Type Inference

```kotlin
fun getNullableInt(): Int? {
    return 42
}

fun smartCast() {
    var x = getNullableInt()
    if (x != null)
        x.inc()
}
```
fun getNullableInt(): Int? {
    return 42
}

fun smartCast() {
    var x = getNullableInt()
    if (x != null)
        x.inc()
}

✓
fun getNullableInt(): Int? {
    return 42
}

fun smartCast() {
    var x = getNullableInt()
    run { x = null }
    if (x != null)
        x.inc()
}
fun getNullableInt(): Int? {
    return 42
}

fun smartCast() {
    var x = getNullableInt()
    run { x = null }
    if (x != null)
        x.inc()  // Error here
}

Type Inference
Type Inference

1. val a: Any? = TODO()
2. val b = a
3. if (b is Int) {
4.   a.inc()
5. }

1/8(5)
Type Inference

```scala
1 val a: Any? = TODO()
2 val b = a
3 if (b is Int) {
4   a.inc()  ✓
5 }
```
Type Inference

- Outline the Type Inference of Kotlin in general
- Focus especially on Smart Casts in more detail
- Relate type inference in Kotlin to Hindely-Milner type system and others
- Related to Type Constraint System
Subtyping

```kotlin
fun <T> mk() : T = TODO()

class Foo<A, B : A?> {
    val b: B = mk()
    val bQ: B? = mk()
    val ab: A = b
    val abQ: A = bQ
    val aQb: A? = b
    val aQbQ: A? = bQ
}
```
fun <T> mk() : T = TODO()

class Foo<A, B : A?> {
    val b: B = mk()
    val bQ: B? = mk()
    val ab: A = b ☒
    val abQ: A = bQ
    val aQb: A? = b
    val aQbQ: A? = bQ
}
fun <T> mk() : T = TODO()

class Foo<A, B : A?> {
    val b: B = mk()
    val bQ: B? = mk()
    val ab: A = b
    val abQ: A = bQ
    val aQb: A? = b
    val aQbQ: A? = bQ
}
Subtyping

```scala
fun <T> mk() : T = TODO()

class Foo<A, B : A?> {
  val b: B = mk()
  val bQ: B? = mk()
  val ab: A = b  // Error
  val abQ: A = bQ  // Error
  val aQb: A? = b
  val aQbQ: A? = bQ
}
```
fun <T> mk() : T = TODO()

class Foo<A, B : A?> {
  val b: B = mk()
  val bQ: B? = mk()
  val ab: A = b  
  val abQ: A = bQ
  val aQb: A? = b  
  val aQbQ: A? = bQ
}
Subtyping

- Outline the *Subtyping* relation important for the type system
- Describe the relation to *Type Containment*
- Detail the function of *Type Decaying*, *Union-* and *Intersection-Types*
- Present the significance of *Integer Literal Types*
- Related to other type system related topics
Type Parameters

```typescript
interface Consumer<A>

interface Producer<A>

var numConsumer: Consumer<Number>? = TODO()
var intConsumer: Consumer<Int>? = numConsumer

var intProducer: Producer<Int>? = TODO()
var numProducer: Producer<Number>? = intProducer
```
Type Parameters

```java
interface Consumer<A>
interface Producer<A>

var numConsumer: Consumer<Number>? = TODO()  // X
var intConsumer: Consumer<Int>? = numConsumer  // X

var intProducer: Producer<Int>? = TODO()  // X
var numProducer: Producer<Number>? = intProducer  // X
```
Type Parameters

```java
interface Consumer<in A>
interface Producer<A>

var numConsumer: Consumer<Number>? = TODO()
var intConsumer: Consumer<Int>? = numConsumer ✓

var intProducer: Producer<Int>? = TODO()
var numProducer: Producer<Number>? = intProducer ✗
```
Type Parameters

```java
interface Consumer<in A>
interface Producer<out A>

var numConsumer: Consumer<Number>? = TODO()
var intConsumer: Consumer<Int>? = numConsumer ✓

var intProducer: Producer<Int>? = TODO()
var numProducer: Producer<Number>? = intProducer ✓
```
Type Parameters

