type member lower bound and the method parameter type are contravariant – as is standard. In rule (FUN), we allow some dependence between the parameter type and the return type of a method, when the argument is a variable.

If a variable x can be assigned a type member L, then the type selection $x.L$ is valid, and can be compared with any upper bound when it is on the left, and with any lower bound when it is on the right, via rules (SEL1) and (SEL2). Furthermore, a type selection is reflexively a subtype of itself via rule (SELX). This rule is explicit, so that even abstract type members can be compared to themselves.

Finally, recursive self types can be compared to each other via (BINDX). They can also be

DOT Syntax and Type System

Syntax

x, y, z	Variable
L	Type label
m	Method label
$\Gamma ::= \emptyset \mid \Gamma, x : T$	Context

$S, T, U ::=$	Type
$\top$	top type
$\bot$	bottom type
$L : S..U$	type member
$m(x : S) : U^x$	method member
$x.L$	type selection
$\{z \Rightarrow T^z\}$	recursive self type
$T \wedge T$	intersection type
$T \vee T$	union type

$t, u ::=$	Term
x	variable reference
$\{z \Rightarrow \overline{d}\}$	new instance
$t.m(t)$	meth. invocation
$d ::=$	Initialization
$L = T$	type initialization
$m(x) = t$	meth. init.

Subtyping $\boxed{\Gamma \vdash S <: U}$

Lattice structure

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

$$\frac{\Gamma \vdash T_1 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{ (And11)} \qquad \frac{\Gamma \vdash T <: T_1}{\Gamma \vdash T <: T_1 \vee T_2} \text{ (Or21)}$$

$$\frac{\Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \wedge T_2 <: T} \text{ (And12)} \qquad \frac{\Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \vee T_2} \text{ (Or22)}$$

$$\frac{\Gamma \vdash T <: T_1 \ , \ T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \text{ (And2)} \qquad \frac{\Gamma \vdash T_1 <: T \ , \ T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T} \text{ (Or1)}$$

Type and method members

$$\frac{\Gamma \vdash S_2 <: S_1 \ , \ U_1 <: U_2}{\Gamma \vdash L : S_1..U_1 <: L : S_2..U_2} \text{ (Typ)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x : S_2 \vdash U_1^x <: U_2^x}{\Gamma \vdash m(x : S_1) : U_1^x <: m(x : S_2) : U_2^x} \text{ (Fun)}$$

Path selections

$$\Gamma \vdash x.L <: x.L \quad \text{(SelX)}$$

$$\frac{\Gamma_{[x]} \vdash x :_! (L : T..\top)}{\Gamma \vdash T <: x.L} \text{ (Sel2)} \qquad \frac{\Gamma_{[x]} \vdash x :_! (L : \bot..T)}{\Gamma \vdash x.L <: T} \text{ (Sel1)}$$

Recursive self types

$$\frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2^z}{\Gamma \vdash \{z \Rightarrow T_1^z\} <: \{z \Rightarrow T_2^z\}} \text{ (BindX)}$$

$$\frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2}{\Gamma \vdash \{z \Rightarrow T_1^z\} <: T_2} \text{ (Bind1)}$$

Transitivity

$$\frac{\Gamma \vdash T_1 <: T_2 \ , \ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \text{ (Trans)}$$

Type assignment $\boxed{\Gamma \vdash t :_{(!)} T}$

Variables, self packing/unpacking

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

$$\frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \{z \Rightarrow T^z\}} \text{ (VarPack)} \qquad \frac{\Gamma \vdash x :_{(!)} \{z \Rightarrow T^z\}}{\Gamma \vdash x :_{(!)} T^x} \text{ (VarUnpack)}$$

Subsumption

$$\frac{\Gamma \vdash t :_{(!)} T_1 \ , \ T_1 <: T_2}{\Gamma \vdash t :_{(!)} T_2} \text{ (Sub)}$$

Method invocation

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2^x) \ , \ y : T_1}{\Gamma \vdash t.m(y) : T_2^y} \text{ (TAppVar)}$$

$$\frac{\Gamma \vdash t : (m(x : T_1) : T_2) \ , \ t_2 : T_1}{\Gamma \vdash t.m(t_2) : T_2} \text{ (TApp)}$$

Object creation

$$\frac{\text{(labels disjoint)} \qquad \Gamma, x : T_1^x \wedge \ldots \wedge T_n^x \vdash d_i : T_i^x \qquad \forall i, 1 \le i \le n}{\Gamma \vdash \{x \Rightarrow d_1 \ldots d_n\} : \{z \Rightarrow T_1^z \wedge \ldots \wedge T_n^z\}} \text{ (TNew)}$$

Member initialization $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma \vdash T <: T}{\Gamma \vdash (L = T) : (L : T..T)} \text{ (DTyp)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (m(x) = t) : (m(x : T_1) : T_2)} \text{ (DFun)}$$

Figure 4.1 – DOT Syntax and Type System

dropped if the self identifier is not used, via rule (BIND1). During type assignment, the rules for variable packing and unpacking, (VARPACK) and (VARUNPACK), serve as introduction and elimination rules, enabling recursive types to be compared to other types as well. Since types can be recursive, and since subtype comparisons may introduce temporary bindings that may need to be unpacked, there are two *contractiveness restrictions* on type selections that ensure well-founded induction in the proofs (Section 4.4.2). First, the typing assignment judgement used in type selections, $\Gamma \vdash x :_! T$, forbids (VARPACK). Note that this typing is used to match an unpacked type member type, anyways. Second, type selections restrict the environment to disregard bindings introduced after the self. In general, environments have the form $\Gamma = \overline{y : T}, \overline{z : T}$, with proper term bindings y followed by bindings z introduced by subtype comparisons. The notation $\Gamma_{[x]}$ in rules (SEL1) and (SEL2) signifies that all $z : T$ bindings to the right of x are dropped from Γ, where x can be either a proper term binding y or a z binding. While these restrictions are necessary for the proofs, they do not limit expressiveness of the type system in any significant way. Similar, and in many cases more impeding, restrictions are standard in type systems with recursive types [Pierce, 2002]. Intuitively, note that the transitivity rule can always be used to relate type selections across the context. Note as well that there is no (BIND2) rule, symmetric to (BIND1), which is another kind of contractiveness restriction. We conjecture that these contractiveness restrictions could be lifted without breaking soundness, since we can always construct explicit conversion functions that use rules (VARPACK) and (VARUNPACK) on proper term bindings. However, removing these contractiveness restrictions would likely require different and harder to mechanize proof techniques such as a coinductive interpretation of subtyping.

In the version of DOT presented here, we make the transitivity rule (TRANS) explicit, although, as we will see in Section 4.2, we will sometimes need to ensure that we can eliminate uses of this rule from subtyping derivations so that the last rule is a structural one.

The aim of DOT is to be a simple, foundational calculus in the spirit of FJ [Igarashi et al., 2001], without committing to specific decisions for nonessential things. Hence, implementation inheritance, mixin strategy, and prototype vs. class dispatch are not considered.

4.2 Static Properties

Having introduced the syntax and static semantics of DOT, we turn to its metatheoretic properties. Our main focus of interest will be type safety: establishing that well-typed DOT programs do not go wrong. Of course, type safety is only meaningful with respect to a dynamic semantics, which we will discuss in detail in Section 4.3. Here, we briefly touch some general static properties of DOT and then discuss specific properties of the subtyping relation, which (or their absence!) makes proving type safety a challenge.

Decidability Type assignment and subtyping are undecidable in DOT. This follows directly from the fact that DOT can encode $F_{<:}$, and that these properties are undecidable there.

Type Inference DOT has no principal types and no global Hindley-Milner style type inference procedure. But as in Scala, local type inference based on subtype constraint solving [Pierce and Turner, 2000, Odersky and Läufer, 1996, Odersky et al., 2001] is possible, and in fact easier than in Scala due to the existence of universal greatest lower bounds and least upper bounds through intersection and union types. For example, in Scala, the least upper bound of the two types `C` and `D` is approximated by an infinite sequence:

```
1 trait A { type T <: A }
2 trait B { type T <: B }
3 trait C extends A with B { type T <: C }
4 trait D extends A with B { type T <: D }
5 lub(C, D) ~ A with B { type T <: A with B { type T <: ... } }
```

DOT's intersection and union types remedy this brittleness.

While the term syntax and type assignment given in Figure 4.1 is presented in Curry-style, without explicit type annotations except for type member initializations, a Church-style version with types on method arguments (as required for local type inference) is possible just as well. Our mechanized proof handles both Curry and Church style via optional parameter and return types. The Curry-style presentation glosses over explicit upcasts – a feature that can easily be added, for example through a typed `let` construct. Upcasts – whether explicit or implicit in the surface syntax – matter for nominality by ascription, as discussed in Section 2.3.

4.2.1 Properties of Subtyping

The relevant static properties we are interested in with regard to type safety are transitivity, narrowing, and inversion of subtyping and type assignment. They are defined as follows.

Inversion of subtyping (example for functions):

$$\frac{\Gamma \vdash m(x:S_1):U_1^x <: m(x:S_2):U_2^x}{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x:S_2 \vdash U_1^x <: U_2^x} \quad \text{(InvFun)}$$

Transitivity of subtyping:

$$\frac{\Gamma \vdash T_1 <: T_2\ ,\ T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \quad \text{(Trans)}$$

Narrowing:

$$\frac{\Gamma_1 \vdash T_1 <: T_2 \qquad \Gamma_2 \vdash T_3 <: T_4 \qquad \Gamma_1 = \Gamma_2(x \to T_1) \qquad \Gamma_2(x) = T_2}{\Gamma_1 \vdash T_3 <: T_4} \quad \text{(Narrow)}$$

where $\Gamma_1 = \Gamma_2(x \to T_1)$ denotes that Γ_1 is like Γ_2 except the binding for x maps to T_1.

On a high-level, the basic challenge for type safety is to establish that some value that e.g. has

a function type actually is a function, with arguments and result corresponding to the given type. This is commonly known as the *canonical forms* property. Inversion of subtyping is required to extract the argument and result types from a given subtype relation between two function types, in particular to derive

$$T_2 <: T_1 \text{ and } U_1 <: U_2$$

from

$$m(x : T_1) : U_1 <: m(x : T_2) : U_2$$

when relating method types from a call site and the definition site.

Transitivity is required to collapse multiple subsumption steps that may have been used in type assignment. Narrowing can be seen as an instance of the Liskov substitution principle, preserving subtyping if a binding in the environment is replaced with a subtype. Narrowing is required for application, when the argument type is a subtype of the declared parameter type.

Unfortunately, as we will show next, these properties do *not* hold simultaneously in DOT, at least not in their full generality.

4.2.2 Inversion, Transitivity and Narrowing

First of all, let us take note that these properties are mutually dependent. In Figure 4.1, we have included (TRANS) as an axiom. If we drop this axiom, then we obtain key rules like (INVFUN) by direct inversion of the corresponding typing derivation, as only the structural rule (FUN) can match the pattern of comparing two function types. But then, we would need to prove (TRANS) as a lemma.

Transitivity and narrowing are also mutually dependent. Transitivity requires narrowing in the following case, where we are given

$$\{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\} <: \{z \Rightarrow T_3\}$$

and want to derive:

$$\{z \Rightarrow T_1\} <: \{z \Rightarrow T_3\}$$

By inversion we obtain

$$z : T_1 \vdash T_1 <: T_2 \qquad z : T_2 \vdash T_2 <: T_3$$

and we narrow the right-hand derivation to $z : T_1 \vdash T_2 <: T_3$ before we apply transitivity inductively to obtain $z : T_1 \vdash T_1 <: T_3$ and thus $\{z \Rightarrow T_1\} <: \{z \Rightarrow T_3\}$.

Narrowing depends on transitivity in the case for type selections

$$x.L <: T \text{ or its counterpart } T <: x.L$$

Assume that we want to narrow x's binding from T_2 in Γ_2 to T_1 in Γ_1, with $\Gamma_1 \vdash T_1 <: T_2$. By inversion we obtain

$$x : (L : \bot..T)$$

and, disregarding rules (VARPACK) and (VARUNPACK) we can deconstruct this assignment as

$$\Gamma_2(x) = T_2 \qquad \Gamma_2 \vdash T_2 <: (L : \bot..T).$$

We first apply narrowing inductively and then use transitivity to derive the new binding

$$\Gamma_1(x) = T_1 \qquad \Gamma_1 \vdash T_1 <: T_2 <: (L : \bot..T).$$

On first glance, the situation appears to be similar to simpler calculi like $F_{<:}$, for which the transitivity rule can be shown to be admissible, i.e. implied by other subtyping rules and hence proved as a lemma and dropped from the definition of the subtyping relation. Unfortunately this is not the case in DOT.

4.2.3 Good Bounds, Bad Bounds

The transitivity axiom (or subsumption step in type assignment) is essential and cannot be dropped. Let us go through and see why we cannot prove transitivity directly.

First of all, observe that transitivity can only hold if all types in the environment have "good bounds", i.e. only members where the lower bound is a subtype of the upper bound. Through "bad" bounds in the context and subtyping transitivity, the subtyping lattice can collapse. For example, assume a binding with "bad" bounds like `Int..String`. Then the following subtype relation holds via transitivity

$$x : \{A = \texttt{Int..String}\} \vdash \texttt{Int} <: x.A <: \texttt{String}$$

but `Int` is not a subtype of `String`. Of course core DOT does not have `Int` and `String` types, but any other incompatible types can be taken as well.

But even if we take good bounds as a precondition, we cannot show

$$\{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\} <: \{z \Rightarrow T_3\}$$

because we would need to use $x : T_1$ in the extended environment for the inductive call, but we do not know that T_1 has indeed good bounds.

Of course we could modify the $\{z \Rightarrow T_1\} <: \{z \Rightarrow T_2\}$ rule (BINDX) to require T_1 to have good bounds. But then this evidence would need to be narrowed, which unfortunately is not

possible. Again, here is a example, considering a type under a context:

$$x : \{A : \bot..\top\} \vdash x.A \wedge \{B = \texttt{Int}\}$$

This type has good bounds, but when narrowing x in the context to the smaller type $\{A = \{B = \texttt{String}\}\}$ (which also has good bounds), its bounds become contradictory.

In summary, even if we assume good bounds in the environment, and strengthen our typing rules so that only types with good bounds are added to the environment, we may still end up with bad bounds due to narrowing and intersection types. This refutes one conjecture about possible proof avenues from earlier work on DOT [Amin et al., 2014].

Intuitively, all the questions related to bad bounds have a simple answer: We ensure good bounds at object creation time, so *why do we need to worry in such a fine-grained way?*

4.2.4 No (Simple) Substitution Lemma

As a final negative result, we illustrate how typing is not preserved by straightforward substitution in DOT, because of path-dependent types.

$$\Gamma, x : \{z \Rightarrow L : S^z..U^z \wedge m(\_) : T^z\} \vdash t^x : x.L$$

Now, consider substituting x in t with u, given

$$\Gamma \vdash u : \{z \Rightarrow L : S^z..U^z \wedge m(\_) : T^z\}$$

First, t^x might invoke method $x.m(\_)$ and therefore require assigning type $(m(\_) : T^x)$ to x. However, the self bind layer can only be removed if an object is accessed through a variable, otherwise we would not be able to substitute away the self identifier. Second, we cannot derive $\Gamma \vdash t^u : x.L$ with x removed from the environment, but we also cannot replace x in $x.L$ with a term u that is not an identifier. Hence, u cannot be an arbitrary term, because path-dependent types are not syntactically valid for arbitrary terms.

Intuitively, there is still hope for substitution, but only with forms that preserve syntactic validity of path-dependent types. Another option is to settle for a relaxed notion of path-dependent types that allows type selections on values like in Chapter 3.

4.2.5 There is Still Hope: Key Observations

Based on the discussion so far, we can make the following observations, which reveal a possible avenue for proving type soundness despite the difficulties:

Observation 1. If objects can only be created with type aliases ($L = T$), not arbitrary bounds $T_1..T_2$, then runtime objects cannot have bad bounds.

Observation 2. If we assume a call-by-value evaluation strategy, then whenever we execute a method call at runtime, all variables are bound to existing objects.

Observation 3. The only place in a soundness proof where we rely on the (INVFUN) property is for the evaluation of such method calls.

Taken together, these observations suggest that making a clear distinction between runtime values and other terms is a prerequisite for a successful soundness proof. Observation 3 further suggests that subtyping transitivity and narrowing can be treated as axioms in subtyping comparisons and when assigning types to static terms, as long as we can recover the (INVFUN) property in contexts that consist exclusively of runtime objects.

Small-Step Operational Semantics and Store Typing

Syntax...

$x ::=$ Variable
- y Concrete Variable (i.e. Store Location)
- z Abstract Variable

$v ::= y$ Value (Store Location)

$\rho ::= \emptyset \mid \rho, y : \overline{d}$ Store

Reduction $\boxed{\rho_1\ t_1 \longrightarrow t_2\ \rho_2}$

$$\begin{array}{llll} \rho & \{z \Rightarrow \overline{d^z}\} \longrightarrow & v & \rho, (v : \overline{d^v}) \quad \textbf{with } v \text{ fresh} \\ \rho & v_1.m(v_2) \longrightarrow & t^{v_2} & \rho \quad \textbf{if } \rho(v_1) \ni m(x) = t^x \\ \rho_1 & e[t_1] \longrightarrow & e[t_2] & \rho_2 \quad \textbf{if } \rho_1\ t_1 \longrightarrow t_2\ \rho_2 \end{array}$$

where $e ::= [\,] \mid [\,].m(t) \mid v.m([\,])$

Subtyping... $\boxed{\rho\ \Gamma \vdash S <: U}$

$$\frac{\rho(y) \ni (L = U) \qquad \rho\ \emptyset \vdash T <: U}{\rho\ \Gamma \vdash T <: y.L}\ (\textsc{SSel2})$$

$$\frac{\rho(y) \ni (L = U) \qquad \rho\ \emptyset \vdash U <: T}{\rho\ \Gamma \vdash y.L <: T}\ (\textsc{SSel1})$$

Type assignment... $\boxed{\rho\ \Gamma \vdash t :_{(!)} T}$

$$\frac{(y, \overline{d^y}) \in \rho \qquad \text{(labels disjoint)}\ \ \forall i, 1 \le i \le n \qquad \rho\ \emptyset, (x : T_1^x \wedge \ldots \wedge T_n^x) \vdash d_i^x : T_i^x}{\rho\ \Gamma \vdash y : T_1^y \wedge \ldots \wedge T_n^y}\ (\textsc{TLoc})$$

Figure 4.2 – Small-Step Operational Semantics and Store Typing

4.3 Operational Semantics

Type soundness is only meaningful with respect to a notion of evaluation: an operational semantics. In order to realize the soundness proof strategy outlined in Section 4.2.5, we need to pick a semantics that allows us to distinguish runtime values from normal terms. While such a distinction is particularly natural in big-step evaluators, the particular style of semantics does not matter so much, and different styles of semantics can be formally inter-derived using the techniques of Danvy and Johannsen [2010], Danvy et al. [2012], Ager et al. [2003]. In our development, we have defined both big-step evaluators and small-step reduction semantics, and used them as the basis for mechanized proofs. Here, we focus our presentation on a small-step reduction semantics, shown in Figure 4.2. In Chapter 5, we discuss soundness proofs with definitional interpreters.

