Bachelor’s thesis

Enhancing Variadic Functions in Kotlin

Oleg Makeev
Constructor University

Supervisor: Prof. Dr. Anton Podkopaev
Industry Advisor: Daniil Berezun, JetBrains Research

Functions with variadic arguments

Kotlin provides a special vararg keyword for creating functions with variadic number of arguments, which helps to avoid creating additional collections the call site:

```
fun printVararg(vararg x: Int) {
    x.forEach {
        print(it)
    }
}

printVararg(1, 2, 3)
// prints 123
```

```
fun printList(x: List<Int>) {
    x.forEach {
        print(it)
    }
}

printList(listOf(1, 2, 3))
// prints 123
```

Varargs are arrays under the hood

Vararg parameters are replaced with arrays that store all the passed arguments:

```
fun foo(vararg x: Int){
  ... // foo(x: IntArray)
}

fun <T>bar(vararg x: T){
  ... // bar(x: Array<out T>)
}
```

Varargs with primitive types use the corresponding array type:

`IntArray`
`BooleanArray`
`ByteArray`
`CharArray`
`DoubleArray`
`FloatArray`
`LongArray`
`ShortArray`

For any other type boxed arrays are used:

`Array<out T>`

Spread operator

Spread operator (*) is used for unpacking elements of a collection to the vararg parameter. The type of the passed array must match the type of the vararg.

```
fun printAll(vararg x: Int) {
    x.forEach {
        print(it)
    }
}

val x = intArrayOf(1, 2, 3)
printAll(*x) // prints 123
printAll(0, *x, *intArrayOf(4, 5, 6)) // prints 0123456
```

Spread operator only works on Array types

Casts to arrays perform an additional copy

```
fun <T>printAll(vararg x: T) {...}

fun printAllInt(vararg x: Int) {...}

val x = listOf(1, 2, 3)
printAll(*x.toTypedArray()) // Double copy
printAllInt(*x.toIntArray()) // Double copy
```

Overriding generic vararg methods

Vararg methods with template parameter cannot be overridden with any primitive type

```
interface A<T> {
    fun foo(vararg x : T) // foo(x: Array<out T>)
}

class B : A<Int> {
    override fun foo(vararg x : Int) { } // foo(x: IntArray)
    /\ /\ /\
    'foo' overrides nothing
}
```

Community demands for a solution

- Unnecessary copying
- Prohibited constructions with overriding
- Non-native Kotlin collection for the variadic parameter
- Even more copying than when using collection type parameter
- A number of issues are reported on YouTrack and Kotlin Forums

https://youtrack.jetbrains.com/issue/KT-2462

Goals of the project

- **Goal**: Enhance the functionality of vararg functions
- **Objectives**:
  - Allow using **spread operator on non-array types**
  - **Reduce** the amount of **type** incompatibility **issues**
  - **Reduce** the amount of **copying** performed
  - Step away from using **Java arrays** in pure **Kotlin code**
  - **Implement** a working compiler **prototype** along with a proposal

Variadic functions in other languages: C

- Are based on pointer arithmetics wrapped into macros
- There is no way to determine the number of passed arguments
- The dedicated parameter for the number of arguments is needed
- **printf** function

```c
// C

double sum(int count, ...) {
  double sum = 0;

  va_list ap;
  va_start(ap, count);
  for (int j = 0; j < count; j++) {
    sum += va_arg(ap, int);
  }
  va_end(ap);
  return sum;
}

sum(3, 1, 2, 3) // 6
```

Variadic functions in other languages: Java

- Variadic parameter with three dots
- Must be on the last position
- Only one existing collection can be passed

