The Trouble with Types

Martin Odersky

EPFL and Typesafe

strangeloop

·2013·

Types

Everyone has an opinion on them

Industry:

– Used to be the norm (C/C++, Java).

– Today split about evenly with dynamic.

Academia:

– Static types are more common.

Static: Points in Favor

- More efficient
- Better tooling
- Fewer tests needed
- Better documentation
- Safety net for maintenance

Dynamic: Points in Favor

- Simpler languages
- Fewer puzzling compiler errors
- No boilerplate
- Easier for exploration
- No type-imposed limits to expressiveness

What is Good Design?

– Clear

– Correct

– Minimal

– The opposite of “random”

Great designs are often discovered, not invented.

Elements Of Great Designs:

Patterns

&

Constraints

Example: Bach Fugues

César Franck : Prélude, Choral & Fugue, premières mesures

What Is A Good Language for Design?

One that helps discovering great designs.

What Is A Good Language for Design?

One that helps discovering great designs.

Patterns → Abstractions

Constraints → Types

Example

– Functional Collections
– And their generalizations, e.g. monads, applicatives

Powerful patterns made safe by types.

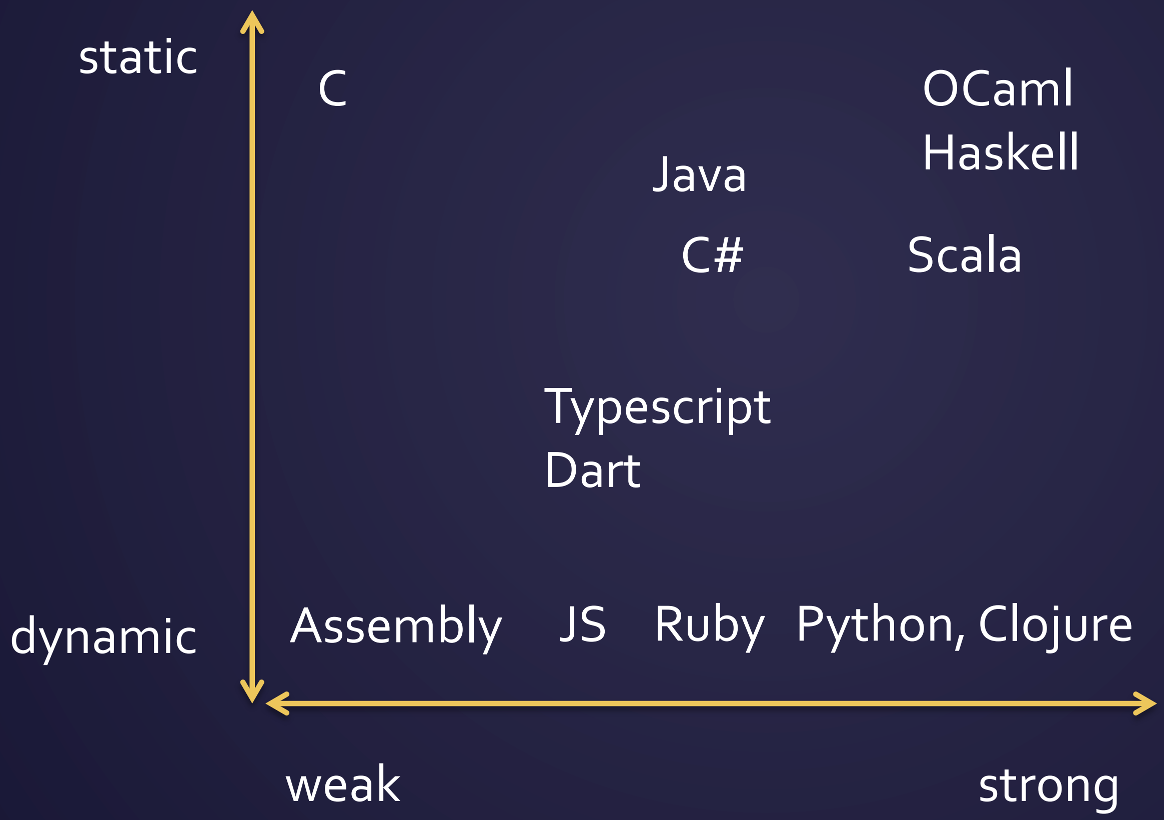
But...

Type systems are hairy.

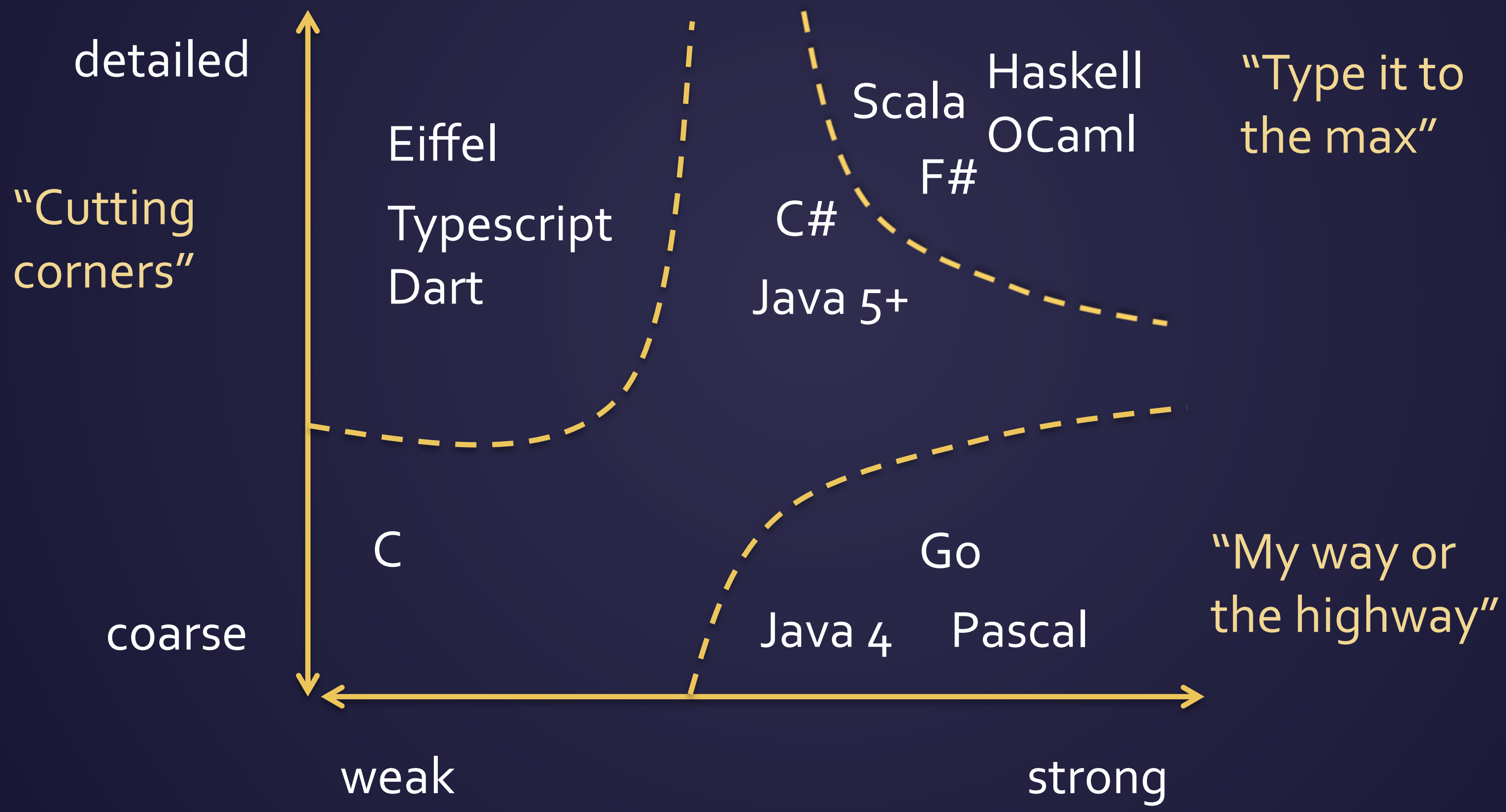
Otherwise there would not be so many different ones.