We allocate all objects in a store-like structure, which grows monotonically. In this setting, store locations are the only values, i.e. passed around between functions and stored in environments. Figure 4.2 also shows necessary extensions to subtyping and type assignment, which we explain next.

A store-less alternative is also possible as shown in Chapter 3. The key ideas are to remove the store indirection by treating values as concrete, and variables as abstract, and to relax the

syntax of path-dependent types to also allow values as paths, in addition to variables.

4.3.1 Concrete Variables in Typing and Subtyping

To assign types to terms that occur during evaluation, we need to be able to handle path-dependent types that refer to variables bound in the store ρ, and type store locations. Accordingly, we extend the type system judgements with an additional store parameter ρ. The rules from Figure 4.1 are still valid, and merely thread the extra store parameter ρ unchanged. The additional cases are shown in Figure 4.2.

For concrete variables (i.e. store locations) in type selections, we use the precise type member given at object creation time by looking it up in the store, as defined by the new rules (SSEL1) and (SSEL2).

In this extended subtyping relation, we continue to use the same rules for type selections on variables bound in Γ, (SEL1) and (SEL2) from Figure 4.1. Hence, the auxiliary judgement $\rho\ \Gamma \vdash z :_! T$, which excludes (VARPACK), is only used for typing abstract variables.

Finally, we also need a new case for concrete variables in typing, (TLOC). It resembles (TNEW), except that it looks up the definitions in the store, it does not create a recursive type (because we can reuse the store location in the result type instead), and it does not consider the abstract context while checking the definitions (because all free variables in the store must be concrete).

4.4 Type Soundness

We discuss in detail our proofs for type soundness (Theorem 5), highlighting challenges and insights. Our statement of type safety is the following, folding together the usual notions of *progress* and *preservation* [Wright and Felleisen, 1994]:

Definition 8 (Type Safety)**.** If a term t typechecks to some type T in a store and empty context $\rho\ \emptyset$ (i.e. $\rho\ \emptyset \vdash t : T$), then either the term t is a variable or the store-term configuration ρ, t steps to a configuration ρ', t' where the new store ρ' extends the old store ρ and the resulting configuration typechecks to the same type T (i.e. $\rho'\ \emptyset \vdash t' : T$).

Note that Definition 8 assumes deterministic execution. Otherwise the statement would need to be modified to consider *all possible* following configurations.

We outline the main strategic choices next, with pointers to further subsections that describe the technical details. We reflect again on the higher-level strategy in Section 4.4.5.

Lenient Well-Formedness Any syntactically valid form (type or term) is considered well-formed, if all its free variables are bound in environments. Our strategy is to impose no

semantics (e.g. good bounds for type members) on well-formedness, and ensure desired properties by construction only when they're needed (e.g. for runtime objects).

Lemma 17 (Regularity)**.** If a judgement holds, all its forms are well-formed.

Lemma 18 (Subtyping Reflexivity)**.** Any well-formed type is a subtype of itself.

This leniency pays off. As we saw in Section 4.2.3, trying to enforce semantic properties can break narrowing and other monotonicity properties. For example, the subtyping lattice is full just by definition, since the greatest lower bound and least upper bound of two types always exist, just by syntactic constructs (intersection and union).

"Pushback" For type soundness, we need to establish Canonical Forms (Section 5.2.3) properties: that if a concrete variable has a structural member type, then its definition in the store include a matching member definition.

This is challenging, as the evidence from the typing derivation might be indirect, because of subsumption and subtyping transitivity. Hence, we need to "pushback" such indirections.

We have two choices: pushback transitivity in subtyping (described in Section 4.4.1), or pushback value typing into directly invertible evidence (an independent alternative described in Section 4.4.6).

In both options, a key insight is that we only need to do the pushback in an empty abstract context, which circumvents the impossibility results of Section 4.2. This is where it pays off to distinguish between runtime values and only static types. Plus, for runtime values, we can rely on any properties enforced during their construction, in particular "good bounds".

Substitution By design, substitution is only needed to replace an abstract variable with a concrete one. Because the abstract variable might have a less precise type, there's also a narrowing step involved. Furthermore, type selections on concrete variables are defined more precisely than for abstract ones, which requires converting the evidence from subtyping with abstract type selections to concrete. We describe further in Section 4.4.2

4.4.1 Narrowing and Transitivity Pushback

In simpler type systems like $F_{<:}$, transitivity can be proved as a lemma, together with narrowing, in a mutual induction on the size of the middle type in a chain $T_1 <: T_2 <: T_3$ (see e.g. the POPLmark challenge documentation [Aydemir et al., 2005]).

Unfortunately, for subtyping in DOT, the same proof strategy fails, because subtyping may involve a type selection as the middle type: $T_1 <: z.L <: T_3$. Since proving transitivity becomes much harder, we adopt a strategy from previous DOT developments [Amin et al., 2014]: admit

transitivity as an axiom (TRANS), but prove a 'pushback' lemma that allows to push uses of the axiom further up into a subtyping derivation, so that the top level becomes invertible. We denote this as *precise* subtyping $T_1 <!\ T_2$.

Definition 9 (Precise Subtyping)**.** If $\rho\ \Gamma \vdash T_1 <: T_2$ and the derivation does not end in rule (TRANS) then we say that $\rho\ \Gamma \vdash T_1 <!\ T_2$.

Such a strategy is reminiscent of cut elimination in natural deduction, and in fact, the possibility of cut elimination strategies is already mentioned in Cardelli's original work on $F_{<:}$ [Cardelli et al., 1994].

The narrowing lemma does not have any dependencies:

Lemma 19 (Narrowing)**.**

$$\frac{\rho\ \Gamma_1 \vdash T_1 <: T_2 \qquad \rho\ \Gamma_2 \vdash T_3 <: T_4 \qquad \Gamma_1 = \Gamma_2(z \to T_1) \qquad \Gamma_2(z) = T_2}{\rho\ \Gamma_1 \vdash T_3 <: T_4}$$

Proof. By structural induction, and using the (TRANS) axiom. □

Lemma 20 (Transitivity Pushback)**.**

$$\frac{\rho\ \emptyset \vdash T_1 <: T_2}{\rho\ \emptyset \vdash T_1 <!\ T_2}$$

Proof. We prove an auxiliary lemma by induction on the subtyping derivation $T_1 <: T_2$:

$$\frac{\rho\ \emptyset \vdash T_1 <: T_2\ ,\ T_2 <!\ T_3}{\rho\ \emptyset \vdash T_1 <!\ T_3}$$

using narrowing (Lemma 44) in cases (FUN), (BINDX), (BIND1). Transitivity pushback follows from the special case $T_2 = T_3$. □

Lemma 21 (Inversion of Subtyping)**.** For example for function types:

$$\frac{\rho\ \emptyset \vdash m(x : S_1) \to U_1^x <: m(x : S_2) \to U_2^x}{\rho\ \emptyset \vdash S_2 <: S_1 \qquad \rho\ x : S_2 \vdash U_1^x <: U_2^x}$$

Proof. By transitivity pushback (Lemma 45) and inversion of the resulting derivation. □

Inversion of subtyping is only required in a concrete runtime context, without abstract component ($\Gamma = \emptyset$). Therefore, transitivity pushback is only required then. Transitivity pushback requires narrowing, but only for abstract bindings (those in Γ, never in ρ). Narrowing requires these bindings to be potentially imprecise, so that the transitivity axiom can be used to inject a

step to a smaller type without recursing into the derivation. In summary, we need both (actual, non-axiom) transitivity and narrowing, but not at the same time.

The insight that transitivity pushback is only required in a concrete-only runtime context is crucial for supporting language features such as intersection types that may collapse the subtyping lattice in unrealizable contexts.

4.4.2 Bootstrapping Substitution and Canonical Forms

To mirror the effect of reduction $\rho\ x.m(y) \longrightarrow t^y\ \rho$ given $\rho(x) \ni m(z) = t^z$, we need a substitution lemma for terms, but we also need to mirror the reduction on the type level, since types can refer to variables via type selections. More precisely, we need a substitution lemma that enables us to replace all type selections on an abstract variable $z.A$ with concrete ones $y.A$. For this, we need to ensure that for a concrete variable with a structural type-member type, its definitions in the store include a corresponding type member – a Canonical Forms property for type members (see Section 5.2.3).

We cannot prove either of these properties directly and therefore need to bootstrap a mutual induction. The key induction measure will be an upper bound on the uses of (VarPack) in a typing derivation.

Definition 10 (VarPack Metric)**.** Let $\rho\ \Gamma \vdash_{\leq m}\ y : S$ denote a derivation $\rho\ \Gamma \vdash\ y : S$ with no more than m uses of (VarPack).

Definition 11 (Substitution)**.** Let Subst(m) denote:

$$\frac{\rho\ \emptyset \vdash_{\leq m_1}\ y : S^y \qquad m_1 < m \qquad \rho\ z : S^z, \Gamma^z \vdash\ {T_1}^z <: {T_2}^z}{\rho\ \Gamma^y \vdash\ {T_1}^y <: {T_2}^y}$$

We go on to prove preliminary canonical forms lemmas, assuming the ability to perform substitution for a given m.

Lemma 22 (Pre-canonical Forms for Recursive Types)**.**

$$\frac{\rho\ \emptyset \vdash_{\leq m}\ y : \{z \Rightarrow T^z\} \qquad \text{Subst}(m)}{\rho\ \emptyset \vdash_{\leq m-1}\ y : T^y}$$

Proof. Any trailing uses of (Sub) can be accumulated into a single subtyping statement $\rho\ \emptyset \vdash \{z \Rightarrow T'^z\} <: \{z \Rightarrow T^z\}$, and the remaining derivation must end in (VarPack). By inversion we obtain $\rho\ \emptyset \vdash_{\leq m-1}\ y : T'^y$. After transitivity pushback (Lemma 45), the subtyping derivation must end in (BindX), from which we obtain $\rho\ z : T'^z \vdash\ T'^z <: T^z$. We can now apply our Subst(m) capability to obtain $\rho\ \emptyset \vdash\ T'^y <: T^y$. Applying subsumption (Sub) yields $\rho\ \emptyset \vdash_{\leq m-1}\ y : T^y$. □

Lemma 23 (Pre-canonical Forms for Type Members)**.**

$$\frac{\rho\ \emptyset \vdash_{\leq m}\ y : \{L : S..U\} \qquad \mathrm{Subst}(m)}{\rho(y) \ni (L = T) \qquad \rho\ \emptyset \vdash S <: T <: U}$$

Proof. Again, we accumulate trailing uses of (Sub) into an invertible subtyping. If we hit (VarPack), the resulting subtyping derivation must collapse into (Bind1). We finish the case by applying Lemma 22, Subst(m), and (Sub). Case (TLoc) is immediate from inversion of member case (DTyp). □

We are now ready to prove our substitution lemma, together with two helpers. The proof is by simultaneous induction, first over m, and then an inner induction over the size of the <: or :! derivation.

Lemma 24 (Substitution for <:)**.** $\forall m.\mathrm{Subst}(m)$, i.e.:

$$\frac{\rho\ \emptyset \vdash_{\leq m}\ y : S^y \qquad \rho\ z : S^z, \Gamma^z \vdash\ {T_1}^z <: {T_2}^z}{\rho\ \Gamma^y \vdash\ {T_1}^y <: {T_2}^y}$$

Proof. The key challenge is translating from the rules (Sel1) / (Sel2) to the rules (SSel1) / (SSel2). For (Sel2), if we are selecting $z.A$, we have $\rho\ z : S^z \vdash\ z :_! (L : T^z..\top)$, and with Lemma 26 we obtain $\rho\ \emptyset \vdash_{\leq m}\ y : (L : T^y..\top)$. We invoke canonical forms (Lemma 23), which yields $\rho(y) \ni (L = U)$ and $\rho\ \emptyset \vdash\ T^y <: U$. Note that we can apply Lemma 23 because of our outer induction on m. If we are selecting $z'.A$ with $z' \neq z$, we apply Lemma 25. Case (Sel1) is analogous. Note that the (Sel1)/(Sel2) rules are carefully set up so that $\Gamma_{[z]} = z : S$. □

Lemma 25 (Substitution for :!)**.**

$$\frac{\rho\ \emptyset \vdash_{\leq m}\ y : S^y \qquad z' \neq z \qquad \rho\ z : S^z, \Gamma^z \vdash\ z' :_!\ T^z}{\rho\ \Gamma^y \vdash\ z' :_!\ T^y}$$

Proof. Straightforward. Case (Sub) uses Lemma 24. □

Lemma 26 (Substitution for :)**.**

$$\frac{\rho\ \emptyset \vdash_{\leq m}\ y : S^y \qquad \rho\ z : S^z, \Gamma^z \vdash\ z :_!\ T^z}{\rho\ \emptyset \vdash_{\leq m}\ y : T^y}$$

Proof. Case (Var) is immediate. In case (VarUnpack), we have $\rho\ z : S^z, \Gamma^z z :_!\ \{z_2 \Rightarrow T^{z_2}\}^z$ and by induction we obtain $\rho\ \emptyset \vdash_{\leq m}\ y : \{z_2 \Rightarrow T^{z_2}\}^y$, as required. In case (Sub), we have $\rho\ z : S^z, \Gamma^z z :_!\ T'^z$ and $\rho\ z : S^z, \Gamma^z z :_!\ T'^z <: T^z$. Applying the induction hypothesis and Lemma 24 and using (Sub) we get $\rho\ \emptyset \vdash_{\leq m}\ y : T'^y$. □

Finally, we can prove a substitution on terms.

Lemma 27 (Substitution in Term Typing)**.**

$$\frac{\rho\ (z : S), \Gamma^z \vdash\ t^z : T^z \qquad \rho \vdash\ y :_! S}{\rho\ \Gamma^y \vdash\ t^y : T^y}$$

Proof. Straightforward induction. Case (SUB) uses substitution of subtyping (Lemma 24). □

4.4.3 Inversion of Value Typing (Canonical Forms)

As a corollary of substitution (Lemma 24), we can extend the preliminary canonical forms lemmas to all m.

Lemma 28 (Canonical Forms for Type Members)**.**

$$\frac{\rho\ \emptyset \vdash\ y : \{L : S..U\}}{\rho(y) \ni (L = T) \qquad \rho\ \emptyset \vdash\ S <: T <: U}$$

Proof. Follows directly from Lemma 23 and Lemma 24. □

We also prove an additional canonical forms lemma for function members, which will be required by the main proof.

Lemma 29 (Canonical Forms for Method Members)**.**

$$\frac{\rho\ \emptyset \vdash\ y : m(x : S_2) \to U_2^x}{\begin{array}{c}\rho(y) \ni m(x) = t \\ \rho\ \emptyset, x : S_1 \vdash\ t : U_1^x \\ \rho\ \emptyset \vdash\ m(x : S_1) \to U_1^x\ <:\ m(x : S_2) \to U_2^x\end{array}}$$

Proof. Analogous to Lemma 23. Case (TLOC) eliminates the self variable in the abstract context by applying substitution of term typing (Lemma 27). □

4.4.4 The Main Soundness Proof

Our main type safety theorem combines the usual *progress* and *preservation* lemmas into a single unified statement.

Theorem 5 (Type Safety)**.** If $\rho\ \emptyset \vdash\ t : T$, then either $t = y$ and $\rho(y) = \{\overline{d}\}$ or $\rho\ t \longrightarrow t'\ \rho'$ and $\rho'\ \emptyset \vdash\ t' : T$

Proof. By structural induction. The most interesting case is the dependent application (TAPPVAR), if the receiver of the call is already evaluated to a store location y. We have $\rho\ \emptyset \vdash\ y.m(y_1) : U_2^{y_1}$ and, by inversion, $\rho\ \emptyset \vdash\ y : (m(x : S_2) : T_2^x)$ and $\rho\ \emptyset \vdash\ y_1 : S_1$. By canonical

forms for method members (Lemma 29) we obtain $\rho(y) \ni m(x) = t^x$, $\rho\ x : S_1 \vdash t^x : U_1^x$, and $\rho \vdash m(x : S_1) : U_1^x <: m(x : S_2) : U_2^x$. We know that $\rho\ y.m(y_1) \longrightarrow t^{y_1}\ \rho$, so we need to show that the result is well-typed with the same type: $\rho\ \emptyset \vdash t^{y_1} : U_2^{y_1}$. We apply inversion of subtyping (Lemma 21) and obtain $\rho\ \emptyset \vdash S_2 <: S_1$ and $\rho\ x : S_2 \vdash U_1^x <: U_2^x$. With (SUB), we have $\rho\ \emptyset \vdash y_1 : S_1$, and we can apply substitution of term typing (Lemma 27), to obtain $\rho\ \emptyset \vdash t^{y_1} : U_2^{y_1}$. With (SUB) and applying substitution of subtyping (Lemma 24) to derive $\rho\ \emptyset \vdash U_1^{y_1} <: U_2^{y_1}$, we arrive at the required $\rho\ \emptyset \vdash t^{y_1} : U_2^{y_1}$. □

4.4.5 Some Reflection

The soundness proof was set up carefully to avoid cycles between required lemmas. Where cycles did occur, as with transitivity and narrowing, we broke them using a pushback technique. Where this was cumbersome to do, as with substitution and canonical forms, we used mutual induction. A key property of the system is that, in general, we are very lenient about things outside of the concrete runtime store. The only place where we invert a dependent function type and go from abstract to concrete is in showing safety of the corresponding type assignment rules. This enables subtyping inversion and pushback to disregard abstract bindings for the most part.

