Decidable Subtyping for Path Dependent Types

A/Prof Alex Potanin

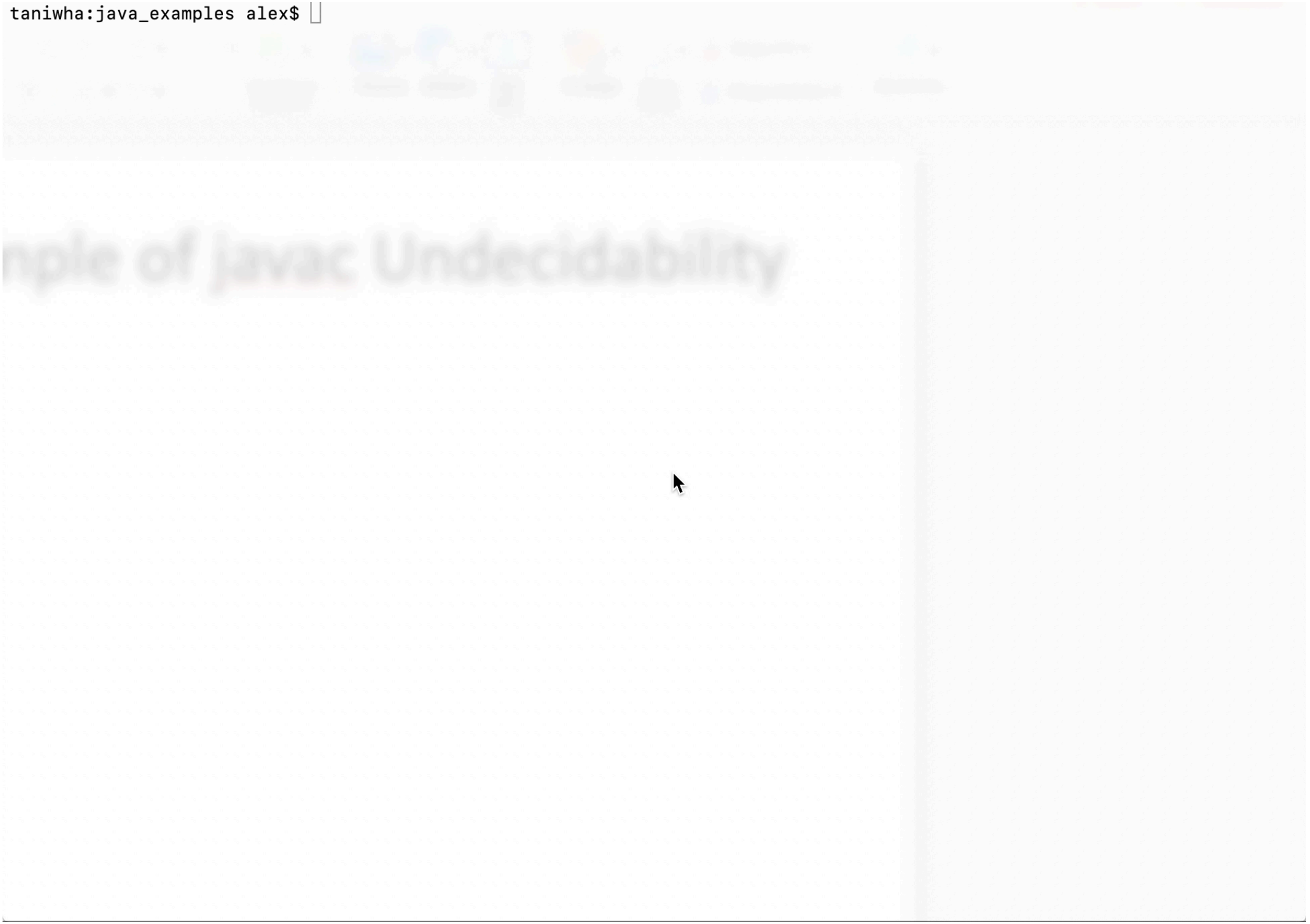
Decidability of Subtyping

```
interface Equatable<T extends Object> { }

interface List<T extends Object>
  extends Equatable <List <? extends Equatable <? super T>>> { }

class ArrayList <T extends Object> implements List<T> { }
class Tree extends ArrayList <Tree> { }

public class Function {
    public void func(Equatable<? super Tree> e) { }
    public static void main(String[] aaaaargh) {
        Tree t = new Tree(); Function f = new Function();
        f.func(t); // what do you think would happen here?
    }
}
```


taniwha:java_examples alex$

Decidability of Subtyping

$$
\begin{array}{c}
\texttt{Tree} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{ArrayList<Tree>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{List<Tree>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{Equatable<List<Equatable<Tree>>>} <: \texttt{Equatable<Tree>} \\
\Downarrow (\textit{contravariance}) \\
\texttt{Tree} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{ArrayList<Tree>} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{inheritance}) \\
\texttt{List<Tree>} <: \texttt{List<Equatable<Tree>>} \\
\Downarrow (\textit{covariance}) \\
\texttt{Tree} <: \texttt{Equatable<Tree>} \\
\vdots
\end{array}
$$

Shapes and Materials

Getting F-Bounded Polymorphism into Shape

Ben Greenman
Cornell University
blg59@cornell.edu

Fabian Muehlboeck
Cornell University
fabianm@cs.cornell.edu

Ross Tate
Cornell University
ross@cs.cornell.edu

Abstract

We present a way to restrict recursive inheritance without sacrificing the benefits of F-bounded polymorphism. In particular, we distinguish two new concepts, *materials* and *shapes*, and demonstrate through a survey of 13.5 million lines of open-source generic-Java code that these two concepts never actually overlap in practice. With this *Material-Shape Separation*, we prove that even naïve type-checking algorithms are sound and complete, some of which address problems that were unsolvable even under the existing proposals for restricting inheritance. We illustrate how the simplicity of our design reflects the design intuitions employed by programmers and potentially enables new features coming into demand for upcoming programming languages.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Inheritance, Polymorphism