I'm not against types, but I don't know of any type systems that aren't a complete pain, so I still like dynamic typing [Alan Kay]

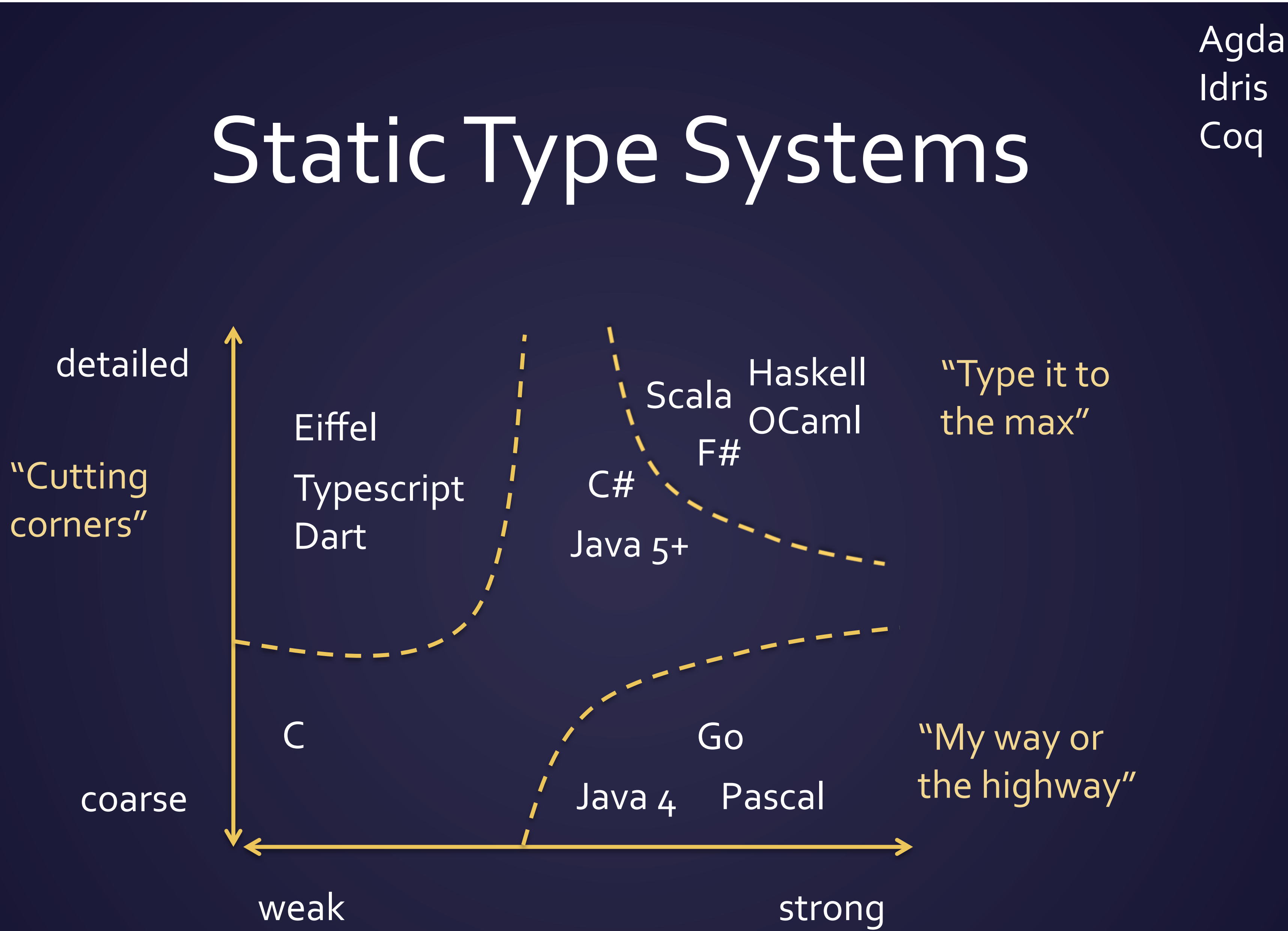
Type Systems Landscape



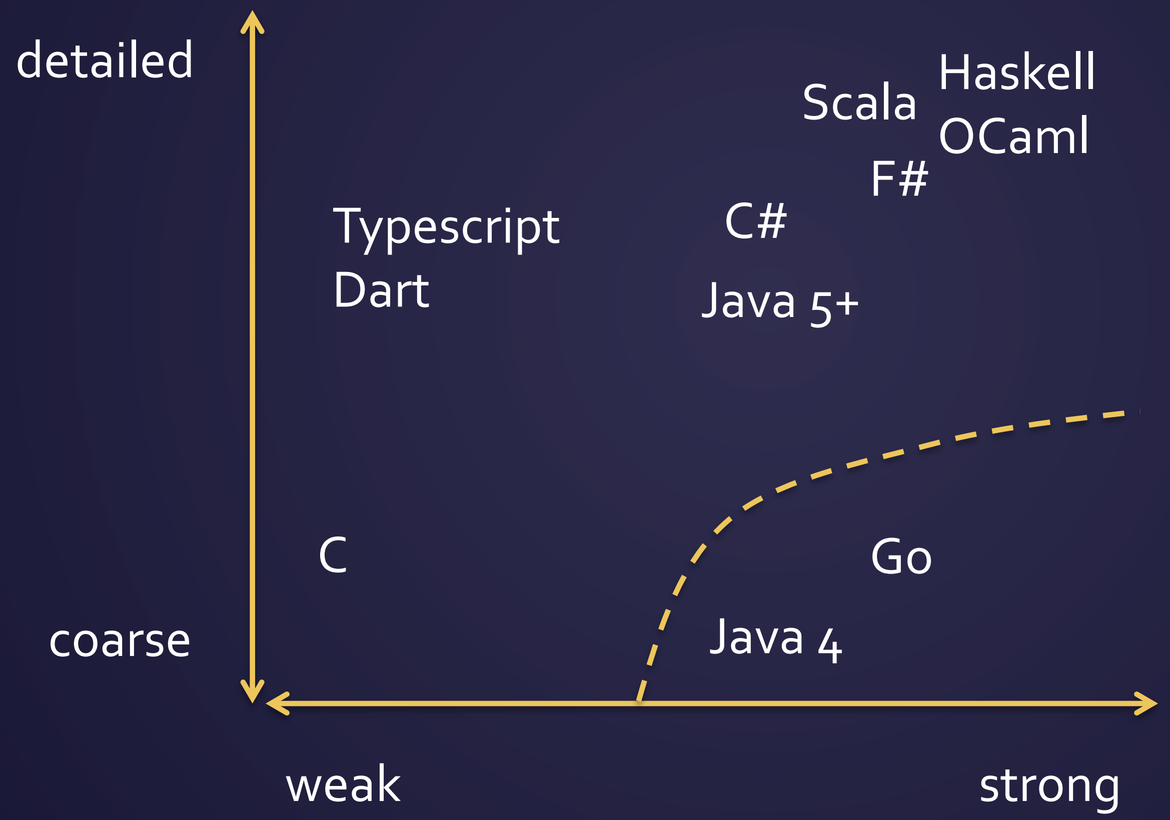
Static Type Systems



Static Type Systems



(1) My Way or the Highway



