

Enhancing Vararg Functions in Kotlin

by

Oleg Makeev

Bachelor Thesis in Computer Science

Submission: May 20, 2024

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

Statutory Declaration

Family Name, Given/First Name	Makeev, Oleg
Matriculation number	30006552
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

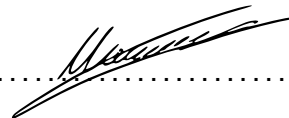
German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

... 20.05.2024

Date, Signature



Abstract

Kotlin is a high-level programming language created by *JetBrains* for the *JVM* platform. It provides a keyword `vararg` that is applied to a function parameter and allows functions to have a variable number of arguments. These elements are stored using an array inside the function and are accessible using its reference. Such an approach allows passing several arguments to one parameter without the need to create intermediate collections on the call site, as long as they are of the same type.

It is also possible to "unwrap" an existing collection and pass its elements to the variadic function using the *spread operator* (`*`), such spread arguments can be mixed with regular single elements and other spread collections.

However, when the function is expected to mostly receive elements from a single already existing collection, developers prefer using a parameter of a normal collection type instead of a variadic one. This is due to the fact that varargs use various array types to store the passed elements depending on the variadic parameter type and expect the spread collections to be of the same type. This causes type incompatibility issues and unnecessary copying required for type casts. Additionally, these underlying arrays are not native *Kotlin* collections and feel out of place.

During this thesis project, we were able to solve this problem by introducing a prototypes for the *Kotlin* compiler that improve code consistency and enhance the user experience. It was done by allowing using spread operators on `Iterable` collections, which is the base interface for most of the standard *Kotlin* collections, and by reducing the amount of copying performed for these collections.

Contents

1	Introduction	1
1.1	Variadic Functions in Kotlin	1
2	Statement and Motivation of Research	4
2.1	Motivation	4
2.1.1	Details of the Current Implementation	4
2.1.2	Issues with Variadic Functions in Kotlin	5
2.2	Aims of the Project	8
2.3	Variadic Functions in Other Languages	8
2.3.1	C	9
2.3.2	Python	10
2.3.3	Java	11
2.4	Existing Solutions	12
2.5	Kotlin Compiler Structure	14
2.5.1	Front-end	14
2.5.2	Back-end	16
2.6	Current Implementation of Variadic Functions	16
2.6.1	Front-end	16
2.6.2	Back-end	17
3	Description of the Investigation	17
3.1	Allowing Spread Operator on Iterable Collections	18
3.1.1	Handling the Argument Checking	18
3.1.2	Handling Vararg Lowering Phase	20
3.1.3	Results	28
3.2	Using Boxed Arrays Only	29
4	Evaluation and Analysis	30
4.1	Criteria for the Solution	30
4.2	Performance Evaluation	31
5	Conclusions	33

1 Introduction

1.1 Variadic Functions in Kotlin

*Kotlin*¹ is a high-level programming language created by *JetBrains*² for the *JVM* platform. It offers a variety of comprehensive tools for handling collections and functions.

In programming languages (including Kotlin) there is a concept of functions with indefinite arity, i.e., which accept a variable number of arguments. Such functions are called variadic [7]. The idea of such functions is present in many modern programming languages and helps to avoid creating unnecessary intermediate collections as opposed to parameters of a collection type.

Here, it's important to distinguish between the function parameter and the function argument: the former is a variable in the function declaration, while the latter is an element on the function call site [1].

```
1 // These are function parameters (s, b, x)
2 //      \/          \/          \/
3 fun foo(s: String, b: List<Char>, vararg x: Int) {
4 }
5
6
7 fun main() {
8 // These are function arguments ("Hello, world", listOf(), 1, 2, 3)
9 //      \/          \/  \/  \/  \/
10     foo("Hello, world", listOf(), 1, 2, 3)
11 }
```

Listing 1: The difference between arguments and parameters

Kotlin provides a special keyword `vararg` for writing functions with a variable number of arguments, which is applied to the corresponding function parameter [4].

¹Kotlin Programming Language <https://github.com/JetBrains/kotlin>

²JetBrains Company <https://www.jetbrains.com/>

```

1 fun fooVariadic(vararg x: Int) {
2     x.forEach {
3         print(it)
4     }
5 }
6
7 fun fooList(x: List<Int>) {
8     x.forEach {
9         print(it)
10    }
11 }
12
13 fun main() {
14     fooVariadic(1, 2, 3) // prints 123
15     fooList(listOf(1, 2, 3)) // prints 123
16 }

```

Listing 2: Kotlin example of variadic function 'foo' and its non-vararg version 'fooNonVararg'

On the listing, there are two functions: `fooVariadic(vararg x: Int)` and `fooList(x: List<Int>)`. The first one accepts a variadic parameter `x` of type `Int` and the second one accepts a normal parameter `x` of type `List<Int>`, each of them writes each element from its parameter to the console. In the main function, `fooVariadic` is being called with three variadic arguments and `fooList` is called on a list of three integers. As we can see, the call to `fooList` introduces additional overhead, as we have to create an immediate collection on the call site.

In Kotlin, every function can have at most one `vararg` parameter and all the passed elements have to be of the same type (the one stated by the type of the `vararg` parameter).

In addition to variadic functions, *Kotlin*, along with many languages, introduces a special *spread operator* (`*`) to unwrap existing collections when calling variadic functions. It is needed to pass the elements of the collection as distinct arguments, as otherwise the function would consider the passed collection as an argument itself. Such spread arguments can be mixed with regular single elements and other spread collections.

```

1 fun foo(vararg x: Int) {
2     x.forEach {
3         print(it)
4     }
5 }
6
7 fun main() {
8     val x = intArrayOf(1, 2, 3)
9     val y = intArrayOf(4, 5, 6)
10    foo(*x) // prints 123
11    foo(*x, *y) // prints 123456
12    foo(0, *x, 4) // prints 01234
13 }

```

Listing 3: Kotlin example of spread operator

On the listing [3] readers can observe a variadic function `foo(vararg x: Int)` that accepts a `vararg` parameter `x` of type `Int` and prints all of its arguments to the console. In the `main` function, two `Int`Arrays are initialized: `x` and `y`, which are passed to the `foo` call using spread operator in different combinations.

This Kotlin feature seems convenient and useful for handling variadic behaviour. However, it presents a number of related problems, which make it challenging to utilize. We will discuss these problems in the next chapter.

2 Statement and Motivation of Research

2.1 Motivation

This section contains the background of the research topic and reveals the existing problems regarding `vararg` functions in *Kotlin*.

2.1.1 Details of the Current Implementation

To preserve the compatibility with *Java*, variadic functions in *Kotlin* are designed to store all the passed `vararg` elements in the corresponding *Java* array type, depending on the parameter type. Functions with primitive types (`Boolean`, `Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`) use the corresponding specialized array type (`IntArray`, `CharArray`, etc), while for other types a templated boxed array `Array<out T>` is used, that can hold an element, which type is a subtype of `T` generic type:

```
1 fun fooInt(vararg x: Int) {  
2     println(x.javaClass.kotlin.qualifiedName) // kotlin.IntArray  
3 }  
4  
5 fun <T>fooTemplate(vararg x: T) {  
6     println(x.javaClass.kotlin.qualifiedName) // kotlin.Array  
7 }  
8  
9 class A  
10 fun fooA(vararg x: A) {  
11     println(x.javaClass.kotlin.qualifiedName) // kotlin.Array  
12 }
```

Listing 4: Kotlin uses different collections to store different `varargs`

Listing [4] demonstrates the type inconsistency for different variadic functions.

- `fooInt` function has a `vararg` parameter `x` with elements of `int` type, the parameter has a type of `IntArray` inside the function.
- `fooTemplate` function has a `vararg` parameter `x` with elements of a templated type, the parameter has a type of `Array<out T>` inside the function.
- `fooA` function has a `vararg` parameter `x` with elements of type `A`, the parameter has a type of `Array<out A>` inside the function.