When seeking to unpack recursive self types within lookups of type selections in subtype comparisons, the option of disregarding abstract bindings is no longer so easy. Every lookup of a variable, while considering a path-dependent type, may potentially need to unfold and invert self types. In particular, the substitution lemma itself that converts imprecise into precise bounds may unfold a self type. Then it will be faced with an abstract variable that first needs to be converted to a concrete variable. More generally, whenever we have a chain

$$\{z \Rightarrow T_1\} <: T <: \{z \Rightarrow T_2\},$$

we first need to apply transitivity pushback to perform inversion. But then, the result of inversion will yield another imprecise derivation

$$T_1 <: U <: T_2$$

which may be bigger than the original derivation due to transitivity pushback. So, we cannot process the result of the inversion further during an induction. This increase in size is a well-known property of cut elimination: removing uses of a cut rule (like our transitivity axiom) from a proof term may increase the proof size exponentially.

This is why type selections use the abstract variable type assignment, which disallows packing, relying on unpacking and subtyping instead.

Furthermore, for sub-derivations on a self type, the abstract context must be restricted to not include any binding added after the binding for the self type, so that the pushback lemma can be applied during substitution. In fact, this last restriction in the model is not just a technical

device, it seems reasonable for soundness. Indeed, a self type might use a type that has bad bounds within a definition (for example, in a function parameter type, or a type member alias). When subtyping two recursive types, such nonsensical types might be added to the abstract context, but we do not want to unpack a self type by using such temporary bad evidence.

4.4.6 Alternative: Invertible Concrete Variable Typing

In the preceding sections, the presence of the subsumption rule in type assignment required us to prove various canonical forms lemmas. It is also possible to turn this around: design a type assignment relation for concrete variables that is directly invertible, and prove the subsumption property (upwards-closure with respect to subtyping) as a lemma.

Here we sketch this alternative, focusing on the set up of the auxiliary relation and lemmas rather than the proof details. Chapter 3 also follows this alternative.

Invertible Concrete Variable Type Assignment We define the concrete variable type assignment, $\rho \vdash y :_! T$, to be directly invertible by excluding subsumption, and instead relating a value in the store to each of its possible type. There is one case per syntactic form, except no case for $\bot$ and two cases for union types.

Definition 12 (Concrete Variable Type Assignment)**.** $\rho \vdash y :_! T$ is defined inductively by the following rules.

$$\frac{y \in \rho}{\rho \vdash y :_! \top} \quad \text{(V-Top)}$$

$$\frac{\rho(y) \ni (L = T) \qquad \rho \not\emptyset \vdash S <: T \ , \ T <: U}{\rho \vdash y :_! (L : S..U)} \quad \text{(V-Typ)}$$

$$\frac{\begin{array}{c}(y, \overline{d^y}) \in \rho \\ \text{(labels disjoint)} \qquad \forall i, 1 \le i \le n \\ \rho \not\emptyset, (x : T_1^x \wedge \ldots \wedge T_n^x) \vdash d_i^x : T_i^x \\ \exists j,\ d_j^x = (m(z : S) = t^z) \quad T_j^x = m(z : S^x) : U^{x,z} \\ \rho \not\emptyset \vdash S' <: S^y \qquad \rho \not\emptyset, (z : S') \vdash U^{y,z} <: U'^z\end{array}}{\rho \vdash y :_! (m(x : S') : U'^x)} \quad \text{(V-Fun)}$$

$$\frac{\rho(x) \ni (L = T) \qquad \rho \vdash y :_! T}{\rho \vdash y :_! (x.L)} \quad \text{(V-Sel)}$$

$$\frac{\rho \vdash y :_! T^y}{\rho \vdash y :_! \{z \Rightarrow T^z\}} \quad \text{(V-Bind)}$$

$$\frac{\rho \vdash y :_! T_1 \ , \ y :_! T_2}{\rho \vdash y :_! T_1 \wedge T_2} \quad \text{(V-And)}$$

$$\frac{\rho \vdash y :_! T_1}{\rho \vdash y :_! T_1 \vee T_2} \quad \text{(V-OR1)}$$

$$\frac{\rho \vdash y :_! T_2}{\rho \vdash y :_! T_1 \vee T_2} \quad \text{(V-OR2)}$$

Bootstrapping Substitution and Widening We proceed in a way similar to Section 4.4.2 to bootstrap a mutual induction, but this time with swapped assumptions: canonical forms properties are now immediate, but we need to assume a widening (i.e. subsumption) capability.

Definition 13 (V-BIND Metric)**.** Let $\rho \vdash_{\leq m} y :_! S$ denote a derivation $\rho \vdash y :_! S$ with no more than m uses of (V-BIND).

Definition 14 (Widening)**.** Let Widen(m) denote:

$$\frac{\rho \vdash_m y :_! T \qquad \rho\, \emptyset \vdash T <: U}{\rho \vdash_{\leq m} y :_! U}$$

We prove three substitution lemmas, analogous to Lemmas 24,25, and 26, but predicated on Widen(m), and with respect to concrete variable typing ($\rho \vdash y :_! T$) instead of term typing ($\rho\, \emptyset \vdash y : T$).

Lemma 30 (Substitution for Subtyping)**.**

$$\frac{\rho \vdash_{\leq m} y :_! S^y \qquad \text{Widen}(m) \qquad \rho\, z : S^z, \Gamma^z \vdash {T_1}^z <: {T_2}^z}{\rho\, \Gamma^y \vdash {T_1}^y <: {T_2}^y}$$

Lemma 31 (Substitution for Abstract Variable Typing)**.**

$$\frac{\rho \vdash_{\leq m} y :_! S^y \qquad \text{Widen}(m) \qquad z' \neq z \qquad \rho\, z : S^z, \Gamma^z \vdash z' :_! T^z}{\rho\, \Gamma^y \vdash z' :_! T^y}$$

Lemma 32 (Substitution for Concrete Variable Typing)**.**

$$\frac{\text{Widen}(m) \qquad \rho \vdash_{\leq m} y :_! S^y \qquad \rho\, z : S^z, \Gamma^z \vdash z :_! T^z}{\rho \vdash_{\leq m} y :_! T^y}$$

Note that the abstract context in the premise of Lemma 32 disappears entirely from the conclusion, which is about concrete type assignment. This gives another intuition why the model restricts the abstract context when typing recursive types for type selections. Indeed, the extra abstract bindings that come from subtyping recursive types within recursive types might cause more derivations to hold, via lattice collapsing, in the abstract than in the concrete.

We can now prove that a general widening or subsumption rule is admissible.

Lemma 33 (Concrete Variable Widening)**.** $\forall m.\text{Widen}(m)$, i.e.:

$$\frac{\rho \vdash_m y :!\ T \qquad \rho\ \emptyset \vdash T <: U}{\rho \vdash_{\leq m} y :!\vdash U}$$

Finally, we can relate the normal typing relation and our concrete variable typing.

Lemma 34 (Concrete Variable Typing Inversion)**.** Term typing of a concrete variable in an empty abstract context implies the same concrete variable type assignment.

$$\frac{\rho\ \emptyset \vdash y : T}{\rho \vdash y :!\ T}$$

4.5 Perspectives

4.5.1 DOT is Sound, but is Scala Sound?

It is not always clear how well results from a small formal model translate to a realistic language. In the case of DOT, the interleaved process of designing and proving sound a prescriptive core model of Scala's type system has provided valuable insights into the design space. Through debugging DOT models, we have uncovered several soundness issues in Scala. We have already discussed problems related to `null` values in Section 2.3. While it can be argued that `null` is a fundamentally unsound feature anyways, and therefore soundness issues involving `null` may be acceptable, we give two further examples here, which use only safe language features and thus constitute uncontroversial soundness violations.

In DOT, type members in new instances are restricted to aliases, so that "good bounds" are enforced syntactically rather than semantically. This restriction was added after it became clear that subtle situations could arise during object initialization with recursive types, where runtime contexts would be polluted by "bad" evidence that is temporary or justified only through circular reasoning. For a similar reason, there is no susumption in member initialization, so type-checking at object creation (TNEW) can tie the recursive knot, without needing to check bounds.

Scala was designed and implemented before all these corner cases became apparent and allows more flexible bounds. Bounds are checked to be "good", but these checks are not sufficient. Here is an example where a concrete object with "bad bounds" slips through in Scala 2.11.8, causing a cast exception at runtime.

```
1 trait O { type A >: Any <: B; type B >: A <: Nothing }
2 val o = new O {}
3 def id(a: Any): Nothing = (a: o.B)
4 val n: Int = id("Boom!") // runtime cast exception
```

As another example, Scala allows `lazy vals` in paths of type selections, while trying to enforce realizability to prevent unsoundness due to "bad bounds" on non-terminating lazy paths that

are never forced but appear in types. But from DOT, we know that realizability is not preserved by narrowing, and with a bit of work, we can exploit this insight and demonstrate the pitfalls of this approach.

```
trait A { type L <: Nothing }
trait B { type L >: Any}
trait U {
  type X <: B
  val p: X
  def id(x: Any): p.L = x // used in plausible context
}
trait V extends U {
  type X = B with A // unrealizable
  lazy val p: X = p // non-terminating
}
val v = new V {}
val n: Int = v.id("Boom!") // runtime cast exception
```

To conclude this section, we believe that formal work on core language models is important and, even though we do not and cannot consider a full language, this work still helps making full languages safer. In addition, formal work can also chart new territory and lead to more general and more regular full languages, by (cautiously) suggesting that a feature may be safer than previously thought and relaxing some constraints. For example, structural recursive types are more expressive in our DOT model than in Scala.

4.5.2 Scaling up: The Road Ahead

While we have seen in Section 2.3 that DOT can encode a large class of realistic Scala programming patterns, a number of non-quintessential but practically relevant features have been (sometimes deliberately) left out of DOT's formal model. Below is an attempt to classify these features according to their ease of integration with DOT and potential implications for type soundness.

Largely Orthogonal Features (1) Traits, Classes, Inheritance and Mixins: DOT does not model any "implementation reuse" mechanisms such as inheritance and mixins (which Scala has) and prototype dispatch (which does not currently exist in Scala, but would be interesting to consider). Most likely, such extensions will be through encodings, showing type preservation of a translation, and not through extending the DOT meta-theory itself. Some challenges: type member inheritance vs "good bounds" at creation time, open construction of nominal hierarchies. (2) Mutable State: this will require standard extensions to the operational semantics, and it is important that type selections remain *stable*, i.e. exclude mutable variables. As is standard, the interaction of mutation and polymorphism requires care [Summers, 2009], but previous work [Rompf and Amin, 2015] has shown that mutation can be added to DOT-like type systems without interfering with soundness. (3) Implicits: Scala has both implicit parameters and implicit conversions, which are automatically inserted by the compiler based

on types and scopes. Implicit parameters are useful for modeling type classes. Implicits do not introduce new types as typing rules and are unlikely to interfere with soundness.

Fitting Extensions (1) Full Paths: extend type selections to include chains of immutable field selections `x.a.b.C` in addition to variables `x.A`. Some challenges: path equality and reduction in type selections, field initialization, circular reasoning with self occurrences. (2) Singleton Types: `x.type` in addition to `x.A`, denoting the type that is only inhabited by the value of `x`. An interesting question is whether singleton types are already encodable in DOT.

Restricted Extensions (1) Laziness: DOT relies on the assumption of call-by-value evaluation. To model Scala's `lazy vals`, some restrictions are necessary. As a first approximation one can forbid type selections on `lazy vals`, but relaxations in combination with traits and class types may be possible, though challenging in terms of balancing flexibility and safety. Another avenue is deliberately forcing evaluation of `lazy vals` that occur in types. (2) By-Name Arguments: must not occur in type selections.

Debatable Extensions Type Projections, Type Lambdas and Higher-Kinded Types: These features are most likely not faithfully encodable in DOT. Type projections `T#A` are similar to type selections `x.A`, but crucially lose the guarantee of variable `x` pointing to an existing object. Thus, it is presently unclear how to approach soundness of type projections. For type lambdas, what is missing is a way to calculate the type resulting from a type application. This could be encoded through type projections or through dependent types beyond just paths. Simplified models are readily doable (as outlined in the introduction of Section 2.3), so it is not clear whether the unrestricted encoding is worth the extra complexity.

Incompatible Extensions As discussed in Section 2.3 and elsewhere [Amin and Tate, 2016], Scala's oblivious treatment of `null` pointers, which mainly exists for backwards compatibility with Java, seems to be fundamentally unsound. Future version of Scala can achieve soundness by (1) reflecting nullability in the type system, e.g. via union types, (2) preventing nullable expressions in type selections, similar to mutable variables or by-name arguments, and (3) tightening the rules for object member initialization so that `null` values cannot be observed during object creation.

5 Type Soundness Proofs with Definitional Interpreters

While type soundness proofs are taught in every graduate PL class, the gap between realistic languages and what is accessible to formal proofs is large. In the case of Scala, it has been shown that its formal model, the Dependent Object Types (DOT) calculus, cannot simultaneously support key metatheoretic properties such as environment narrowing and subtyping transitivity, which are usually required for a type soundness proof. Moreover, Scala and many other realistic languages lack a general substitution property.

The first contribution of this chapter is to demonstrate how type soundness proofs for advanced, polymorphic, type systems can be carried out with an operational semantics based on high-level, definitional interpreters, implemented in Coq. We present the first mechanized soundness proofs in this style for System $F_{<:}$ and several extensions, including mutable references. Our proofs use only straightforward induction, which is significant, as the combination of big-step semantics, mutable references, and polymorphism is commonly believed to require coinductive proof techniques.

The second main contribution of this chapter is to show how DOT-like calculi emerge from straightforward generalizations of the operational aspects of $F_{<:}$, exposing a rich design space of calculi with path-dependent types inbetween System F and DOT, which we dub the System D Square.

By working directly on the target language, definitional interpreters can focus the design space and expose the invariants that actually matter at runtime. Looking at such runtime invariants is an exciting new avenue for type system design.

The main contribution of this chapter is to demonstrate how type soundness for advanced, polymorphic, type systems can be proved with respect to an operational semantics based on high-level, definitional interpreters, implemented directly in a total functional language like Coq. While this has been done before for very simple, monomorphic, type systems [Siek, 2013, Danielsson, 2012], we are the first to demonstrate that, with some additional machinery, this approach scales to realistic, polymorphic type systems that include subtyping, abstract types,

types with binders, mutable references, exceptions, and certain forms of dependent types.

Our motivation is twofold. The first is intellectual: It is commonly believed that proofs based on big-step semantics (of which definitional interpreters are a proper sub-category) are difficult or unsatisfactory. One criticism is that big-step semantics do not distinguish errors from nontermination. Thus, if taken as the basis of a type soundness proof, what is shown is only preservation, but not progress. Hence, the result is significantly weaker than a comparable small-step proof. Another commonly held belief is that advanced features such as mutable references require coinduction or other non-standard proof techniques in big-step.

With this chapter, we present convincing evidence that, contrary to this view, definitional interpreters have no inherent drawbacks to small-step semantics for deterministic languages: making evaluation functional, instead of relational, and parameterizing over a step counter to make evaluation total, enables precise distinction between timeouts, errors, and normal values, leading to strong soundness results. Established techniques like monads [Moggi, 1991] can be used to abstract over these case distinctions and keep the interpreter implementation elegant [Wadler, 1992]. Furthermore, auxiliary invariants such as store typings can be used in the same way as in small-step proofs to eliminate the need for non-standard proof techniques, and gradually extend proofs for simple languages with multiple advanced features, without any disruptive changes to the proof structure.

Our second motivation is pragmatic: While type soundness proofs are taught in every graduate PL class, the gap between realistic languages and what is accessible to formal proofs is large. In the case of Scala, it has been shown that its formal model, the Dependent Object Types (DOT) calculus, cannot simultaneously support key metatheoretic properties such as environment narrowing and subtyping transitivity, which are usually required for a type soundness proof. Moreover, Scala and many other realistic languages lack a general substitution property. Thus, applying the Wright & Felleisen method [Wright and Felleisen, 1994] requires ingenuity to extend the language syntax and type system with auxiliary constructs to support subject reduction. It also requires at least an informal argument of adequacy, showing that the syntactic theory faithfully models the intended language semantics.

In the case of Scala, developing a sound formal model that captures the essence of its type system was an open problem for more than a decade. Over the years, a handful of talented post-docs and students tried many variations of syntactic theories, but ultimately failed to find one they could prove sound. The situation changed when looking at definitional interpreters. The split between the static and the runtime world exposed the key invariants that had to be maintained, identified problems with previous proof attempts that were not apparent in small-step, and finally lead to the first soundness proof for DOT. Moreover, starting from the runtime invariants has lead to a simpler and more regular calculus. The resulting proof is easily translated to small-step, and has been presented in detail in Chapter 4. While the resulting syntactic theory looks pleasant and blindingly obvious in hindsight, nobody had thought of this particular variant before.

The lesson we draw from this is again two-fold: First, by working directly on the target language (instead of an augmented version) definitional interpreters can constrain the design space in a good way and expose the invariants that actually matter at runtime. Second, looking at such runtime invariants is an exciting new avenue for type system design. In this chapter, we illustrate how DOT-like calculi emerge as generalizations of the static typing rules to fit the operational typing aspects of the $F_{<:}$ based system—in some cases almost like removing artificial restrictions. We expose a rich design space of calculi with path-dependent types inbetween System F and DOT, which we dub the System D Square. By this, we put Scala and DOT on a firm theoretical foundation grounded in existing, well-studied, type systems, alluding to Wadler's point that "good languages are discovered, not invented" [Wadler, 2015].

We make the following contributions:

- We discuss limitations of the syntactic approach to soundness and review a proof strategy based on high-level definitional interpreters (Section 5.1).
- We demonstrate that this proof strategy scales to advanced polymorphic type systems, presenting the first soundness proof for $F_{<:}$ in this style (Section 5.2).
- We further show that extensions such as mutable references and exceptions can be added gradually and without much difficulty, requiring only straightforward induction, and effectively separating failures from divergence (Section 5.3).
- We now take the opposite direction and investigate how the internal invariants in the $F_{<:}$ proof yield new static type systems. Enabling lower-bounded quantification (System $F_{<:>}$) leads to user-definable subtyping theories. Unifying term and type variables (System D) leads to path-dependent types (Section 5.4).
- We discuss how the combination of these two extensions (System $D_{<:>}$) can be identified as the core of Scala and DOT, which we have rediscovered from first principles. We thus put DOT on a firm theoretical grounding, by showing how it emerges as a generalization of the static typing rules of $F_{<:}$ to its runtime typing aspects (Section 5.5).