General Terms Algorithms, Design, Languages

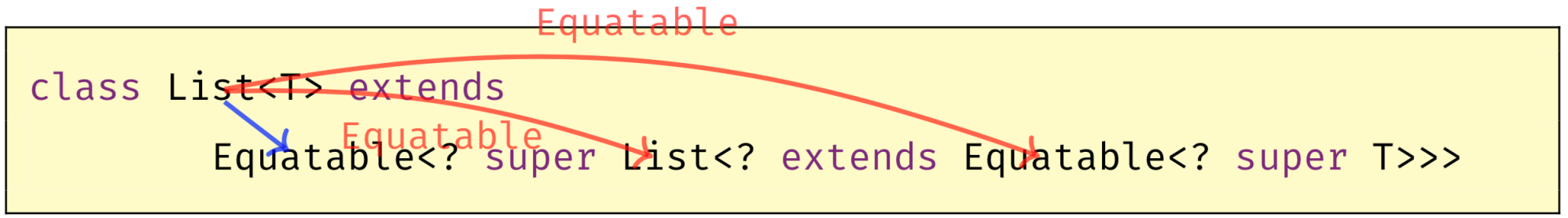
Keywords Materials, Shapes, Separation, Subtyping, F-Bounded Polymorphism, Variance, Decidability, Joins, Higher-Kinded Types

reshaping F-bounded polymorphism [4] to properly match how it is used in practice.

Plain bounded polymorphism is the ability to specify the range of types a type variable can represent. Typically this is done with an upper bound, i.e. a constraint indicating what classes/interfaces instantiations of a type variable must implement. This allows the programmer to guarantee the presence of various methods, such as requiring a type variable to extend `Formattable` so that the programmer can safely use the `format` method. Thus bounded polymorphism enables programmers to impose the same requirements and guarantees on type arguments that they can impose on function parameters and returns.

F-bounded polymorphism is the ability to constrain a type variable by a type expressed in terms of the type variable itself [4]. In other words, F-bounded polymorphism is the ability to use recursive constraints. This subtle addition significantly increases the power of type-variable constraints. In particular, F-bounded polymorphism addresses the issue of *binary methods*, the pattern that operations such as comparison and addition need both arguments to have the same type. With F-bounded polymorphism, one can require a type parameter `T` to extend `Comparable<T>`, where `Comparable<T>` has a comparison method that only accepts arguments of type `T`. This way types such as `Integer` and `String`

Shapes and Materials



- *Shapes*: classes that facilitate cycles in the inheritance hierarchy.
- *Materials*: types of concrete objects and may be freely used as normal.
- Shapes are restricted from use in class type parameters.
- Java corpus study confirmed that this aligns with what programmers already do intuitively.

Path Dependent Types

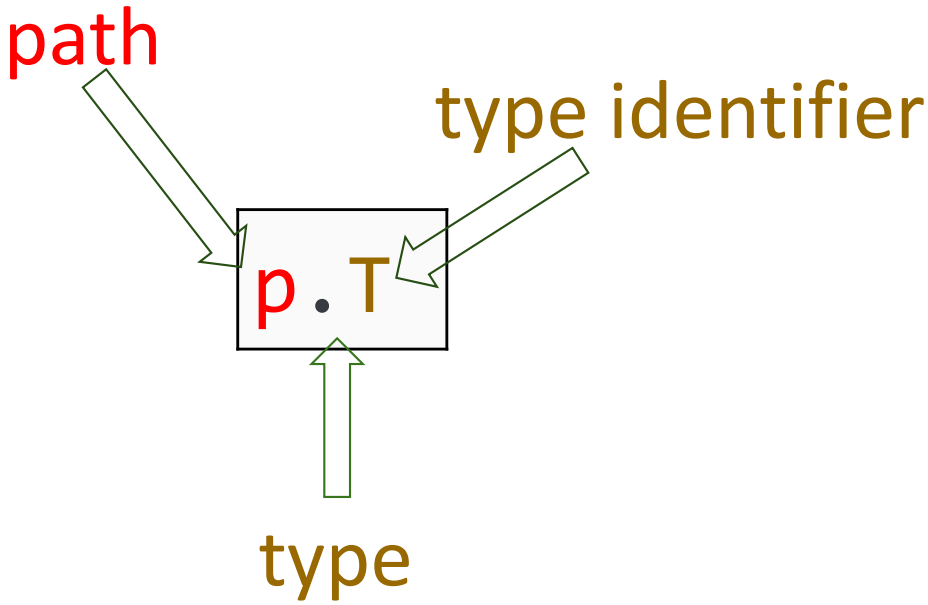
```
1 class Object{}
2 class Cell<E extends Object>{
3     E member;
4 }
5 Cell<int> aCell = new Cell<int>(...)
6 int i = aCell.member
```

⇒

```
1 type Object = {}
2 type Cell = {this ⇒
3         type E <: Object;
4         val member:this.E
5 }
6 Cell aCell = new Cell{...}
7 val i:aCell.E = aCell.member
```

Path Dependent Types

```
val p = {
  val f = ...
  def fn(...) = ...
  type T = ...
}
```



Path Dependent Types

Scalable Component Abstractions

Martin Odersky
EPFL
CH-1015 Lausanne
martin.odersky@epfl.ch

Matthias Zenger
Google Switzerland GmbH
Freigutstrasse 12
CH-8002 Zürich
zenger@google.com

ABSTRACT

We identify three programming language abstractions for the construction of reusable components: abstract type members, explicit selftypes, and modular mixin composition. Together, these abstractions enable us to transform an arbitrary assembly of static program parts with hard references between them into a system of reusable components. The transformation maintains the structure of the original system. We demonstrate this approach in two case studies, a subject/observer framework and a compiler front-end.

Categories and Subject Descriptors

D.3.3 **[Programming Languages]**: Language constructs and features – Classes and objects; inheritance; modules; packages; polymorphism; recursion.