During the compilation process, the compiler performs a number of checks to ensure the correctness of the program, including type compatibility. It's important to mention that at the type checking stage, all `vararg` parameters have array types and not the type of their elements.

The topic of `vararg` functions is particularly interesting, as the Kotlin standard library itself actively makes use of them. All the type element constructors, like `listOf`, `setOf`, `intArrayOf`, use a `vararg` parameter in their signature. Some variadic functions from the standard library directly delegate their functionality to the compiler. For example, the

implementation of different array casts in Kotlin-Native directly relies ³ on the underlying primitive arrays usage for `vararg` parameters of primitive types.

```
1  /**
2   * Returns an array containing the specified [Float] numbers.
3   */
4   @Suppress("NOTHING_TO_INLINE")
5   public inline fun floatArrayOf(vararg elements: Float) = elements
6
7   /**
8   * Returns an array containing the specified [Long] numbers.
9   */
10  @Suppress("NOTHING_TO_INLINE")
11  public inline fun longArrayOf(vararg elements: Long) = elements
12
13  /**
14   * Returns an array containing the specified [Int] numbers.
15   */
16  @Suppress("NOTHING_TO_INLINE")
17  public inline fun intArrayOf(vararg elements: Int) = elements
```

Listing 5: Kotlin standard library relies on the `vararg` implementation.

Code listing [5] contains a part of the Kotlin-Native standard library implementation of several functions

- `floatArrayOf(vararg elements: Float): FloatArray`
- `longArrayOf(vararg elements: Long): LongArray`
- `intArrayOf(vararg elements: Int): IntArray`

These functions take a variable number of arguments via `vararg` parameter and return a specialized array containing the passed elements. It can be seen on the snippet that all these functions just return their `vararg` parameter and rely on the compiler to construct the required array.

These primitive arrays were introduced to increase performance, as they are directly compiled into *Java* arrays with the corresponding primitive type on the *Java* bytecode level, while boxed arrays (`Array<out T>`) are compiled into arrays of wrapper objects (`Object[]`). For example, `IntArray` is compiled into `int[]`, while `Array<Int>` is compiled into `Integer[]`.

2.1.2 Issues with Variadic Functions in Kotlin

The main concern regarding the `vararg` functions is the constraint applied to spread arguments. The collection passed via spread operator must have the same type as the `vararg` array. In pure *Kotlin* code arrays are not widely used, as the language provides a set of its own convenient collection types in the standard library. This implies that the cast to the array type is often needed, which performs an additional copying.

³Kotlin-Native Arrays <https://github.com/JetBrains/kotlin/blob/0938b46726b9c6938df309098316ce741815bb55/kotlin-native/runtime/src/main/kotlin/kotlin/Arrays.kt>

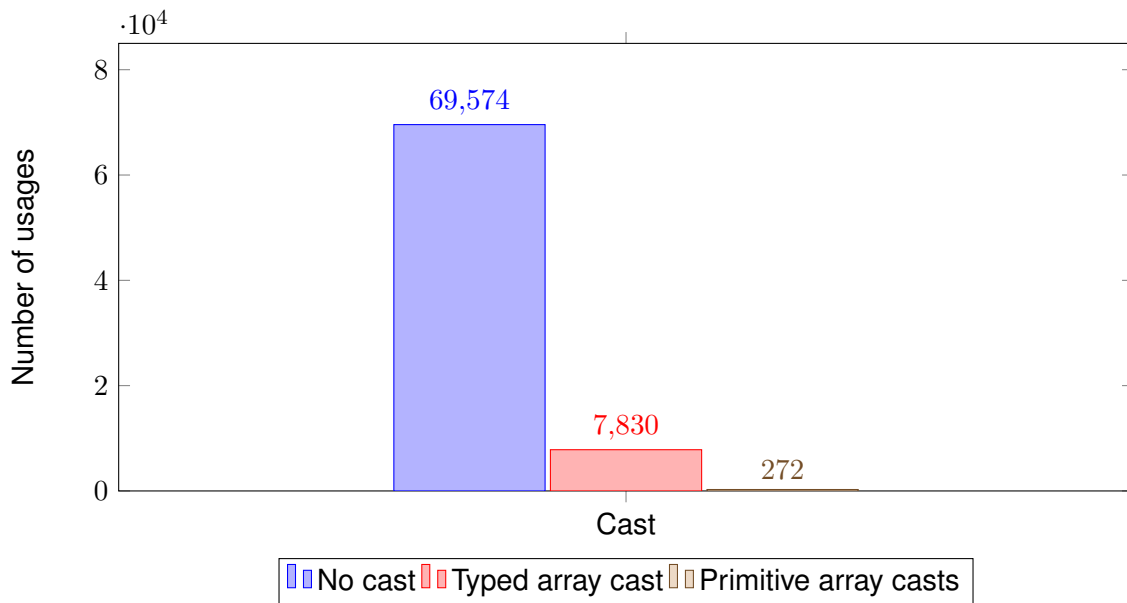
Listing [6] shows two variadic functions: `fooInt` that accepts `Int vararg` parameter and `<T>fooTemplate` that accepts templated parameter `T`. In the `main` function, we initialize `list` variable with the type `List<Int>`. After that, we are trying to pass `list` to both of these functions using the spread operator. However, due to the type incompatibility, these calls do not succeed. After that, both functions are called on `list` using the spread operator and needed casts.

```
1 fun fooInt(vararg x: Int) {
2     println(x.javaClass.kotlin.qualifiedName) // kotlin.IntArray
3 }
4
5 fun <T>fooTemplate(vararg x: T) {
6     println(x.javaClass.kotlin.qualifiedName) // kotlin.Array
7 }
8
9 fun main() {
10     val list = listOf(1, 2, 3)
11
12     fooInt(*list)
13     // Type mismatch, IntArray expected, List<Int> found
14
15     fooTemplate(*list)
16     // Type mismatch, Array<out Int> expected, List<Int> found
17
18     fooInt(*list.toIntArray())
19     // OK, performs an additional copy
20
21     fooTemplate(*list.toTypedArray())
22     // OK, performs an additional copy
23 }
```

Listing 6: Variadic functions expect spread collections to have corresponding array types

This approach not only adds unnecessary overhead and inconsistency, as we have to call different cast methods for different functions. Plot [7] shows the usage statistics of different casts on spread arguments right on the call site. It was collected using Sourcegraph ⁴ utility that allows searching for a regular expression in open repositories. As we can see, in 10.4 percents of all spread operator usages, a cast was inserted on the call site. Most of the calls that didn't involve calling any casts were calls to *Java* APIs and libraries or used a pass-through variadic parameter.

⁴sourcegraph.com <https://sourcegraph.com/search>



Listing 7: Usages of various casts on spread arguments on the call site

Additionally, the inconsistency of the underlying collection type prohibits some constructions.

One of such constructions is overriding generic `vararg` method with a primitive type [9].

Listing [8] shows a templated interface `A<T>` with a single variadic method `foo(vararg x: T)`. Class `B` is derived from `A<T>` specializing the templated type with `Int`. As `B` is derived from `A<Int>`, it has to override `foo(vararg x: T)` method using `Int` type. However, on the type checking stage, the original method has signature `foo(x: Array<out T>)` and the method in `B` class has a divergent signature `foo(x: IntArray)`, so the compiler doesn't consider the `B.foo` method as an override of the base method.

```

1 interface A<T> {
2     fun foo(vararg x: T)
3     // Array<out T> expected
4 }
5
6 class B : A<Int> {
7     override fun foo(vararg x: Int) { }
8     // Error, method overrides nothing, IntArray expected
9 }

```

Listing 8: Method `B.foo` overrides nothing, as the signature differs

Nowadays, people tend to use variadic functions mostly for wrapping *Java* API and not as independent code. For other purposes, developers prefer using parameters of collection types (list, set, etc.). Due to the mentioned issues and overhead, this Kotlin feature is not widely used in the community. At the moment, all known issues are tracked using the YouTrack issue tracker [8].

