From OO, to OO

Subtyping as a Cross-cutting Language Feature

Paul Zhu

School of Software,
Tsinghua University

May 10, 2019

Today

1. Extending $\lambda_{\rightarrow}$ ($\lambda_{<:}$)
   - Records
   - Bottom Types
2. Bounded Quantification ($F_{<:}$)
   - Kernel and Full System
   - Properties
3. Variances
4. Intersection & Union Types

Contents

Motivating Example

Declare two classes, where `A` inherits `B`:

```
class B
class A extends B
```

A method whose argument has type `B`

```
def foo(x: B): Int = ???
```

could also be called with an expression of type `A`:

```
val a = new A
foo(a)
```

Remarks:

- "`A extends B`" means what it says: all members of `B` are also members of `A`.
- Invoking `foo(a)` is safe because `a` has all members of `B`.

Opinions

- Many people think that inheritance is the core of OO. But it shouldn't be, as Grady Booch once said, "Inheritance is highly overrated".
- On the other side, subtyping is a key feature of inheritance.

The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.

– Mark Mannasse

Subtype Relation

A subtyping is a pre-order relation, i.e., a binary relation that is reflexive and transitive. Formally, we write $S <: T$ to pronounce "S is a subtype of T", or "T is a supertype of S".

$$\text{S-Refl}\ \frac{}{S <: S}$$

$$\text{S-Trans}\ \frac{S <: U \qquad U <: T}{S <: T}$$

$\lambda_{\to}$ with Subtyping ($\lambda_{<:}$)

$$\text{Term } t ::= x \mid (t_1 t_2) \mid (\lambda x : T.t)$$

$$\text{Type } T ::= B \mid \top \mid T_1 \to T_2$$

- Evaluation rules are same with $\lambda_{\to}$.
- Subtyping rules: S-Refl, S-Trans and

$$\text{S-Top}\ \frac{}{S <: \top} \qquad \text{S-Arrow}\ \frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

- Typing rules:

$$\text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\ \frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash (\lambda x : T.t) : T \to T'}$$

$$\text{T-App}\ \frac{\Gamma \vdash t_1 : T \to T' \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1 t_2) : T'} \qquad \text{T-Sub}\ \frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T}$$

Contents

Record Extension with Subtyping ($\lambda^{\{\}}_{<:}$)

$$\text{Term } t ::= \cdots \mid \{l_1 = t_1, \ldots, l_n = t_n\} \mid t.l_i$$

$$\text{Type } T ::= \cdots \mid \{l_1 : T_1, \ldots, l_n : T_n\}$$

New subtyping rules:

$$\text{S-RecWidth}\ \frac{}{\{l_1 : T_1, \ldots, l_n : T_n, l_{n+1} : T_{n+1}, \ldots, l_m : T_m\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\text{S-RecDepth}\ \frac{\forall i. S_i <: T_i}{\{l_1 : S_1, \ldots, l_n : S_n\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

$$\text{S-RecPerm}\ \frac{\{k_1 : S_1, \ldots, k_n : S_n\} \text{ is a permutation of } \{l_1 : T_1, \ldots, l_n : T_n\}}{\{k_1 : S_1, \ldots, k_n : S_n\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

Type Checking in $\lambda_{<:}^{\{\}}$

$$
\begin{aligned}
subtype(S, T) = \;& \text{if } T = \top \text{ then true} \\
& \text{else if } S = S_1 \to S_2 \text{ and } T = T_1 \to T_2 \\
& \quad \text{then } subtype(T_1, S_1) \wedge subtype(S_2, T_2) \\
& \text{else if } S = \{k_1 : S_1, \ldots, k_m : S_m\} \text{ and } T = \{l_1 : T_1, \ldots, l_n : T_n\} \\
& \quad \text{then } dom(T) \subseteq dom(S) \wedge \forall i. \exists 1 \leqslant j \leqslant m. (k_i = l_j \wedge subtype(S_i, T_j)) \\
& \text{else false}
\end{aligned}
$$

Theorem

(Soundness and completeness) $S <: T \iff subtype(S, T) = true$.
(Termination) The algorithm subtype terminates on all inputs.