My Way or The Highway

Simple type systems

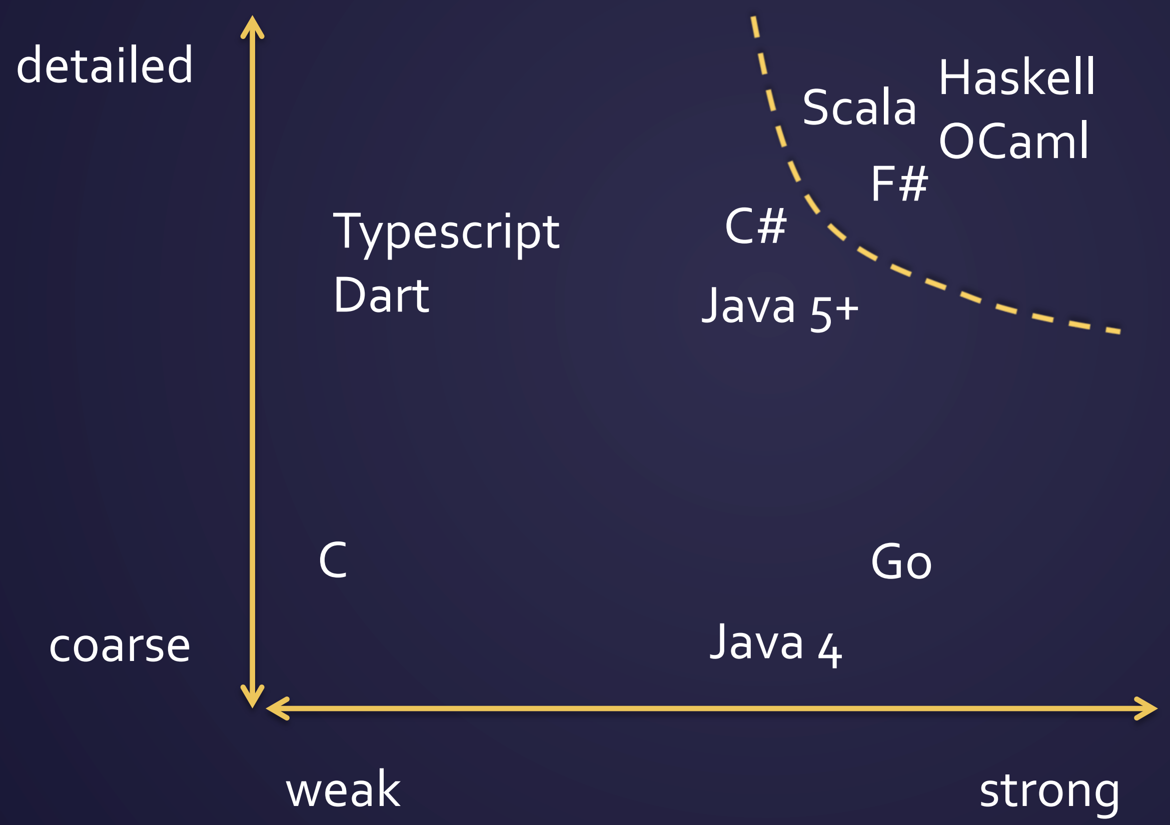
No generics

Not that extensible by users

→ Simpler tooling

→ Highly normative

(3) Type it to the Max



Type it to the Max

Rich* language to write types

Type combination forms, including generics.

Type systems often inspired by logic.

