subtyping

```
fn (x : t) => e
```

```
e1(e2)
```

$$\lambda\,(x : \tau)\, e$$

$$e_1(e_2)$$

```
fn (x : 'a) => e
```

```
(e1 : t1 -> t2)(e2)
```

$$
\begin{aligned}
\tau \; ::= \; & t \\
\mid \; & \tau_1 \to \tau_2 \\
\mid \; & \forall t.\tau
\end{aligned}
$$

System F

```
'a -> 'b
```

$$\forall \alpha . \forall \beta . \alpha \to \beta$$

```
signature ANIMAL = sig
  type t
  val name : t -> string
  val num_legs : t -> int
  val sound : t -> string
end
```

```
signature ANIMAL = sig
  type t
  val name : t -> string
  val num_legs : t -> int
  val sound : t -> string
end

structure Cat : ANIMAL
structure Dog : ANIMAL

val cats : Cat.t list = [luna, bella, oliver, charlie]
val dogs : Dog.t list = [max, cooper, lucy, daisy]
```

no interoperability between Cat and Dog, despite common interface

```
signature ANIMAL = sig
  type t
  val name : t -> string
  val num_legs : t -> int
  val sound : t -> string
end

structure Animal : ANIMAL = struct
  datatype t = Cat of string | Dog of string
  fun name (Cat c) = c | name (Dog d) = d
  fun num_legs _ = 4
  fun sound (Cat _) = “meow” | sound (Dog _) = “woof”
end

val animals = [luna, max, lucy, charlie]
```

cannot distinguish between Cat and Dog

```
signature ANIMAL = sig
  type 'a t
  val name : 'a t -> string
  val num_legs : 'a t -> int
  val sound : 'a t -> string
end

structure Animal : ANIMAL = struct
  datatype 'a t = Cat of string | Dog of string
  (* ... *)
end

datatype cat = Cat and dog = Dog
val cats : cat Animal.t list = [luna, bella, oliver]
val dogs : dog Animal.t list = [max, cooper, lucy]
```

```
signature ANIMAL = sig
  type 'a t
  val name : 'a t -> string
  val num_legs : 'a t -> int
  val sound : 'a t -> string
end

structure Animal : ANIMAL = struct
  datatype 'a t = Cat of string | Dog of string
  (* ... *)
end

datatype cat = Cat and dog = Dog
val cats : cat Animal.t list = [luna, bella, oliver]
val dogs : dog Animal.t list = [max, cooper, lucy]
```

phantom type

would be better with GADTs (next week)

want to **separate** instances of Cat from Dog…

without making them completely **non-interoperable**

why aren’t **polymorphic types** enough?

subsumption

if e has type σ

and σ is a **subtype** of τ

then e has type τ

"any cat is an animal,
and any dog is an animal"

subsumption

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

(reflexive, antisymmetric, transitive)

```
type big = { x: t1, y: t2, z: t3 }
type small = { x: t1, y: t2 }

fun f (s : small) = #x s
val b = { x = 1, y = 2, z = 3 }

(* f b *)
```

width

$$\frac{}{\left\{ l_i : \tau_i^{\; i \in 1..n+k} \right\} <: \left\{ l_i : \tau_i^{\; i \in 1..n} \right\}}$$

depth

$$\frac{\sigma_i <: \tau_i}{\left\{ l_i : \sigma_i{}^{i \in 1..n} \right\} <: \left\{ l_i : \tau_i{}^{i \in 1..n} \right\}}$$

variant width

$$\frac{}{\left\langle l_i : \tau_i{}^{i \in 1..n} \right\rangle <: \left\langle l_i : \tau_i{}^{i \in 1..n+k} \right\rangle}$$

$$\frac{}{\left\langle l_i : \tau_i^{\ i\in 1..n} \right\rangle <: \left\langle l_i : \tau_i^{\ i\in 1..n+k} \right\rangle}$$

$$\frac{}{\left\{ l_i : \tau_i^{\ i\in 1..n+k} \right\} <: \left\{ l_i : \tau_i^{\ i\in 1..n} \right\}}$$

```
datatype big =
    Xb of t1
  | Yb of t2
  | Zb of t3

datatype small =
    Xs of t1
  | Ys of t2
```