Algorithmic Subtyping

The above type checking algorithm could also be described as an algorithmic subtyping relation, of the form $\triangleright S <: T$, pronounced "S is algorithmically a subtype of T":

$$\text{SA-Top}\ \frac{}{\triangleright S <: \top} \qquad \text{SA-Arrow}\ \frac{\triangleright T_1 <: S_1 \qquad \triangleright S_2 <: T_2}{\triangleright S_1 \to S_2 <: T_1 \to T_2}$$

$$\text{SA-Rec}\ \frac{dom(T) \subseteq dom(S) \qquad \triangleright S_i <: T_j \text{ for every } k_i = l_j}{\triangleright(S \triangleq \{k_1 : S_1, \ldots, k_m : S_m\}) <: (\{l_1 : T_1, \ldots, l_n : T_n\} \triangleq T)}$$

Theorem

(Soundness and completeness) $S <: T \iff \triangleright S <: T$.
(Termination) The algorithmic subtyping derivation terminates on all inputs.

Properties of $\lambda_{<:}^{\{\}}$

Lemma

(Inversion lemma)
(1) If $S <: T_1 \to T_2$*, then* S *has the form* $S_1 \to S_2$*, with* $T_1 <: S_1$ *and* $S_2 <: T_2$*.*
(2) If $S <: \{l_1 : T_1, \ldots, l_n : T_n\} \triangleq T$*, then* S *has the form* $\{k_1 : S_1, \ldots, k_m : S_m\}$*, such that* $dom(T) \subseteq dom(S)$ *and* $S_i <: T_j$ *for every* $k_i = l_j$*.*

Theorem

(Progress) For any term t *and type* T*, if* $\vdash t : T$*, then either* t *is a value, or* $t \Rightarrow t'$ *for some term* t'*.*

Theorem

(Preservation) If $\vdash t : T$ *and* $t \Rightarrow t'$*, then* $\vdash t' : T$*.*

Obviously, type uniqueness is violated.

Exercises

- How many different supertypes does $\{l_1 : \top, l_2 : \top\}$ have?
  $\{l_1 : \top\}$, $\{l_2 : \top\}$, $\{\}$, $\{l_1 : \top, l_2 : \top\}$, $\{l_2 : \top, l_1 : \top\}$, $\top$.
- Can you find an infinite descending chain in the subtype relation?
  $\{\} <: \{l_1 : \top\} <: \{l_1 : \top, l_2 : \top\} <: \cdots$
- What about an infinite ascending chain?
  $\{\} \to \top <: \{l_1 : \top\} \to \top <: \{l_1 : \top, l_2 : \top\} \to \top <: \cdots$
- Is there a type that is a subtype of every other type?
  No. By inversion lemma.
- Is there an arrow type that is a supertype of every other arrow type?
  No. If there were such an arrow type $T_1 \to T_2$, then T_1 would have to be a subtype of every other type, which we have just seen is impossible.

Contents

$\lambda^{\{\}}_{<:}$ with Bottom Types

$$\text{Type } T ::= \cdots \mid \bot$$

New subtyping rule:

$$\text{S-Bot}\ \frac{}{\bot <: T}$$

Remarks:

- There are no closed values of type $\bot$. Suppose there is one, say v, then $\vdash v : \top \to \top$, which is impossible.
- In practice, $\bot$ indicates no return value, or whatever value.

Top & Bottom Types in Scala

```scala
// top types
abstract class Any
class AnyRef extends Any // AnyRef = java.lang.Object
final class AnyVal extends Any

// bottom types
abstract final class Nothing extends Any
def ???: Nothing = throw new NotImplementedError
def methodToBeImplemented(x: Int): Int = ???
```

Contents

Contents

System F with Subtyping ($F_{<:}^{\mathrm{kernel}}$) I

$$\text{Term } t ::= x \mid (t_1 t_2) \mid (\lambda x : T.t) \mid t\ [T] \mid \lambda X <: T.t$$
$$\text{Type } T ::= B \mid X \mid \top \mid T_1 \to T_2 \mid \forall X <: U.T$$
$$\text{Context } \Gamma ::= \varnothing \mid \Gamma, x : T \mid \Gamma, X <: T$$