5.1 Definitional Interpreters for Type Soundness

Today, the dominant method for proving soundness of a type system is the syntactic approach of Wright and Felleisen [Wright and Felleisen, 1994]. Its key components are the progress and preservation lemmas with respect to a small-step operational semantics based on term rewriting. While this syntactic approach has a lot of benefits, as described in great detail in the original 1994 paper [Wright and Felleisen, 1994], there are also some drawbacks. An important one is that reduction semantics often pose a question of adequacy: realistic language implementations do not proceed by rewriting, so if the aim is to model an existing language, at least an informal argument needs to be made that the given reduction relation faithfully implements the intended semantics. Furthermore, few realistic languages actually enjoy the subject reduction property. If simple substitution does not hold, the syntactic approach is more difficult to apply and requires stepping into richer languages in ways that are often non-obvious. Again, care must be taken that these richer languages are self-contained and match the original intention.

To motivate a substitution-free semantics, we present three examples next where naive substitution does not preserve types:

Example 1: Return statements Consider a simple program in a language with `return` statements:

```
def fun(c) = if (c) return x; y
fun(true)
```

Taking a straightforward small-step execution strategy, this program will reduce to:

```
⟶ if (true) return x; y
```

But now the `return` has become unbound. We need to augment the language and reduce to an auxiliary construct like this:

```
⟶ scope { if (true) return x; y }
```

This means that we need to work with a richer language than we had originally intended, with additional syntax, typing, and reduction rules like the following:

```
scope E[ return v ] ⟶ v          scope v ⟶ v
```

Example 2: Private members As another example, consider an object-oriented language with access modifiers.

```
class Foo { private val data = 1;  def foo(x) = x * this.data }
```

Starting with a term

```
val a = new Foo; a.foo(7) / S
```

where `S` denotes a store, small-step reduction steps may lead to:

```
1    ⟶ l0.foo(7)                / S, (l0 -> Foo(data=1))
2    ⟶ x * l0.data              / S, (l0 -> Foo(data=1))
```

But now there is a reference to private field `data` outside the scope of class `Foo`.

We need a special rule to ignore access modifiers for 'runtime' objects in the store, versus other expresssions that happen to have type `Foo`. We still want to disallow `a.data` if `a` is a normal variable reference or some other expression.

Example 3: Method overloading Looking at a realistic language, many type preservation issues are documented in the context of Java, which were discussed at length on the Types mailing list [von Oheimb, 1998], back in the time when Java's type system was an object of study.

Most of these examples relate to static method overloading, and to Java's conditional expressions `c ? a : b`, which require `a` and `b` to be in a subtype relationship because Java does not have least upper bounds. It is worth noting that these counterexamples to preservation are not actual type safety violations.

5.1.1 Alternative Semantic Models

So what can we do if our object of study does not naturally fit a rewriting and substitution model of execution? Of course one option is to make it fit (perhaps with force), but an easier path may be to pick a different semantic model. Before the syntactic approach, denotational semantics [Scott, 1982] and Kahn's natural semantics (or 'big-step' semantics) [Kahn, 1987] were the tools of the trade.

Big-step semantics in particular has the benefit of being more 'high-level', in the sense of being closer to actual language implementations. Environment-based formulations are as natural as substitution-based ones, and have the additional advantage of working with unmodified syntax of the target language. The downside of big-step semantics for soundness proofs is that failure cases and nontermination are not easily distinguished. This often requires tediously enumerating all possible failure cases, which may cause a blow-up in the required rules and proof cases. Moreover, in the history of big-step proofs, advanced language features such as recursive references have required specific proof techniques (e.g. coinduction) [Tofte, 1988] which made it hard to compose proofs for different language features. In general, polymorphic type systems pose difficulties for substitution-free semantics, because type variables need to be related across different contexts.

But the circumstances have changed since 1994. Today, most formal work is done in proof assistants such as Coq, and no longer with pencil and paper. This means that we can use software implementation techniques like monads (which, ironically were developed in the context of denotational semantics [Moggi, 1991]) to handle the complexity of failure cases

[Wadler, 1992]. Moreover, using simple but clever inductive techniques such as step counters we can avoid the need for coinduction and other complicated techniques in many cases.

In the following, we present our approach to type soundness proofs with definitional interpreters in the style of Reynolds [Reynolds, 1998]: high-level evaluators implemented in a (total) functional language. As we will see, in a functional system such as Coq or Agda, we can implement such evaluators quite naturally.

5.1.2 Simply Typed Lambda Calculus: Siek's 3 Easy Lemmas

We build our exposition on Siek's type safety proof for a dialect of simply typed lambda calculus (STLC) [Siek, 2013], which in turn takes inspiration from Ernst, Ostermann and Cook's semantics in their formalization of virtual classes [Ernst et al., 2006].

The starting point is a fairly standard definitional interpreter for STLC, shown in Figure 5.1 together with the STLC syntax and typing rules. We opt to show the interpreter in actual Coq syntax, but stick to formal notation for the language definition and typing rules. The interpreter consists of three functions: one for primitive operations (which we elide), one for variable lookups, and one main evaluation function `eval`, which ties everything together. Instead of working exclusively on terms, as a reduction semantics would do, the interpreter maps terms to a separate domain of values `v`. Values include primitives, and closures, which pair a term with an environment.

Notions of Type Soundness What does it mean for a language to be type safe? We follow Wright and Felleisen [Wright and Felleisen, 1994] in viewing a static type system as a filter that selects well-typed programs from a larger universe of untyped programs. In their definition of type soundness, a partial function `evalp` defines the semantics of untyped programs, returning `Error` if the evaluation encounters a type error, or any other answer for a well-typed result. We assume here that the result in this case will be `Val v`, for some value `v`. For evaluations that do not terminate, `evalp` is undefined.

The simplest soundness property states that well-typed programs do not go wrong.

Definition 15 (Weak soundness)**.**

$$\frac{\emptyset \vdash e : T}{\text{evalp } e \neq \text{Error}}$$

A stronger soundness property states that if the evaluation terminates, the result value must have the same type as the program expression, assuming that values are classified by types as well.

Definition 16 (Strong soundness)**.**

Syntax

$$
\begin{array}{lcl}
T & ::= & B \mid T \to T \\
t & ::= & c \mid x \mid \lambda x : T.t \mid t\ t \\
v & ::= & c \mid \langle H, \lambda x : T.t \rangle \\
r & ::= & \text{Timeout} \mid \text{Done}\,(\text{Error} \mid \text{Val}\ v) \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T \\
H & ::= & \emptyset \mid H, x : v
\end{array}
$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash c : B$$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2\ ,\ t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

Consistent environments $\boxed{\Gamma \vDash H}$

$$\emptyset \vDash \emptyset$$

$$\frac{\Gamma \vDash H \qquad v : T}{\Gamma, x : T \vDash H, x : v}$$

Value type assignment $\boxed{v : T}$

$$c : B$$

$$\frac{\Gamma \vDash H \qquad \Gamma, x : T_1 \vdash t : T_2}{\langle H, \lambda x : T_1.t \rangle : T_1 \to T_2}$$

Definitional Interpreter

```
(* Coq data types and auxiliary functions elided *)
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}:
option (option vl) :=
  DO n1 ⇐ FUEL n;                              (* totality    *)
  match t with
    | tcst c      ⇒ DONE VAL (vcst c)          (* constant    *)
    | tvar x      ⇒ DONE (lookup x env)        (* variable    *)
    | tabs x ey   ⇒ DONE VAL (vabs env x ey)   (* lambda      *)
    | tapp ef ex  ⇒                            (* application *)
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex;
        match vf with
          | (vabs env2 x ey) ⇒
            eval n1 ((x,vx)::env2) ey
          | _ ⇒ ERROR
        end
  end.
```

Figure 5.1 – STLC: Syntax and Semantics

$$\frac{\emptyset \vdash e : T \qquad \text{evalp } e = r}{r = \text{Val } v \qquad v : T}$$

In our case, assigning types to values is achieved by the rules in the lower half of Figure 5.1.

Partiality Fuel To reason about the behavior of our interpreter, and to implement it in Coq in the first place, we have to get a handle on potential nontermination, and make the interpreter a total function. Again we follow Siek [Siek, 2013] by first making all error cases explicit by wrapping the result of each operation in an `option` data type with alternatives `Val v` and `Error`. This leaves us with possible nontermination. We parameterize the interpreter over a step index or 'fuel value' `n`, which bounds the amount of work the interpreter is allowed to do. If it runs out of fuel, the interpreter returns `Timeout`, otherwise `Done r`, where `r` is the option type introduced above.

It is convenient to treat this type of answers as a (layered) monad and write the interpreter in monadic `do` notation (as done in Figure 5.1). The `FUEL` operation in the first line desugars to a simple non-zero check:

```
1    match n with
2      | z ⇒ TIMEOUT
3      | S n1 ⇒ ...
4    end
```

There are other ways to define monads that encode partiality (e.g. a coinductively defined partiality monad [Danielsson, 2012]), but this simple method has the benefit of enabling easy inductive proofs about all executions of the interpreter, by performing a simple induction over n. If a fact is proved for all executions of length n, for all n, then it must hold for all finite executions. Specifically, infinite executions are by definition not 'stuck'.

Proof Structure For the type safety proof, the 'three easy lemmas' [Siek, 2013] are as follows. There is one lemma per function of the interpreter.

Lemma 35. Well-typed primitive operations are not stuck and preserve types.

We are omitting primitive operations for simplicity, so we skip this lemma.

Lemma 36. Well-typed environment lookups are not stuck and preserve types.

$$\frac{\Gamma \vDash H \qquad \text{lookup } x\ \Gamma = \text{Some } T}{\text{lookup } x\ H = \text{Val } v \qquad v : T}$$

Proof. By structural induction over the environment and case distinction on whether the lookup succeeds. □

Lemma 37. For all n, if the interpreter returns a result that is not a timeout, then the result is a value (i.e. not stuck) and it is well-typed.

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \text{eval } n\ H\ e = \text{Done } r}{r = \text{Val } v \qquad v : T}$$

Proof. By induction on n, and case analysis on the term e. □

It is easy to see that this lemma corresponds to the strong soundness notion (Definition 16). In fact, we can define a partial function evalp e to probe eval $n\ \emptyset\ e$ for all $n = 0, 1, 2, \ldots$ and return the first non-timeout result, if one exists. Restricting to the empty environment then yields *exactly* Wright and Felleisen's statement of strong soundness [Wright and Felleisen, 1994]:

Theorem 6. (Soundness of STLC)

$$\frac{\emptyset \vdash e : T \qquad \text{evalp } e = r}{r = \text{Val } v \qquad v : T}$$

Using classical reasoning we can conclude that either evaluation diverges (i.e. it times out for all n), or there exists an n for which the result is well-typed.

The mechanized soundness proof is available from http://sound-big-step-stlc.namin.net.

5.2 Type Soundness for System $F_{<:}$

We now turn our attention to System $F_{<:}$ [Cardelli et al., 1994], moving beyond type systems that have been previously formalized with definitional interpreters. We pick $F_{<:}$ because it combines a range of interesting features, because its type soundness is well-studied, in particular through the POPLmark challenge [Aydemir et al., 2005], and because $F_{<:}$ can serve as a basis for extensions that lead up to formalizations of key Scala features (see Sections 5.4 and 5.5).

The syntax and static typing rules of $F_{<:}$ are defined in Figure 5.2. In addition to STLC, we have type abstraction and type application, and subtyping with upper bounds. The calculus is more expressive than STLC and more interesting from a formalization perspective, in particular because it contains type variables. These are bound in the environment, which means that we need to consider types in relation to the environment they were defined in.

5.2.1 Operational Semantics: The Definitional Interpreter

What would be a suitable runtime semantics for passing type arguments to type abstractions? The usual small-step semantics uses substitution to eliminate type arguments (Figure 5.3):

$$(\Lambda Y <: T_1.t)[T_2] \longrightarrow [T_2/Y]t$$

We could do the same in our definitional interpreter, which we extend from Figure 5.1 to add a new case for type application:

```
(* ... *)
    | ttapp ef T ⇒
      DO vf ⇐ eval n1 env ef;
        match vf with
          | (vtabs env2 x ey) ⇒
              (* ... first attempt ... *)
              eval n1 env2 (substitute ey x T)
          | _ ⇒ ERROR end
(* ... *)
```

But then, the interpreter would have to modify program terms at runtime. This would be odd for an interpreter, which is meant to be simple, and it would conflict with our goal of a *substitution-free* semantics, to circumvent difficulties in formal reasoning due to substitution in small-step semantics (see Section 5.1).

Types in Environments A better idea, more consistent with an environment-passing interpreter, is to put the type argument into the runtime environment as well:

```
              (* ... second attempt ... *)
              eval n1 ((x,vty T)::env2) ey
```

Syntax

$$
\begin{array}{lcl}
X & ::= & Y \mid Z \\
T & ::= & X \mid \top \mid T \to T \mid \forall Z <: T.T^Z \\
t & ::= & x \mid \lambda x : T.t \mid \Lambda Y <: T.t \mid t\ t \mid t\ [T] \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X <: T
\end{array}
$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash S <: \top$$

$$\Gamma \vdash X <: X$$

$$\frac{\Gamma \ni X <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash S_2 <: S_1\ ,\ T_1 <: T_2}{\Gamma \vdash (S_1 \to T_1) <: (S_2 \to T_2)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, Z <: S_2 \vdash T_1^Z <: T_2^Z}{\Gamma \vdash (\forall Z <: S_1.T_1^Z) <: (\forall Z <: S_2.T_2^Z)}$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x : S.t : S \to T}$$

$$\frac{\Gamma \vdash t_1 : S \to T\ ,\ t_2 : T}{\Gamma \vdash t_1\ t_2 : T}$$

$$\frac{\Gamma, Y <: S \vdash t : T^Y}{\Gamma \vdash \Lambda Y <: S.t : \forall Z <: S.T^Z}$$

$$\frac{\Gamma \vdash t_1 : \forall Z <: U.T^Z\ ,\ T_2 <: U}{\Gamma \vdash t_1[T_2] : T^{T_2}}$$

$$\frac{\Gamma \vdash t : S\ ,\ S <: T}{\Gamma \vdash t : T}$$

Figure 5.2 – $F_{<:}$: syntax and static semantics

Value Syntax

$$v \quad ::= \quad \lambda x : T.t \mid \Lambda Y <: T.t$$

Reduction $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2} \qquad \frac{t_1 \longrightarrow t_1'}{t_1[T_2] \longrightarrow t_1'[T_2]}$$

$$(\Lambda Y <: T_1.t_1)[T_2] \longrightarrow [T_2/Y]t_1$$

$$(\lambda x : T_1.t_1)\ v_2 \longrightarrow [v_2/x]t_1$$

Figure 5.3 – $F_{<:}$: small-step semantics (call-by-value)

However, this leads to a problem: the type `T` may refer to other type variables that are bound in the current environment at the call site (`env`), while evaluation switches to the environment at definition site (`env2`) of the type abstraction called. We could potentially resolve all the references, and substitute their occurrences in the type, but this will no longer work if types can be recursive. In any case, we wouldn't want to do such type resolution and manipulation in the interpreter, i.e. during evaluation. So instead, we pass the caller environment along with the type.

```
1               (* ... final attempt ... *)
2               eval n1 ((x,vty env T)::env2) ey
```

In effect, type arguments $\langle H, T \rangle$ (written `vty env T` in the code above) become very similar to function closures, in that they close over their defining environment. Intuitively, this makes a lot of sense, and is consistent with the overall workings of our interpreter.

Semantic Equivalence As a reassurance that our definitional interpreter faithfully implements the well-known small-step call-by-value semantics of $F_{<:}$ (Figure 5.3), we prove a semantic equivalence theorem.

We first define a correspondence relation $\sim$ between closure values of the interpreter and closed lambda terms and type abstractions, such that the closure environment is recursively substituted into the term. This makes the nature of the runtime environment as delayed substitution explicit. The $\sim$ relation is then lifted to interpreter results r, such that `Done (Val v)` corresponds to a term value `v`, `Done Error` to any stuck term (irreducible non-value), and `Timeout` to any term.

Theorem 7 (Semantic Equivalence)**.** If $e \rightarrow^* e'$, then $\exists n.\text{eval}\ n\ \emptyset\ e \sim e'$, and if $\text{eval}\ n\ \emptyset\ e = r$, then $\exists e'.e \rightarrow^* e'$ with $r \sim e'$.

Proof. The first direction is by induction over the $\rightarrow^*$ derivation, the second direction is by

Syntax

$$\begin{array}{lcl} v & ::= & \langle H, \lambda x : T.t \rangle \mid \langle H, \Lambda Y <: T.t \rangle \\ H & ::= & \emptyset \mid H, x : v \mid H, Y = \langle H, T \rangle \\ J & ::= & \emptyset \mid J, Z <: \langle H, T \rangle \end{array}$$

Runtime subtyping $\boxed{J \vdash H_1\ T_1 <: H_2\ T_2}$

$$\frac{}{J \vdash H_1\ T <: H_2\ \top} \qquad \frac{J \vdash H_2\ S_2 <: H_1\ S_1 \qquad J \vdash H_1\ T_1 <: H_2\ T_2}{J \vdash H_1\ (S_1 \to T_1)\ <:\ H_2\ (S_2 \to T_2)}$$

$$\frac{J \vdash H_2\ S_2 <: H_1\ S_1 \qquad J, Z <: \langle H_2, S_2 \rangle \vdash H_1\ T_1^Z\ <:\ H_2\ T_2^Z}{J \vdash H_1\ (\forall Z <: S_1.T_1^Z)\ <:\ H_2\ (\forall Z <: S_2.T_2^Z)}$$

Abstract type variables

$$\frac{}{J \vdash H_1\ Z <: H_2\ Z} \qquad \frac{J \ni Z <: \langle H, U \rangle \qquad J \vdash H\ U <: H_2\ T}{J \vdash H_1\ Z <: H_2\ T}$$

Concrete type variables

$$\frac{H_1 \ni Y_1 = \langle H, T \rangle \qquad H_2 \ni Y_2 = \langle H, T \rangle}{J \vdash H_1\ Y_1 <: H_2\ Y_2} \qquad \frac{J \vdash H_1\ T <: H\ S \qquad H_2 \ni Y = \langle H, S \rangle}{J \vdash H_1\ T <: H_2\ Y}$$

$$\frac{H_1 \ni Y = \langle H, U \rangle \qquad J \vdash H\ U <: H_2\ T}{J \vdash H_1\ Y <: H_2\ T}$$

Transitivity

$$\frac{J \vdash H_1\ T_1 <: H_2\ T_2 \qquad H_2\ T_2 <: H_3\ T_3}{J \vdash H_1\ T_1 <: H_3\ T_3}$$

Consistent environments $\boxed{\Gamma \vDash H\ J}$

$$\frac{}{\emptyset \vDash \emptyset\ \emptyset} \qquad \frac{\Gamma \vDash H\ \emptyset \qquad \emptyset \vdash H_1\ T_1 <: H\ T}{\Gamma, Y <: T \vDash (H, Y = \langle H_1, T_1 \rangle)\ \emptyset}$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad H \vdash v : T}{\Gamma, x : T \vDash (H, x : v)\ \emptyset} \qquad \frac{\Gamma \vDash H\ J}{\Gamma, Z <: T \vDash H\ (J, Z <: \langle H, T \rangle)}$$

Value type assignment $\boxed{H \vdash v : T}$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, x : S \vdash t : T}{H \vdash \langle H, \lambda x : S.t \rangle : S \to T} \qquad \frac{H_1 \vdash v : T_1 \qquad \emptyset \vdash H_1\ T_1 <: H_2\ T_2}{H_2 \vdash v : T_2}$$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, Y <: S \vdash t : T^Y}{H \vdash \langle H, \Lambda Y <: S.t \rangle : \forall Z <: S.T^Z}$$

Figure 5.4 – $F_{<:}$: runtime typing

induction over n. □

5.2.2 Runtime Invariants

Having defined a suitable interpreter, the next step is to identify and model runtime invariants including value typing that will enable our soundness proof. For $F_{<:}$, the key novel features compared to STLC are subtyping and abstract types.