* Often, turing complete

Type it to the Max

Where dynamic languages had the upper hand:

- No type-imposed limits to expressiveness
  - → Rich type system + escape hatches such as casts
- No boilerplate
  - → Type Inference
- Easier for exploration
  - → Bottom type Nothing, ???

Making Good Use of Nothing

```
def f(x: Int) = ???
```

Making Good Use of Nothing

```
def f(x: Int): Nothing = ???

if (x < 0) ??? else f(x)
```

Other Strengths of Dynamic

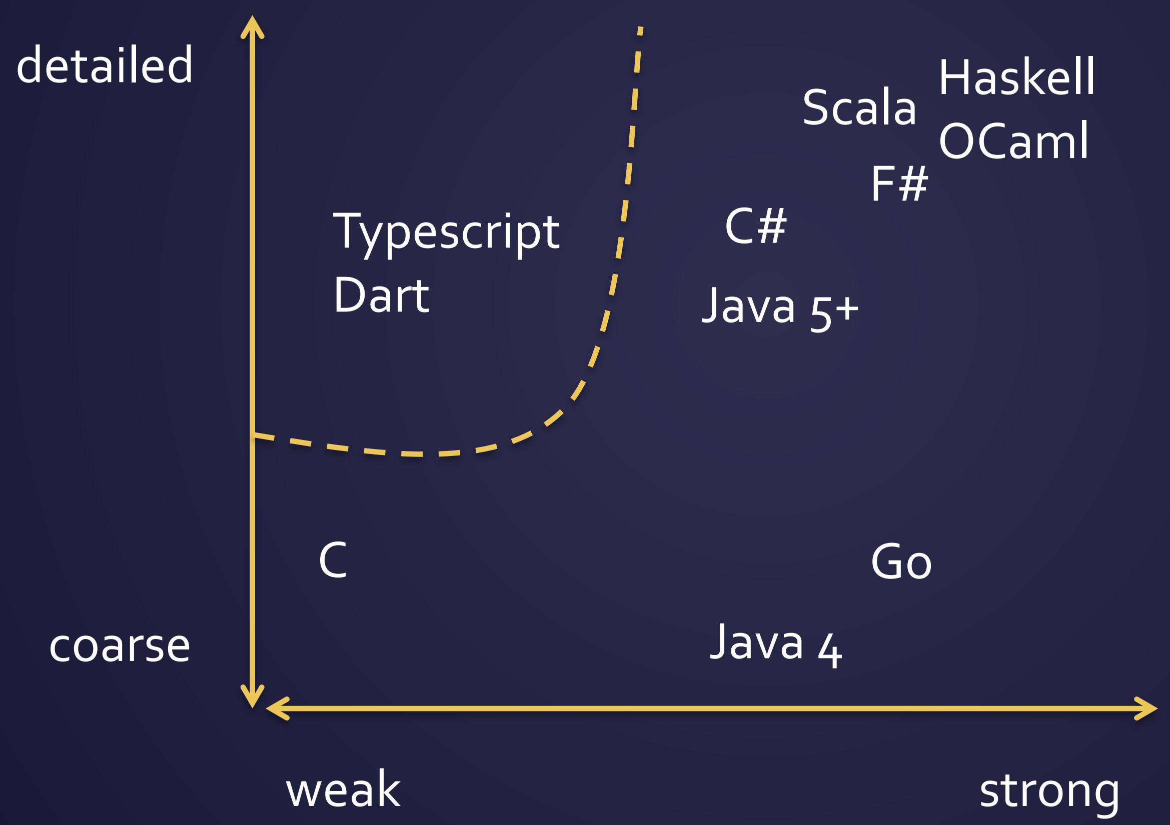
- Simpler languages
  - Rich types add complexity
- Fewer puzzling compiler errors

```
5862.scala:36: error: type mismatch;
 found   : scala.collection.mutable.Iterable[_ >: (MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 } <: Object] with
scala.collection.mutable.Builder[(MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 },scala.collection.mutable.Iterable[_ >: (MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 } <: Object] with
scala.collection.mutable.Builder[(MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 },scala.collection.mutable.Iterable[_ >: (MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 } <: Object] with
scala.collection.mutable.Builder[(MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 },scala.collection.mutable.Iterable[_ >: (MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]]) with test.TaggedMapper[_$1,_$2,_$3] forSome
{ type _$1; type _$2; type _$3 } <: Object] with
scala.collection.mutable.Builder[(MapReduceJob.this.DataSource,
scala.collection.mutable.Set[test.TaggedMapper[_, _, _]])
```

Need a Type Debugger!

and so on for another 200 lines

(3) Cutting Corners



Cutting Corners

- Appeal to user's intuitions (covariant generics).
  - Employee are Persons
  - So functions from Employees to Employers are also functions from Persons to Employers, right?
- Embrace unsoundness.
- Easy, and superficially simple.
- But, fundamentally, a hack.
- Can we build great designs on false theories?

Precision

Soundness

Simplicity

Take Any Two?

Abstractions

Two fundamental forms

– Parameters (positional, functional)

– Abstract Members (name-based, object-oriented)

