arrow-kt.io

Super-charge your build with

Arrow Analysis

Alejandro Serrano Mena – Tweag I/O

```
?.let { it + " is awesome" }
```

One of the most celebrated Kotlin features is nullability analysis

```
?.let { it + " is awesome" }
```

One of the most celebrated Kotlin features is nullability analysis

`?.let { it + " is awesome" }`

One of the most celebrated Kotlin features is nullability analysis

- Mark possibly-nullable values with `?`
- Safe access via Elvis operators `?.` and `?:`

?.let { it + " is awesome" }

One of the most celebrated Kotlin features is nullability analysis

- Mark possibly-nullable values with ?
- Safe access via Elvis operators ?. and ?:
- Powerful static data and control flow analysis

```
if (list ≠ null) {
  list.map { it + 1 }  // no ?. required
}
```

Why stop there?

Nullability analysis saves us from `NullReferenceException`

Wrong indexing leads to `IndexOutOfBoundsException`

```
list.get(2)  // what if fewer elements?
```

Bad initialization leads to `IllegalArgumentException`

```
Person(age = -1)  // not really an age
```

Arrow Analysis to the rescue!

Nullability analysis saves us from `NullReferenceException`

Wrong indexing leads to `IndexOutOfBoundsException`

```
list.get(2)   // what if fewer elements?
```

Bad initialization leads to `IllegalArgumentException`

```
Person(age = -1)   // not really an age
```

Flow-aware static analyzer for pre-, postconditions and invariants for Kotlin

Flow-aware static analyzer for pre-, postconditions and invariants for Kotlin

- Runs as part of your build, as a compiler plug-in

Flow-aware static analyzer for pre-, postconditions and invariants for Kotlin

- Runs as part of your build, as a compiler plug-in
- Checks information provided as annotations
  - As with nullable types and `require`

Flow-aware static analyzer for pre-, postconditions and invariants for Kotlin

- Runs as part of your build, as a compiler plug-in
- Checks information provided as annotations
  - As with nullable types and `require`
- Understands the order and branching of your code
  - As opposed to a simpler linter

🚦 Counting semaphore

Incrementing the semaphore should only happen for positive numbers

```
import arrow.analysis.pre

fun increment(x: Int): Int {
  pre(x > 0) { "value must be positive" }
  return x + 1
}
```

🚦 Counting semaphore

Incrementing the semaphore should only happen for positive numbers

```
val example = increment(-1)

e: pre-condition `value must be positive`
    is not satisfied in `increment(-1)`
    → unsatisfiable constraint: `(-1 > 0)`
```

□ Flow-awareness

Flow (`if`, `when`, other calls) is taken into account when deciding whether a condition is satisfied

```
val new = if (current > 1) {
  increment(current)  // fine
} else {
  log("weird...")
  0
}
```

↪️ After the call

The following code is rejected

```
increment(increment(1))
```

because `increment` makes no promises about its result

```
fun increment(x: Int): Int {
  pre(x > 0) { "value must be positive" }
  return x + 1
}
```

↪️ After the call

The following code is rejected

```
increment(increment(1))
```

what can we promise?

because `increment` makes no promises about its result

```
fun increment(x: Int): Int {
  pre(x > 0) { "value must be positive" }
  return x + 1
}
```

↪️ Post-conditions

The following code is accepted

```
increment(increment(1))
```

because `increment` makes promises about its result

```
fun increment(x: Int): Int {
  pre(x > 0) { "value must be positive" }
  return (x + 1)
    .post({ it > 0 }) { "result is positive" }
}
```

Post-conditions are checked

```
fun increment(x: Int): Int {
  val new = if (x < 0) {
    0   // fails to satisfy the post-condition
  } else {
    x + 1
  }
  return new.post({ it > 0 }) { "positive" }
}
```

Post-conditions are checked

```
fun increment(x: Int): Int {
  val new = if (x < 0) {
    0   // fails to satisfy the post-condition
  } else {
    x + 1
  }
  return new.post({ it > 0 }) { "positive" }
}
```

🔮 The magic

How does Arrow Analysis know that this holds?

```
fun increment(x: Int): Int {
  pre(x > 0) { "value must be positive" }
  return (x + 1)
    .post({ it > 1 }) { "result is positive" }
}
```

~~The magic~~ The reasoning engine

How does Arrow Analysis know that this holds?

```
fun increment(x: Int): Int {
  pre(x > 0) { "value must be positive" }
  return (x + 1)
    .post({ it > 1 }) { "result is positive" }
}
```

$$x > 0 \;\&\&\; result = x + 1 \implies result > 1$$

?

SMT solver

🤖 SMT solvers

Satisfiability Modulo Theories

Specialized software for automatic reasoning

Very fast but limited to a few theories:

#️⃣numbers 🧮 bitvectors regular expressions ...