Runtime Subtyping Since we model runtime type arguments as closure, we need to account for each defining environment when comparing types at runtime. In our approach, on each side, the runtime subtyping judgement takes not only a type but also its defining environment:

$$J \vdash H_1\ T_1 <: H_2\ T_2$$

The judgement pairs each type T with a corresponding runtime environment H. Each runtime environment H maps a term variable x to a value v or a type variable Y to a "type" closure, which pairs a (runtime) environment with a type. In the Coq code above, we use the `vty` marker to distinguish between term variables and type variables in the mapping. We will explain the role of the J environments shortly, in the section on abstract types below.

The runtime typing rules are shown in Figure 5.4. Note that the rules labeled 'concrete type variables' are entirely structural: different type variables Y_1 and Y_2 are treated equal if they map to the same $H\ T$ pair. In contrast to the surface syntax of $F_{<:}$, there is not only a rule for $Y <: T$ but also a symmetric one for $T <: Y$, i.e. with a type variable on the right hand side. This symmetry is necessary to model runtime type equality through subtyping, which gives us a handle on those cases where a small-step semantics would rely on type substitution.

Another way to look at this is that the runtime subtyping relation removes abstraction barriers (nominal variables, only one-sided comparison with other types) that were put in place by the static subtyping relation.

We further note in passing that formal reasoning involving subtyping transitivity becomes more difficult in the runtime subtyping compared to the static $F_{<:}$ subtyping, because of type variables in the middle of a chain $T_1 <: Y <: T_2$, which should contract to $T_1 <: T_2$. We will get back to this question and related ones in Section 5.2.3 and specifically Lemma 45.

Abstract Types So far we have seen how to handle types that correspond to existing type objects. We now turn to subtyping for $\forall$ types. In the static subtyping relation, this rule introduces a new type binding: in the premise that compares the result types, the context is extended with a binding for the quantified type variable.

How can we support this rule at runtime? We cannot quite use only the facilities discussed

so far, because this would require us to 'invent' new hypothetical objects $\langle H, T \rangle$, which are not actually created during execution, and insert them into another runtime environment. Furthermore, the premise subtyping the result types should hold not just *for some* hypothetical object, but *for all* hypothetical objects that satisfy the bound on the type variable. Semantically, we do not have an exact type instantiation, only an upper bound, and we should be careful about the distinction.

Our solution is rather simple: we split the runtime environment into abstract (J) and concrete (H) parts. While the environments H, as discussed above, map type variables to concrete type values created at runtime, we use a shared environment J, that maps type variables to *hypothetical* objects: $\langle H, T \rangle$ pairs that may or may not correspond to an actual object created at runtime. For clarity, we use two disjoint alphabets for type variable names: Y for variables bound in terms and Z for variables bound in types. We use X when referring to either kind. The concrete environment (H) is indexed by variables bound in terms (Y) and extended during evaluation or typing. The abstract environment (J) is indexed by variables bound in types (Z) and extended during subtyping.

Implementation-wise, this approach fits quite well with a locally nameless representation of binders [Charguéraud, 2012] that already distinguishes between free and bound identifiers.

The runtime subtyping rules for abstract type variables in Figure 5.4 correspond more or less directly to their counterparts in the static subtype relation (Figure 5.2), modulo addition of the two runtime environments. In particular, like in static subtyping but unlike for concrete type variables in runtime subtyping, there is no subtyping rule for abstract type variables on the right.

5.2.3 Metatheory: Soundness Proof

How do we adapt the soundness proof from Section 5.1.2 to work with this form of runtime subtyping? We need to show that the runtime rules are consistent with the static rules (Lemma 39) and, due to the possibility of subsumption, the main proof can no longer just rely on case analysis of value typing derivations. Hence, we require proper canonical forms lemmas (Lemmas 40,41,42,43).

Relating Static and Runtime Subtyping First, the runtime subtyping should be consistent with the static subtyping.

Lemma 38 (Static to Runtime Subtyping)**.** Static subtyping implies runtime subtyping in well-formed consistent environments.

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vDash H\ J}{J \vdash H\ T_1 <: H\ T_2}$$

Proof. By straightforward structural induction over the static subtyping derivation. Static

rules on Z variables map to corresponding abstract runtime rules. Static rules on Y variables map to corresponding concrete runtime rules. □

Second, for the cases where $F_{<:}$ type assignment relies on substitution in types – specifically, in the result of a type application – we need to replace a hypothetical binding with an actual value.

Lemma 39 (Abstract to Concrete Substitution for Subtyping)**.**

$$\frac{Z <: \langle H, T\rangle, \emptyset \vdash H_1\ T_1^Z <: H_2\ T_2^Z}{\emptyset \vdash H_1, Y_1 = \langle H, T\rangle\ T_1^{Y_1} <: H_2, Y_2 = \langle H, T\rangle\ T_2^{Y_2}}$$

Proof. By induction and case analysis, first strengthening to allow more type variables in abstract context on the right of type variable Z, to be substituted. □

We actually prove a slighty more general lemma that incorporates the option of weakening, i.e. not extending H_i if Z does not occur in T_i, for each side $i = 1, 2$.

Inversion of Value Typing (Canonical Forms) Due to the presence of the subsumption rule, the main proof can no longer just rely on case analysis of the typing derivation, but we need proper inversion lemmas for term and type abstractions.

Lemma 40 (Inversion of Value Typing for Term Abstraction)**.**

$$\frac{H \vdash\ v : S_2 \to T_2}{\begin{array}{c} v = \langle H_c, \lambda x : S_1.t\rangle \qquad \Gamma_c \vDash H_c\ \emptyset \\ \Gamma_c, x : S_1 \vdash\ t : T_1 \qquad \emptyset \vdash H_c\ (S_1 \to T_1) <: H\ (S_2 \to T_2) \end{array}}$$

Lemma 41 (Inversion of Value Typing for Type Abstraction)**.**

$$\frac{H \vdash\ v : \forall Z <: S_2.T_2^Z}{\begin{array}{c} v = \langle H_c, \Lambda Y <: S_1.t\rangle \qquad \Gamma_c \vDash H_c\ \emptyset \\ \Gamma_c, Y <: S_1 \vdash\ t : T_1^Y \\ \emptyset \vdash H_c\ (\forall Z <: S_1.T_1^Z)\ <:\ H\ (\forall Z <: S_2.T_2^Z) \end{array}}$$

We further need to invert the resulting subtyping derivations, so we need additional inversion lemmas for function and ∀ types.

Lemma 42 (Inversion of Runtime Subtyping for Function Types)**.**

$$\frac{\emptyset \vdash H_1\ (S_1 \to T_1) <: H_2\ (S_2 \to T_2)}{\emptyset \vdash H_2\ S_2 <: H_1\ S_1\ ,\ H_1\ T_1 <: H_2\ T_2}$$

Lemma 43 (Inversion of Runtime Subtyping for ∀ Types)**.**

$$\frac{\emptyset \vdash H_1\ (\forall Z{<:}S_1.T_1^Z)\ <:\ H_2\ (\forall Z{<:}S_2.T_2^Z)}{\begin{array}{c}\emptyset \vdash H_2\ S_2 <: H_1\ S_1 \\ \emptyset, Z{<:}\langle H_2, S_2\rangle \vdash H_1\ T_1^Z\ <:\ H_2\ T_2^Z\end{array}}$$

The inversion lemmas we need here depend in a crucial way on transitivity and narrowing properties of the subtyping relation (similar to small-step proofs for $F_{<:}$ [Aydemir et al., 2005]).

Transitivity Pushback and Cut Elimination For the static subtyping relation of $F_{<:}$, transitivity can be proved as a lemma, together with narrowing, in a mutual induction on the size of the middle type in a chain $T_1 <: T_2 <: T_3$ (see e.g. the POPLmark challenge documentation [Aydemir et al., 2005]).

Unfortunately, for the runtime subtyping version, the same proof strategy fails, because runtime subtyping may involve a type variable as the middle type: $T_1 <: Y <: T_3$. This setting is very similar to issues that arise with path-dependent types in Scala and DOT, but here it surprisingly arises already when just looking at the runtime semantics of $F_{<:}$. Since proving transitivity becomes much harder, we adopt a strategy from previous DOT developments [Amin et al., 2014]: admit transitivity as an axiom, but prove a 'pushback' lemma that allows to push uses of the axiom further up into a subtyping derivation, so that the top level becomes invertible. We denote this as *precise* subtyping $T_1 <!\ T_2$.

Definition 17 (Precise Subtyping)**.** If $J \vdash H_1\ T_1 <: H_2\ T_2$ and the derivation does not end in the transitivity rule, then we say that $J \vdash H_1\ T_1 <!\ H_2\ T_2$.

Such a strategy is reminiscent of cut elimination in natural deduction, and in fact, the possibility of cut elimination strategies is already mentioned in Cardelli's original $F_{<:}$ paper [Cardelli et al., 1994].

Lemma 44 (Narrowing)**.**

$$\frac{\begin{array}{cc}J_1 \vdash H_1\ T_1 <: H_2\ T_2 & J_2 \vdash H_3\ T_3 <: H_4\ T_4 \\ J_1 = J_2(Z \to \langle H_1\ T_1\rangle) & J_2 \ni Z <: \langle H_2\ T_2\rangle\end{array}}{J_1 \vdash H_3\ T_3 <: H_4\ T_4}$$

Proof. By structural induction, and using the transitivity axiom. □

Lemma 45 (Transitivity Pushback)**.**

$$\frac{\emptyset \vdash H_1\ T_1 <: H_2\ T_2}{\emptyset \vdash H_1\ T_1 <!\ H_2\ T_2}$$

Proof. By structural induction on the derivation using narrowing (Lemma 44) in the case for $\forall$ types. □

This completes the proofs for the inversion Lemmas 40,41,42,43.

Inversion of subtyping is only required in a concrete runtime context, without abstract component ($J = \emptyset$). Therefore, transitivity pushback is only required then. Transitivity pushback requires narrowing, but only for abstract bindings (those in J, never in H). Narrowing requires these bindings to be potentially imprecise, so that the transitivity axiom can be used to inject a step to a smaller type without recursing into the derivation. In summary, we need both (actual, non-axiom) transitivity and narrowing, but not at the same time. This is a major advantage over a purely static setting where these properties can become much more entangled, with no obvious way to break cycles. Though transitivity pushback can be shown to hold in a mixed concrete/abstract runtime context for $F_{<:}$, the insight that it is only required in a concrete-only runtime context is crucial for extensions, in which subtyping can collapse in unrealizable contexts (see Section 5.4.1).

Finishing the Soundness Proof We now have everything in place to complete the soundness proof.

Theorem 8 (Soundness of $F_{<:}$)**.** For all n, if the interpreter returns a result that is not a timeout, then the result is a value (i.e. not stuck), and it is well-typed.

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \text{eval } n\ H\ e = \text{Done } r}{r = \text{Val } v \qquad H \vdash v : T}$$

Proof. By induction over n. The interesting case is the one for type application. We use the inversion lemma for type abstractions (Lemma 41) and then invert the resulting subtyping relation on $\forall$ types to relate the actual type at the call site with the expected type at the definition site of the type abstraction. In order to do this, we invoke pushback (Lemma 45) once and obtain an invertible subtyping on $\forall$ types. But inversion then gives us evidence for the return type in a mixed, concrete/abstract environment, with J containing the binding for the quantified type variable. Since the abstract component J is non-empty, we cannot apply transititivy pushback again directly. So we first apply the substitution lemma (Lemma 39) to replace the abstract variable reference with a concrete reference to the actual type in a runtime environment H. After that, the abstract component is gone ($J = \emptyset$) and we can use pushback again to further invert the inner derivations. □

The mechanized soundness proof is available from http://sound-big-step-fsub.namin.net.

5.3 Standard Type System Extensions

In this section we consider two type system extensions that are relevant for practical languages: mutable references and exceptions. While the functionality of these extensions are standard, they are thought to require different proof methods or pose other significant challenges in

big-step style. Here we show that our definitional interpreter approach scales gracefully, and that straightforward inductive proofs are sufficient.

5.3.1 Mutable References

We extend the syntax to support ML-style mutable references:

$$\begin{array}{lll} T & ::= & \cdots \mid \text{Ref } T \\ t & ::= & \cdots \mid \text{ref } t \mid !t \mid t := t \\ v & ::= & \cdots \mid \text{loc } i \end{array}$$

The extension of the syntax and static typing rules is standard, with a new syntactic category of store locations. The evaluator is threaded with a runtime store ρ, and reading or writing to a location accesses the store. How do we assign a runtime type to a store location? The key difficulty is that store bindings may be recursive, which has lead Tofte to discover coinduction as a proof technique for type soundness [Tofte, 1988]. We sidestep this issue by assigning types to values (in particular store locations) with respect to a *store typing* $\mathscr{S}$ instead of the store itself. Store typings consist of $H\ T$ pairs, which can be related through the usual runtime subtyping judgements. The value type assignment judgement now takes the form $\mathscr{S}\ H \vdash v : T$ and since subtyping depends on value type assignment, it is parameterized by the store typing as well: $\mathscr{S}\ J \vdash H_1\ T_1 <: H_2\ T_2$. The type assignment rule for store locations simply looks up the correct type from the store typing:

$$\frac{\mathscr{S}(i) = \langle H, T \rangle}{\mathscr{S}\ H \vdash \text{loc } i : \text{Ref } T}$$

When new bindings are added to the store, they are assigned the type and environment from their creation site in the store typing. When accessing the store, bindings in the store typing are always preserved, i.e. store typings are invariant under reads and updates.

Objects in the store ρ must conform to the store typing $\mathscr{S}$ at all times: $\mathscr{S} \vDash \rho$. With that, an update only has to provide a subtype of the type in the store typing, and it will not change the type of that slot. So if an update creates a cycle in the store, this does not introduce circularity in the store typing.

A canonical forms lemma states that if a value v has type Ref T, v must be a store location with type T in the store typing.

Lemma 46 (Inversion of Value Typing for Store Location)**.**

$$\frac{\mathscr{S}\ H \vdash\ v : \text{Ref } T}{\begin{array}{c} v = \text{loc } i \qquad \mathscr{S}(i) = \langle H_i, T_i \rangle \\ \mathscr{S}\ \emptyset \vdash H_i\ T_i <: H\ T\ ,\ H\ T <: H_i\ T_i \end{array}}$$

The main soundness statement is modified to guarantee that the interpreter threads a store

that corresponds to an extension of the initial store typing, regardless of whether it terminates or not. Thus, the store typing is required to grow monotonically, while the values in the store may change arbitrarily within the constraints given by the store typing. Furthermore, as before, if the interpreter terminates, then the result is a value (i.e. not stuck), and it is well-typed.

Theorem 9 (Soundness of $F_{<:}$ with Mutable References)**.**

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash \mathscr{S}\ H \qquad \mathscr{S} \vDash \rho \qquad \text{eval}\ n\ \rho\ H\ e = (\rho', r)}{\begin{array}{c}\mathscr{S}' = \mathscr{S} \ldots \qquad \mathscr{S}' \vDash \rho' \\ r = \text{Timeout} \vee (r = \text{Done Val}\ v \wedge \mathscr{S}'\ H \vdash v : T)\end{array}}$$

We believe that the ease with which we were able to add mutable references is a further point in favor of definitional interpreters. Back in 1991, the fact that different proof techniques were thought to be required for references in big-step style was a major criticism by Wright and Felleisen and a problem their syntactic approach sought to address [Wright and Felleisen, 1994]. While there is precedent for a simply-inductive big-step soundness proof of polymorphic references [Harper, 1994], this earlier proof omits explicit wrong and nonterminating cases (see followup note [Harper, 1995]) and thus suffers from the usual criticism towards big-step arguments of proving only preservation but not progress, and furthermore, of giving no guarantees for nonterminating evaluation (while, like in small step, we still ensure the invariant on store typing).

Finally, note that the store typing is just another invariant of runtime typing. We do not need to expose store typing in the static typing of terms, since locations are not surface terms.

The mechanized soundness proof is available from http://sound-big-step-mut.namin.net.

5.3.2 Exceptions

While strong soundness promises to guarantee the absence of all runtime errors in well-typed programs, realistic languages need to support a well-defined set of benign runtime failures. These commonly include division-by-zero or file-not-found conditions, which are hard to check with static type systems.

We extend the language with support for exceptions, which abort the current flow of execution and potentially resume at an enclosing exception handler:

$$\begin{array}{rcl} t & ::= & \cdots \mid \text{raise} \mid \text{try}\ t\ \text{catch}\ t \\ r & ::= & \cdots \mid \text{Done Raise} \end{array}$$

To the term syntax, we add a facility to raise an exception, as well as try/catch blocks. The set of possible interpreter results now includes the option of returning a (benign) exception, in addition to values, timeouts, and actual errors, which we still seek to prevent.