Abstractions

Two fundamental forms

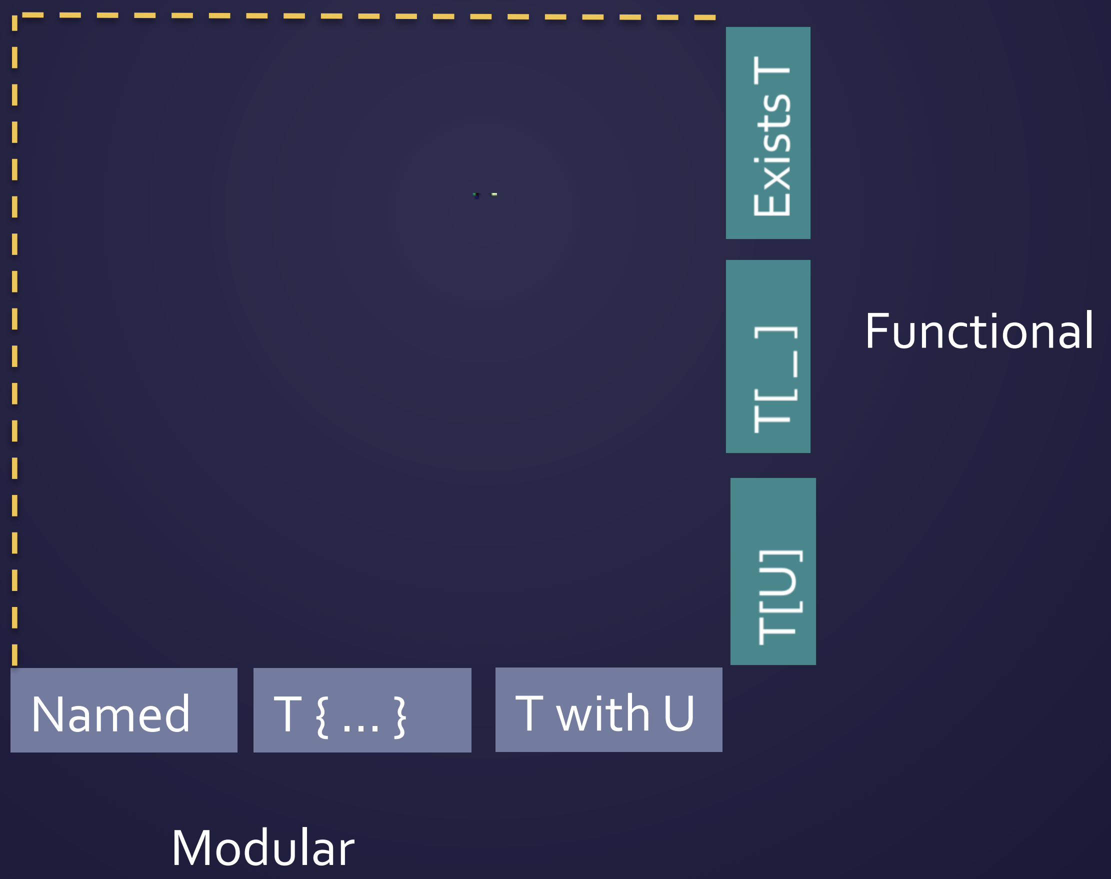
– Parameters (positional, functional)

– Abstract Members (name-based, modular)

Types in Scala

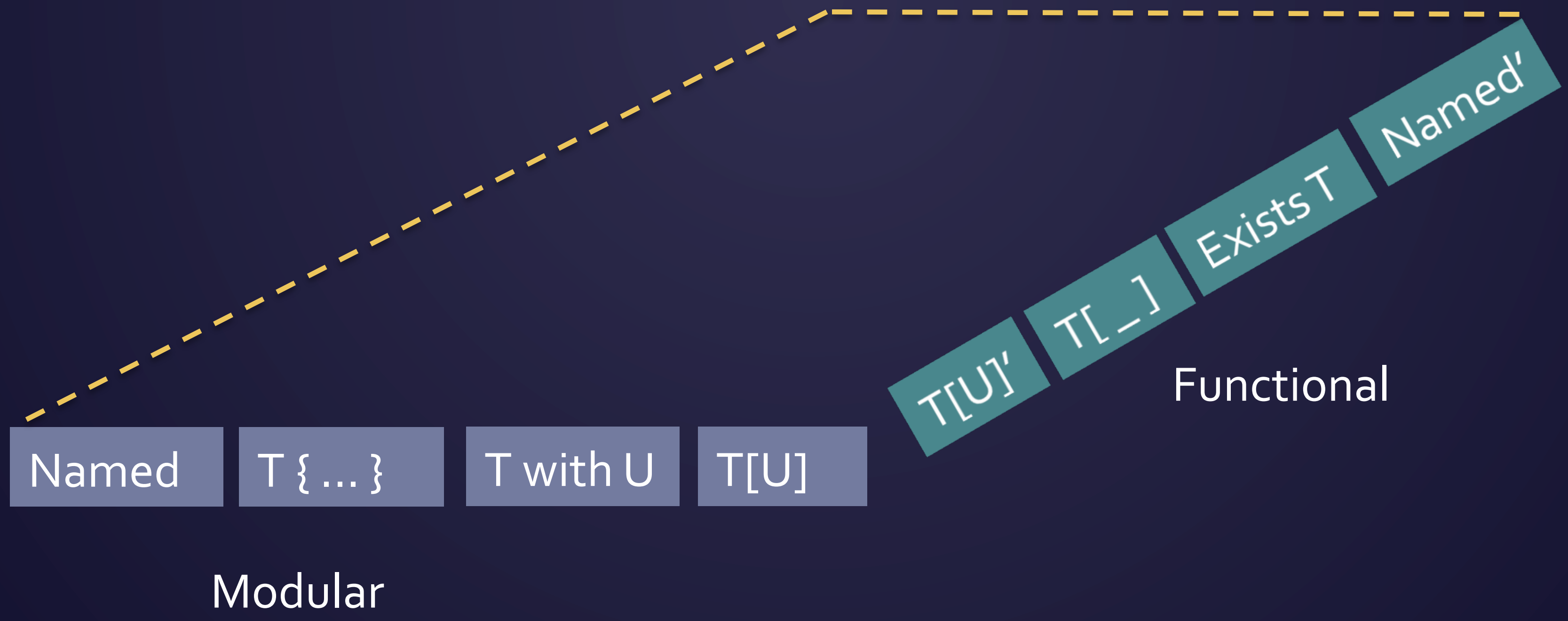
`scala.collection.BitSet`	Named Type	Modular
`Channel with Logged`	Compound Type	Modular
`Channel { def close(): Unit }`	Refined Type	Modular
`List[String]`	Parameterized	Functional
`List[T] forSome { type T }`	Existential Type	Functional
`List`	Higher-Kinded	Functional