- Subtyping rules now have form $\Gamma \vdash S <: T$

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

$$\text{S-Top}\ \frac{}{\Gamma \vdash S <: \top} \qquad \text{S-TVar}\ \frac{X <: T \in \Gamma}{\Gamma \vdash X <: T}$$

$$\text{S-Arrow}\ \frac{\Gamma \vdash T_1 <: S_1 \quad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \qquad \text{S-}\forall\ \frac{\Gamma, X <: U \vdash S <: T}{\Gamma \vdash (\forall X <: U.S) <: (\forall X <: U.T)}$$

System F with Subtyping ($F_{<:}^{\text{kernel}}$) II

- Typing rules:

$$\text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\ \frac{\Gamma, x : T \vdash t : T'}{\Gamma \vdash (\lambda x : T.t) : T \to T'}$$

$$\text{T-App}\ \frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash (t_1\ t_2) : T'} \qquad \text{T-TAbs}\ \frac{\Gamma, X <: U \vdash t : T}{\Gamma \vdash \lambda X <: U.t : \forall X <: U.T}$$

$$\text{T-TApp}\ \frac{\Gamma \vdash t : \forall X <: U.T' \qquad \Gamma\, T <: U}{\Gamma \vdash t\ [T] : T'[X := T]} \qquad \text{T-Sub}\ \frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T}$$

Scoping

The scoping of type variables is yet not obvious from the typing rules above. Consider the following typing contexts (given $\mathtt{Nat}$ and $\mathtt{Bool}$ are base types):

$$\Gamma_1 \triangleq X <: \top, y : X \to \mathtt{Nat}$$

$$\Gamma_2 \triangleq y : X \to \mathtt{Nat}, X <: \top$$

$$\Gamma_3 \triangleq X <: \{a : \mathtt{Nat}, b : X\}$$

$$\Gamma_4 \triangleq X <: \{a : \mathtt{Nat}, b : Y\}, Y <: \{c : \mathtt{Bool}, d : X\}$$

Which should be considered to be well-scoped?

- Γ_1 is well-scoped.
- Γ_2 and Γ_4 are ill-scoped.
- Γ_3 could be well-scoped using *F-bounded quantification*.

Bounded & Unbounded

- In $F_{<:}^{\text{kernel}}$, the unbounded universal type $\forall X.T$ we have seen in F has disappeared.
- The reason is that we do not need it, as "unbounded" is interpreted as "bounded by $\top$". Thus, we specify an abbreviation:

$$\forall X.T \triangleq \forall X <: \top.T$$

Full System ($F_{<:}^{\text{full}}$)

In S-$\forall$ rule of $F_{<:}^{\text{kernel}}$, the bounds of the two quantifiers being compared must be identical. If we think of a quantifier as a sort of arrow type (whose elements are functions from types to terms), we could extend this rule to allow the bounds to be distinct:

$$\text{S-}\forall\ \frac{\Gamma \vdash U_2 <: U_1 \qquad \Gamma, X <: U_2 \vdash S <: T}{\Gamma \vdash (\forall X <: U_1.S) <: (\forall X <: U_2.T)}$$

Recall that

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

Type Bounds in Scala

In Scala, type parameters can be restricted to be a subtype/supertype of some type.

```
// Implicit conversions from a Java Array into collection.Mutable.ArrayOps
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): ArrayOps[T] =
  new ArrayOps.ofRef[T](xs)

// getOrElse method in Option
sealed abstract class Option[+A] extends Product with Serializable {
  @inline final def getOrElse[B >: A](default: => B): B = { /* ... */ }
}
```

Contents

Properties of $F_{<:}$

In both the kernel and full system, the following two properties hold:

Theorem

(Progress) For any term t and type T, if $\vdash t : T$, then either t is a value, or $t \Rightarrow t'$ for some term t'.

Theorem

(Preservation) If $\vdash t : T$ and $t \Rightarrow t'$, then $\vdash t' : T$.