small product   big sum

big product   small sum

Any

Enum Animal Container

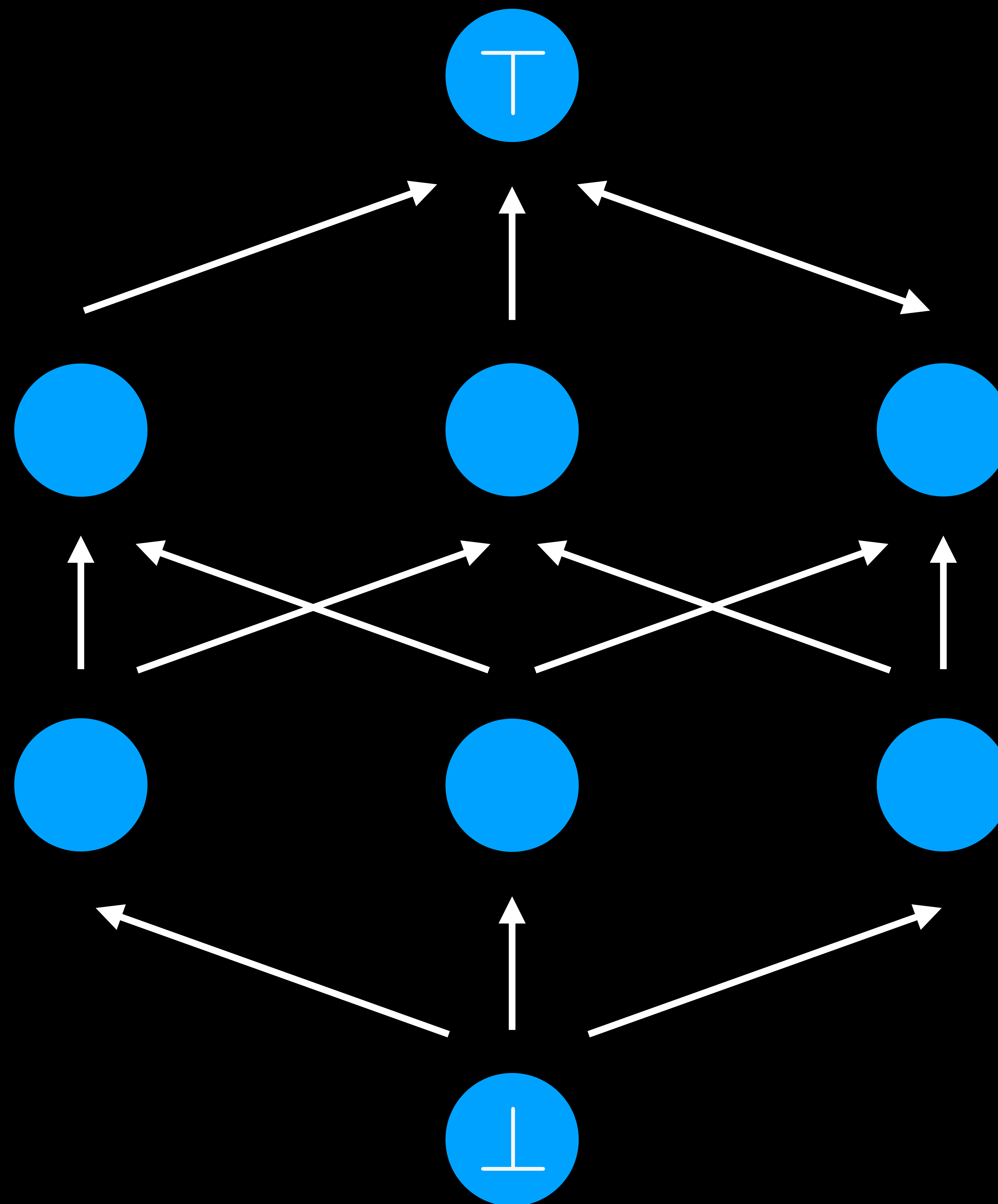
Int Bool Cat Dog Socket Image List

TCPSocket LinkedList

Nothing

any value is an instance of **Any**

what is an instance of **Nothing**?



lattices

joins and meets

consequences of the bottom type?

consequences of the top type?

ascription and casting

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \text{ as } \tau : \tau}$$

ascription and casting

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \text{ as } \tau : \tau}$$

completely useless, right?

ascription and casting

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e \text{ as } \tau : \tau}$$

ascription and casting

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e \text{ as } \tau : \tau}$$

how do we make this safe at runtime?

functions

$$\frac{\sigma_1 <: \tau_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2}$$

functions

functions

$$\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

functions are ***covariant*** in the ***codomain***
and ***contravariant*** in the ***domain***!

references

$$\frac{\sigma <: \tau}{\sigma \ \mathsf{ref} <: \tau \ \mathsf{ref}}$$

references

references

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma \text{ ref} <: \tau \text{ ref}}$$

references

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\sigma\ \texttt{ref} <: \tau\ \texttt{ref}}$$

in effect, the types have to be ***equal***,
making references ***invariant***!
(modulo field reordering)

sources and sinks

$$\frac{\Gamma \vdash e : \tau\ \mathtt{src}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \mathtt{sink} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \coloneqq e_2 : \mathtt{unit}}$$

also called “capabilities”

sources and sinks

$$\frac{\sigma <: \tau}{\sigma\ \mathsf{src} <: \tau\ \mathsf{src}} \qquad \frac{\tau <: \sigma}{\sigma\ \mathsf{sink} <: \tau\ \mathsf{sink}}$$

$$\frac{}{\tau\ \mathsf{ref} <: \tau\ \mathsf{src}} \qquad \frac{}{\tau\ \mathsf{ref} <: \tau\ \mathsf{sink}}$$

arrays

generalized references, so invariant?

arrays

$$\frac{\sigma <: \tau}{\sigma[] <: \tau[]}$$

what problems does this cause?