Orthogonal Design

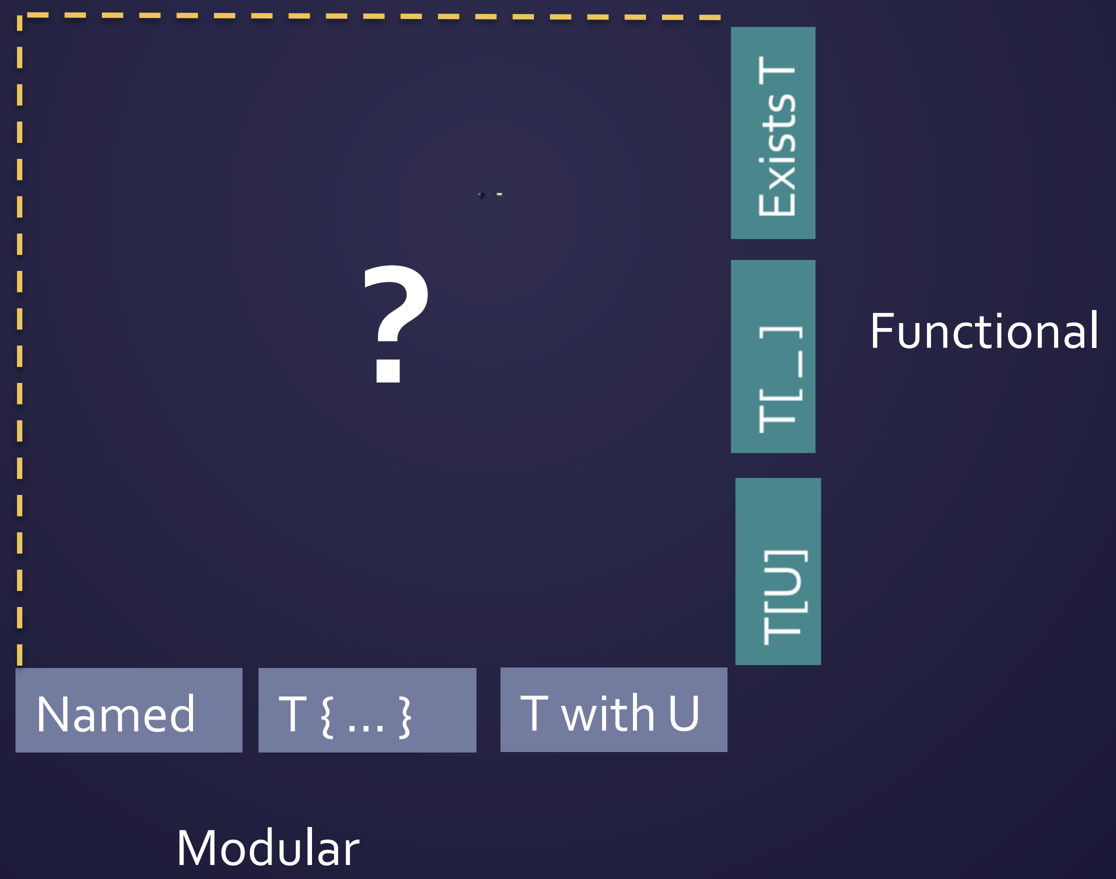


Non-Orthogonal Design

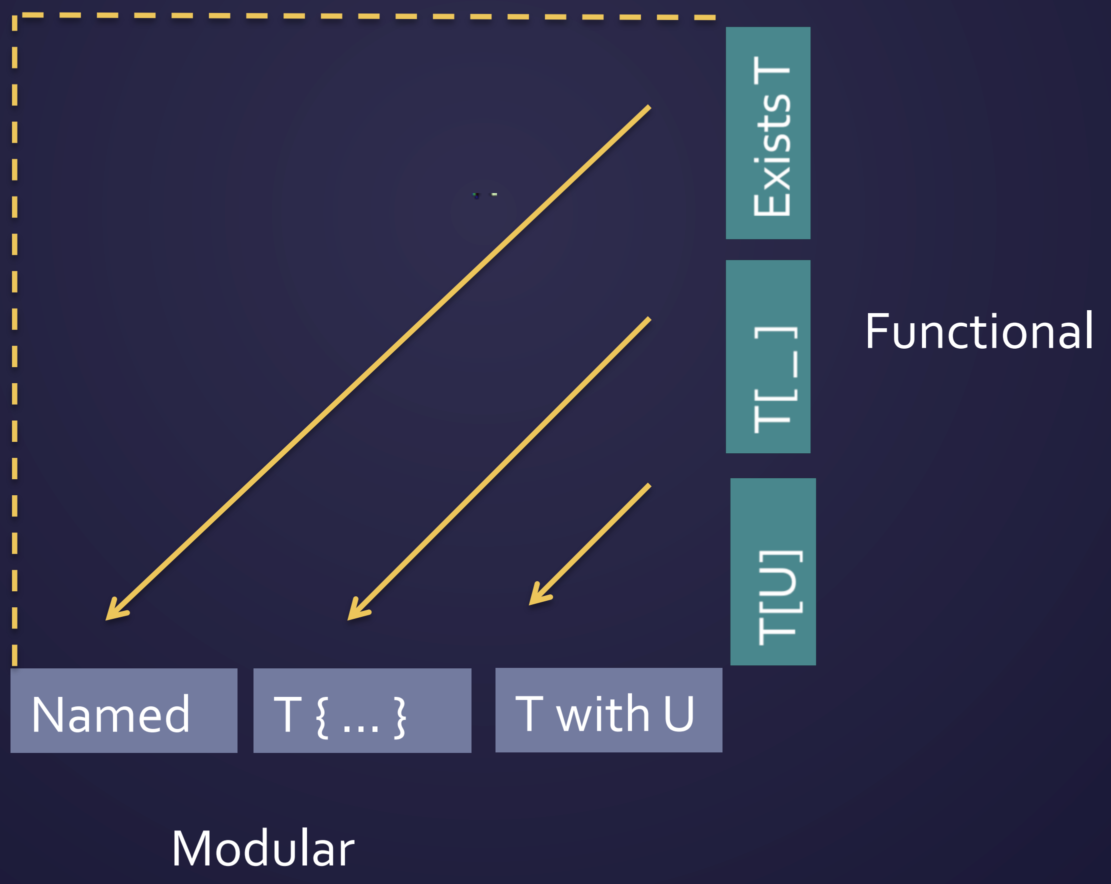
More Features
Fewer combinations



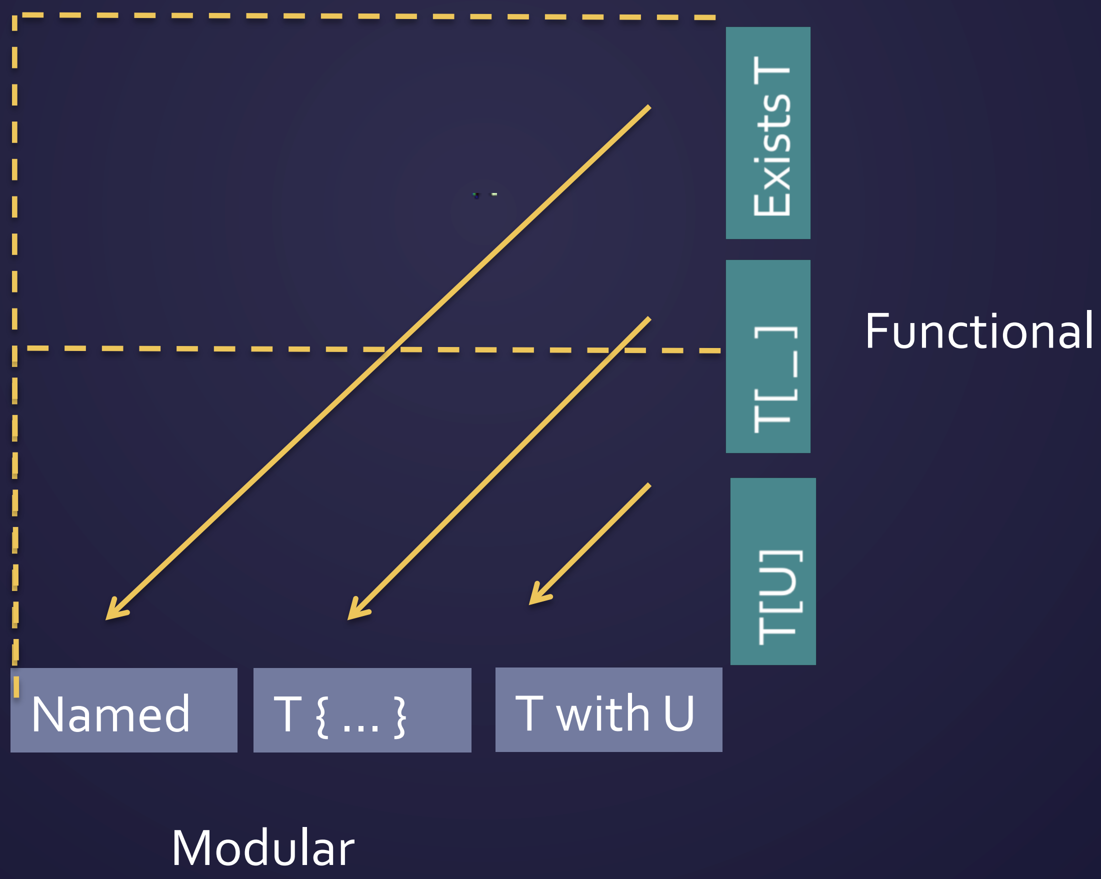
Too Many Combinations?



Projections Reduce Dimensionality



Projections Help Remove Features



Dot and Dotty

DOT: Calculus for Dependent Object Types

Dotty: A Scala-Like Language with DOT as its core

[FOOL 2012]

Dependent Object Types

Towards a foundation for Scala's type system

Nada Amin Adriaan Moors Martin Odersky
EPFL
first.last@epfl.ch

Abstract

We propose a new type-theoretic foundation of Scala and languages like it: the Dependent Object Types (DOT) calculus. DOT models Scala's path-dependent types, abstract type members and its mixture of nominal and structural typing through the use of refinement types. The core formalism makes no attempt to model inheritance and mixin composition. DOT normalizes Scala's type system by unifying the constructs for type members and by providing classical intersection and union types which simplify greatest lower bound and least upper bound computations.

In this paper, we present the DOT calculus, both formally and informally. We also discuss our work-in-progress to prove type-safety of the calculus.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Abstract data types, Classes and objects, polymorphism; D.3.1 [*Formal Definitions and Theory*]: Syntax, Semantics; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]; F.3.3 [*Studies of Program Constructs*]: Object-oriented constructs, type structure; F.3.2 [*Semantics or Programming Languages*]: Operational semantics

General Terms Languages, Theory, Verification

Keywords calculus, objects, dependent types

1. Introduction

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. An essential ingredient of this unification is objects with type members. Given a stable path to an object, its type members can be accessed as types, called path-dependent types.

This paper presents Dependent Object Types (DOT), a small object calculus with path-dependent types. In addition to path-dependent types, types in DOT are built from refinements, intersections and unions. A refinement extends a type by (re-)declaring members, which can be types, values or methods.