Type Checking in $F_{<:}^{\text{kernel}}$

This time, the algorithmic subtyping relation needs a typing context, i.e., of the form $\Gamma \triangleright S <: T$:

$$\text{S-Refl}\ \frac{}{\Gamma \triangleright S <: S} \qquad \text{SA-Top}\ \frac{}{\Gamma \triangleright S <: \top}$$

$$\text{SA-Trans}\ \frac{S <: U \in \Gamma \qquad \Gamma \triangleright U <: T}{\Gamma \triangleright S <: T} \qquad \text{SA-Arrow}\ \frac{\Gamma \triangleright T_1 <: S_1 \qquad \Gamma \triangleright S_2 <: T_2}{\Gamma \triangleright S_1 \to S_2 <: T_1 \to T_2}$$

$$\text{SA-}\forall\ \frac{\Gamma, X <: U \triangleright S <: T}{\Gamma \triangleright (\forall X <: U.S) <: (\forall X <: U.T)}$$

Theorem

(Soundness and completeness) $\Gamma \vdash S <: T \iff \Gamma \triangleright S <: T$.

(Termination) The algorithmic subtyping derivation terminates on all inputs.

Type Checking in $F_{<:}^{\text{full}}$

We only need to update the SA-$\forall$ rule:

$$\text{SA-}\forall\ \frac{\Gamma \triangleright U_2 <: U_1 \qquad \Gamma, X <: U_2 \triangleright S <: T}{\Gamma \triangleright (\forall X <: U_1.S) <: (\forall X <: U_2.T)}$$

Theorem

(Soundness and completeness) $\Gamma \vdash S <: T \iff \Gamma \triangleright S <: T$.

Type Checking in $F_{<:}^{\text{full}}$ is Diverging

An example (Ghelli, 1995):

- Let $\neg S \triangleq \forall X <: S.X$.
- By S-$\forall$, we see that $\Gamma \vdash \neg S <: \neg T \iff \Gamma \vdash T <: S$.
- Let $T \triangleq \forall X.\neg(\forall Y <: X.\neg Y)$.
- To show

  $$X_0 <: T \triangleright X_0 <: \forall X_1 <: X_0.\neg X_1,$$

  we end up in an infinite regress of larger and larger subgoals, say it reduces to

  $$X_0 <: T, X_1 <: X_0 \triangleright X_0 <: \forall X_2 <: X_0.\neg X_2,$$

  which makes the derivation diverge.

Type Checking in $F_{<:}^{\text{full}}$ is Undecidable

Worse yet, it can be shown that there is no subtyping algorithm that is sound and complete and that terminates on all inputs:

Theorem

(Pierce, 1994) For every two-counter machine, there exists a subtyping statement such that it is derivable in $F_{<:}^{full}$ iff the execution of the two-counter machine halts.

Contents

Motivating Questions

- Suppose we know that $S <: T$, should $[S] <: [T]$ (where $[\cdot]$ indicates a list type)?
- Is the answer applicable to other situations of generic types?

Three Kinds of Variances

A type constructor is

- covariant if it preserves the ordering of types ($S <: T$ impiles $X[S] <: X[T]$);
- contravariant if it reverses this ordering ($S <: T$ impiles $X[T] <: X[S]$);
- invariant if neither of the above applies.

In Scala and Java:

Variances	Scala	Java
Covariant	`+T`	`? extends T`
Contravariant	`-T`	`? super T`
Invariant	`T`	`T`

Functions Under the Hood

Recall that

$$\text{S-Arrow}\ \frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

In scala, a function is implemented as a trait:

```scala
trait Function1[-T1, +R] extends AnyRef {
  def apply(v1: T1): R
}

class B
class A extends B

val foo: B => Int = new Function1[B, Int] {
  def apply(x: B) = ???
}
val a = new A
foo(a)
```

Comparer is Contravariant

Suppose we have a comparer for comparing two objects of some class `A`. Then we should also have this comparer for all the subclasses of `A`. In C#, this is achieved by declaring the `IComparer` interface as a contravariant.

```
public interface IComparer<in T> {
  int Compare(T left, T right);
}
```