- Outline the relation of \textit{in-/co-/contravariant types} to subtyping
- Describe \textit{Mixed-Site-Variance} in this context
- Detail \textit{Type Capturing}
- Related to \textit{Type Inference} and \textit{Subtyping}
fun main() {
    val x: Int
    var y: Int
    if (cond) {
        x = 40
        y = 4
    } else {
        x = 20
    }
    y = 5
    val z = x + y
}
fun main() {
    val x: Int
    var y: Int
    if (cond) {
        x = 40
        y = 4
    } else {
        x = 20
    }
    y = 5
    val z = x + y
}
fun main() {
    val x: Int
    var y: Int
    while (cond) {
        x = 40
        y = 4
    }
    val z = x + y
}
fun main() {
    val x: Int
    var y: Int
    while (cond) {
        x = 40
        y = 4
    }
    val z = x + y
}
fun main() {
    val x: Int
    var y: Int
    while (cond) {
        x = 40  // Error: x is already assigned
        y = 4
    }
    val z = x + y  // Error: y is in a var scope
}
Null Safety

- Outline the function of *Variable Initialisation Analysis*
- Explain the relation between *Nullable* and *Non-nullable Types*
- Describe approaches to handle nullable Types, e.g. *Elvis Operator*
- Related to *Type Inference*, esp. *Smart-Casts*
Contracts

```kotlin
fun Any?.isValidString(): Boolean {
    return this != null && this is String && this.length > 0
}

fun(getString() : String? {
    // Somehow get the string, which might be null.
}

fun testString() {
    val test = getString()
    if (test.isValidString()) {
        // Does not compile:
        // Type mismatch. Required: String. Found: String?.
        val result: String = test
    }
}
```

In the example:
- nullability checks and cast checks performed in `isValidString`
- however, this is not propagated to call site
In the example:

- nullability checks and cast checks performed in `isValidString`
- however, this is not propagated to call site

Contracts

- give additional information to e.g. function and lambda calls
- help with smart casts
- verification of contracts at declaration site?

Topic

- assess contract capabilities
- how about verification?
- details on how are they used?
- details on how are they implemented, esp. on JVM?
Coroutines

```kotlin
var c: Continuation<Unit>? = null

suspend fun suspendMe() = suspendCoroutine<Unit> { continuation ->
    println("Suspended")
    c = continuation
}

fun main() {
    val lambda: suspend () -> Unit = {
        suspendMe()
        println(1)
        suspendMe()
        println(2)
    }
    lambda()
}
```

In the example:
- first compiled to CPS
- then implemented via one-shot continuations
- Continuation is exposed via API
Coroutines

```kotlin
var c: Continuation<Unit>? = null

suspend fun suspendMe() = suspendCoroutine<Unit> { continuation ->
    println("Suspended")
    c = continuation
}

fun builder(c: suspend () -> Unit) {
    c.startCoroutine(object: Continuation<Unit> {
        override val context = EmptyCoroutineContext
        override fun resumeWith(result: Result<Unit>) {
            result.getOrThrow()
        }
    })
}

fun main() {
    val lambda: suspend () -> Unit = {
        suspendMe()
        println(1)
        suspendMe()
        println(2)
    }
    builder {
        lambda()
    }
    c?.resume(Unit)
    c?.resume(Unit)
}
```

In the example:
- first compiled to CPS
- then implemented via one-shot continuations
- Continuation is exposed via API

**Topic**
- assess capabilities of Coroutines
- shed light on the implementation concept
- highlight implementation consequences on JVM
Overload Resolution

```kotlin
interface Y {
    fun Y.foo() {} // `foo` is an extension for Y, // needs extension receiver to be called
    fun bar() {
        foo() // `this` reference is both // the extension and the dispatch receiver
    }
}

class X : Y {
    fun Y.foo() {}
}

fun main() {
    val x: X = mk()
    val y: Y = mk()
    with (x) {
        y.foo() // OK!
    }
}
```

In the example:

▶ E.g. Extension Methods complicate Overload Candidate Set
fun foo(a: Foo, b: Bar) {
    (a + b)(42)
    // Such a call is handled as if it is
    // (a + b).invoke(42)
}
fun f(arg: Int, arg2: String) {} // (1)
fun f(arg: Any?, arg2: CharSequence) {} // (2)
...
f(2, "Hello")

fun f1(arg: Int, arg2: String) {
    f2(arg, arg2) // VALID: can forward both arguments
}

fun f2(arg: Any?, arg2: CharSequence) {
    f1(arg, arg2) // INVALID: function f1 is not applicable
}

In the example:
- E.g. Extension Methods complicate *Overload Candidate Set*
- Operators / Infix Receivers
- Concept of *Most Specific Candidate* refined
Overload Resolution

In the example:

- E.g. Extension Methods complicate Overload Candidate Set
- Operators / Infix Receivers
- Concept of Most Specific Candidate refined
- Lambda return types also affect the MRO

Topic

- explore Kotlin extension Methods
- elaborate on Overload Candidate Set Determination
- elaborate how to pick the Most Specific Candidate
Type Constraints

For each call, we determine function applicability via one of the following constraint systems:

▶ For every non-lambda argument inferred to have type $T_i$, corresponding to the function parameter of type $U_j$, a constraint $T_i \leq U_j$ is constructed.