We propose DOT as a new type-theoretic foundation of Scala and languages like it. The properties we are interested in modeling are Scala's path-dependent types and abstract type members, as well as its mixture of nominal and structural typing through the use of refinement types. Compared to previous approaches [5, 14], we make no attempt to model inheritance or mixin composition. Indeed we will argue that such concepts are better modeled in a different setting.

The DOT calculus does not precisely describe what's currently in Scala. It is more normative than descriptive. The main point of deviation concerns the difference between Scala's compound type formation using **with** and classical type intersection, as it is modeled in the calculus. Scala, and the previous calculi attempting to model it, conflates the concepts of compound types (which inherit the members of several parent types) and mixin composition (which build classes from other classes and traits). At first glance, this offers an economy of concepts. However, it is problematic because mixin composition and intersection types have quite different properties. In the case of several inherited members with the same name, mixin composition has to pick one which overrides the others. It uses for that the concept of linearization of a trait hierarchy. Typically, given two independent traits T_1 and T_2 with a common method m, the mixin composition T_1 **with** T_2 would pick the m in T_2, whereas the member in T_1 would be available via a super-call. All this makes sense from an implementation standpoint. From a typing standpoint it is more awkward, because it breaks commutativity and with it several monotonicity properties.

In the present calculus, we replace Scala's compound types by classical intersection types, which are commutative. We also complement this by classical union types. Intersections and unions form a lattice wrt subtyping. This addresses another problematic feature of Scala: In Scala's current type system, least upper bounds and greatest lower bounds do not always exist. Here is an example: given two traits `A` and `B`, where each declares an abstract upper-bounded type member `T`,

```
trait A { type T <: A }
trait B { type T <: B }
```

the greatest lower bound of `A` and `B` is approximated by the infinite sequence

```
A with B { type T <: A with B { type T <: A with B {
  type T < ...
}}}
```

The limit of this sequence does not exist as a type in Scala.