Sink for Contravariant

In real-world event-driven software systems, when an event is fired, all its "super levels" should also be notified. To achieve this, we can declare the trait `Sink` as a contravariant. [1]

```
trait Event
trait UserEvent extends Event
trait SystemEvent extends Event
trait ApplicationEvent extends SystemEvent
trait ErrorEvent extends ApplicationEvent

trait Sink[-In] { def notify(o: In) }

def appEventFired(e: ApplicationEvent, s: Sink[ApplicationEvent]) = s.notify(e)
def errorEventFired(e: ErrorEvent, s: Sink[ErrorEvent]) = s.notify(e)
```

[1] http://blog.petruescu.com/programming/types/scala-types-contravariance/

Sink is Contravariant

```scala
trait SystemEventSink extends Sink[SystemEvent]
val ses = new SystemEventSink {
  override def notify(o: SystemEvent): Unit = ???
}

trait GenericEventSink extends Sink[Event]
val ges = new GenericEventSink {
  override def notify(o: Event): Unit = ???
}

// You can call:
appEventFired(new ApplicationEvent {}, ses)
errorEventFired(new ErrorEvent {}, ges)
appEventFired(new ApplicationEvent {}, ges)
```

Contents

Motivation

- A type can be interpreted as a set, which contains all closed values of that type.
- Since we have intersection and union operations on sets, can we add them to types?

Intersection Types

The inhabitants of an intersection type $T_1 \cap T_2$ are terms belonging to both T_1 and T_2, formulated by the following subtyping rules:

$$\text{S-}\cap\text{Proj1}\ \frac{}{T_1 \cap T_2 <: T_1} \qquad \text{S-}\cap\text{Proj2}\ \frac{}{T_1 \cap T_2 <: T_2}$$

$$\text{S-}\cap\text{Form}\ \frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \cap T_2}$$

Remark: in words of lattice theory, $\cap$ is a *meet* operator.

Intersection Types in Dotty[2]

```scala
trait Resettable { def reset(): this.type }
trait Growable[T] { def add(x: T): this.type }
def f(x: Resettable & Growable[String]) = {
  x.reset()
  x.add("first")
}

trait A { def children: List[A] }
trait B { def children: List[B] }
class C extends A with B {
  def children: List[A & B] = ???
}
val x: A & B = new C
val ys: List[A & B] = x.children
// Since List is covariant, List[A] & List[B] = List[A & B]
```

[2] http://dotty.epfl.ch/

Union Types

The inhabitants of a union type $T_1 \cup T_2$ are terms belonging to either T_1 or T_2, formulated by the following subtyping rules:

$$\text{S-}\cap\text{Form1}\ \frac{}{T_1 <: T_1 \cup T_2} \qquad \text{S-}\cap\text{Form2}\ \frac{}{T_2 <: T_1 \cup T_2}$$

$$\text{S-}\cap\text{Proj}\ \frac{T_1 <: S \qquad T_2 <: S}{T_1 \cup T_2 <: S}$$

Remark: in words of lattice theory, $\cup$ is a *join* operator.

Union Types in Dotty

```scala
case class UserName(name: String) {
  def lookup(admin: Admin): UserData
}

case class UID(id: Id) {
  def lookup(admin: Admin): UserData
}

def login(input: UserName | UID) = {
  val user = input match {
    case UserName(name) => name.lookup(admin)
    case UID(id) => id.lookup(admin)
  }
  // ...
}
```

$\lambda_{<:}^{\{\}}$ with If-expressions

$$\text{Type } T ::= \cdots \mid \texttt{Bool}$$

$$\text{Term } t ::= \cdots \mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid \text{true} \mid \text{false}$$

New typing rules:

$$\text{T-True}\ \frac{}{\Gamma \vdash \text{true} : \texttt{Bool}} \qquad \text{T-False}\ \frac{}{\Gamma \vdash \text{false} : \texttt{Bool}}$$

$$\text{T-If}\ \frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T_2 \sqcup T_3 = T}{\Gamma \vdash (\text{if } t_1 \text{ then } t_2 \text{ else } t_3) : T}$$

where $T_2 \sqcup T_3$ is a join of T_2 and T_3.