The interpreter itself requires very little changes, as it already supports abortive behavior due

to timeout and error conditions. We can modify the monadic `DO v <== eval` notation to abstract over exceptions in exactly the same way. But in addition to the bind operator `DO`, which can be viewed as targeting the innermost level of a layered monad, we also need another operator `DOE`, which extracts either `Val v` or `Raise`, conceptually one level up in the layered monad. With that, the only modification to the interpreter is adding new cases for `raise` and `catch`:

```
(* ... *)
    | traise        ⇒ DONE RAISE
    | tcatch et ec  ⇒
      DOE rt ⇐ eval n1 env et;
        match vf with
          | VAL v  ⇒ DONE VAL v
          | RAISE  ⇒ eval n1 env ec
        end
(* ... *)
```

If the 'try' part of a try/catch block raises an exception, control resumes at the 'catch' block.

The soundness statement changes as follows to include the possibility of runtime exceptions:

Theorem 10. For all n, if the interpreter returns a result that is not a timeout, then the result is either an exception or a well-typed value (but it is not stuck)

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vDash H \qquad \text{eval } n\ H\ e = \text{Done } r}{r = \text{Raise} \ \vee \ (r = \text{Val } v \wedge H \vdash v : T)}$$

In conclusion, actual errors are excluded, but benign exceptions are allowed. The proof requires no additional lemmas, just handling the two new syntactic cases in the main proof.

The key take-away is that while distinguishing between non-termination, actual errors, and benign errors is a big problem with relational big-step semantics, a total functional interpreter deals with all these outcomes uniformly and eliminates this entire class of problems by design.

The mechanized soundness proof is available from http://sound-big-step-exceptions.namin.net.

5.4 Novel Type System Extensions

In the preceding sections we have studied the ingredients of type soundess proofs with definitional interpreters, the application of this approach to known and well-studied type system such as $F_{<:}$, as well as to standard extensions. In addition to the static typing of terms on the surface, the approach involves defining runtime invariants such as typing on values. For the latter, we have some leeway: runtime typing needs to be strong enough for the inductive cases in the type safety proof, but also can be more relaxed than the surface static typing.

Up to now, we always started from given static type systems, and we tried to fit the runtime invariants to support a proof. In this section, we take the opposite route. Starting from the interpreter semantics and its runtime invariants, we derive novel and interesting static type systems.

5.4.1 System F, $F_{<:}$, and $F_{<:>}$

The fact that runtime typing can be more relaxed than the surface static typing is particularly striking for subtyping. Depending on how closely the static typing tracks the runtime typing, we can model a spectrum of polymorphic λ-calculi.

System F At one end of the spectrum, we can just not expose subtyping in the static semantics – by restricting the surface rules to System F. In this case, the runtime rules are more powerful than actually needed. They can be tightened by replacing the runtime subtyping relation with a runtime type equality relation. We leave this as an exercise to the reader.

System $F_{<:>}$ At the other end of the spectrum, we can explore what emerges from propelling more of the runtime typing to the surface. Abstract types in $F_{<:}$ can only be bounded from above. But internally, runtime subtyping already needs to support symmetric rules, for (concrete) type variables on either side. We can easily expose that facility on the static level as well, extending upper-bounded quantification to "translucent" quantification, which is both lower- and upper-bounded.

On the left, Figure 5.5 summarizes the changes in syntax and static semantics for System $F_{<:>}$, showing the progression from System F to $F_{<:>}$ as this dimension exposes more subtyping in quantification over types. In particular, we generalize the upper-bounded type variables of $F_{<:}$ from $X <: U$ to $X : S..U$, where the type S is the lower bound and the type U is the upper bound – changing the syntax of type abstraction, $\forall$ type, and binding of type variables in the static environment Γ. We also add a bottom type $\bot$ to recover the usual upper-bounded quantification of $F_{<:}$ by setting the lower bound to $\bot$.

System F

$$\begin{array}{rcl} T & ::= & \top \mid X \\ & \mid & T \to T \mid \forall Z.T^Z \\ t & ::= & x \mid \lambda x:T.t \mid \Lambda Y.t \mid t\ t \mid t\ [T] \end{array}$$

System D

$$\begin{array}{rcl} T & ::= & \top \mid x.\text{Type} \mid \{\text{Type} = T\} \mid \{\text{Type}\} \\ & \mid & (z:T) \to T^z \\ t & ::= & x \mid \{\text{Type} = T\} \mid \lambda x:T.t \mid t\ t \end{array}$$

System $F_{<:}$

$$\begin{array}{rcl} T & ::= & \top \mid X \\ & \mid & T \to T \mid \forall Z <: T.T^Z \\ t & ::= & x \mid \lambda x:T.t \mid \Lambda Y <: T.t \mid t\ t \mid t\ [T] \end{array}$$

System $D_{<:}$

$$\begin{array}{rcl} T & ::= & \top \mid x.\text{Type} \mid \{\text{Type} = T\} \mid \{\text{Type} <: T\} \\ & \mid & (z:T) \to T^z \\ t & ::= & x \mid \{\text{Type} = T\} \mid \lambda x:T.t \mid t\ t \end{array}$$

System $F_{<:>}$

$$\begin{array}{rcl} T & ::= & \bot \mid \top \mid X \mid T \to T \mid \forall Z:T..T.T^Z \\ t & ::= & x \mid \lambda x:T.t \mid \Lambda Y:T..T.t \mid t\ t \mid t\ [T] \end{array}$$

System $D_{<:>}$

$$\begin{array}{rcl} T & ::= & \bot \mid \top \mid x.\text{Type} \mid \{\text{Type}:T..T\} \mid (z:T) \to T^z \\ t & ::= & x \mid \{\text{Type} = T\} \mid \lambda x:T.t \mid t\ t \end{array}$$

System $F_{<:>}$	⟶ **System $D_{<:>}$** ⟵	**System D**
adding lower bounds to System $F_{<:}$		adding first-class type values to System F

Subtyping... $\boxed{\Gamma \vdash S <: U}$

$$\frac{\Gamma \vdash S_1 <: S_2\ ,\ U_2 <: U_1 \qquad \Gamma, Z:S_2..U_2 \vdash T_1^Z <: T_2^Z}{\Gamma \vdash (\forall Z:S_1..U_1.T_1^Z) <: (\forall Z:S_2..U_2.T_2^Z)}$$

(same as System D)

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, z:S_2 \vdash T_1^z <: T_2^z}{\Gamma \vdash (z:S_1) \to T_1^z <: (z:S_2) \to T_2^z}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{\text{Type}:S_1..U_1\}\ <:\ \{\text{Type}:S_2..U_2\}}$$

$$\Gamma \vdash \{\text{Type} = T\}\ <:\ \{\text{Type}\}$$

$$\frac{\Gamma \ni X:S..U \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash x:\{\text{Type}:\bot..U\}}{\Gamma \vdash x.\text{Type} <: U}$$

$$\frac{\Gamma \vdash x:\{\text{Type} = T\}\ ,\ T <: U}{\Gamma \vdash x.\text{Type} <: U}$$

$$\frac{\Gamma \ni X:S..U \qquad \Gamma \vdash T <: S}{\Gamma \vdash T <: X}$$

$$\frac{\Gamma \vdash x:\{\text{Type}:S..\top\}}{\Gamma \vdash S <: x.\text{Type}}$$

$$\frac{\Gamma \vdash x:\{\text{Type} = T\}\ ,\ S <: T}{\Gamma \vdash S <: x.\text{Type}}$$

Type assignment... $\boxed{\Gamma \vdash t:T}$

$$\frac{\Gamma, Y:S..U \vdash t:T^Y}{\Gamma \vdash \Lambda Y:S..U.t:\forall Z:S..U.T^Z}$$

(same as System D)

$$\frac{\Gamma, x:S \vdash t:T^x}{\Gamma \vdash \lambda x:S.t:(z:S) \to T^z}$$

$$\frac{\Gamma \vdash t_1:\forall Z:S..U.T^Z\ ,\ S <: T_2, T_2 <: U}{\Gamma \vdash t_1[T_2]:T^{T_2}}$$

(same as System D)

$$\frac{\Gamma \vdash t:(z:S) \to T^z\ ,\ t_2:S}{\Gamma \vdash t_1\ t_2:T^{t_2}}$$

$$\Gamma \vdash \{\text{Type} = T\}:\{\text{Type}:T..T\}$$

$$\Gamma \vdash \{\text{Type} = T\}:\{\text{Type} = T\}$$

Figure 5.5 – The System D Square: syntax and static semantics

User-Defined Subtyping Relations Now, a key question is what constraints should be put on lower and upper bounds at their declaration site in type abstractions. If we do not enforce "good bounds", i.e. do not require $S <: U$ for bounds $S..U$, then we can use this facility to enable fully user-defined subtyping relations between abstract types. Here is an example: in a context

$$\Gamma = A : \bot..\top,\ B : \bot..\top,\ C : \bot..\top$$

where A, B, C are fully abstract, we can add new bindings via

$$\Lambda U : A..C.\ \Lambda V : A..B.t$$

such that in the body t of this abstraction, A is a subtype of both B and C via transitivity, while B and C remain incomparable. With only upper-bounded quantification as in $F_{<:}$, we could not achieve the same flexibility.

But of course this absence of constraints on bounds also means that user-defined subtyping relations might be contradictory. Under a type abstraction with bad bounds, for example $X : \top..\bot$, we can effectively collapse the subtyping relation via transitivity chains on X and subsumption. So why does this work at all?

Soundness Interestingly, we do not need to enforce "good bounds" for soundness. The key is that we do check that the instantiated type is in between the lower bound and upper bound during a *type application*. Therefore, any code path that was type checked using bad bounds will never be executed. The proof strategy from Section 5.2 already only needs canonical forms and subtyping inversion in an empty abstract context, so bad bounds do not pose any particular difficulty. In fact, the whole proof setup for $F_{<:}$ and the runtime invariants suggest at no point that checking bounds would be necessary, so the unconstrained system comes out as a natural choice.

In runtime subtyping (not shown in Figure 5.5), we do not need to change the concrete variable rules, while the other changes (on abstract type variable and $\forall$ type) are analogous to static subtyping. For example, bindings in the runtime environment J change from $Z <: \langle H, U\rangle$ to $Z : \langle H, S..U\rangle$.

It is instructive to compare this with a small-step proof strategy. Since there is no a priori distinction between type assignment time and runtime, the unconstrained System $F_{<:>}$ would not fall out naturally as an extension of $F_{<:}$'s soundness proof, which makes key use of transitivity and narrowing lemmas that need to work in arbitrary contexts. Of course the small-step proof can be adapted to follow a transitivity pushback strategy in empty contexts, similar to Section 5.2, but such a strategy would require a radical departure from the small-step proof whereas it is self-suggesting with a definitional interpreter approach.

More generally, with the definitional interpreter approach, we can first generalize runtime typing, and then find a way to incorporate a consistent set of changes back into static typing.

The mechanized soundness proof is available from http://sound-big-step-fsubsup.namin.net.

Language Design Trade-Offs We have seen that we gain expressiveness by not checking bounds at declarations sites, but what do we lose? First, a type checker cannot decide whether user-defined subtyping relations make sense. Hence, errors might only manifest when trying (and failing) to instantiate the type abstractions. Second, while $F_{<:}$ permits arbitrary beta-reduction, $F_{<:>}$ is more restricted. For example, $F_{<:>}$ cannot support a normal-order reduction strategy which would require reductions under type lambdas, to transform a term into head normal form. With conflicting bounds, such as `String <: Int`, a term like $3 + \text{“d'oh”}$ would be well-typed but stuck.

5.4.2 System D, $D_{<:}$, and $D_{<:>}$

We now consider an extension that starts with relaxing the interpreter semantics itself. We observe that type arguments, implemented as $\langle H, T \rangle$ pairs, are already treated de-facto as first-class values at runtime. So why not let them loose and make their status explicit?

Refactoring the Interpreter In the interpreter, we do not strictly need the distinction between term and type abstraction. Closures for term abstraction `vabs` and closures for type abstraction `vtabs` below hold the same data:

```
(* ... *)
    | tabs x ey   ⇒ DONE VAL (vabs env x ey)   (* lambda      *)
    | ttabs x ey  ⇒ DONE VAL (vtabs env x ey)  (* type lambda *)
    | tapp ef T ⇒                              (* application *)
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex; match vf with
          | (vabs env2 x ey) ⇒  eval n1 ((x,vx)::env2) ey
          | _ ⇒ ERROR end
    | ttapp ef T ⇒                              (* type app   *)
      DO vf ⇐ eval n1 env ef; match vf with
          | (vtabs env2 x ey) ⇒  eval n1 ((x,vty env T)::env2) ey
          | _ ⇒ ERROR end
(* ... *)
```

Thus, we can do with only one data type, and settle for `vabs`. To also fuse term and type application, we need to make the evaluation of the argument uniform, regardless of whether it is a type or a term. We thus introduce a new syntactic form $\{\text{Type} = T\}$, for a term that holds a type (`ttyp T` below) and evaluate it to a `vty env T`, which gains status as first-class value:

```
(* ... *)
    | tapp ef T ⇒
      DO vf ⇐ eval n1 env ef;
        DO vx ⇐  eval n1 env ex; match vf with
          | (vabs env2 x ey) ⇒  eval n1 ((x,vx)::env2) ey
          | _ ⇒ ERROR end
    | ttyp T ⇒ (vty env T)
```

```
8 (* ... *)
```

First-Class Type Values and Path-Dependent Types What happens at the type level? In System $D_{<:>}$, the term $\{\text{Type} = T\}$ introduces a corresponding type $\{\text{Type} : T..T\}$. Now, since we unify term and type abstraction at the term level, we also unify function type and universal type into one dependent function type: $(x : S) \to T^x$. How can T depend on the term variable x? Instead of a universal type $\forall X : S..U.T^X$, we write $(x : \{\text{Type} : S..U\}) \to T^x$, where x is a regular term variable. For occurrences of type variable X in T, we need to be able to select the type within the term variable x – eliminating a term holding a type at the type level: x.Type. We have re-discovered path-dependent types, like in Scala and DOT but with a unique, global label Type.

On the right and center, Figure 5.5 summarizes the changes in syntax and static semantics for System D and System $D_{<:>}$, showing the progression in the System D Square along this second dimension: by adding first-class types, System D, $D_{<:}$, and $D_{<:>}$ unify the term and type abstractions of System F, $F_{<:}$, and $F_{<:>}$. Note that even System D requires a bit of subtyping (even though System F does not) to account for matching a concrete type value with an abstract type value type: $\{\text{Type} = T\} <: \{\text{Type}\}$, which we motivate next.

"D-ing F": Encoding of System F, $F_{<:}$, $F_{<:>}$ The modified interpreter semantics trivially generalizes evaluation of System F and its variants, but we still need to show how that the static typing of System D, $D_{<:}$, $D_{<:>}$ generalizes System F, $F_{<:}$, and $F_{<:>}$.

To establish this encoding, we replace references to type variables X with path-dependent types x.Type. Type abstractions become term lambdas with a type-value argument, universal types become dependent function types, and type application becomes dependent function application with a concrete type value:

$$\begin{array}{rcl} \Lambda X.t^X & \rightsquigarrow & \lambda x : \{\text{Type}\}.t^{x.\text{Type}} \\ \forall X.T^X & \rightsquigarrow & (x : \{\text{Type}\}) \to T^{x.\text{Type}} \\ t\,[T] & \rightsquigarrow & t\,\{\text{Type} = T\} \end{array}$$

But how do we actually type check a type application? Let us assume that f is the polymorphic identity function, and we apply it to type T. Then we would like the following to be an admissible type assignment:

$$\frac{f : (x : \{\text{Type}\}) \to (z : x.\text{Type}) \to x.\text{Type}}{f\,\{\text{Type} = T\} : (z : T) \to T}$$

In most dependently typed systems there is a notion of reduction or normalization on the type level. Based on our definitional interpreter construction, we observe that we can just as well use subtyping. For this application to type check using standard dependent function

Syntax

$$\begin{array}{lll} v & ::= & \langle H, \lambda x : T.t \rangle \mid \langle H, T \rangle \\ H & ::= & \emptyset \mid H, x : v \\ J & ::= & \emptyset \mid J, z : \langle H, T \rangle \end{array}$$

Runtime Subtyping... $\boxed{J \vdash H_1\ T_1 <: H_2\ T_2}$

$$\frac{J \vdash H_2\ S_2 <: H_1\ S_1 \qquad J \vdash H_1\ U_1 <: H_2\ U_2}{J \vdash H_1\ \{\text{Type} : S_1..U_1\} <:\ H_2\ \{\text{Type} : S_2..U_2\}}$$

$$\frac{J(z) = \langle H, T \rangle \qquad J \vdash H\ T <: H_2\ \{\text{Type} : \bot..U\}}{J \vdash H_1\ z.\text{Type} <: H_2\ U} \qquad \frac{H_1(x) = v \quad H \vdash v : T \qquad J \vdash H\ T <: H_2\ \{\text{Type} : \bot..U\}}{J \vdash H_1\ x.\text{Type} <: H_2\ U}$$

$$\frac{J(z) = \langle H, T \rangle \qquad J \vdash H\ T <: H_1\ \{\text{Type} : S..\top\}}{J \vdash H_1 S <: H_2\ z.\text{Type}} \qquad \frac{H_2(x) = v \quad H \vdash v : T \qquad J \vdash H\ T <: H_2\ \{\text{Type} : S..\top\}}{J \vdash H_1\ S <: H_2\ x.\text{Type}}$$

...

Value type assignment $\boxed{H \vdash v : T}$

$$\frac{\Gamma \vDash H\ \emptyset \qquad \Gamma \vdash \{\text{Type} = T\} : \{\text{Type} : S..U\}}{H \vdash \langle H, T \rangle : \{\text{Type} : S..U\}} \qquad \frac{\Gamma \vDash H\ \emptyset \qquad \Gamma, x : S \vdash t : T^x}{H \vdash \langle H, \lambda x : S.t \rangle : (z : S) \to T^z}$$

Figure 5.6 – $D_{<:>}$: runtime typing (excerpt)

types, we need to establish $T <: x.\text{Type} <: T$ and $\{\text{Type} = T\} <: \{\text{Type}\}$, using the rules on the right in Figure 5.5.