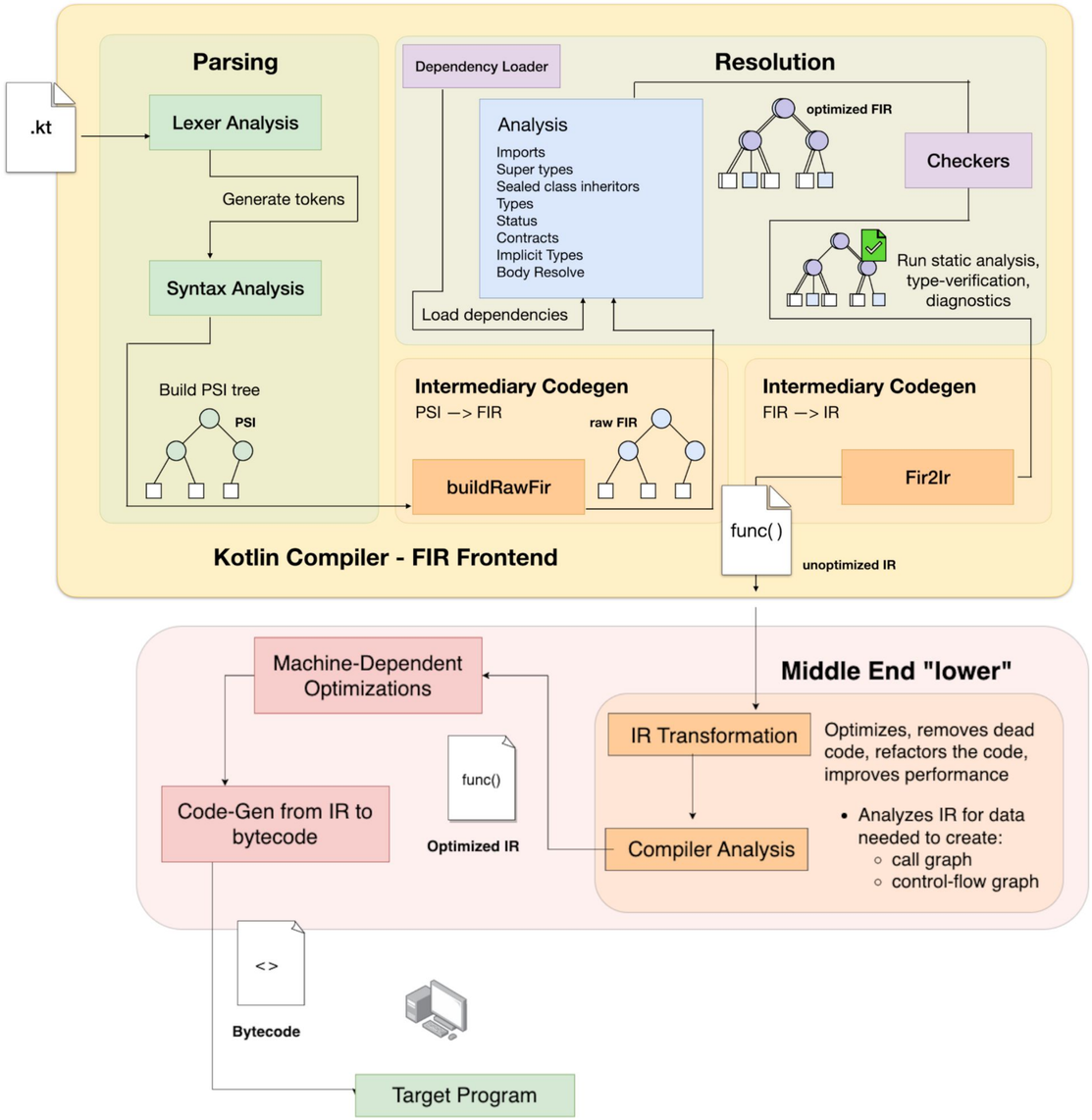
```java
// Java

public class Main {
  public static void printAll(int... x){
    for (Object i:x) {
      System.out.print(i);
    }
  }

  public static void main(String[] args) {
    printAll(1, 2, 3); // 123
    int[] x = {4, 5, 6};
    printAll(x); // 456
  }
}
```

Kotlin compiler

- Front-end and Back-end
- On the FE, Front-end Intermediate Representation tree is built and resolved
- Two important resolution phases:
  - **TYPES** - global names types resolution
  - **BODY_RESOLVE** - function calls resolution
- On the BE, another IR is built and transformed using *lowerings*, which desugar higher-order concepts



Variadic functions implementation

We can break it down into three steps

FE TYPES Phase

Vararg type is replaced with an array type

```
fun foo(vararg x: Int) {}
val x = intArrayOf(4, 5)
foo(1, 2, 3, *x)
```

```
fun foo(x: IntArray) {}
val x = intArrayOf(4, 5)
foo(1, 2, 3, *x)
```

FE BODY_RESOLVE Phase

Candidate functions are found for each call

```
fun foo(x: IntArray) {}
val x = intArrayOf(4, 5)
foo(1, 2, 3, *x)
```

```
fun foo(x: IntArray) {}
val x = intArrayOf(4, 5)
foo(1, 2, 3, *x)
```

IR Vararg Lowering Phase

Creates a single array argument and copies all the elements to it

```
fun foo(x: IntArray) {}
val x = intArrayOf(4, 5)
foo(1, 2, 3, *x)
```

```
fun foo(x: IntArray) {}
val x = intArrayOf(4, 5)
foo(IntArray{1, 2, 3, 4, 5})
```

Solution idea

Two steps:

- Refine the argument type checking when looking for function candidates
- Copy all spread collections on Vararg Lowering phase