General Terms

Languages

Keywords

Components, classes, abstract types, mixins, Scala.

passing.

An important requirement for components is that they are *reusable*; that is, that they should be applicable in contexts other than the one in which they have been developed. Generally, one requires that component reuse should be possible without modifiying a component's source code. Such modifications are undesirable because they have a tendency to create versioning problems. For instance, a version conflict might arise between an adaptation of a component in some client application and a newer version of the original component. Often, one goes even further in requiring that components are distributed and deployed only in binary form [43].

To enable safe reuse, a component needs to have *interfaces* for provided as well as for required services through which interactions with other components occur. To enable flexible reuse in new contexts, a component should also minimize "hard links" to specific other components which it requires for its functioning.

We argue that, at least to some extent, the lack of progress in component software is due to shortcomings in the programming languages used to define and integrate components. Most existing languages offer only limited support for component abstraction and composition. This holds in

Path Dependent Types

```
1  trait Cell{ this =>
2          type E <: Any
3          val member : this.E
4  }
5  object intCell extends Cell{
6          type E = Int
7          val member : this.E = 5
8  }
9  def returnMember(c : Cell) : c.E = c.member
```

Path Dependent Types

```
1  type Protocol {this =>
2          type Address
3          type Data
4          def send(a:Address, d:Data)
5          def receive(a:Address):Data
6  }
7  module IP : Protocol { ... }
8  module TCP : Protocol { ... }
```

Example: Sets

```
trait Set{ this =>
  type Elem

}
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

```
val flight = new Set{
  type Elem = ANAPassenger
  def checkin = insert
  ...
}
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

```
val flight = new Set{
  type Elem = ANAPassenger
  def checkin = insert
  ...
}
```

```
flight.checkin(alex)
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

```
val flight = new Set{
  type Elem = ANAPassenger
  def checkin = insert
  ...
}
```

```
flight.checkin(alex)
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

```
val flight = new Set{
  type Elem = ANAPassenger
  def checkin = insert
  ...
}
```

```
flight.checkin(alex)
```

```
flight.checkin(AeroflotPassenger)
```

Example: Sets

```
trait Set{ this =>
  type Elem
  def insert(e : this.Elem) : Unit
}
```

```
val flight = new Set{
  type Elem = ANAPassenger
  def checkin = insert
  ...
}
```

```
flight.checkin(alex)
```

```
flight.checkin(AeroflotPassenger)
```

Rich Expressiveness of Path Dependent Types

- Generics
- Bounded Polymorphism
- Nominality
- Family Polymorphism
- Modules

Example: Set Equality

```
trait Eq{ self =>
  type E >: bot
  def equals : E -> bool
}
```

Example: Set Equality

```
trait Eq{ self =>
  type E >: bot
  def equals : E -> bool
}
```

Equality of sets is extensional:

S = U iff S ⊆ U and S ⊇ U

Example: Set Equality

```
trait Eq{ self =>
  type E >: bot
  def equals : E -> bool
}
```

Equality of sets is extensional:

S = U iff S ⊆ U and S ⊇ U

```
trait Set extends Eq{ self =>
  type Elem
  type E >:
    ???
}
```

Example: Set Equality

```
trait Eq{ self =>
  type E >: bot
  def equals : E -> bool
}
```

Equality of sets is extensional:

S = U iff S ⊆ U and S ⊇ U

```
trait Set extends Eq{ self =>
  type Elem
  type E >:
    Set{type Elem <: ??? }
}
```

Example: Set Equality

```
trait Eq{ self =>
  type E >: bot
  def equals : E -> bool
}
```

Equality of sets is extensional:

S = U iff S ⊆ U and S ⊇ U

```
trait Set extends Eq{ self =>
  type Elem
  type E >:
    Set{type Elem <: Eq{type E >: self.Elem}}
}
```

Example: Set Equality … trees as

```
trait Eq{ self =>
  type E >: bot
  def equals : E -> bool
}
```

```
class Tree extends Set{
  type Elem = Tree
  ...
}
```

```
trait Set extends Eq{ self =>
  type Elem
  type E >:
    Set{type Elem <: Eq{type E >: self.Elem}}
}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

```
Eq{type E >: Set{type Elem <: Eq{type E >: Tree}}} <: Eq{type E >: Tree}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

```
Eq{type E >: Set{type Elem <: Eq{type E >: Tree}}} <: Eq{type E >: Tree}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

```
Eq{type E >: Set{type Elem <: Eq{type E >: Tree}}} <: Eq{type E >: Tree}
```

```
Tree <: Set{type Elem <: Eq{type E >: Tree}}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

```
Eq{type E >: Set{type Elem <: Eq{type E >: Tree}}} <: Eq{type E >: Tree}
```

```
Tree <: Set{type Elem <: Eq{type E >: Tree}}
```

```
Set{type Elem <: Tree} <: Set{type Elem <: Eq{type E >: Tree}}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