It is easy to show that System D encodes System F, but not vice versa. For example, the following function does not have a System F equivalent: $\lambda x : \{\text{Type}\}.x$

Runtime Typing For System $D_{<:>}$, we also present modified runtime typing rules in Figure 5.6. Type objects $\langle H, T \rangle$ are now first-class values, just like closures. We no longer need two kinds of bindings in H environments, which now bind variables to values. For J environments, we still consider only abstract structures, that is, only type values, now.

Delta in Meta-Theory (Type Soundness) Most of the changes to the soundness proof are rather minor. However, one piece requires further attention: the previous transitivity pushback proof relied crucially on being able to relate types across type variables:

$$H_1\ T_1 <!\ H\ Y <!\ H_3\ T_3$$

Inversion of this derivation would yield another chain

$$H \ni Y = \langle H_2, T_2 \rangle \qquad H_1\ T_1 <:\ H_2\ T_2 <:\ H_3\ T_3,$$

which, using an appropriate induction strategy, can be further collapsed into $H_1\,T_1 <!\ H_3\,T_3$. But now the situation is more complicated: inversion of $H_1\ T_1 <!\ H\ x.\text{Type} <!\ H_3\ T_3$ yields

$$H(x) = v \qquad H_2 \vdash v : T_2$$

$$H_2\ T_2 <:\ H_1\ \{\text{Type} : T_1..\top\} \qquad H_2\ T_2 <:\ H_3\ \{\text{Type} : \bot..T_3\},$$

but there is no immediate way to relate T_1 and T_3! We would first have to invert the subtyping relations with T_2, but this is not possible because these relations are imprecise and may use transitivity. Recall that they have to be, because they may need to be narrowed—but wait! Narrowing is only required for abstract types, and we only need inversion and transitivity pushback for fully concrete contexts. So, while the imprecise subtyping judgement is required for bounds initially in the presence of abstract types, we can replace it with the precise version once we move to a fully concrete context.

This idea leads to a solution involving another conversion layer. We define an auxiliary relation $T_1 <<: T_2$, which is just like $T_1 <: T_2$, but with precise lookups, e.g. for a concrete variable on the left:

$$\frac{H_1(x) = \langle H, T \rangle \qquad \emptyset \vdash H\ T <<:\ H_2\ U}{\emptyset \vdash H_1\ x.\text{Type} <<!\ H_2\ U}$$

For this relation, pushback and inversion work as before, but narrowing is not supported. To make sure we do not need narrowing inductively for subtyping with precise lookup, we delegate to usual subtyping <: with imprecise lookup and built-in transitivity in the body of the dependent function rule:

$$\frac{\begin{array}{c} J \vdash H_2\ S_2 <:\ H_1\ S_1 \\ J, z : \langle H_2, S_2 \rangle \vdash H_1\ T_1^z <:\ H_2\ T_2^z \end{array}}{J \vdash H_1\ (x : S_1) \to T_1^x\ <<!\ H_2\ (x : S_2) \to T_2^x}$$

In this new relation, we can again remove top-level uses of the transitivity axiom. A derivation $T_1 <: T_2$ can be converted into $T_1 <<: T_2$ and then further into $T_1 <<! T_2$. With that, we can again perform all the necessary inversions required for the soundness proof.

The mechanized soundness proof is available from http://sound-big-step-dsubsup.namin.net.

5.5 Scaling from F to DOT

The DOT (Dependent Object Types) calculus [Amin et al., 2012, 2014, 2016, Rompf and Amin, 2016] has been proposed as a new type-theoretic foundation for blending functional and object-oriented programming, as well as ML modules, based on path-dependent types. Historically, DOT was designed as a formal model of Scala [Amin et al., 2012], based on various rewriting semantics, but very little progress was made towards a soundness proof. The shift to big-step semantics has been instrumental in focusing the requirements, leading to the first soundness results, grounding the theory into known territory, and ultimately resulting in a much cleaner calculus [Rompf and Amin, 2016]. In this section, we sketch how the techniques of this chapter apply to DOT, showing the insights gained from looking at runtime invariants, and from exploiting abstractions becoming transparent at runtime.

From $D_{<:>}$ to DOT DOT consolidates all type and function values into *objects*, which contain type and method members with distinct labels. Object members can refer to the object itself and its other members through a recursive self reference:

$$\begin{array}{lcl} t & ::= & x \mid t.m(t) \mid \{x \Rightarrow \overline{d}\} \\ d & ::= & L = T \mid m(x:T) = t \end{array}$$

We could also add mutable fields based on the handling of references (Section 5.3.1) but we disregard this option for simplicity.

At the type level, DOT adds intersection and union types, as well as recursive self types, which are similar to equi-recursive types but quantify over a term instead of a type. Individual labeled methods and types remain separate at the type level:

$$\begin{array}{lcl} T & ::= & \bot \mid \top \mid T \wedge T \mid T \vee T \mid x.L \\ & & \{L : T..T\} \mid m(z:T) : T^z \mid \{z \Rightarrow T\} \end{array}$$

An object creation term introduces a recursive self type intersecting the member types. We present a few key examples of DOT's expressiveness next.

Objects and First-Class Modules Consider the type of an object containing a type A and a method f that returns an A:

$$\{c \Rightarrow (A : \bot..\top) \wedge (f(u:\top) : c.A)\}$$

We can create a package object or module that provides a type alias C for this type and a way to create objects of such a type:

$$\{p \Rightarrow C = \{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\};$$
$$m(t : \{A : \bot..\top\}) = \{\_ \Rightarrow m(a : t.A) = \{c \Rightarrow$$
$$A = t.A; f(u : \top) = a\}\}\}$$

Nominality through Ascription We can give the package object a type which is transparent for the type member C, using the shorthand $= T$ for $: T..T$:

$$\{p \Rightarrow (C = \{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\}) \wedge$$
$$(m(t : \{A : \bot..\top\}) : (m(a : t.A) : p.C \wedge \{A = t.A\}))\}$$

However, then, given a package object p, anyone could structurally create a type $p.C$ even by-passing the method m from the package – for example, this term would type-check as type $p.C$:

$$\{c \Rightarrow A = \top; f(u : \top) = u\}$$

We can ascribe a more abstract type to a package object. By making the lower bound of the type member C abstract, the type $p.C$ becomes nominal from the outside enabling the package to fully control the creation of objects of types $p.C$:

$$\{p \Rightarrow (C : \bot..\{c \Rightarrow (A : \bot..\top) \wedge (f(u : \top) : c.A)\}) \wedge$$
$$(m(t : \{A : \bot..\top\}) : (m(a : t.A) : p.C \wedge \{A = t.A\}))\}$$

Records and Refinements as Intersections The package could also provide a refinement of type $p.C$, for example one with additional methods. By closing over a recursive type, one can concisely refer to type members from the refined type.

$$\{c \Rightarrow p.C \wedge (g(a : c.A) : p.C)\}$$

5.5.1 Historical Challenges and Fresh Perspective

Extending the $D_{<:>}$ soundness proof to DOT poses only few challenges. The main source of complication are recursive self types, which create mutual dependencies between various previously independent considerations, e.g., between the additional layer with precise lookup (see meta-theory in Section 5.4.2) and abstract to concrete substitution for subtyping (see Lemma 39). These complications are resolved with clever induction metrics and contractiveness restrictions as presented elsewhere [Rompf and Amin, 2016] in Chapter 4. Since the details do not yield further insights into proof techniques with definitional interpreters, we do

not repeat them here.

Instead, we present the historical challenges in proving 'invented' versions of DOT sound and discuss their 'discovered' solutions thanks to the bottom-up exploration with definitional interpreters.

The mechanized soundness proof is available from http://sound-big-step-dot.namin.net.

Substitution – translating back to small-step For path-dependent types, substitution should preserve syntactic validity of paths. In our approach, we do not need substitution of terms, and we only need substitution of a concrete variable for an abstract one in types, so this syntactic preservation is straightforward. What happens when we translate the proof back to small-step? For call-by-value, we need substitution in terms of a value for a variable. So we can either allow values in paths, i.e. relax the syntax from $x.L$ to $(x \mid v).L$, or we can put all values in a store, so that variables are only substituted with other variables, which are bound in the store. In this case, subtyping takes the form $\mathscr{S}\ \Gamma \vdash T_1 <: T_2$ to track the store $\mathscr{S}$. Both options lead to sound and clean syntactic theories [Rompf and Amin, 2016] (see Chapters 3 and 4), which look blindingly obvious in hindsight, but have not been discovered directly despite focused efforts of several person-years.

Abstraction vs. Preservation – exploiting the runtime Branding is an identity function which changes the type, usually from something concrete, e.g. $\top$, to something abstract, e.g. $z.L$. This is valid e.g. if $z : \{L : \top..\top\}$. However, we might only know a less precise type $z : \{L : \bot..\top\}$. In that case, even though we can brand a term $x : \top$ as $x : z.L$ via $z.\mathrm{brand}(x) \longrightarrow x$, we cannot establish $x : z.L$ directly given only the more abstract information. In our approach, thanks to the distinction between static and runtime typing, we can effectively access the most precise information at run-time, thus keeping evaluation and preservation in sync instead of in tension. In small-step, this strategy translates to also bringing down abstraction barriers selectively, i.e. keeping more precise information about values or store locations.

Transitivity vs. Narrowing – pushback in empty abstract context In previous DOT developments, narrowing and subtyping transitivity were a major headache. In fact, a key step was to show that in the presence of intersection types, both statements cannot hold at the same time [Amin et al., 2014] in full generality. With the approach of extending $\mathrm{F}_{<:}$ towards DOT in this chapter, we discovered that we only need to pushback or invert subtyping transitivity in an empty abstract context, and so we can tolerate lattice collapses in non-empty abstract contexts. We have already taken a lenient strategy with respect to "bad bounds" in Section 5.4.1, and therefore adding intersection and union types poses no particular difficulty. However, the distinction between concrete and abstract context is unusual and essential to the strategy here.

Monotonicity Matters – embrace subsumption Finally, in the historical invented as opposed to discovered model of DOT [Amin et al., 2012], the type system had many more judgements: not just typing and subtyping, but also expansion, membership and well-formedness. In the discovered model, well-formedness is so lenient that a judgement is not needed, a type merely needs to follow the syntax and have well-bound variables. For membership, the discovered model relies on subtyping, comparing to the member type. Expansion was used for membership, but also for collecting all member requirements when creating an object (while this remains structural in the discovered model). Because some uses of expansion needed to be precise (e.g. for object creation or for ensuring good bounds), subsumption had to be controlled. However, this was untenable at the end, because such a strategy broke several monotonicity properties. By having a lenient strategy towards bad bounds and embracing subsumption, the design not only finally proved sound but became more principled, powerful, and less tied to Scala.

6 Related Work

6.1 Semantics and Proof Techniques

The textbook Pierce [2002] (TAPL) uses a small-step structural operational semantics, introduced by Plotkin [2004]. As pointed out by Wright and Felleisen [1994], such a semantics provide a recipe for syntactic soundness in terms of term rewriting. In Chapters 3 and 4, our presentation follows the standard recipe.

Historically, we did not manage to find the models and proofs of Chapters 3 and 4 by following the recipe, because we encountered many difficulties – in particular, relating abstract and concrete names across preservation steps. In Chapter 5, we use a big-step approach, as popularized by Siek [2013]. This big-step approach goes back to the natural semantics of Kahn [1987], giving a direct evaluation relation from terms and values. As pointed out by Wright and Felleisen [1994], big-step approaches have the drawback that wrong states and non-termination are not easily distinguished. However, Leroy and Grall [2009] argue for benefits of the big-step approach, in particular in the context of validating the semantics of a compiler, and they show how to augment the big-step semantics with a co-inductive definition to distinguish between wrong states and non-termination. The work of Danielsson [2012] builds on the work of Leroy and Grall [2009], and shows how to encapsulate the non-termination in a partiality monad, so that the semantics can be expressed intuitively and concisely, without repetition among the inductive and co-inductive definitions.

In our big-step work, we use simple inductive step counters, circumventing the need for co-induction. The use of step counters in natural semantics to distinguish between divergence and errors goes back to at least the partial proof semantics of Gunter and Rémy [1993] and has recently been advocated in the context of compiler verification [Owens et al., 2016]. We believe that ours is the first purely inductive big-step strong soundness proof in the presence of mutable state and polymorphism. As discussed earlier (Section 5.3.1), Harper [1994] proves preservation of polymorphic references in big-step using only induction, though not strong soundness as we do. The big-step soundness proof of core ML by Tofte [1988], uses co-induction to deal with the mutable state and their relation to types. In our setting, we get

away with purely inductive proofs, thanks to numeric step indexes or depth bounds, even for mutable references. We find that avoiding co-induction has also helped us keep the proofs conceptually simple.

The survey of interpreter implementations of Midtgaard et al. [2013] discusses further potential drawbacks of rewriting techniques. Big-step evaluators can be mechanically transformed into equivalent small-step semantics following the techniques of Danvy and Johannsen [2010], Danvy et al. [2012], Ager et al. [2003].

Logical relations[Appel and McAllester, 2001, Ahmed, 2004, 2006] are a semantic approach to proving soundness and other properties, for example normalization. Intuitively, related inputs go to related output, and the ingenuity comes in finding a logical relation that captures the property of interest. Care must be taken to ensure well-foundedness of a logical relation.

DOT is also a theory of names, in the sense that names, even for types, are bound by a scope. Our runtime environment construction bears some resemblance to name graphs [Neron et al., 2015] and also to bindings as sets of scopes [Flatt, 2016], which are also theories of names.

6.2 Calculi Related to Dependent Object Types (DOT)

Scala Foundations Much work has been done on grounding Scala's type system in theory. Early efforts included νObj [Odersky et al., 2003], Featherweight Scala [Cremet et al., 2006] and Scalina [Moors et al., 2008], all of them more complex than what is studied here. None of them lead to a mechanized soundness result, and due to their inherent complexity, not much insight was gained why soundness was so hard to prove.

The νObj calculus relies on features beyond the full Scala language, for example classes as first-class values, and contains a comparatively large type language, including distinct notions of singleton types, type selections, record types, class types and compound types, which make νObj rather unwieldy in practice and not very suitable to extensions. In particular, subtyping does not provide unique upper or lower bounds, and mixin composition is not commutative. Type checking in νObj was shown to be undecidable through an encoding of the $F_{<:}$ system. The νObj paper [Odersky et al., 2003] claims a type soundness result, but there is no machine-verified proof. Featherweight Scala was an attempt at a calculus with decidable type checking, but soundness was explicitly left as future work [Cremet et al., 2006]. Scalina [Moors et al., 2008] was proposed as a formal underpinning for introducing higher-kinded types in Scala. Among other constructs, it contains concepts such as un-members, un-types and un-kinds to model contravariant refinements. Soundness of Scalina has not been established.

ML Module Systems 1ML [Rossberg, 2015] unifies the ML module and core languages through an elaboration to System F_ω based on earlier such work [Rossberg et al., 2014]. Compared to DOT, the formalism treats recursive modules in a less general way and it only

models fully abstract vs. fully concrete types, not bounded abstract types. Although an implementation is provided, there is no mechanized proof. In good ML tradition, 1ML supports Hindler-Milner style type inference, with only small restrictions. Path-dependent types in ML modules go back at least to SML [Macqueen, 1986], with foundational work on translucent signatures by Harper and Lillibridge [1994] and Leroy [1994]. MixML [Dreyer and Rossberg, 2008] drops the stratification requirement and enables modules as first-class values. The work on coercion constraints by Cretin and Rémy [2014], Scherer and Rémy [2015] resembles user-defined subtyping theories in DOT, where abstract types can be both lower- and upper-bounded, even leading to inconsistencies. In DOT, we solve subtyping collapse by adopting a weak-reduction strategy and ensuring type safety only in run-time typing contexts. Cretin, Scherer and Rémy consider two flavors of constraints: consistent and inconsistent and only forbid reduction in the later. Their approach is thus more flexible at the expense of a more complex type system.

Other Related Languages Other calculi related to path-dependent types include the family polymorphism of Ernst [2001], virtual classes [Ernst et al., 2006, Ernst, 2003, Nystrom et al., 2004, Gasiunas et al., 2007], and ownership type systems like Tribe [Clarke et al., 2007, Cameron et al., 2010]. Nomality by ascription is also achieved in Grace [Jones et al., 2015]. Like System D, pure type systems [Barendregt, 1992] unify term and type abstraction. Extensions of System $F_{<:}$ related to DOT include intersection types and bounded polymorphism [Pierce, 1991] and higher-order subtyping [Steffen, 1997, Abel, 2008]. Subtyping has also been combined with dependent types albeit without polymorphism [Aspinall and Compagnoni, 2001], motivated by applications in the context of logical frameworks.

Virtual classes abstract not only over the type, but also over classes and inheritance: in a class extension, the actual superclass used at run-time can be a subclass of the class that appears statically in the extends clause. This feature is powerful but also hard to control, because the unknown superclass might introduce bindings which conflict with those in the subclass. Consequently, type systems for virtual classes either need additional restrictions or more type machinery to control these interactions. For example, the *vc* calculus [Ernst et al., 2006] models virtual classes with path-dependent types but restricts paths to start with “this”, though it provides a way (“out”) to refer to the enclosing object.

7 Conclusions

The key aim behind DOT is to build a solid foundation for Scala and similar languages from first principles.

We have presented soundness results for DOT, including recursive type refinement and a subtyping lattice with full intersection types – by exploiting a semantic model that exposes a distinction between static terms and runtime values.

We also think that it is important to convey not just that a calculus is sound in isolation, but also what assumptions the soundness proof relies on in order to evaluate the broader applicability of the work. In particular, our proof relies crucially on runtime values having only type members with good bounds, which the syntax enforces. Because of recursive types, such a property would be difficult to enforce semantically. It also relies on call-by-value semantics, in that it expects all variables that can partake in types to point to runtime values when a method body is evaluated.

Finally, in our own experience with DOT, the process of designing the calculus and proving it sound have been intertwined. As we understood the landscape better, we have been able to make the model more uniform yet powerful.

We have presented type soundness proofs with definitional interpreters, reviewing a proof strategy on STLC, scaling it to System $F_{<:}$, and leading all the way to DOT via the System D Square. For this approach, one defines static typing on terms and runtime typing on values, ensuring that static typing approximates runtime typing. For existing systems, the approach thus focuses the creative search for a soundness proof in devising an appropriate runtime typing on values. For novel systems or extensions, the approach suggests first working out the runtime typing and gradually exposing it into the static typing. We have seen how DOT emerges from System $F_{<:}$ through relatively gentle extensions in the System D Square. This naturally raises the question what other interesting type systems can arise by devising suitable static rules from the runtime ones given by the interpreter and environment structures of interesting languages. We believe this is an exciting new research angle.