2.2 Aims of the Project

The main goal of this project is to research the possibility of enhancing variadic functions in *Kotlin*. Ideally, it should be possible to use *spread operator* on any collection type, mainly on types that are derived from `Iterable` interface, which is a base interface for most of the standard collections.

Secondly, we should reduce the amount of copying performed during the usage of variadic functions.

As an additional goal, the types of variadic parameters should somehow be compatible to avoid type issues when working with primitive and non-primitive types.

2.3 Variadic Functions in Other Languages

Early languages didn't formally introduce the concept of variadic functions and sometimes didn't require any function declarations in advance. The example of such a language is *B* [5], the predecessor of *C*. Since the function call has no information about the number of arguments, it allows passing a variable number of parameters. Inside the function, one can use a special function `nargs()` that returns the number of passed arguments to the function. It's needed to fill in default values for the parameters that were not initialized. Otherwise, if the number of passed arguments is greater than the number of parameters, all the surplus arguments are discarded.

In modern languages that don't have a dedicated mechanism for variadic functions, variadic behaviour could be mimicked by overloading a function for each required number of arguments. It requires building a call chain from functions with fewer parameters towards the main function of maximum arity, containing all the required logic.

```
1 fun sum(x1: Int): Int = sum(x1, 0)
2
3 fun sum(x1: Int, x2: Int): Int = sum(x1, x2, 0)
4
5 fun sum(x1: Int, x2: Int, x3: Int): Int = x1 + x2 + x3
```

Listing 9: Mocking variadic functions in Kotlin

On the listing [9] readers can observe the example of mocking the variadic behaviour in Kotlin language. There is a function `sum` that calculates the sum of its parameters and that is overloaded three times for one, two and three parameters

- `sum(x1: Int): Int`
- `sum(x1: Int, x2: Int): Int`
- `sum(x1: Int, x2: Int, x3: Int): Int`

The version with one parameter calls the function with two parameters, substituting 0 for the second parameter. The same is done when calling the version with three parameters from the second one. The last function returns the sum of all three of its parameters.

However, this approach is not really scalable and efficient, as it contains a lot of boilerplate code and its usage is limited to the maximum number of parameters for which the user has implemented the call chain. So the concept of variadic functions started to become

a widely used native feature in many programming languages. We will consider some of them further.

2.3.1 C

C is a classical compiled low-level programming language. It has one of the first and most known appearances of variadic functions, which is *printf* [1] function that takes a formatting string and arbitrary number of arguments and then prints the arguments according to the formatting string.

Variadic functions in C are marked with three dots (...) in the parameter list right after the last named parameter. One can access variadic arguments inside the function using several special types and macros [2] based on the pointer arithmetic. However, as there is no way to calculate the number of passed arguments, a dedicated parameter for this purpose is needed. In the case of *printf* function, the number of arguments expected is derived from the format string itself.

On code snippet [10] there is `double sum(int count, ...)` function that calculates a sum of variadic arguments. It takes a parameter `count` that represents a number of variadic arguments.

```
1  #include <stdarg.h>
2  #include <stdio.h>
3
4  double sum(int count, ...) {
5      va_list ap;
6      int j;
7      double sum = 0;
8
9      va_start(ap, count);
10     for (j = 0; j < count; j++) {
11         sum += va_arg(ap, int);
12     }
13     va_end(ap);
14
15     return sum;
16 }
17
18 int main() {
19     printf("%f\n", sum(3, 1, 2, 3));
20     return 0;
21 }
```

Listing 10: Variadic function *sum* in C

`va_list` is a dedicated type definition for the type that holds all the required information regarding the variadic parameter of the current function. The usage of variadic parameters is based on several function macros:

- `va_start(list, par)` — takes a `va_list` object and a reference to the function's last parameter. In the latest language standard, the second parameter is no longer

required. It initialises the `va_list` object for use by `va_arg` or `va_copy`

- `va_arg(list, type)` — takes two parameters, a `va_list` object and a type descriptor. It expands to the next variable argument, and has the specified type. Successive invocations of `va_arg` allow processing each of the variable arguments in turn. Unspecified behavior occurs if the type is incorrect or there is no next variable argument.
- `va_end(list)` — takes one parameter, a `va_list` object, and resets it. For example, to scan the variable arguments multiple times, the programmer would need to re-initialize the `va_list` object by calling `va_end` and then `va_start` again on it.
- `va_copy(list1, list2)` — takes two parameters, both of them `va_list` objects. It clones the second (which must have been initialised) into the first.

However, the `printf` function was considered dangerous by many experts for its vulnerability called ‘format string attack’ [6]. After the invocation, the function puts all the elements on the stack and parses the format string and searches for “%” specifiers in the string to embed arguments. Then it pops the element from the stack and prints it. However, many developers made a mistake by writing ‘`printf(buffer)`’ instead of ‘`printf(“%s”, buffer)`’ in order to print the ‘buffer’ string. In this case, `printf` considers the passed string as a format string. If it contains an element embedding “%” it tries to pop an element from the stack, which is not presented, so it pops further. This hack allows attackers to access the program stack and change the stored data.

2.3.2 Python

Python is one of the most popular languages nowadays. It’s a high-level general-purpose interpreted language. It has two options for variadic parameters:

1. Non-keyword variadic parameters. These are marked with an *asterisk* (*) before the parameter name in the function declaration. Inside the function, users can access the passed arguments via the variadic parameter name. The type of the parameter is tuple.

Code on listing [11] shows a function `foo(*args)` that takes a variadic parameter `args` and prints its elements to the console. Then the function is called on three elements 1, 2, 3.

```
1 def foo(*args): # type(args) == <class 'tuple'>
2     for element in args:
3         print(element, end='')
4
5 foo(1, 2, 3) # prints 123
```

Listing 11: *Python* example of non-keyword variadic parameter

2. Keyword variadic parameters. Such parameters are marked with double *asterisk* (**). It allows passing named arguments to the function. The type of such a parameter inside the function is a dictionary.

Listing [12] demonstrates function `foo(**kwargs)` that takes a variadic keyword parameter `kwargs` and prints variables “a” and “b” to the console. The function is

called with two parameters `a = "hello"` and `b = 1`.

```
1 def foo(**kwargs): # type(kwargs) == <class 'dict'>
2     print(kwargs["a"]) // hello
3     print(kwargs["b"]) // 1
4
5 foo(a = "hello", b = 1)
```

Listing 12: *Python* example of keyword variadic parameter

Two mentioned approaches can be used together. But named and non-named arguments have to be in the same order as the parameters defined in the function signature.

Python also provides a way to pass existing collections to these parameters. To unpack a collection to a non-named parameter, a single *asterisk* (*) is used. For named arguments, the *double asterisks* (**) operator is used.

On the following listing [13] function `foo(*args, **kwargs)` takes a non-keyword variadic parameter `args` and a keyword variadic parameter `kwargs`, printing them to the console. The function is called on a list `list` and on a dictionary `dict` using spread operators. Note that there are other non-spread parameters being passed.

```
1 def foo(*args, **kwargs):
2     print(args) # (0, 1, 2, 3)
3     print(kwargs) # {'a': 1, 'b': 2, 'c': 3}
4
5 list = [1, 2]
6 dict = {"a": 1, "b": 2}
7 foo(0, *list, 3, **dict, c = 3)
8
```

Listing 13: *Python* example of spread operators

2.3.3 Java

In *Java*, each function can have at most one variadic parameter, and it has to be in the last position in the function signature. Such a parameter is marked with *triple dots* (...) after the parameter type.

Code snippet [14] demonstrates class `Main` with two methods: variadic method `printAll(int... things)` and the main function `main(String[] args)`. Method `printAll(int... things)` accepts a variadic number of `int` arguments and prints all of them to the console. It's called from the `main` function with three arguments `1, 2, 3`.

```

1 public class Main {
2     public static void printAll(int... things){
3         for(Object i:things){
4             System.out.print(i);
5         }
6     }
7
8     public static void main(String[] args) {
9         printAll(1, 2, 3); // prints 123
10    }
11 }

```

Listing 14: Java example of variadic function

It is also possible to pass elements of an array to the variadic parameters. However, this can only be done for one array per call without the possibility for any other arguments to be passed.