```
Eq{type E >: Set{type Elem <: Eq{type E >: Tree}}} <: Eq{type E >: Tree}
```

```
Tree <: Set{type Elem <: Eq{type E >: Tree}}
```

```
Set{type Elem <: Tree} <: Set{type Elem <: Eq{type E >: Tree}}
```

```
Tree <: Eq{type E >: Tree}
```

Example: Set Equality … trees as nested sets

```
Tree <: Eq{type E >: Tree}
```

```
Set{type Elem <: Tree} <: Eq{type E >: Tree}
```

```
Eq{type E >: Set{type Elem <: Eq{type E >: Tree}}} <: Eq{type E >: Tree}
```

```
Tree <: Set{type Elem <: Eq{type E >: Tree}}
```

```
Set{type Elem <: Tree} <: Set{type Elem <: Eq{type E >: Tree}}
```

```
Tree <: Eq{type E >: Tree}
```

Julian Mackay's PhD Thesis

- Path Dependent Types are central to Scala (and Wyvern)
- 15 years of formalising the foundations resulted in Dependent Object Types (DOT) calculi only recently proven to be sound
- Subtyping remained undecidable until 2019…

Julian Mackay's PhD Thesis

Decidable Subtyping for Path Dependent Types

ANONYMOUS AUTHOR(S)

Path dependent types have long served as an expressive component of the Scala programming language. They allow for the modelling of both bounded polymorphism and a degree of nominal subtyping. Nominality in turn provides the ability to capture first class modules. Thus a single language feature gives rise to a rich array of expressiveness. Recent work has proven path dependent types sound in the presence of both intersection and recursive types, but unfortunately typing remains undecidable, posing problems for programmers who rely on the results of type checkers. The Wyvern programming language is an object oriented language with path dependent types, recursive types and first class modules. In this paper we define two variants of Wyvern that feature decidable typing, along with machine checked proofs of decidability. Despite the restrictions, our approaches retain the ability to encode the parameteric polymorphism of Java generics along with many idioms of the Scala module system.

Additional Key Words and Phrases: keyword1, keyword2, keyword3

POPL 2020

1 INTRODUCTION

A *type member* is a named member within an object that denotes a type. We can refer to the type denoted by type member `L` of some object `o` with the notation `o.L`. The semantics of `L` are

Three Decidable Variants:

1. Wyv_{fix} – fixes types to the contexts they are defined in, thus eliminating expansive environments
2. $Wyv_{non\text{-}\mu}$ – removes recursive subtyping, thus removing the key source of expansive environments during subtyping
3. Wyv_{μ} – places syntactic restriction of the usage of recursive types

Formalising Path Dependent Types

The Essence of Dependent Object Types*

Nada Amin*, Samuel Grütter*, Martin Odersky*, Tiark Rompf†, and Sandro Stucki*

*EPFL, Switzerland: {first.last}@epfl.ch
†Purdue University, USA: {first}@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_{<:}$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_{<:}$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Keywords: Calculus, Dependent Types, Scala

1 Introduction

While hiking together in the French alps in 2013, Martin Odersky tried to explain to Phil Wadler why languages like Scala had foundations that were not directly related via the Curry-Howard isomorphism to logic. This did not go over well. As you would expect, Phil strongly disapproved. He argued that anything that was useful should have a grounding in logic. In this paper, we try to approach this goal.

We will develop a foundation for Scala from first principles. 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 instance,

Formalising Path Dependent Types

s, t, u	$::=$	**Term**
	x	*variable*
	v	*value*
	$x.f$	*selection*
	$\texttt{let}\ x = t\ \texttt{in}\ u$	*let binding*
v	$::=$	**Value**
	$\nu(x : \tau).d$	*object*
	$\lambda(x : \tau).t$	*abstraction*
d	$::=$	**Definition**
	$\{f = t\}$	*field definition*
	$\{L = \tau\}$	*type definition*
	$d \cap d$	*aggregate definition*

τ	$::=$	**Type**
	$\mu(x : \tau)$	*recursive type*
	$\forall(x : \tau).\tau$	*dependent function*
	$\{f : \tau\}$	*field definition*
	$\{L : \tau \ldots \tau\}$	*type definition*
	$x.L$	*type selection*
	$\tau \cap \tau$	*intersection*
	$\top$	*top*
	$\bot$	*bottom*

Figure 2.20: Wadlerfest DOT Syntax

Formalising Path Dependent Types

Type Soundness for Dependent Object Types (DOT)

Tiark Rompf * Nada Amin†

*Purdue University: {firstname}@purdue.edu
†EPFL: {first.last}@epfl.ch

Abstract

Scala's type system unifies aspects of ML modules, object-oriented, and functional programming. The Dependent Object Types (DOT) family of calculi has been proposed as a new theoretic foundation for Scala and similar expressive languages. Unfortunately, type soundness has only been established for restricted subsets of DOT. In fact, it has been shown that important Scala features such as type refinement or a subtyping relation with lattice structure break at least one key metatheoretic property such as environment narrowing or invertible subtyping transitivity, which are usually required for a type soundness proof.

The main contribution of this paper is to demonstrate how, perhaps surprisingly, even though these properties are lost in their full generality, 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.

starts, the first mechanized soundness proof for a calculus of the DOT (Dependent Object Types) [4] family was finally presented in 2014 [5].

The calculus proved sound by Amin et al. [5] is μDOT, a core calculus that distills the essence of Scala's objects that may contain *type members* in addition to methods and fields, along with *path-dependent types*, which are used to access such type members. μDOT models just these two features–records with type members and type selections on variables–and nothing else. This simple system already captures some essential programming patterns, and it has a clean and intuitive theory. In particular, it satisfies the intuitive and mutually dependent properties of environment narrowing and subtyping transitivity, which are usually key requirements for a soundness proof.

Alas, Amin et al. also showed that adding other important Scala features such as type refinement, mixin composition, or just a bottom type breaks at least one of these properties, which makes a soundness proof much harder to come by.

The main contribution of this paper is to demonstrate how, perhaps surprisingly, even though these properties are lost in their full generality, a richer DOT calculus that in-

Type Refinements

```
type List = {def head : Object
             def tail : List}
type AppendableList = List{def append (o : Object) : List}
```

Parametric Polymorphism

```
1 def append[E >: ⊥]
2            (e : E,
3             l : List[E])
```

⇒

```
1 def append(x : {type E >: ⊥},
2            e : x.E,
3            l : List{type E = x.E})
```

```
type List = {type E <: ⊤}
def copy(l : List) : List{type E = l.E}{
    new List{type E = l.E}
}
```

Method Parameters would result in too much boilerplate code, especially if many types.

Nominality