Dealing with the argument checking

- Currently, all the spread arguments types are checked against the type of the vararg parameter
- Now we will compare them just by the types of their elements
- This approach provides a more proper error message with elements types

```
fun foo(x: IntArray) {}
val x = listOf(“Hello”, “World”)

foo(*x)
Type mismatch: inferred type is
List<String> but IntArray was expected

// The prototype only compares element
// types (String and Int)
foo(*x)
Type mismatch: inferred type is String but
Int was expected
```

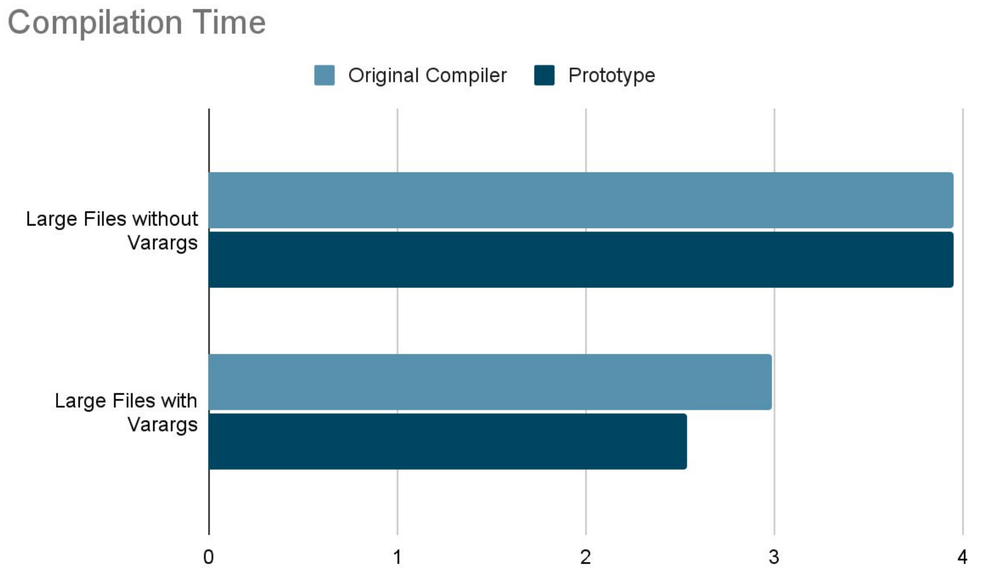
Copying the contents

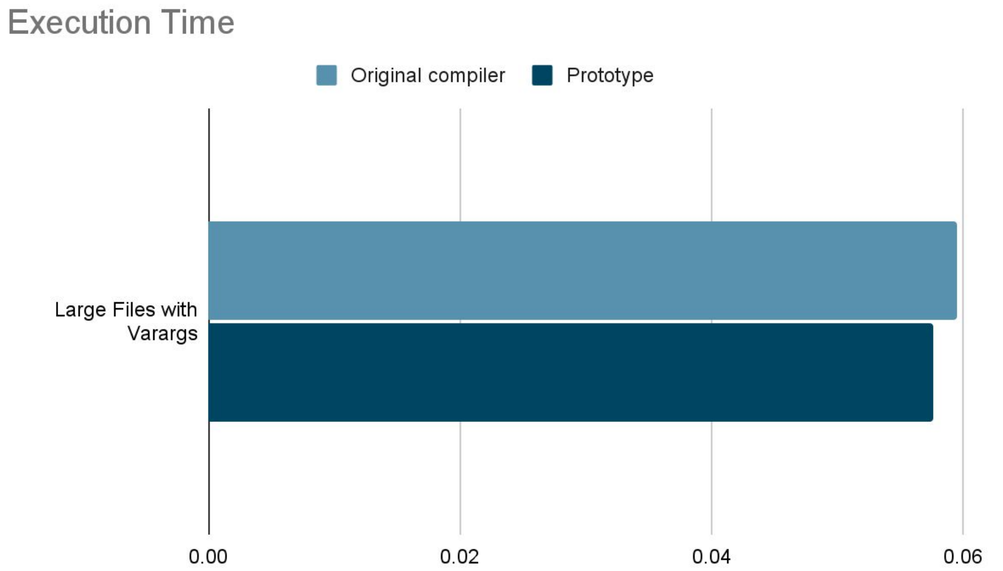
- The Vararg Lowering Phase uses *IrArrayBuilder* class that generates an array with the provided elements
- Utilizes two additional builders *SpreadBuilder* and *PrimitiveSpreadBuilder* in complex cases
- *SpreadBuilder* already supported almost all collection types (expect for primitive arrays)
- All three of these classes were extended to accept *Iterable* collections