On the listing [15] the reader can observe class Main with a variadic method `printAll(int... things)` and the main function `main(String[] args)`. Method `printAll(int... things)` takes a variadic parameter `things` of type `int` and prints all the arguments to the console. In the main function `printAll` is called on a single array `arr` with three parameters 1, 2, 3.

```

1 public class Main {
2     public static void printAll(int... things){
3         for(Object i:things){
4             System.out.print(i);
5         }
6     }
7
8     public static void main(String[] args) {
9         int[] arr = {1,2,3};
10        printAll(arr); // prints 123
11    }
12 }

```

Listing 15: In Java there is no spread operator, only one array containing all the elements can be passed to the function

In such cases, arrays are passed by reference, just like any other normal argument. Inside the function, all the elements are accessed using the `vararg` parameter, which has an array type with elements being of the type of the variadic parameter.

2.4 Existing Solutions

To solve the mentioned issues [2.1.2] [2.1.2] in *Kotlin* using existing tools, the community proposed several workarounds:

- To be able to spread non-array collections, developers have to call a cast to the corresponding array type.

```

1 fun fooInt(vararg x: Int) {
2     // Some code
3 }
4
5 fun <T>fooTemplate(vararg x: T) {
6     // Some code
7 }
8
9 fun main() {
10     val list = listOf(1, 2, 3)
11     fooInt(*list.toIntArray())
12     fooTemplate(*list.toTypedArray())
13 }

```

Listing 16: Variadic functions expect spread collections to have corresponding array types

- In case of overriding a generic `vararg` method with a primitive type, one should create a method that takes an array with the desired type instead of a `vararg`. On the type checking stage, the signature of the abstract method and the derived method is identical, so the compiler won't throw any errors. However, to keep the variadic behavior, another method that takes `vararg` parameter of the required type should be created. Now all the logic should be kept in one of these functions, while the other one references it.

```

1 abstract class A<T> {
2     // Base method
3     abstract fun broken(vararg values: T): String
4 }
5
6 class B : A<Int>() {
7     // Override method
8     override fun broken(values: Array<out Int>): String {
9         return broken(*values.toIntArray())
10    }
11
12    // Method that keeps the variadic behaviour
13    fun broken(vararg values: Int): String {
14        return values.joinToString { it.toString() }
15    }
16 }

```

Listing 17

- Another workaround helps to omit copying a single collection argument. The idea is to write all the logic in *Kotlin*, but call the variadic function via a special proxy function

in *Java*. In the example below, `VarargUtil` has a single method that accepts an array of `int` and calls the *Kotlin* variadic function with the received array. With such an approach, the argument won't be copied and will be passed by reference.

```
1 import VarargUtil.passArrayAsVarargs
2
3 fun foo(vararg ids: Int) {
4     // Some code
5 }
6
7 fun main() {
8     val array: IntArray = intArrayOf(1, 2, 3)
9     passArrayAsVarargs(array)
10 }
```

Listing 18: Main.kt

```
1 public class VarargUtil {
2     public static void passArrayAsVarargs(int[] ids) {
3         MainKt.foo(ids);
4     }
5 }
```

Listing 19: VarargUtil.java

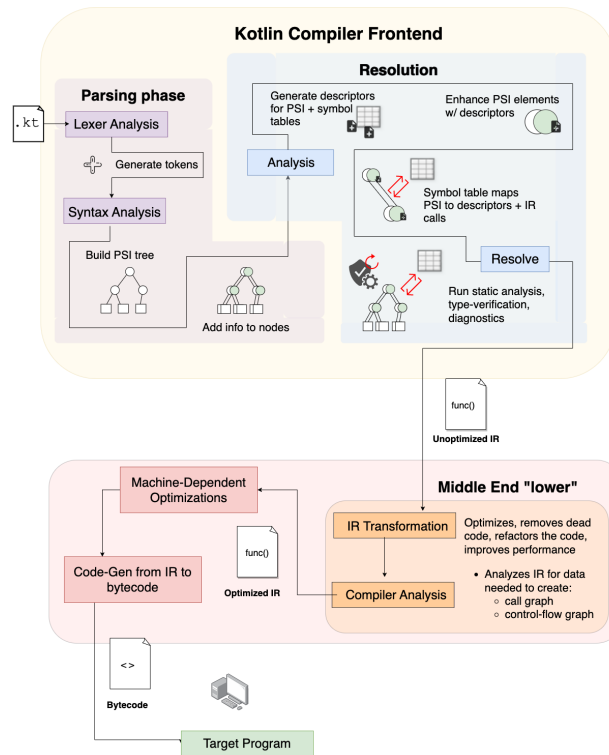
This approach works as follows: the function call to `passArrayAsVarargs` is just a usual function call, so the array is passed by reference. Then, inside `VarargUtil` *Java Kotlin* function `foo` is also variadic. However, in *Java* you can only pass one collection to the variadic function and it is done by reference. The drawback of this approach is the need to create a new *Java* method for each *Kotlin* variadic function, which implies a lot of boilerplate code.

2.5 Kotlin Compiler Structure

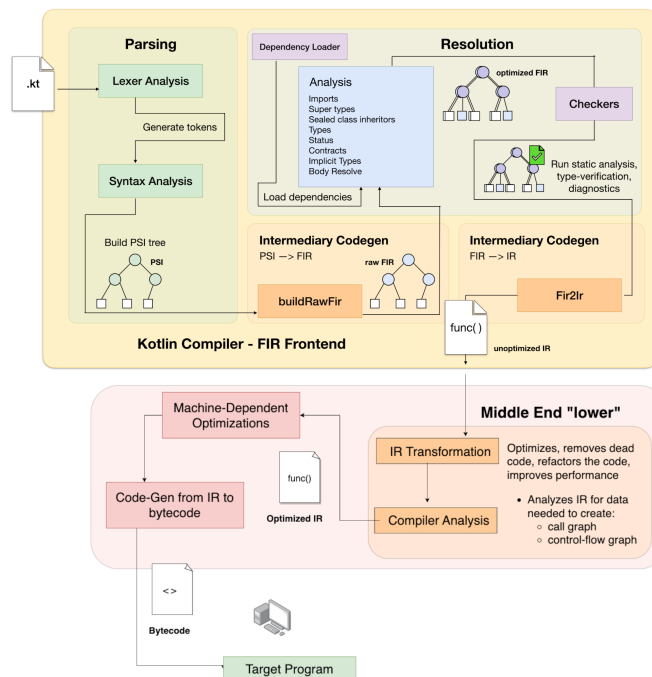
The Kotlin compiler follows the classical compiler structure: front-end and back-end [3]. The front-end part is responsible for parsing the source code, initial analysis and resolution. After that the intermediate representation is generated, on which a lot of optimizations are performed. On the back-end part, the compiler generates executable code for the target platform. Kotlin is famous for having several target platforms: Kotlin-JVM is compiled into Java byte code, Kotlin-native is directly compiled into the executable file using LLVM infrastructure, Kotlin/JS is compiled into JavaScript for web development. All these platforms utilize the same front-end, but have different back ends implemented.

2.5.1 Front-end

The front-end part is responsible for parsing the source code, verifying syntax, initial analysis and resolution. There are two generations of the Kotlin compiler front-end: K1 (also known as FE10) and K2.



Listing 20: K1 compiler pipeline [3]



Listing 21: K2 compiler pipeline [3]

The main difference between the two is that K2 introduces a new intermediate tree concept: FIR (front-end intermediate representation) tree. It was made to increase the overall performance and to be able to represent the source code semantically, not syntactically.

K2 aims to take the load off the back-end part, moving a lot of unified analysis on the front-end. In this project, we will focus only on the second version of the compiler.

The FIR resolution stage consists of several phases (as depicted on the listing [21]). In terms of this project, we are mainly interested in TYPES and BODY_RESOLVE stages:

- TYPES — resolves explicit default types of declarations, including function parameter types.
- BODY_RESOLVE — analyses bodies of all other functions and properties, including resolving the function calls and finding candidate functions for each one.