```
type ListAPI = {
    type List <: { this ⇒ type Elem }
    def nil(): List{ type Elem = Bot }
}
```

Transitivity

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \text{ (S-Trans)}$$

$$\frac{\Gamma \vdash x : \{L : \_ \ldots \tau\}}{\Gamma \vdash x.L <: \tau} \text{ (S-Upper)}$$

$$\frac{\Gamma \vdash x : \{L : \tau \ldots \_\}}{\Gamma \vdash \tau <: x.L} \text{ (S-Lower)}$$

$\Rightarrow$

$$\frac{\Gamma \vdash x : \{L : \_ \ldots \tau_1\} \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash x.L <: \tau_2} \text{ (S-Upper)}$$

$$\frac{\Gamma \vdash x : \{L : \tau_2 \ldots \_\} \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash \tau_1 <: x.L} \text{ (S-Lower)}$$

"Syntax directed" rather than general transitivity as inspired by Kernel $F_{<:}$

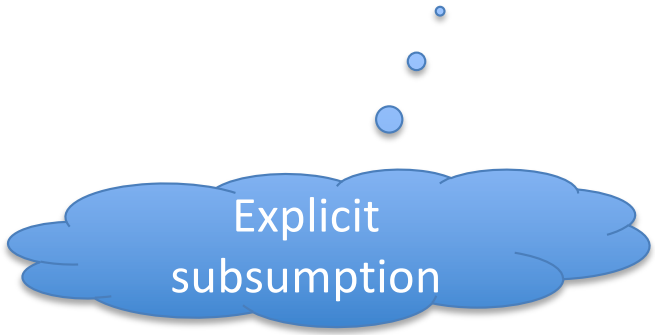
Subsumption

$$\frac{\Gamma \vdash x \ : \ \tau \qquad \Gamma \vdash \tau \ <: \ \tau'}{\Gamma \vdash x \ : \ \tau'} \ (\text{S-SUB})$$

$\Rightarrow$

$$\frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash x \ : \ \{[x/z]\sigma\}} \ (\text{T-REC})$$

$$\frac{\Gamma \vdash x \ : \ y.L \qquad \Gamma \vdash y \ : \ \{L \ : \_ \ldots \tau\}}{\Gamma \vdash x \ : \ \tau} \ (\text{T-SEL})$$



Single Bounds

σ	::=	**Declaration Type**
	$\vdots$	
	$L : \tau \ldots \tau$	

$\Rightarrow$

σ	::=	**Declaration Type**	
	$\vdots$		
	$L \leqslant \tau$		*upper bound*
	$L \geqslant \tau$		*lower bound*
	$L = \tau$		*specific type*

Wyv_{core}

τ	$::=$	**Type**
	$\forall(x : \tau).\tau$	*dependent function*
	$\{z \Rightarrow \overline{\sigma}\}$	*top refinement*
	$x.L\{z \Rightarrow \overline{\sigma}\}$	*selection refinement*
	$x.L$	*type selection*
	$\top$	*top*
	$\bot$	*bottom*
σ	$::=$	**Declaration Type**
	$L \leqslant \tau$	*upper bound*
	$L \geqslant \tau$	*lower bound*
	$L = \tau$	*equality*
	$l : \tau$	*value*
Γ	$::=$	**Environment**
	$\emptyset$	
	$\Gamma, x : \tau$	

Figure 3.9: *Wyv*$_{core}$ Syntax

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

$$\frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash x \ : \ \{[x/z]\sigma\}} \quad \text{(T-Rec)}$$

$$\frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\}}{\Gamma \vdash x \ : \ \tau} \quad \text{(T-Rfn)}$$

$$\frac{\Gamma \vdash x \ : \ y.L \qquad \Gamma \vdash y \ : \ \{L \ \leqslant \tau\}}{\Gamma \vdash x \ : \ \tau} \quad \text{(T-Sel)}$$

$$\frac{\Gamma \vdash x \ : \ \{L \ = \ \tau\}}{\Gamma \vdash x \ : \ \{L \ \leqslant \tau\}} \quad \text{(T-Eq}_1\text{)}$$

$$\frac{\Gamma \vdash x \ : \ \{L \ = \ \tau\}}{\Gamma \vdash x \ : \ \{L \ \geqslant \tau\}} \quad \text{(T-Eq}_2\text{)}$$

Figure 3.10: *Wyv*$_{core}$ Typing

Wyv$_{core}$

$$\Gamma \vdash x.L \ <: \ x.L \quad \text{(S-Rfl)} \qquad \Gamma \vdash \bot <: \tau \quad \text{(S-Bot)} \qquad \Gamma \vdash \tau <: \top \quad \text{(S-Top)}$$

$$\frac{\Gamma \vdash x \ : \ \{L \ \leqslant \tau'\} \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash x.L \ <: \ \tau} \ \text{(S-Upper)}$$

environment narrowing

$$\frac{\Gamma \vdash x \ : \ \{L \ \geqslant \tau'\} \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \tau \ <: \ x.L} \ \text{(S-Lower)}$$

contravariance

$$\frac{\Gamma \vdash \tau_2 \ <: \tau_1 \qquad \Gamma, x : \tau_2 \vdash \tau_1' \ <: \ \tau_2'}{\Gamma \vdash \forall (x : \tau_1).\tau_1' \ <: \ \forall (x : \tau_2).\tau_2'} \ \text{(S-All)}$$

$$\frac{\Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 <: \overline{\sigma}_2}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} <: \tau\{z \Rightarrow \overline{\sigma}_2\}} \ \text{(S-Rfn)} \qquad \frac{\Gamma \vdash \tau_1 \ \leqslant:: \ \tau \qquad \Gamma \vdash \tau \ <: \ \tau_2}{\Gamma \vdash \tau_1 \ <: \ \tau_2} \ \text{(S-Ext)}$$

environment narrowing

$$\frac{\forall \ \sigma_2 \ \in \ \overline{\sigma}_2, \ \exists \ \sigma_1 \ \in \ \overline{\sigma}_1 \ s.t. \ \Gamma \vdash \sigma_1 \ <: \sigma_2}{\Gamma \vdash \overline{\sigma}_1 <: \ \overline{\sigma}_2} \ \text{(S-Decls)}$$

$$\frac{\Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash L \leqslant\!/= \tau_1 <: \ L \ \leqslant \tau_2} \ \text{(S}_\sigma\text{-Upper)} \qquad \frac{\Gamma \vdash \tau_2 \ <: \ \tau_1}{\Gamma \vdash L \geqslant\!/= \tau_1 <: \ L \ \geqslant \tau_2} \ \text{(S}_\sigma\text{-Lower)}$$

$$\frac{\Gamma \vdash \tau_2 \ <: \ \tau_1 \qquad \Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash L \ = \ \tau_1 <: \ L \ = \ \tau_2} \ \text{(S}_\sigma\text{-Equal)} \qquad \frac{\Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash l : \tau_1 <: \ l : \tau_2} \ \text{(S}_\sigma\text{-Value)}$$

Undecidability of System $F_{<:}$

Bounded Quantification is Undecidable

Benjamin C. Pierce
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
bcp@cs.cmu.edu

Abstract

$F_{\leq}$ is a typed λ-calculus with subtyping and bounded second-order polymorphism. First proposed by Cardelli and Wegner, it has been widely studied as a core calculus for type systems with subtyping.

Curien and Ghelli proved the partial correctness of a recursive procedure for computing minimal types of $F_{\leq}$ terms and showed that the termination of this procedure is equivalent to the termination of its major component, a procedure for checking the subtype relation between $F_{\leq}$ types. This procedure was thought to terminate on all inputs, but the discovery of a subtle bug in a purported proof of this claim recently reopened the question of the decidability of subtyping, and hence of typechecking.