Java and Scala's Type Systems are Unsound *

The E

Nada Amin
EPFL, Switzerland
nada.amin@epfl.ch

Abstract

We present short programs that demonstrat
ness of Java and Scala's current type syst
ular, these programs provide parametricall
functions that can turn any type into any
(down)casting. Fortunately, parametric poly
not integrated into the Java Virtual Machine
examples do not demonstrate any unsoundn
Nonetheless, we discuss broader implicatior
ings on the field of programming languages.

Categories and Subject Descriptors D.3.2
Languages]: Language Classifications—
languages; D.3.3 [*Programming Langua*
Constructs and Features—Polymorphism

General Terms Design, Languages, Reliability, Security

Keywords Unsoundness, Java, Scala, Null, Existential

dence and productivity.

On that note, another lesson from this experience is that one well understood feature should be refined. In particular, the design of constrained generics is itself overconstrained.

Java Generics are Turing Complete

Radu Grigore
University of Kent, United Kingdom

Abstract

This paper describes a reduction from the halting problem of Turing machines to subtype checking in Java. It follows that subtype checking in Java is undecidable, which answers a question posed by Kennedy and Pierce in 2007. It also follows that Java's type checker can recognize any recursive language, which improves a result of Gil and Levy from 2016. The latter point is illustrated by a parser generator for fluent interfaces.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords Java, subtype checking, decidability, fluent interface, parser generator, Turing machine

Theorem 1. *It is undecidable whether $t <: t'$ according to a given class table.*

Theorem 2. *Given is a context free grammar G that describes a language $\mathcal{L} \subseteq \Sigma^*$ over an alphabet Σ of method names. We can construct Java class definitions, a type T, and expressions Start, Stop such that the code*

$$T\ \ell \quad = \quad Start.f^{(1)}().f^{(2)}()\ldots f^{(m)}().Stop$$

type checks if and only if $f^{(1)} f^{(2)} \ldots f^{(m)} \in \mathcal{L}$. Moreover, the class definitions have size polynomial in the size of G, and the Java code can be type-checked in time polynomial in the size of G.

Theorem 1 is proved by a reduction from the halting problem of Turing machines to subtype checking in Java (Section 5). The

why is Java so complex?

let’s return to System **F**…

$$\forall t . \tau$$

$$\forall t <: T.\tau$$

bounded quantification

```
List<?>    // just 'a list
```

```java
List<? extends Animal>
```

```
List<? extends Any>  // same as List<?>
```

```java
List<? super Cat>
```

```
List<? super Nothing>
```

```java
List<? extends Cat & Dog>
```

intersection type

```
List<? extends Animal & Comparable<?>>
```

F-bounded polymorphism

System $F_{<:}$

polymorphism + subtyping

universals: “kernel”

$$\frac{\Gamma, t <: u \vdash \sigma <: \tau}{\Gamma \vdash \forall t <: u . \sigma <: \forall t <: u . \tau}$$

works only if u does not change

universals: “full”

$$\frac{\Gamma \vdash \tau_1 <: \sigma_1 \quad \Gamma, t <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Gamma \vdash \forall t <: \sigma_1 . \sigma_2 <: \forall t <: \tau_1 . \tau_2}$$

contravariant domain, like functions

full $F_{<:}$ seems easily more powerful…

why have kernel version at all?

theorem (Pierce, 1993).

typechecking for full System $F_{<:}$ is undecidable.

proof.

first define $\neg\sigma \triangleq \forall x <: \sigma . x$

now attempt to check:

$$\forall x_0 . \forall x_1 . \forall x_2 . \neg(\forall y_0 <: x_0 . \forall y_1 <: x_1 . \forall y_2 <: x_2 . \neg x_0)$$

$$<: \forall x_0 . \forall x_1 <: P . \forall x_2 <: Q . \neg(\forall y_0 . \forall y_1 . \forall y_2 . \neg(\forall z_0 <: y_0 . \forall z_1 <: y_2 . \forall z_2 : y_1 . U))$$

proof.

first define $\neg\sigma \triangleq \forall x <: \sigma.x$

claim.

the subtype checking terminates if and only if a counter machine with registers P and Q terminates on input U.

now attempt to check:

$$\forall x_0.\forall x_1.\forall x_2.\neg(\forall y_0 <: x_0.\forall y_1 <: x_1.\forall y_2 <: x_2.\neg x_0)$$

$$<: \forall x_0.\forall x_1 <: P.\forall x_2 <: Q.$$

$$\neg(\forall y_0.\forall y_1.\forall y_2.\neg(\forall z_0 <: y_0.\forall z_1 <: y_2.\forall z_2 : y_1.U))$$

theorem (Pierce, 1994).

for each two-counter machine ***M*** there exists a typing judgment ***S(M)*** such that ***S(M)*** is derivable iff ***M*** halts.

```java
List<? extends Integer & GreaterThan<One>>
```

dependent types

further reading

ch. 24, 25 of *PFPL* (by Harper)

ch. 15, 16, 26, 28 of *TaPL* (by Pierce)