2.5.2 Back-end

The backend is responsible for generating executables for the target architecture and performing a number of architecture-specific optimizations. After the front-end resolution, FIR-tree is converted into the middle-end representation. At this stage, a special representation is generated for each FIR node using the same transformer pattern. At this stage, more static analyzers are applied, such as dead-code elimination and constant propagation. To optimize the intermediate representation, "lowering" phases are used. These lowerings desugar higher order concepts into lower ones.

2.6 Current Implementation of Variadic Functions

The key elements for the implementation of variadic parameters in Kotlin are located on both the front-end and the back-end of the compiler. We can divide the whole varargs implementation into three steps.

2.6.1 Front-end

After the FIR is built, transformations take place, including a special node type transformer *FirTypeResolveTransformer* called on TYPES resolve phase. It takes the type of the function's `vararg` parameter and transforms it into an array type: primitive types are being transformed into primitive arrays, all other types use the boxed array type⁵.

```
1 // Before
2 fun foo(vararg x: Int): Unit
3
4 // After
5 fun foo(x: IntArray): Unit
```

Listing 22: Variadic function signature before and after transformations

After that, on the BODY_RESOLVE stage, the compiler tries to resolve all the function bodies, including nested function calls, which we are interested in. When resolving a function call, the compiler has to find the corresponding function if there is any. For this purpose, it tries to find a set of candidate functions based on many compatibility criteria: value parameters, type parameters, parameter of the receiver etc. One of the checkers

⁵`FirTypeResolveTransformer.transformProperty` <https://github.com/JetBrains/kotlin/blob/a511bbcddeb1919ad6fcbcc492a9b85588553a9d6/compiler/fir/resolve/src/org/jetbrains/kotlin/fir/resolve/transformers/FirTypeResolveTransformer.kt#L219>

for each candidate is the value arguments checker⁶. It checks whether the type of each argument on the call site can be substituted to the function parameter. If the parameter type is not the subtype of the expected argument type, the checker reports an error using a special report service. We will actively use this aspect of the compiler in several solutions. For spread arguments, it checks their types against the type of the `vararg` parameter. For single arguments, it unwraps the element type of the `vararg` array and checks the argument type against it.

2.6.2 Back-end

After the initial intermediate representation generation, the middle-end transformations take place. Some transformations are IR lowering phases — these are the stages where high-level abstractions are turned into lower-level ones. One of these lowering stages is the `vararg` lowering phase `VarargLowering`⁷. Our function currently accepts an array type; however, we are passing several elements to it. To handle this, the lowering phase creates a new IR array and copies all the elements from the call site to it. It is implemented in terms of the transformer pattern. When it visits `IrFunctionAccessExpression` it goes through the call parameters and tries to find a `vararg` parameter if there is one. If there is one, the compiler should generate IR for instantiating the needed array and copying all the elements into it. For this purpose, it calls `IrArrayBuilder`⁸ and provides it the `JvmIrBuilder` and the required array type. After that, depending on the number of arguments, `IrArrayBuilder` chooses the best strategy for each case:

- When there are no `vararg` arguments, it instances an empty array
- In case when there are no spread arguments, it instances a new array and generates calls to `Array.set` to set each element of the created array
- When there is only one spread argument, it passes it by reference if it is an in-place array, otherwise it copies it. This was made so that no one could accidentally change the array using its reference somewhere outside the function.
- Otherwise if there are several mixed elements, it instances a special array builder, which copies all the elements to one unified array.

After the call to the array builder, the newly created array is placed at the function call IR.

3 Description of the Investigation

In this chapter, we present the solution idea along with the way to achieve it in the Kotlin compiler.

⁶`Arguments.kt` <https://github.com/JetBrains/kotlin/blob/a511bbcdeb1919ad6fcbcc492a9b85588553a9d6/compiler/fir/resolve/src/org/jetbrains/kotlin/fir/resolve/calls/Arguments.kt>

⁷`VarargLowering` <https://github.com/JetBrains/kotlin/blob/a511bbcdeb1919ad6fcbcc492a9b85588553a9d6/compiler/ir/backend.jvm/lower/src/org/jetbrains/kotlin/backend/jvm/lower/VarargLowering.kt#L34>

⁸`IrArrayBuilder` <https://github.com/JetBrains/kotlin/blob/a511bbcdeb1919ad6fcbcc492a9b85588553a9d6/compiler/ir/backend.jvm/src/org/jetbrains/kotlin/backend/jvm/ir/IrArrayBuilder.kt#L27>

3.1 Allowing Spread Operator on Iterable Collections

One of the most straightforward ways to allow calling vararg functions with any collections using spread operator consists of two steps:

1. To bypass the argument type checking when searching for function candidates.
2. To handle non-array types on the `vararg` lowering phase.

3.1.1 Handling the Argument Checking

All the function calls in FIR tree are instances of `FirFunctionCall` class [23]. This class contains all the information regarding the function call.

```
1 abstract class FirFunctionCall : FirQualifiedAccessExpression(), FirCall {
2     @UnresolvedExpressionTypeAccess
3     abstract override val coneTypeOrNull: ConeKotlinType?
4     abstract override val typeArguments: List<FirTypeProjection>?
5     abstract override val argumentList: FirArgumentList
6     abstract override val calleeReference: FirNamedReference
7     // Other fields
8
9     ...
10
11 }
```

Listing 23: `FirFunctionCall` class from the Kotlin compiler

One of the information stored in `FirFunctionCall` [23] is `calleeReference`. This is the reference to the function that is being called. All the references are resolved on the "BODY_RESOLVE" stage. Before that, it contains a dummy reference of type `FirSimpleNamedReference`, that contains just the function name and the information about which source the FIR node was generated from.

On the "BODY_RESOLVE" stage, the calls are resolved and the `callableReference` in the FIR node are turned into `FirResolvedNamedReference`. This class has an additional field `resolvedSymbol` of type `FirBasedSymbol<T>` that contains the reference to the function.

For function call resolving `org.jetbrains.kotlin.fir.FirCallResolver` class is used. It has a special method `resolveCallableReference` that takes the function call FIR node and finds the most suitable candidate function based on many features. For each candidate, several checking stages are being run with argument types compatibility being checked.

Currently, all the spread arguments types are checked against the type of the `vararg` array, which is the source of many issues. The more suitable and natural way to resolve this is to compare only the elements' types.

```

1 private fun checkApplicabilityForArgumentType(
2     // Not relevant parameters are hidden
3     argument: FirExpression,
4     argumentTypeBeforeCapturing: ConeKotlinType,
5     initialExpectedType: ConeKotlinType?,
6     // Added parameter
7     isSpread: Boolean = false
8 ) {
9     var expectedType = initialExpectedType ?: return
10    var argumentType =
11        captureFromTypeParameterUpperBoundIfNeeded(
12            argumentTypeBeforeCapturing,
13            expectedType,
14            context.session
15        )
16
17    ////////////////
18    // Inserted code:
19    if (isSpread && !argumentType.isNullable) {
20        argumentType =
21            ↪ argumentType.spreadableCollectionElementType()?.also {
22                expectedType =
23                    expectedType.arrayElementType()
24                    ?: error(
25                        "Could not retrieve expected element type for
26                        ↪ vararg parameter. "
27                    )
28            } ?: argumentType
29    }
30    ////////////////
31    ...
32 }

```

Listing 24: The argument type checking part of the prototype

Code snippet [24] shows a part of the prototype that is responsible for the argument type checking. `expectedType` represents the expected type of the argument, `argumentType` represents the type of the passed argument. `isSpread` boolean flag represents whether the argument was passed using spread operator and whether the corresponding parameter is, indeed, a `vararg` one. If this flag is true and the argument type is not nullable (it's important as nullable types are not allowed to be passed using spread operator), both the parameter and argument types are replaced with the type of their elements. For this purpose, `spreadableCollectionElementType()` and `arrayElementType()` methods are used to retrieve the element type for either a spreadable collection or an array. The rest of the argument checking is the same. This approach is beneficial, as now the type checking stage checks only whether the element types are compatible with each other and not the collection types. Moreover, now the compiler produces a more meaningful error message [25].

```

1 fun intVararg(vararg xs: Int) {}
2
3 fun main() {
4     val stringArray = arrayOf("Hello", "World")
5
6     // error: argument type mismatch: actual type is 'kotlin.String',
7     // but 'kotlin.Int' was expected.
8     //      \
9     intVararg(*stringArray)
10 }

```

Listing 25: The compiler throws a meaningful error

The rest of candidate processing stays the same. Note that in the listing [24] the collection element is retrieved using `spreadableCollectionElementType` method. This method returns an element type for all array types and iterable collections. For other types, the method returns `null`.