This question is settled here in the negative, using a reduction from the halting problem for two-counter Turing machines to show that the subtype relation of $F_{\leq}$ is undecidable.

and Ghelli [20, 23] address a number of syntactic properties of $F_{\leq}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [3], Martini [29], Breazu-Tannen, Coquand, Gunter, and Scedrov [1], Cardone [17], Cardelli and Longo [13], Cardelli, Martini, Mitchell, and Scedrov [14], and Curien and Ghelli [20, 21]. $F_{\leq}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [15], Bruce [2], Cardelli [12], and Canning, Cook, Hill, Olthoff, and Mitchell [5]; an extension with intersection types [19, 34] is the subject of the present author's Ph.D. thesis [32]. Bounded quantification also plays a key role in Cardelli's programing language *Quest* [10, 13] and in the *Abel* language developed at HP Labs [4, 5, 6, 18].

The original *Fun* was simplified by Bruce and Longo [3], and again by Curien and Ghelli [20]. Curien and Ghelli's formulation, called *minimal Bounded Fun* or $F_{\leq}$ ("F sub"), is the one considered here.

Like other second-order λ-calculi, the terms of $F_{\leq}$ include variables, abstractions, applications, type abstrac-

Undecidability of System $F_{<:}$

e	::=	**Expressions**
	$\vdots$	
	$\Lambda\alpha \leqslant \tau.e$	*type abstraction*

τ	::=	**Types**
	$\vdots$	
	$\top$	*top*
	$\forall(\alpha \leqslant \top).\tau$	*all*

Figure 2.11: System $F_{<:}$ Syntax extension over System F

$$\Delta \vdash \tau <: \top \quad \text{(S-Top)} \qquad \Delta \vdash \tau <: \tau \quad \text{(S-Refl)} \qquad \frac{\Delta(\alpha) = \tau}{\Delta \vdash \alpha <: \tau} \quad \text{(S-Var)}$$

$$\frac{\Delta \vdash \tau_1 <: \tau \quad \Delta \vdash \tau <: \tau_2}{\Delta \vdash \tau_1 <: \tau_2} \quad \text{(S-Trans)} \qquad \frac{\Delta \vdash \tau_2 <: \tau_1 \quad \Delta \vdash \tau_1' <: \tau_2'}{\Delta \vdash \tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \quad \text{(S-Arr)}$$

contravariance

environment narrowing

$$\frac{\Delta \vdash \tau_2 <: \tau_1 \quad \Delta, \alpha \leqslant \tau_2 \vdash \tau_1' <: \tau_2'}{\Delta \vdash \forall(\alpha \leqslant \tau_1).\tau_1' <: \forall(\alpha \leqslant \tau_2).\tau_2'} \quad \text{(S-All)}$$

Figure 2.13: System $F_{<:}$ Subtyping Rules

Undecidability of System $F_{<:}$

$$\frac{\Gamma \vdash (\forall \alpha \leqslant T_1.\top) <: (\forall \alpha \leqslant T_2.\top)}{\Gamma \vdash \neg T_1 <: \neg T_2} \ (\text{S-NEG}) \qquad \text{let } T_0 = (\forall \alpha \leqslant \top.\neg(\forall \beta \leqslant \alpha.\neg\beta))$$

$$\begin{array}{lll}
\emptyset & \vdash (\forall \alpha_0 \leqslant \top.\alpha_0) & <: (\forall \alpha_0 \leqslant T_0.(\forall \alpha_1 \leqslant \alpha_0.\neg\alpha_1)) \\
\alpha_0 \leqslant T_0 & \vdash \alpha_0 & <: (\forall \alpha_1 \leqslant \alpha_0.\neg\alpha_1) \\
\alpha_0 \leqslant T_0 & \vdash (\forall \alpha_1 \leqslant \top.\neg(\forall \beta \leqslant \alpha_1.\neg\beta)) & <: (\forall \alpha_1 \leqslant \alpha_0.\neg\alpha_1) \\
\alpha_0 \leqslant T_0, \ \alpha_1 \leqslant \alpha_0 & \vdash \neg(\forall \beta \leqslant \alpha_1.\neg\beta) & <: \neg\alpha_1 \\
\alpha_0 \leqslant T_0, \ \alpha_1 \leqslant \alpha_0 & \vdash \alpha_1 & <: (\forall \beta \leqslant \alpha_1.\neg\beta) \\
 & \vdots &
\end{array}$$

Wyv_{core} Subtype Undecidability

$$
\begin{array}{lcl}
\mathcal{F}(\alpha) & = & \alpha.A \\
\mathcal{F}(T_1 \to T_2) & = & \forall(\_ : \mathcal{F}(T_1)).\mathcal{F}(T_2) \\
\mathcal{F}(\forall(\alpha \leqslant T_1).T_2) & = & \forall(\alpha : \{A \ \leqslant \mathcal{F}(T_1)\}).\mathcal{F}(T_2) \\
\mathcal{F}(\emptyset) & = & \emptyset \\
\mathcal{F}(\alpha \leqslant T, \Delta) & = & \alpha \ : \ \{A \ \leqslant \mathcal{F}(T)\}, \ \mathcal{F}(\Delta)
\end{array}
$$

Figure 3.15: Encoding System $\mathrm{F}_{<:}$ in Wyv_{core}

Wyv_{core} Subtype Undecidability

$$
\begin{array}{lcl}
F'(\top) & = & \top \\
F'(\alpha) & = & \alpha.A \\
F'(T_1 \ \rightarrow \ T_2) & = & \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_2) \end{array} \right\} \\
F'(\forall(\alpha \leqslant T_1).T_2) & = & \neg \left\{ \alpha \Rightarrow \begin{array}{l} A \ \leqslant F'(T_1) \\ B \ \geqslant F'(T_2) \end{array} \right\} \\
\text{where } \neg\tau & = & \{X \ \geqslant \tau\} \\
\mathcal{F}(\emptyset) & = & \emptyset \\
\mathcal{F}(\alpha \leqslant T, \Delta) & = & \alpha \ : \left\{ \begin{array}{l} A \ \geqslant F'(T) \\ B \ \leqslant F'(\_) \end{array} \right\}, \ \mathcal{F}(\Delta)
\end{array}
$$

Figure 3.18: Encoding System $\text{F}_{<:}$ in Wyv_{core} without functions

Core Issues with Wyv_{core}

contravariance

environment narrowing

$$\frac{\Gamma \vdash \tau_2 \;<:\; \tau_1 \qquad \Gamma, x : \tau_2 \vdash \tau_1' \;<:\; \tau_2'}{\Gamma \vdash \forall(x : \tau_1).\tau_1' \;<:\; \forall(x : \tau_2).\tau_2'} \;(\text{S-ALL})$$

environment narrowing

recursive types

$$\frac{\Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \;<:\; \overline{\sigma}_2}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} \;<:\; \tau\{z \Rightarrow \overline{\sigma}_2\}} \;(\text{S-RFN})$$

Recursive Types