▶ All declaration-site type constraints for the function are also added to the constraint system.

▶ For every lambda argument with the number of lambda arguments known to be $K$, corresponding to the function parameter of type $U_m$, a special constraint of the form

$$\left( FT(L_1, \ldots, L_K) \rightarrow R \land FTR(FT(RT, L_1, \ldots, L_n) \rightarrow R) \leq U_m \right)$$

is added to the constraint system, where $R, RT, L_1, \ldots, L_K$ are fresh type variables.

▶ For each lambda argument with an unknown number of lambda arguments (that is, being equal to 0 or 1), corresponding to the function parameter of type $U_n$, a special constraint of the form

$$\left( FT() \rightarrow R \land FT(L) \rightarrow R \land FTR(FT(RT) \rightarrow R \land FTR(RT, L) \rightarrow R) \leq U_m \right)$$

is added to the constraint system, where $R, RT, L$ are fresh type variables.

In the example:

Type constraints stem from several applications:

▶ Applicability of a function during collection of Overload Candidate Sets.
Type Constraints

During MSC selection, for every two distinct members of the candidate set F1 and F2, the following constraint system is constructed and solved:

- For every non-default argument of the call and their corresponding declaration-site parameter types $X_1, \ldots, X_N$ of F1 and $Y_1, \ldots, Y_N$ of F2, a type constraint $X_K \leq Y_K$ is built unless both $X_K$ and $Y_K$ are built-in integer types. If both $X_K$ and $Y_K$ are built-in integer types, a type constraint $\text{Widen}(X_K) \leq \text{Widen}(Y_K)$ is built instead, where $\text{Widen}$ is the integer type widening operator. During construction of these constraints, all declaration-site type parameters $T_1, \ldots, T_M$ of F1 are considered bound to fresh type variables $T_1, \ldots, T_M$, and all type parameters of F2 are considered free;

- If F1 and F2 are extension callables, their extension receivers are also considered non-default arguments of the call, even if implicit, and the corresponding constraints are added to the constraint system as stated above. For non-extension callables, only declaration-site parameters are considered;

- All declaration-site type constraints of $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ are also added to the constraint system.

In the example:

Type constraints stem from several applications:

- Applicability of a function during collection of *Overload Candidate Sets*
- comparing two candidate function signatures during determination of *Most Specific Candidate*
Type Constraints

During MSC selection, for every two distinct members of the candidate set F1 and F2, the following constraint system is constructed and solved:

- For every non-default argument of the call and their corresponding declaration-site parameter types $X_1, \ldots, X_N$ of F1 and $Y_1, \ldots, Y_N$ of F2, a type constraint $X_K \leq Y_K$ is built unless both $X_K$ and $Y_K$ are built-in integer types. If both $X_K$ and $Y_K$ are built-in integer types, a type constraint $\text{Widen}(X_K) \leq \text{Widen}(Y_K)$ is built instead, where $\text{Widen}$ is the integer type widening operator. During construction of these constraints, all declaration-site type parameters $T_1, \ldots, T_M$ of F1 are considered bound to fresh type variables $\tilde{T}_1, \ldots, \tilde{T}_M$, and all type parameters of F2 are considered free;

- If F1 and F2 are extension callables, their extension receivers are also considered non-default arguments of the call, even if implicit, and the corresponding constraints are added to the constraint system as stated above. For non-extension callables, only declaration-site parameters are considered;

- All declaration-site type constraints of $X_1, \ldots, X_N$ and $Y_1, \ldots, Y_N$ are also added to the constraint system

In the example:

Type constraints stem from several applications:

- Applicability of a function during collection of Overload Candidate Sets
- comparing two candidate function signatures during determination of Most Specific Candidate

Topic

- give examples for noteworthy applications of constraint solving
- assess the expressivity of constraints, as well as solvable classes
- point out how constraint solving is implemented in Kotlin
Topic Selection

1. Type Inference
2. Subtyping
3. Type Parameters
4. Null Safety
5. Contracts
6. Coroutines
7. Overload Resolution
8. Type Constraints

Proceeding

In order to vote for your preferred topics:

1. Sort the topics in the order that you prefer them,
2. assign them numbers, with your most preferred topic #1 and the least preferred topic #8,
3. Send an email with your ranking to petter@in.tum.de until Tue April 11th 23:59