Several industrial-grade solvers

We interface with them using `java-smt`

📋 Fields

We can refer to fields or properties of values

```
fun List<Double>.average() {
  pre(this.isNotEmpty()) { "list not empty" }
  return this.sum() / this.size
}
```

📋 Fields

We can refer to fields or properties of values

```
fun List<Double>.average() {
    pre(this.isNotEmpty()) { "list not empty" }
    return this.sum() / this.size
}
```

Otherwise `ArithmeticException` may be thrown

0️⃣ division by zero

📋 Fields

We can refer to fields or properties of values

```
fun List<Double>.average() {
  pre(this.isNotEmpty()) { "list not empty" }
  return this.sum() / this.size
}
```

Arrow Analysis knows about the relationship

```
this.isNotEmpty() ⟺ this.size > 0
```

📋 Fields

We can refer to fields or properties of values

```
fun List<Dou                e() {
  pre(this.              { "list not empty" }
  return th              is.size
}
```

HOW?

Arrow Analysis knows about the relationship

```
this.isNotEmpty() ⟺ this.size > 0
```

⚖️ Laws

Encode pre- and post-conditions separate from implementation

```
@Law
inline fun <E> List<E>.getLaw(index: Int): E {
  pre(index ≥ 0 && index < size) { "within bounds" }
  return get(index)
}
```

Inspired by TypeScript declaration files,
but using JVM-specific features, like annotations

⚖️ Laws

Encode pre- and post-conditions separate from implementation

```
@Law
inline fun <E> List<E>.getLaw(index: Int): E {
  pre(index ≥ 0 && index < size) { "within bounds" }
  return get(index)
}
```

Currently annotated: standard library

Next in our radar: (much larger, help more than welcome)

Invariants

Conditions which apply to a whole type

```
data class Positive(val value: Int) {
  init { require(value > 0) }
}
```

Useful to avoid repeating conditions again and again

```
data class Person(val age: Positive, ...)
```

Invariants

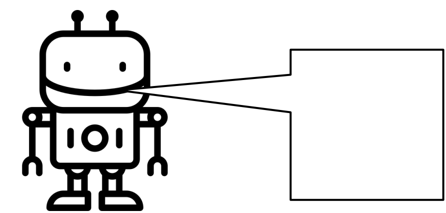
Invariants are assumed when using that type

```
data class Positive(val value: Int) {
  init { require(value > 0) }

  operator fun plus(other: Positive) =
    Positive(this.value + other.value)
}
```

```
this.value > 0 && other.value > 0
  ⟹ this.value + other.value > 0
```

?



Invariants in value classes

More compile checks with the same runtime performance!

```
@JvmInline
value class Positive(val value: Int) {
  init { require(value > 0) }
  ...
}
```

Defining your domain more strictly is A Good Thing™

🐘 Practical matters

How do I add Arrow Analysis to my project?

Our Gradle plug-in adds it to your build
Instructions available at `arrow-kt.io/docs/analysis`

How much does it add to my compile times?

Our (very) preliminary results say that around 3x

Does it integrate with IntelliJ?

Tighter integration

Does it integrate with IntelliJ?

```
arrow-analysis-mpp-template [build]: faile 20 sec, 944 ms
  :compileKotlinJvm 2 errors 3 sec, 687 ms
    example.kt src/commonMain/kotlin 2 errors
      pre-condition `index within bounds` is not satisfied
      pre-condition `index within bounds` is not satisfied
```

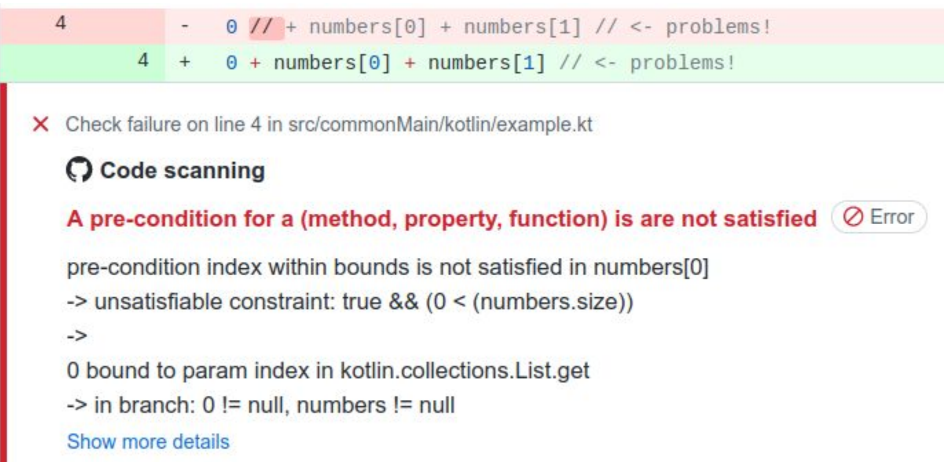
```
e: /home/serras/arrow/arrow-analysis-mpp-template/src/commonMa
  → unsatisfiable constraint: `true && (0 < (numbers.size))`
  →
`0` bound to param `index` in `kotlin.collections.List.get`
  → in branch: 0 ≠ null, numbers ≠ null
```

The new Kotlin Frontend IR promises better integration

Our plan is to migrate once the API stabilizes

Tighter integration

Does it integrate with GitHub?



CORE

Functional companion to Kotlin's Standard Library

FX

Functional Effects Framework companion to KotlinX Coroutines

OPTICS

Deep access and transformations over immutable data

ANALYSIS

Pre-, post-condition, and invariant checks for your Kotlin code

arrow-kt.io