```
type Map[K <: T, V <: T]
type Node[E <: T, V <: T] = Map[self.E, Node]
```


```
type Graph = {type Node <: {val neighbors : Map[Edge, Node]}
              type Edge <: {val origin, destination : Node}}
```

Contravariance

```
def append[E >: ⊥](e : E, l : List[E]) : List[E] =
  match l with
    nil = e :: nil
    e'::t = e'::(append e t)
```

Environment Narrowing

- Required in the subtyping of both universally quantified and recursive types (S-ALL and S-RFN before)…
- Environments are narrowed when evaluating functions, requiring well-formedness to be maintained in the presence of a narrower type…

WHAT IF WE DO NOT NARROW ENVIRONMENT???

Decidable Wyvern 1: Wyv_{fix}

$$\frac{\Gamma_2 \vdash \tau_2 \;<:\; \tau_1 \dashv \Gamma_1 \qquad \Gamma_1, x : \tau_1 \vdash \tau_1' \;<:\; \tau_2' \dashv \Gamma_2, x : \tau_2}{\Gamma_1 \vdash \forall(x : \tau_1).\tau_1' \;<:\; \forall(x : \tau_2).\tau_2' \dashv \Gamma_2} \text{(S-All)}$$

a.k.a. “Doubleheaded Subtyping”

$$\frac{\Gamma_1, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \;<:\; \overline{\sigma}_2 \dashv \Gamma_2, z : \tau\{z \Rightarrow \overline{\sigma}_2\}}{\Gamma_1 \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} \;<:\; \tau\{z \Rightarrow \overline{\sigma}_2\} \dashv \Gamma_2} \text{(S-Rfn)}$$

WHAT IF WE WANT TYPE SOUNDNESS/SAFETY???

Decidable Wyvern 2: $\text{Wyv}_{\text{non-}\mu}$

$$\frac{\Gamma, x : \tau \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash \forall(x : \tau).\tau_1 \ <: \ \forall(x : \tau).\tau_2} \ (\text{S-ALL})$$

No Recursive Types!!!

$$\frac{\Gamma \vdash \overline{\sigma}_1 <: \overline{\sigma}_2}{\Gamma \vdash \tau\{\overline{\sigma}_1\} <: \tau\{\overline{\sigma}_2\}} \ (\text{S-RFN})$$

WHAT ABOUT THAT SHAPE/MATERIAL STUFF???

Adding Nominality to Wyv_{core}