Joins & Meets I

- A type J is called a join of a pair of types S and T, written $J = S \sqcup T$, if $S <: J$, $T <: J$, and for all types U, if $S <: U$ and $T <: U$, then $J <: U$.
- A type M is called a meet of a pair of types S and T, written $M = S \sqcap T$, if $M <: S$, $M <: T$, and for all types L, if $L <: S$ and $L <: T$, then $L <: M$.

Theorem

(1) For every pair of types S and T, there is some type J such that $J = S \sqcup T$.
(2) For every pair of types S and T with a common subtype, there is some type M such that $M = S \sqcap T$.

Joins & Meets II

Proof.

(1) By case analysis on the shapes of S and T:

- Case $S = T = \texttt{Bool}$: $J = \texttt{Bool}$.
- Case $S = S_1 \to S_2$ and $T = T_1 \to T_2$: $J = (S_1 \sqcap T_1) \to (S_2 \sqcup T_2)$.
- Case $S = \{k_1 : S_1, \ldots, k_m : S_m\}$ and $T = \{l_1 : T_1, \ldots, l_n : T_n\}$: $J = \{j_1 : U_1, \ldots, j_q : U_q\}$ where $dom(J) = dom(S) \cap dom(T)$, and $U_i = S_s \sqcup T_t$ for every $j_i = k_s = l_t$.
- Otherwise, $J = \top$.

$\square$

Joins & Meets III

Proof.

(2) We first propose an algorithm for computing M:

- Case $S = \top$: $M = T$.
- Case $T = \top$: $M = S$.
- Case $S = T = \texttt{Bool}$: $M = \texttt{Bool}$.
- Case $S = S_1 \to S_2$ and $T = T_1 \to T_2$: $J = (S_1 \sqcup T_1) \to (S_2 \sqcap T_2)$.
- Case $S = \{k_1 : S_1, \ldots, k_m : S_m\}$ and $T = \{l_1 : T_1, \ldots, l_n : T_n\}$: $U = \{j_1 : U_1, \ldots, j_q : U_q\}$ where $dom(J) = dom(S) \cup dom(T)$, and (i) $U_i = S_s \sqcap T_t$ for every $j_i = k_s = l_t$; (ii) $U_i = S_s$ if $j_i = k_s$ occurs only in S; (iii) $U_i = T_t$ if $j_i = l_t$ occurs only in T.
- Otherwise, fails.

Then, we can show the above algorithm never fails if S and T have a common subtype, by the lemma shown in the next page. □

Joins & Meets IV

Lemma

If $L <: S$ and $L <: T$, then $M = S \sqcap T$ for some M.

Proof.

By induction on S, with a case analysis on the shapes of S and T. Suppose either is $\top$, the case is trivial. Otherwise, they have the same shape, or else L has inconsistent shapes by inversion lemma.

- If both are `Bool`, the case is trivial.
- If both are arrow types, say $S = S_1 \to S_2$ and $T = T_1 \to T_2$, we only need to check if $S_2 \sqcap T_2$ exists. By inversion lemma, we have $L = L_1 \to L_2$ with $L_2 <: S_2$ and $L_2 <: T_2$. Thus, we are done by inductive hypothesis.
- Otherwise, both are record types. We only need to check if the meet operations invoked by the algorithm always succeed. By inversion lemma, L include all labels of S and T. Moreover, for every common label, the corresponding type in L must be a common subtype of those in S and T. Again, we are done by inductive hypothesis.

$\square$

OO is More Complicated Than You Thought

> *Early object design books, including Designing Object-Oriented Software, speak of finding objects by identifying things (noun phrases) written about in a design specification. In hindsight, this approach seems naive. Today, we don't advocate underlining nouns and simplistically modelling things in the real world. It's much more complicated than that.*
>
> – *Rebecca Wirfs-Brock*

Perhaps functional programming (better to be pure) is much easier than OOP!