Results

```
fun foo(vararg x: Int) {}

// No casts, no additional copying!
fun main() {
    val list = arrayOf(1, 2, 3)
    val set = setOf(4, 5, 6)
    val intArray = intArrayOf(7, 8, 9)
    val array = arrayOf(10, 11, 12)
    foo(
        0,
        *list,
        *set,
        *intArray,
        *array
    )
}
```





Future Steps of Work

Future goals:

- Implement a working prototype of variadic functions with alternative and unified underlying collection
- Rewrite another type checking pipeline that is used for diagnostics
- Develop more tests
- Complete **Kotlin** Enhancement Proposals
- Integrate the prototypes into the **Kotlin** compiler

Additional Slides

Variadic functions in other languages: Python

- Offers non-keyword and keyword variadic arguments
- Has spread operators for each type
- Allows mixing spread and non-spread arguments

```python
# Python

def foo(*args, **kwargs):
    print(args)
    # (0, 1, 2, 3)
    print(kwargs)
    # {'a': 1, 'b': 2, 'c': 3}

list = [1, 2]
dict = {"a": 1, "b": 2}
foo(0, *list, 3, **dict, c = 3)
```

Workaround: passing spread argument by reference

```
// Java
public class VarargUtil {
  public static void passArrayAsVarargs(@NotNull String[] ids) {
      MainKt.processIds(ids); // This proxy passes collection by reference
  }
}
// Kotlin
fun processIds(vararg ids: String) { ... }
fun main() {
  val ids: List<String> = listOf("...")
  passArrayAsVarargs(ids)
}
```

Eliminating primitive arrays

- It is possible to only use boxed array type
- No more inconsistency
- One line fix
- Breaks the backward compatibility
- Still not native Kotlin collection
- Worsens the performance

```
FirCallableDeclaration.transformTypeToArrayType()
{
    …
    type  = ConeKotlinTypeProjectionOut(returnType)
        .createArrayType(
            createPrimitiveArrayTypeIfPossible = false
        )
    …
}
```

Variadic Functions in Kotlin Library

The standard Kotlin library actively utilizes variadic functions

```
public fun <T> listOf(vararg elements: T): List<T> =
  if (elements.size > 0) elements.asList() else emptyList()


// Here it directly relies on the compiler implementation of
// varargs
public inline fun intArrayOf(vararg elements: Int) =
elements
```

Test Data

```
fun callMe0(vararg a: Int) { println(a) }

...

fun main() {

  val var0 = setOf(901, 272, 869, 69, 597, 300, 207, 414, 922, 390, 207, 922, 135, 519, 886)

  val var1 = listOf(776, 251, 740, 194, 812, 601, 341, 38, 192, 945, 122, 107, 494, 252)

  val var2 = arrayOf(616, 937, 123, 762, 483)

  ...

  callMe0(*var27, *var8, *var17, *var7, *var9, *var6, *var13, 305, 35, 140, 504, 474, 213, 917,
505, 822, 517, 418, 437, 93, 353, 234, 446, 403, 623, 758, 781)

  ...

}
```

Workaround for overriding generic vararg methods

```
// Problem
interface A<T> {
    fun foo(vararg x : T)
    // foo(x: Array<out T>): Unit
}

class B : A<Int> {
    override fun foo(vararg x : Int) { }
    // foo(x: IntArray): Unit
    'foo' overrides nothing

}
```

```
// Workaround
interface A<T> {
    fun foo(vararg x : T)
}

class B : A<Int> {
    override fun foo(values: Array<out Int>) {
        foo(*values.toIntArray())
    }

    fun foo(vararg x : Int) {
        ...
    }
}
```

Copying when using spread operator

Spread operator needs to copy the input contents to the underlying array

Single immediate array

```
printAll(*intArrayOf(1, 2, 3))
//optimised to printAll(1,2,3)
```

No copy needed, elements are created in-place

Single non-immediate collection

```
val x = intArrayOf(1, 2, 3)
printAll(*x)
// equivalent to
// printAll(x = x)
```

Copying is performed every time

Several elements with at least one spread array

```
val x = intArrayOf(1, 2, 3)
printAll(*x, *x)
printAll(*x, 4, 5, 6)
```

All elements are copied into one internal array

Primitive and non-Primitive Arrays

Primitive arrays were introduced solely for optimization purposes. When targeting JVM, they are compiled into different types.

Primitive arrays are directly compiled to the array of the corresponding primitive type:

IntArray is compiled into int[]

Boxed arrays Array<out T> with primitive types are compiled into arrays of wrapper types

Array<out Int> is compiled into Integer[]