```
new ListAPI{
        type Equatable = { ... }
        type List = Equatable{ self => ... }
}
```

$$\begin{array}{lll} \sigma & ::= & \textbf{Declaration Type} \\ & \vdots & \\ & L \preceq \tau & \textit{nominal definition} \end{array}$$

$$\frac{\Gamma \vdash x : \{L \preceq \tau\}}{\Gamma \vdash x : \{L \leqslant \tau\}} \quad (\text{T-Nom})$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash L \preceq \tau_1 <: L \leqslant \tau_2} \quad (\text{S}_\sigma\text{-Upper}_\preceq)$$

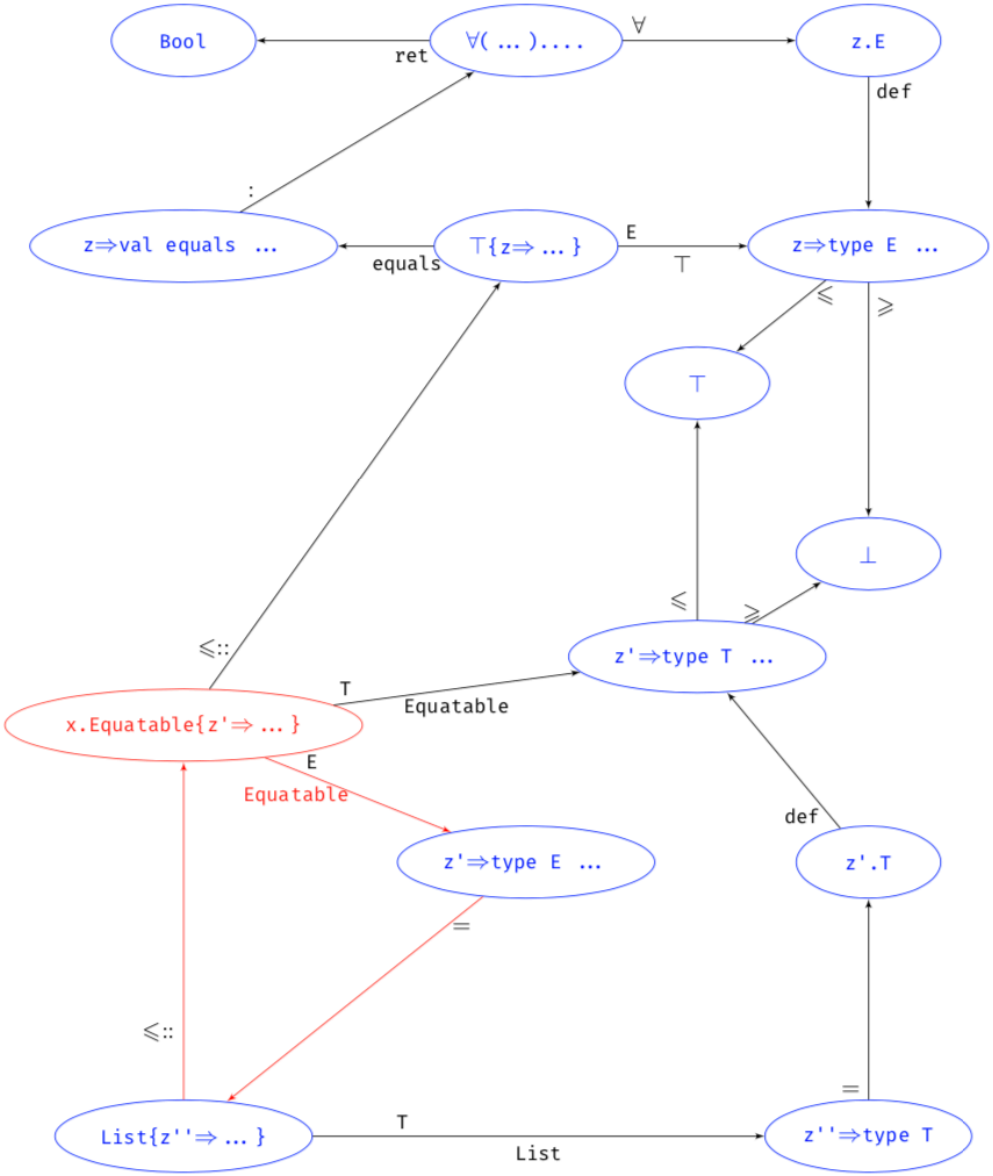
$$\Gamma \vdash L \preceq \tau <: L \preceq \tau \quad (\text{S}_\sigma\text{-Nominal})$$

```
new ListAPI{
        type Equatable = { ... }
        type List ≺ Equatable{ self => ... }
}
```

Adding Nominality to Wyv_{core}

```
1  {x =>
2    type Equatable ≺ ⊤{z =>
3                     type E >: ⊥
4                     val equals : ∀(y : z.E).Bool
5                   }
6    type List ≺ x.Equatable{z' =>
7                     type T <: ⊤
8                     type E = x.List{z'' => type T = z'.T}
9                   }
10 }
```

Decidability with Type Graphs



Material/Shape Separated Wyv_{core}

Type

$$
\begin{array}{lll}
\tau & ::= & \\
 & \eta\{z \Rightarrow \overline{\sigma}\} & \\
 & x.L\{z \Rightarrow \overline{\delta}\} & \\
 & x.M & \\
 & \forall(x : \tau).\tau & \\
 & \top & \\
 & \bot &
\end{array}
$$

Material Type

$$
\begin{array}{lll}
\eta & ::= & \\
 & \eta\{z \Rightarrow \overline{\delta}\} & \\
 & x.M & \\
 & \forall(x : \eta).\eta & \\
 & \top & \\
 & \bot &
\end{array}
$$

Decl. Type

$$
\begin{array}{lll}
\sigma & ::= & \\
 & M \leqslant \tau & \\
 & M \geqslant \eta & \\
 & M = \eta & \\
 & L \preceq \tau & \\
 & l : \eta &
\end{array}
$$

Mat. Decl. Type

$$
\begin{array}{lll}
\delta & ::= & \\
 & M \leqslant \eta & \\
 & M \geqslant \eta & \\
 & M = \eta & \\
 & L \preceq \eta & \\
 & l : \eta &
\end{array}
$$

$$
\begin{array}{lll}
L & ::= & \textbf{Type Name} \\
 & M & \textit{material} \\
 & S & \textit{shape}
\end{array}
$$

Decidable Wyvern 3: Wyv_{μ}

$$
\begin{array}{lll}
\eta & ::= & \textbf{Material Type} \\
 & \vdots & \\
 & \eta\{\overline{\delta}\} &
\end{array}
$$

$$
\frac{\Gamma, x : \tau \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash \forall (x : \tau).\tau_1 \ <: \ \forall (x : \tau).\tau_2} \quad \text{(S-ALL)}
$$

Future Work

- Type Safety for Wyv_{fix}
- Transitivity
- Intersection and Union Types
- *Decidable Scala?*