3.1.2 Handling Vararg Lowering Phase

Let's move on to the lowering part. The `VarargLowering` phase has two main responsibilities:

- To create an empty array for empty `vararg` argument list, when there is no default value for the parameter provided.
- To create an array containing all the passed elements when there are any.

We are interested only in the second aspect.


```

1  override fun visitVararg(expression: IrVararg): IrExpression =
2      createBuilder(expression.startOffset,
3          ↪ expression.endOffset).irArray(expression.type) {
4          ↪ addVararg(expression) }
5
6  inline fun JvmIrBuilder.irArray(
7      arrayType: IrType,
8      block: IrArrayBuilder.() -> Unit
9  ): IrExpression =
10     IrArrayBuilder(this, arrayType).apply { block() }.build()
11
12 private fun IrArrayBuilder.addVararg(expression: IrVararg) {
13     loop@ for (element in expression.elements) {
14         when (element) {
15             is IrExpression -> +element.transform(this@VarargLowering,
16                 ↪ null)
17             is IrSpreadElement -> {
18                 val spread = element.expression
19                 if (spread is IrFunctionAccessExpression
20                     && spread.symbol.owner.isArrayOf()) {
21                     // Skip empty arrays and don't copy immediately
22                     ↪ created arrays
23                     val argument = spread.getValueArgument(0) ?:
24                         ↪ continue@loop
25                     if (argument is IrVararg) {
26                         addVararg(argument)
27                         continue@loop
28                     }
29                 }
30                 addSpread(spread.transform(this@VarargLowering, null))
31             }
32         }
33     }
34     else -> error("Unexpected IrVarargElement subclass")
35 }
36 }
37 }

```

Listing 26: Part of logic contained in the VarargLowering class

VarargLowering class contains a method for transforming IrVararg element [26]. It creates a new JvmIrBuilder instance and calls irArray method, which builds a new array of the desired type using IrArrayBuilder [27].

```

1 private class IrArrayElement(val expression: IrExpression, val isSpread:
  ↳ Boolean)
2
3 private val elements: MutableList<IrArrayElement> = mutableListOf()
4
5 private val hasSpread
6     get() = elements.any { it.isSpread }
7
8 operator fun IrExpression.unaryPlus() = add(this)
9 fun add(expression: IrExpression) =
  ↳ elements.add(IrArrayElement(expression, false))
10
11 fun addSpread(expression: IrExpression) =
  ↳ elements.add(IrArrayElement(expression, true))
12
13 fun build(): IrExpression {
14     val array = when {
15         elements.isEmpty() -> newArray(0)
16         !hasSpread -> buildSimpleArray()
17         elements.size == 1 -> copyArray(elements.single().expression)
18         else -> buildComplexArray()
19     }
20     return coerce(array, arrayType)
21 }

```

Listing 27: Part of logic contained in the IrArrayBuilder class

Listing [27] contains a part of IrArrayBuilder class. It contains a list of passed variadic arguments `elements: MutableList<IrArrayElement>`. When all arguments are added, the `build` method is called, that returns the array containing all the elements. Depending on the number of non-spread and spread arguments, it chooses the best strategy. We are interested in cases when there is only one spread argument and when there are many arguments with at least one being spread.

Let's start with the second case, as in the first case (for single non-array spread arguments we will still call `buildComplexArray()`). Let's take a look at this method.

```

1 private fun buildComplexArray(): IrExpression {
2     val spreadBuilder = if (unwrappedArrayType.isBoxedArray)
3         builder.irSymbols.spreadBuilder
4     else
5         builder.irSymbols.primitiveSpreadBuilders.getValue(elementType)
6
7     val addElement = spreadBuilder.functions.single {
8         it.owner.name.asString() == "add"
9     }
10    val addSpread = spreadBuilder.functions.single {
11        it.owner.name.asString() == "addSpread"
12    }
13    val toArray = spreadBuilder.functions.single {
14        it.owner.name.asString() == "toArray"
15    }
16
17    return builder.irBlock {
18        val spreadBuilderVar = ... // Call to constructor
19
20        for (element in elements) {
21            +irCall(if (element.isSpread) addSpread else addElement).apply
22            ↪ {
23                dispatchReceiver = irGet(spreadBuilderVar)
24                putValueArgument(
25                    0,
26                    coerce
27                        (
28                            element.expression,
29                            if (element.isSpread)
30                                unwrappedArrayType
31                                else elementType
32                        )
33                )
34            }
35
36            val toArrayCall = irCall(toArray).apply {
37                ...
38            }
39
40            if (unwrappedArrayType.isBoxedArray)
41                +builder.irImplicitCast(toArrayCall, unwrappedArrayType)
42            else
43                +toArrayCall
44        }
45    }
46

```

Listing 28: Part of logic contained in the IrArrayBuilder class

Snippet [28] shows `buildComplexArray` method of `IrArrayBuilder` class. It can be seen that it retrieves some spread builder by the type of the expected array. Then it retrieves three methods `"add"`, `"addSpread"` and `"toArray"`. The first two are used for adding an element to the array, the last one is used for instantiating the resulting array. Then it creates several calls to these methods and adds them to builder: `JvmIrBuilder`.

```

1  public class SpreadBuilder {
2
3      private final ArrayList<Object> list;
4
5      public SpreadBuilder(int size) {
6          list = new ArrayList<Object>(size);
7      }
8
9      @SuppressWarnings("unchecked")
10     public void addSpread(Object container) {
11         if (container == null) return;
12
13         if (container instanceof Object[]) {
14             Object[] array = (Object[]) container;
15             if (array.length > 0) {
16                 list.ensureCapacity(list.size() + array.length);
17                 Collections.addAll(list, array);
18             }
19         }
20         else if (container instanceof Collection) {
21             list.addAll((Collection) container);
22         }
23         else if (container instanceof Iterable) {
24             for (Object element : (Iterable) container) {
25                 list.add(element);
26             }
27         }
28         else if (container instanceof Iterator) {
29             for (Iterator iterator = (Iterator) container;
30                  ↪ iterator.hasNext(); ) {
31                 list.add(iterator.next());
32             }
33         else {
34             throw new UnsupportedOperationException(
35                 "Don't know how to spread " + container.getClass()
36             );
37         }
38     }
39
40     public int size() {
41         return list.size();
42     }
43
44     public void add(Object element) {
45         list.add(element);
46     }
47
48     public Object[] toArray(Object[] a) {
49         return list.toArray(a);
50     }
51 }

```

You can notice that `SpreadBuilder` [29] is a Java class and has a `ArrayList<Object>` `list` with all the elements and just adds all the elements to it. `toArray` method adds all these elements to the passed array. We can observe that it already supports almost all collections. What is left to add is the support for primitive arrays. We only have to add the following if branch [30] to `addSpread` method for each primitive type. This code checks the type of the container, casts it to this type and then adds all the elements to the `list` array field.

```
1  else if (container instanceof int[]) {
2      int[] array = (int[]) container;
3      if (array.length > 0) {
4          list.ensureCapacity(list.size() + array.length);
5          Collections.addAll(list, array);
6      }
7  }
```