This is problematic because greatest lower bounds and least upper bounds play a central role in Scala's type inference. For example, in order to infer the type of an **if** expression such as

```
if (cond) ((a: A) => c: C) else ((b: B) => d: D)
```

type inference tries to compute the greatest lower bound of `A` and `B` and the least upper bound of `C` and `D`. The absence of universal greatest lower bounds and least upper bounds makes type inference more brittle and more unpredictable.

FOOL '12 October 22, 2012, Tucson, AZ, USA.

Syntax

x, y, z	Variable
l	Value label
m	Method label
$v ::=$	Value
$\quad x$	variable
$t ::=$	Term
$\quad v$	value
$\quad \textbf{val}\ x = \textbf{new}\ c;\ t$	new instance
$\quad t.l$	field selection
$\quad t.m(t)$	method invocation
$p ::=$	Path
$\quad x$	variable
$\quad p.l$	selection
$c ::= T_c \left\{\overline{d}\right\}$	Constructor
$d ::=$	Initialization
$\quad l = v$	field initialization
$\quad m(x) = t$	method initialization
$s ::= \overline{x \mapsto c}$	Store

$L ::=$	Type label
$\quad L_c$	class label
$\quad L_a$	abstract type label
$S, T, U, V, W ::=$	Type
$\quad p.L$	type selection
$\quad T\left\{z \Rightarrow \overline{D}\right\}$	refinement
$\quad T \wedge T$	intersection type
$\quad T \vee T$	union type
$\quad \top$	top type
$\quad \bot$	bottom type
$S_c, T_c ::=$	Concrete type
$\quad p.L_c \mid T_c\left\{z \Rightarrow \overline{D}\right\} \mid T_c \wedge T_c \mid \top$	
$D ::=$	Declaration
$\quad L : S..U$	type declaration
$\quad l : T$	value declaration
$\quad m : S \to U$	method declaration
$\Gamma ::= \overline{x : T}$	Environment

Reduction $\boxed{t \mid s \ \to\ t' \mid s'}$

$$\frac{y \mapsto T_c\left\{\overline{l = v'}\ \overline{m(x) = t}\right\} \in s}{y.m_i(v) \mid s \ \to\ [v/x_i]t_i \mid s} \quad \text{(MSEL)}$$

$$\textbf{val}\ x = \textbf{new}\ c;\ t \mid s \ \to\ t \mid s, x \mapsto c \quad \text{(NEW)}$$

$$\frac{y \mapsto T_c\left\{\overline{l = v}\ \overline{m(x) = t}\right\} \in s}{y.l_i \mid s \ \to\ v_i \mid s} \quad \text{(SEL)}$$

$$\frac{t \mid s \ \to\ t' \mid s'}{e[t] \mid s \ \to\ e[t'] \mid s'} \quad \text{(CONTEXT)}$$

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

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

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

$$\frac{\Gamma \vdash t \ni l : T'}{\Gamma \vdash t.l : T'} \quad \text{(SEL)}$$

$$\frac{\Gamma \vdash t \ni m : S \to T \qquad \Gamma \vdash t' : T' ,\ T' <: S}{\Gamma \vdash t.m(t') : T} \quad \text{(MSEL)}$$

$$\frac{y \notin fn(T') \qquad \Gamma \vdash T_c\ \textbf{wfe}\ ,\ T_c \prec_y \overline{L : S..U}, \overline{D} \qquad \Gamma, y : T_c \vdash \overline{S <: U}\ ,\ \overline{d : D}\ ,\ t' : T'}{\Gamma \vdash \textbf{val}\ y = \textbf{new}\ T_c\left\{\overline{d}\right\};\ t' : T'} \quad \text{(NEW)}$$

Declaration Assignment $\boxed{\Gamma \vdash d : D}$

$$\frac{\Gamma \vdash v : V' ,\ V' <: V}{\Gamma \vdash (l = v) : (l : V)} \quad \text{(VDECL)}$$

$$\frac{\Gamma \vdash S\ \textbf{wfe} \qquad \Gamma, x : S \vdash t : T' ,\ T' <: T}{\Gamma \vdash (m(x) = t) : (m : S \to T)} \quad \text{(MDECL)}$$

Types in Dotty

`scala.collection.BitSet`	Named Type
`Channel & Logged`	Intersection Type
`Channel { def close(): Unit }`	Refined Type
(`List[String]`	Parameterized)
~~`List[T] forSome { tpe T }`~~	~~Existential Type~~
~~`List`~~	~~Higher-Kinded~~

Modelling Generics

```
class Set[T] { ... }      →   class Set { type $T }
Set[String]               →   Set { type $T = String }

class List[+T] { ... }    →   class List { type $T }
List[String]              →   List { type $T <: String }
```

Parameters → Abstract members

Arguments → Refinements

Making Parameters Public

```
class Set[type Elem] {...}          class Set { type Elem ...}
Set[String]                          Set { type Elem = String }

class List[type +Elem] {...}        class List { type Elem ...}
List[String]                         List { type Elem <: String }
```

Analogous to "val" parameters:

```
class C(val fld: Int)                class C { val fld: Int }
```

Expressing Existentials

What is the type of Lists with arbitrary element type?

Previously: `List[_]`

`List[T] forSome { type T }`

Now: `List`

(Types can have abstract members)

Expressing Higher-Kinded

- What is the type of List constructors?
- Previously: List
- Now: List

- Can always instantiate later:

```
type X = List
X { type T = String }
X[String]
```

In a Nutshell

In this system,

Existential = Higher-kinded

In fact, both are just types with abstract members.

We do not distinguish between types and type constructors.

Subtyping

Fundamental relation:

$T1 <: T2$

$T1$ is a subtype of $T2$.

Comes in many guises:

```
Implementation matches Interface
Type class extension
Signature ascription
```

Native Meets and Joins

- The horrible type error message came from a computed join of two types.
- Problem: In Scala, the least upper bound of two types can be infinitely large.
- Adding native & and | types fixes that.

Will this Be Scala?

- Hopefully. Depends on how compatible we can make it.
- Note: SIP 18 already forces you to flag usages of existentials and higher-kinded types in Scala.
- This should give you a some indication how much effort would be needed to convert.

The Essence of Scala

Harness the power of naming

A small language struggling to get out

Types Are Trouble

– Tooling

– Error messages

– Conceptual complexity

– Scope for misuse

But I believe they are worth it,

because they can lead to great designs.