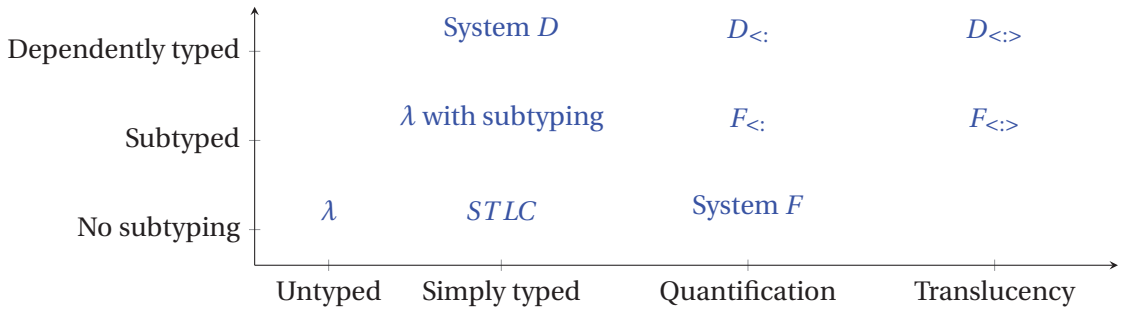

Figure 7.1 – exploring the landscape

Finally, let's try to sum up the dimensions. Figure 7.1 shows some of them [1]. At the beginning, there was the lambda-calculus. Then we can extend along several dimensions. We can add more subtyping. We can add more types like in System $F_{<:}$. So we have lambda-calculus is to lambda-calculus with subtyping as System F is to System $F_{<:}$. We can add more mirror symmetry – that is extending a dimension over another. Even if the expressivity turns out to be equivalent, we get a new way of expressing concepts with different trade-offs. For example, from upper-bounded $F_{<:}$ to upper- and lower-bounded $F_{<:>}$. The expressivity landscape varies in unexpected way. So we get System $F_{<:>}$. Finally, the System D Square explores an alternative universe with only quantification over terms – not types – to paraphrase Martin Odersky.

And dimensions do not stop. There is Church vs. Curry type annotations and inference. There's syntax restrictions such as A-Normal Form. And mutation. A leitmotiv is to cut across as much as possible by mechanizing the easiest way and then abstracting away.

As Yuri Manin said, a good proof is one that makes us wiser. In this spirit, we conclude with a retrospective on proving soundness.

- Static semantics should be monotonic. All attempts to prevent bad bounds broke monotonicity.
- Embrace subsumption, don't require precise calculations in arbitrary contexts.
- Create recursive objects concretely, enforcing good bounds and shape syntactically not semantically. Then abstract, if desired.
- Inversion lemmas need only hold for realizable environments.
- To resolve the tension between preservation and abstraction, rely on a precise static environment that corresponds to the runtime.

[1] Courtesty of André Videla.

Bibliography

Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18:797–822, 10 2008.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, 2003.

Amal J. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, 2004.

Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *POPL*, 2017.

Nada Amin and Ross Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA* , 2016.

Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *FOOL*, 2012.

Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *OOPSLA*, 2014.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, 2016.

Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1):273–309, 2001.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics*, TPHOLs, 2005.

Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaeta. Basic theory of f-bounded quantification. *Information and Computation*, 153(2):173–237, 1999.

H. P. Barendregt. Handbook of logic in computer science. chapter Lambda Calculi with Types. Oxford University Press, 1992.

Nicholas R. Cameron, James Noble, and Tobias Wrigstad. Tribal ownership. In *OOPSLA*, 2010.

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1/2), 1994.

Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3), 2012.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *Aspect-Oriented Software Development*, AOSD, 2007.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Mathematical Foundations of Computer Science*, MFCS, 2006.

Julien Cretin and Didier Rémy. System F with coercion constraints. In *CSL-LICS*, 2014.

Nils Anders Danielsson. Operational semantics using the partiality monad. In *ICFP*, 2012.

Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *JCSS*, 76(5), 2010.

Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *TCS*, 435, 2012.

Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *ICFP*, 2008.

Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.

Erik Ernst. Family polymorphism. In *ECOOP*, 2001.

Erik Ernst. Higher-order hierarchies. In *ECOOP*, 2003.

Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL*, 2006.

Matthew Flatt. Binding as sets of scopes. In *POPL*, 2016.

Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA*, 2007.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, 1972.

C. A. Gunter and D. Rémy. A proof-theoretic assesment of runtime type errors. Technical Report Technical Memo 11261-921230-43TM, AT&T Bell Laboratories, 1993.

Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201 – 206, 1994.

Robert Harper. A simplified account of polymorphic references – followup. https://www.cs.cmu.edu/~rwh/papers/refs/ipl-followup.pdf, 1995.

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL*, 1994.

Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1), 2002.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *TOPLAS*, 23(3), 2001.

Timothy Jones, Michael Homer, and James Noble. Brand objects for nominal typing. In *ECOOP*, 2015.

Gilles Kahn. Natural semantics. In *Symposium on Theoretical Aspects of Computer Science*, STACS, 1987.

Xavier Leroy. Manifest types, modules and separate compilation. In *POPL*, 1994.

Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2), 2009.

David Macqueen. Using dependent types to express modular structure. In *POPL*, 1986.

Jan Midtgaard, Norman Ramsey, and Bradford Larsen. Engineering definitional interpreters. In *PPDP*, 2013.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

Adriaan Moors, Frank Piessens, and Martin Odersky. Safe type-level abstraction in Scala. In *FOOL*, 2008.

Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *ESOP*, 2015.

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, 2004.

Martin Odersky. The trouble with types. Presentation at Strange Loop, 2013. http://www.infoq.com/presentations/data-types-issues.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL*, 1996.

Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with Scala. *CACM*, 57(4), 2014.

Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *POPL*, 2001.

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0521663504.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP*, 2016.

Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991.

Benjamin C Pierce. Bounded quantification with bottom. Technical Report 492, Indiana University, 1997.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

Benjamin C. Pierce and David N. Turner. Local type inference. *TOPLAS*, 22(1), 2000.

Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61, 2004.

John C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, 1974.

John C. Reynolds. Definitional interpreters for higher-order programming languages. *HOSC*, 11(4), 1998.

Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University, 2015.
http://arxiv.org/abs/1510.05216.

Tiark Rompf and Nada Amin. Type soundness for dependent object types. In *OOPSLA* , 2016.

Andreas Rossberg. 1ML - core and modules united (f-ing first-class modules). In *ICFP*, 2015.

Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *JFP*, 24(5), 2014.

Gabriel Scherer and Didier Rémy. Full reduction in the face of absurdity. In *ESOP*, 2015.

Dana S. Scott. Domains for denotational semantics. In *Automata, Languages and Programming*, 1982.

Jeremy Siek. Type safety in three easy lemmas, May 2013.
http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html.

Martin Steffen. *Polarized higher-order subtyping*. PhD thesis, University of Erlangen-Nuremberg, 1997.

Alexander J Summers. Modelling java requires state. In *Formal Techniques for Java-like Programs*, FTfJP, 2009.

Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, 1988.

David von Oheimb. Re: Subject reduction fails in java. http://www.seas.upenn.edu/~sweirich/types/archive/1997-98/msg00452.html, 1998.

Philip Wadler. The essence of functional programming. In *POPL*, 1992.

Philip Wadler. Propositions as types. Presentation at Strange Loop, 2015.

Goeffrey Alan Washburn. SI-1557: Another type soundness hole. https://issues.scala-lang.org/browse/SI-1557, 2008.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.

avenue Trembley 3
CH-1209 Genève
+41 79 943 37 72
☎ +41 22 733 08 05
✉ namin@alum.mit.edu
namin.net

Nada AMIN

Research interests

Meta-Theory of Programming Languages, Generative Programming, Verification.

Education

2011–present **Ph.D., Computer Science**, *EPFL*, Switzerland.
Advisor: Martin Odersky. Thesis: "Dependent Object Types."

2001-2008 **B.S. & M.Eng., Computer Science, Music Minor.**, *MIT*, USA.
Advisor: Saman Amarasinghe. Thesis: "Computer-Aided Design for Multilayer Microfluidic Chips."

Spring 2007 **Visiting Student, Mathematics**, *EPFL*, Switzerland.

Spring 2004 **Visiting Student, Computer Science**, *EPFL*, Switzerland.

Fall 2003 **Visiting Student, Computer Science**, *Ecole Polytechnique*, France.

Publications

POPL'17 **Nada Amin** and Tiark Rompf. LMS-Verify: Abstraction without Regret for Verified Systems Programming, *Principles of Programming Languages.*

POPL'17 **Nada Amin** and Tiark Rompf. Type Soundness Proofs with Definitional Interpreters, *Principles of Programming Languages*.

OOPSLA'16 Tiark Rompf and **Nada Amin**. Type Soundness for Dependent Object Types. *Object-Oriented Programming Systems, Languages and Applications*.

OOPSLA'16 **Nada Amin** and Ross Tate. Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. *Object-Oriented Programming Systems, Languages and Applications*.

Wadlerfest'16 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf and Sandro Stucki. The Essence of Dependent Object Types. *A List of Successes That Can Change the World*: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday.

ICFP '15 Tiark Rompf and **Nada Amin**. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. *International Conference on Functional Programming*.

OOPSLA '14 **Nada Amin**, Tiark Rompf and Martin Odersky. Foundations of Path-Dependent Types. *Object-Oriented Programming Systems, Languages and Applications*.

TAP '14 Nada Amin, Rustan Leino and Tiark Rompf. Computing with an SMT solver. *Tests & Proofs*.

HOSC '13 Tiark Rompf et al. Scala-Virtualized: Linguistic reuse for deep embeddings. *Higher Order and Symbolic Computation*.

Scala '13 Sandro Stucki, **Nada Amin**, Manohar Jonnalagedda and Tiark Rompf. What are the Odds?: probabilistic programming in Scala.

FTfJP '13 Lukas Rytz, **Nada Amin** and Martin Odersky. A Flow-Insensitive, Modular Effect System for Purity. *Formal Techniques for Java-like Programs*.

POPL '13 Tiark Rompf et al. Optimizing Data Structures in High-Level Programs. *Principles of Programming Languages*.

FOOL '12 **Nada Amin**, Adriaan Moors and Martin Odersky. Dependent Object Types. *Foundations of Object-Oriented Languages*.

ECOOP '12 Grzegorz Kossakowski, **Nada Amin**, Tiark Rompf and Martin Odersky. JavaScript as an Embedded DSL. *European Conference on Object-Oriented Programming*.

111

ICCD '09 **Nada Amin**, William Thies and Saman Amarasinghe. Computer-Aided Design for Microfluidic Chips Based on Multilayer Soft Lithography. *International Conference on Computer Design*.

Genome Res. '03 Paul Shannon et al. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research*.

Nature Bio. '03 Owen Ozier, **Nada Amin** and Trey Ideker. Global architecture of genetic interactions on protein network. *Nature Biotechnology*.

Talks and Meetings

- LMS: Generative Programming in Scala, EPFL - Novi Sad *SCOPES* meeting, Lausanne, Switzerland (June 2016).
- The DOT Calculus, ECOOP PC Workshop, Providence, RI, USA (February 2016).
- IFIP *WG2.16* (Working Group on Language Design), Los Angeles, CA, USA (January 2016).
- LMS: a Perspective on Generative Programming, Invited talk at *PEPM*, St Petersburg, FL, USA (January 2016).
- Programming should eat itself, *Strange Loop* keynote, St Louis, MO, USA (September 2014).
- Batteries Included: Generative Programming with Scala and LMS (with Tiark Rompf), *CUFP* tutorial, Gothenburg, Sweden (September 2014).
- Implicits in Practice (with Tiark Rompf), *ML Family Workshop*, Gothenburg, Sweden (September 2014).
- The DOT Calculus, *Scala Days*, Berlin, Germany (June 2014).
- Patterns for Generative Programming, *EcoCloud* Annual Event, Lausanne, Switzerland, (June 2014)
- *NII Shonan Meeting Seminar on Staging and High-Performance Computing*, Japan (May 2014).
- The DOT Calculus, *flatMap*, Oslo, Norway (May 2014).
- Mind the Gap, *Off the Beaten Track (OBT)*, San Diego, CA, USA (January 2014).
- Meta-Programming in Logic Programming (with William Byrd), *codemesh.io*, London, UK (December 2013).
- From Greek to Clojure! (with William Byrd), *Clojure/conj*, Alexandria, VA, USA (November 2013).
- Program Transformations (with William Byrd), *Hacker School (Recurse Center)*, New York, NY, USA (July 2013).
- Staging: Runtime code generation for "abstraction without regret", *Hacker School (Recurse Center)*, New York, NY, USA (July 2013).
- Program Transformations (with William Byrd), *Lambda Jam* tutorial, Chicago, IL, USA (July 2013).
- How to write your next POPL paper in Dafny, *Microsoft Research*, Redmond, WA, USA (July 2013).
- Lightweight Modular Staging (with Tiark Rompf et al.), *PLDI* tutorial, Seattle, WA, USA (June 2013).
- core.logic.nominal, *miniKanren Confo co-located with Clojure/West*, Portland, OR, USA (March 2013).
- Dependent Object Types, *FOOL*, Tucson, AZ, USA (October 2012).
- *Oregon Programming Languages Summer School (OPLSS)*, Eugene, OR, USA (July 2012).
- JavaScript as an Embedded DSL, *ECOOP*, Beijing, China (June 2012).
- JavaScript as an Embedded DSL, *Scala Days*, London, UK (April 2012).

Professional Service

Program Committee Member

- *Scala Symposium* 2016.
- *Higher-order, typed, inferred, strict: ML Family Workshop (ML)* 2016.
- *Symposium on Trends in Functional Programming (TFP)* 2016.

- *European Conference on Object-Oriented Programming (ECOOP)* 2016.
- *Off the Beaten Track (OBT)* 2016.
- *Workshop on Generic Programming (WGP)* 2014.
- *Scheme and Functional Programming Workshop (Scheme)* 2013.

Reviewer

- *Principles of Programming Languages (POPL) – External Review Committee (ERC)* 2017.
- *Science of Computer Programming (SCP)* 2015.
- *International Conference on Functional Programming (ICFP)* 2014, 2015.
- *International Conference on Generative Programming (GPCE)* 2013, 2014.
- *Symposium on Principles and Practice of Declarative Programming (PPDP)* 2014.
- *Symposium on Trends in Functional Programming (TFP)* 2013.

Positions Held

Research

2011–present **Research Assistant, Programming Methods Laboratory**, *EPFL*, Lausanne, VD, CH.
Research in Programming Languages with Prof. Martin Odersky.

2013 (June 24–July 5) **Visiting Research Scholar, RiSE**, *Microsoft Research*, Redmond, WA, USA.
Invited by Dr. K. Rustan M. Leino.

2006–2008 **Research Assistant, Computer Architecture Group**, *MIT CSAIL*, Cambridge, MA, USA.
Research in Design Automation for Programmable Microfluidic Chips with Prof. Saman Amarasinghe.

Summer 2003 & Jan. 2004 **Computational Molecular Biology**, *Max-Planck Institute for Molecular Genetics*, Berlin, Germany.
Research in Computational Biology with Prof. Martin Vingron.

Summer 2002 & Fall 2002 **Supercomputing Technologies Group**, *MIT CSAIL*, Cambridge, MA, USA.
Research in Computer Systems with Prof. Charles Leiserson.

Fall 2001 & Spring 2002 **Ideker Lab**, *Whitehead Institute for Biomedical Research*, Cambridge, MA, USA.
Research in Computational Biology with Dr. Trey Ideker.

Industry

2009–2011 **Software Engineer**, *Google*, Zürich, Switzerland.
Part of the Gmail and Closure Compiler teams.

Summer 2008 **Intern Software Engineer**, *Google*, Zürich, Switzerland.
Improved type inference and checking in Closure Compiler.

Summer 2006 **Intern Software Development Engineer**, *Microsoft*, Redmond, WA, USA.
Part of the Visual Studio Data team.

Summer 2004 **Intern in IT Architecture**, *Lombard Odier Darier Hentsch*, Geneva, Switzerland.
Developed a prototype to authenticate network devices with disconnected smartcard readers.

Summer 2000 **Intern**, *ArsDigita Foundation*, Cambridge, MA, USA.
Built and maintained database-backed websites.

Teaching Experience

2012–2014 **Teaching Assistant, Functional Programming Principles in Scala**, *EPFL & Coursera*.

Fall 2015 **Teaching Assistant, Programmation I**, *EPFL*.

Fall 2013 **Teaching Assistant, Principles of Reactive Programming**, *EPFL & Coursera*.

2013 **Resident**, *Hacker School (Recurse Center)*, New York, NY, USA.

Spring 2013 **Teaching Assistant, Informatique II**, *EPFL*.

Spring 2007 **Teaching Assistant, Pattern Classification and Machine Learning**, *EPFL*.

2004–2006 **Lab Assistant & Tutor, Structure and Interpretation of Computer Programs**, *MIT*.

2005 **Tutor, Introduction to Algorithms**, *MIT*.

2000 **Lab Assistant, Database-backed Websites**, *ArsDigita Boot Camp*, Cambridge, MA, USA.

Research Mentoring

Fall 2013 Samuel Grütter, *"Explorations of Type Systems"*.
Spring 2014 Daniel Espino, *"Embedding Logical Frameworks in Scala"*.
Spring 2014 Samuel Grütter, *"Machine-checked typesafety proofs"*.
Spring 2014 Valérian Pittet, *"Scala Music Generation"*.
Spring 2015 Fengyun Liu, *"Dependency Resolution Through SAT Solvers"*.
Fall 2015 Fengyun Liu (co-supervised with Sandro Stucki), *"Type-and-Effect Systems based on Capabilities"*.
Spring 2016 Samuel Grütter, *"Connecting Scala to DOT"*.

Selected Awards

2015 EPFL Teaching Assistant Team Award.
2011 EPFL I&C School Fellowship.
2010 Google Peer Bonus for enabling usage of Gmail's CSS compiler for Android projects.
2008 Google Peer Bonus for "benefiting numerous projects including all of Google Apps".
2005 MIT Letter of Commendation awarded by Prof. Michael Ernst for outstanding performance in the "Laboratory in Software Engineering" class.
1999 ArsDigita Prize finalist for Metis Service.