Listing 30: `SpreadBuilder` extension

For primitive types, `PrimitiveSpreadBuilder` [31] is used. It is a templated interface, which takes a primitive array type as the only type parameter. It has an array of spread array arguments. `toArray(values: T, result: T): T` protected method takes two arrays: `values` - array of single non-spread elements, and `result` - the array to which the copying is performed. An example of this interface implementation for `Int` type can be found on the listing [32]. One can see that it defines a public method `toArray(): IntArray` that calls the `toArray(values: T, result: T): T` method from `PrimitiveSpreadBuilder` with single `Int` elements and a freshly created array of the required size.

```

1 public abstract class PrimitiveSpreadBuilder<T : Any>(private val size:
  ↳ Int) {
2     abstract protected fun T.getSize(): Int
3     protected var position: Int = 0
4
5     @Suppress("UNCHECKED_CAST")
6     private val spreads: Array<T?> = arrayOfNulls<Any>(size) as Array<T?>
7
8     public fun addSpread(spreadArgument: T) { spreads[position++] =
  ↳ spreadArgument }
9
10    protected fun size(): Int {
11        ...
12        return totalLength
13    }
14
15    protected fun toArray(values: T, result: T): T {
16        var dstIndex = 0
17        var copyValuesFrom = 0
18        for (i in 0..size - 1) {
19            val spreadArgument = spreads[i]
20            if (spreadArgument != null) {
21                if (copyValuesFrom < i) {
22                    System.arraycopy(
23                        values,
24                        copyValuesFrom,
25                        result,
26                        dstIndex,
27                        i - copyValuesFrom
28                    )
29                    dstIndex += i - copyValuesFrom
30                }
31                val spreadSize = spreadArgument.getSize()
32                System.arraycopy(spreadArgument, 0, result, dstIndex,
  ↳ spreadSize)
33                dstIndex += spreadSize
34                copyValuesFrom = i + 1
35            }
36        }
37        if (copyValuesFrom < size) {
38            System.arraycopy(
39                values,
40                copyValuesFrom,
41                result,
42                dstIndex,
43                size - copyValuesFrom
44            )
45        }
46
47        return result
48    }
49 }

```

```

1 public class IntSpreadBuilder(size: Int) :
  ↳ PrimitiveSpreadBuilder<IntArray>(size) {
2     private val values: IntArray = IntArray(size)
3
4     public fun add(value: Int) {
5         values[position++] = value
6     }
7
8     public fun toArray(): IntArray = toArray(values, IntArray(size()))
9 }
10

```

Listing 32: IntSpreadBuilder class

It's relatively easy to add the support of other collections here. We will only need to rewrite the base interface [31].

We need to be able to store not only primitive arrays as spread arguments, but any collection type. Since they don't have any common interface, we will use `Any` type [33].

```

1 private val spreads: Array<Any?> = arrayOfNulls(size)
2
3 public fun addSpread(spreadArgument: Any) {
4     spreads[position++] = spreadArgument
5 }

```

Listing 33: Changes in PrimitiveSpreadBuilder<T : Any> interface

The rest of the changes in `PrimitiveSpreadBuilder<T : Any>` are similar to the `SpreadBuilder`, we just cast the argument to the collection class and add all the elements to the storage.

3.1.3 Results

Now let's take a look at what we have achieved so far.


```

1 fun fooInt(vararg x: Int) {
2 }
3
4 fun <T>fooTemplate(vararg x: T) {
5 }
6
7 fun main() {
8     val a = intArrayOf(1, 2, 3)
9     val b = listOf(4, 5, 6)
10    val c = setOf(7, 8, 9)
11    val d = arrayOf(10, 11, 12)
12
13    fooInt(0)
14    fooInt(*a)
15    fooInt(*b)
16    fooInt(1, *d, 2, *c)
17
18    fooTemplate(0)
19    fooTemplate(*a)
20    fooTemplate(*c, *d)
21    fooTemplate(*b, 2, *c, 1)
22 }

```

Listing 34: With the first approach [3.1] this code compiles without errors

Listing [34] demonstrates the code example which will be possible to compile using the new compiler. It contains two variadic functions `fooInt(vararg x: Int)` and `<T>fooTemplate (vararg x: T)`. In the main function, several collections of types `IntArray`, `List<Int>`, `Set<Int>` and `Array<Int>` are initialized and passed to both variadic functions using spread operator in different combinations. The original compiler used to throw type incompatibility errors, but the prototype smoothly compiles and runs the code.

Such a solution takes the responsibility for calling array casts from the user and allows writing simpler code. However, it's not perfect, as it doesn't eliminate the type incompatibility problems [2.1.2]. In the next step, we will discuss the way to eliminate the type issues that occur when combining primitive and non-primitive variadic functions.

3.2 Using Boxed Arrays Only

This approach follows the previous idea [Allowing Spread Operator on Iterable Collections](#), but the only difference is that we will try to eliminate type issues by only using boxed arrays under the hood. And it rather a simple modification.

There is a special `FirCallableDeclaration.transformTypeToArrayType` extension method that is used only for converting the type of `vararg` parameters to array types. Inside it calls `ConeTypeProjection.createArrayType` method on its return type that takes a type and returns an array type with the passed type as an element type. It has a boolean parameter `createPrimitiveArrayTypeIfPossible` that defines whether it should create a primitive array for primitive types or not. The only thing left to do is to set this flag to false. Now the boxed array type always replaces the type of the variadic parameter.

This approach allows taking off users the responsibility of calling corresponding array casts, reduces copying and eliminates presented type incompatibilities.

The main drawback of this approach is that it breaks the backwards compatibility with the existing code. If the user worked with the `vararg` parameter as a variable of `IntArray` type, then the compiler will throw type errors, as this prototype will replace this primitive type with `Array<out T>` [35].

Listing [35] shows a function `someFunctionWithIntArray(x: IntArray)` that accepts an element of `IntArray` type. The listing also demonstrates a variadic function `foo(vararg x: Int)` that accepts a `vararg` argument of `Int` type and passes the `vararg` array to `someFunctionWithIntArray`.

```
1 fun someFunctionWithIntArray(x: IntArray) {  
2 }  
3  
4 fun foo(vararg x: Int) {  
5     someFunctionWithIntArray(x)  
6 }
```

Listing 35: Primitive arrays elimination creates type issues for this code

Additionally, this idea worsens the performance by using wrapper classes for primitive types. This approach is also not the most convenient one, as `Array<out T>` is still not a native Kotlin collection.

4 Evaluation and Analysis

In this chapter, we will state the criteria for a desired solution and evaluate our approach.

4.1 Criteria for the Solution

In the chapter 3 several compiler solutions were presented, each of them having its own advantages and drawbacks. However, we have to state the requirements for the possible solution.

First of all, the new compiler should be compatible with the existing code, as a lot of high-loaded systems rely on the existing implementation of variadic functions, most of them are not under our control.

Secondly, the prototype should not hide too many underlying details from the end-user, as, for example, creating additional copies.

The prototype must not worsen the performance in any way by introducing additional overhead.

The second solution 3.2 doesn't meet these criteria, as it breaks the type compatibility with the existing code and has a poorer performance due to the boxed array usage. That is the reason why we decided to only consider 3.1, which meets all of these requirements.

4.2 Performance Evaluation

The implemented prototype was evaluated based on several benchmarks.

The benchmarks' suite consisted of a set of various large *.kt* files. The files that didn't use variadic functions were used for benchmarking both the original compiler and our prototype.

As for variadic functions, a special script was written that generates a blank variadic function, and then in the `main` function it generates a number of random collections with arbitrary elements. Then it inserts calls to this one variadic function passing previously initialized collections via spread operator along with random single elements.

It's important to note that for the prototype, spread operator was used directly on the created collections without any casts [36]. However, for the original compiler these files were copied and changed such that there is a cast inserted for each collection on the call site [37].

```
1 fun callMe0(vararg a: Int) {  
2     println(a)  
3 }  
4  
5 fun main() {  
6     val var0 = setOf(901, 272, 869, 69, 597, 300, 207, 414, 922, 390)  
7     val var1 = listOf(616, 937, 123, 762, 483)  
8     val var2 = mutableListOf(947, 449, 994, 913)  
9     val var3 = arrayOf(635, 611, 128, 275, 939, 878)  
10    // More collections here  
11  
12    callMe0(*var0, *var1, *var2, *var3, 305, 35, 140, 504, 474, 213, 917)  
13    // More calls here  
14 }
```

Listing 36: Mock example of a test file for the prototype

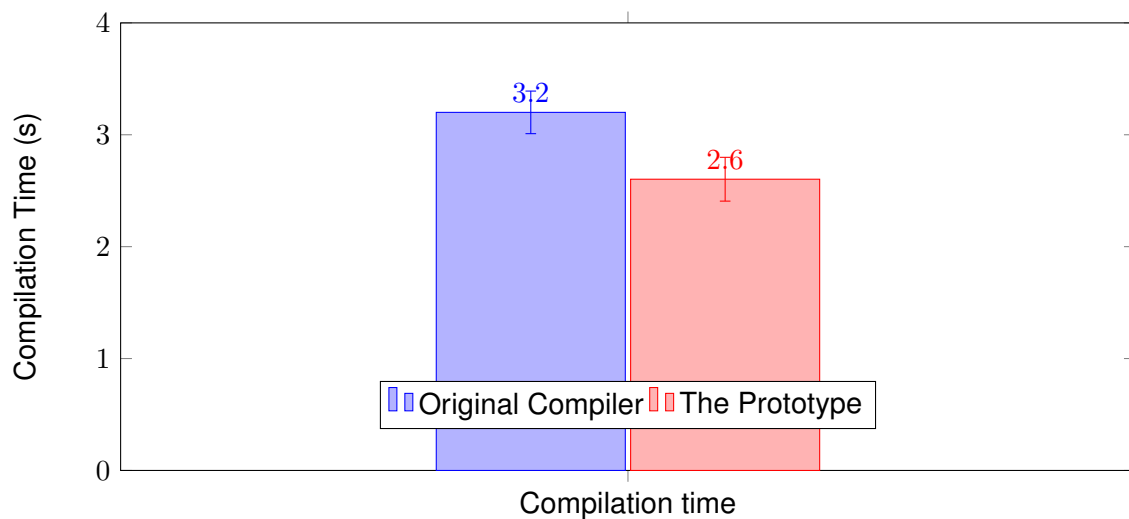
```

1 fun callMe0(vararg a: Int) {
2     println(a)
3 }
4
5 fun main() {
6     val var0 = setOf(901, 272, 869, 69, 597, 300, 207, 414, 922, 390)
7     val var1 = listOf(616, 937, 123, 762, 483)
8     val var2 = mutableListOf(947, 449, 994, 913)
9     val var3 = arrayOf(635, 611, 128, 275, 939, 878)
10    // More collections here
11
12    callMe0(
13        *var0.toIntArray(),
14        *var1.toIntArray(),
15        *var2.toIntArray(),
16        *var3.toIntArray(),
17        305, 35, 140, 504, 474, 213, 917)
18    // More calls here
19 }

```

Listing 37: Mock example of a test file for the original compiler

The first benchmark was the compilation time. For this, the compilation command `kotlinc test.kt` was used. For each file, the benchmark consisted of 5 warmup runs and 10 normal runs. After the execution, the average compile time for each file was taken.

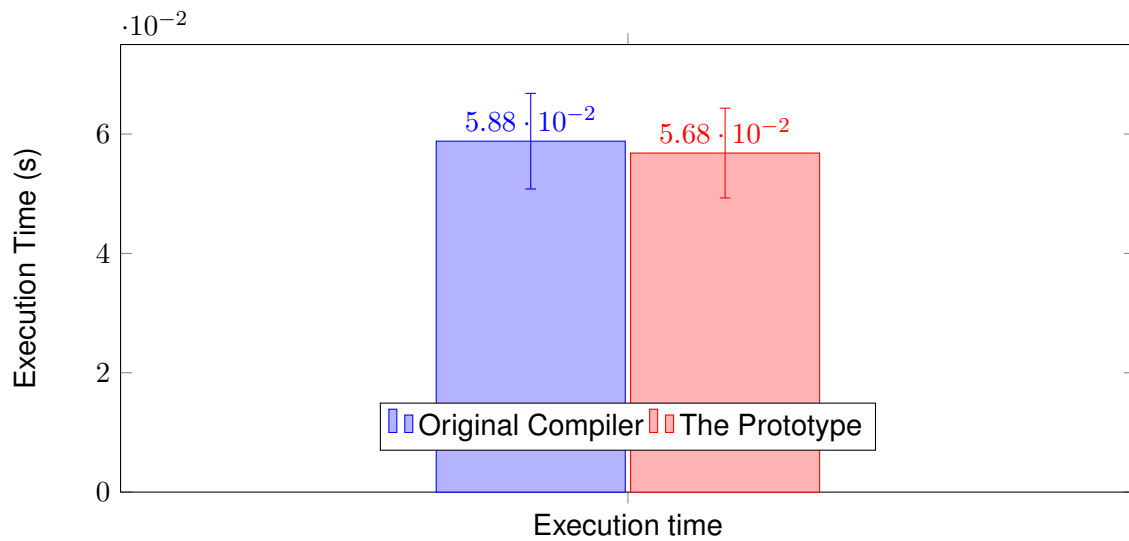


Listing 38: Average compilation time of large files with variadic functions

As we can observe [38], the compilation time was faster by 0.6 seconds on average. For the normal files, the compilation time was equal.

The execution time has also been considered. Firstly, each file was compiled using `/dist/kotlinc/bin/kotlinc-jvm test.kt -include-runtime -d test.jar`. After that, all of them were executed using `java -jar test.jar` command. The pipeline is the

same as for the compilation benchmark: 5 warmup runs and then 10 normal runs. The execution time on the prototype was a bit faster: 0.0568 seconds compared to 0.0588 [39].



Listing 39: Average execution time of large files with variadic functions

5 Conclusions

After the investigation of the problem, we were able to come up with several possible prototypes, each with its own advantages and drawbacks. After deep analysis and consideration of all of them, it was decided that the solution 3.1 meets our requirements the best. It significantly simplifies the usage of variadic functions and provides a much better performance by allowing the usage of spread operator on any collection type. As for the future work, it's vital to carefully test the compiler once again to ensure the lack of dangerous faults and to avoid presenting new issues, both in the `vararg` compilation aspect and in non-related processes. Furthermore, a detailed proposal for integrating this solution into the Kotlin compiler has to be created to smooth the user experience and to motivate developers to actively include variadic functions in their code.

References

- [1] cplusplus.com. *Printf documentation*. URL: <https://cplusplus.com/reference/cstdio/printf/>.
- [2] cplusplus.com. *Variable arguments handling*. URL: <https://cplusplus.com/reference/cstdarg/>.
- [3] JeneaVranceanu & ahinchman1 GitHub. *Kotlin Compiler Crash Course*. URL: <https://github.com/ahinchman1/Kotlin-Compiler-Crash-Course>.
- [4] JetBrains. *Kotlin Documentation, Variadic Functions*. URL: <https://kotlinlang.org/docs/functions.html#variable-number-of-arguments-varargs>.
- [5] Thinkage Ltd. *B Manual: Mechanics of Function Calls*. 1998. URL: https://www.thinkage.ca/gcos/expl/b/manu/manu.html#Section4_3.
- [6] Stanford University. *Exploiting Format String Vulnerabilities*. 2001. URL: <https://cs155.stanford.edu/papers/formatstring-1.2.pdf>.
- [7] Wikipedia. *Variadic Function*. URL: https://en.wikipedia.org/wiki/Variadic_function#In_C.
- [8] YouTrack. *KT-2462, Varargs*. URL: <https://youtrack.jetbrains.com/issue/KT-2462>.
- [9] YouTrack. *KT-9495, vararg and substitution of primitives for type parameters*. 2015. URL: <https://youtrack.jetbrains.com/issue/KT-9495/vararg-and-substitution-of-primitives-for-type-parameters>.