UTRECHT UNIVERSITY

MASTER'S THESIS

Not Too Local type inference

Author:
Maksymilian Demitraszek

Supervisor:
Alejandro Serrano Mena
Wouter Swierstra
Marco Vassena

A thesis submitted in fulfillment of the requirements for Master's degree

in the

Computing Science

July 22, 2024

UTRECHT UNIVERSITY

Abstract

Graduate School of Natural Sciences
Computing Science

Master's degree

Not Too Local type inference

by Maksymilian Demitraszek

The type inference in statically typed, object-oriented languages is local because it needs to know the type of an object when a method is called or a property accessed on it, as multiple classes can implement the method/parameter with the same name. This is necessary to determine the type of the method/parameter. We introduce a language Inferable Featherweight Java (IFJ) which models statically typed, object-oriented programs with type information inferred partially. We define the type inference progression property for IFJ programs. If a program holds this property it has sufficient information inferred, so the type inference algorithm can determine all types of methods/parameters. We prove that type variables substituting phantom type parameters and type parameters used only on contravariant positions do not have to be inferred for a program to hold this property. Furthermore, we use this result to extend the reference type inference algorithm based on the programming language Kotlin, so it does not fail if unable to infer certain type variables based only on the local context. We consolidate those results as a feature proposal for the programming language Kotlin and integrate it partially into Kotlin's compiler. The new type inference algorithm allows certain type annotations to be left out in programs, most notably when working with algebraic data types.

Contents

Chapter 1

Introduction

The most popular statically typed programming languages these days support type inference. It is a huge convenience for the programmer as it allows the compiler to automatically deduce type information, allowing the programmer to skip the explicit type annotations. It results in succinct code with less boilerplate. How powerful the type inference algorithm is and what annotations the programmer can omit may differ between languages.

Haskell is a functional programming language based on the Hindley-Milner type system. Because of that Haskell's compiler can infer types considering the global context. Introducing more advanced features, like arbitrary-rank types, makes type inference harder (Jones et al., 2007), and manual annotations from the programmer are sometimes required.

Kotlin and Scala are statically typed, object-oriented languages based on the Java virtual machine. Kotlin and Scala rely only on the local type inference where variables are inferred considering only a limited, local context. The compilers can infer the type information for the simple cases but often the programmer has to provide the type annotations manually. This limitation is dictated by several factors originating from the fact that those languages are object-oriented. The major factor is that object-oriented languages generally include subtyping, and with its presence type inference algorithms have huge time complexity (Hoang and Mitchell, 1995).

An example situation in Kotlin, when type inference based on a limited context is not sufficient, is when the programmer uses a common pattern of Algebraic Data Types (ADTs). The language does not support algebraic data types natively, but the programmer can emulate them using classes to emulate product types and inheritance hierarchies to emulate sum types. As an example, `Boolean` ADT could be defined as:

```
class Boolean
class True: Boolean
class False: Boolean
```

Classes `True` and `False` inherit from the class `Boolean`, which implies that both are subtypes of the class `Boolean`. It means that both can be used in places when the type `Boolean` is required.

Such a definition exposes a problem of local type inference when dealing with mutable variables.

```
var x = True() // infers type True for x
x = False() // type checking fails, False is not a subtype of True
```

The code presented above will not compile, as the type of `x` will be inferred as `True`, not `Boolean`. It is because the compiler infers the type of the variable considering only a local context, in this case, the context of a single statement. To fix this, the programmer has to provide an explicit type annotation `Boolean` in the first statement, as presented below.

```
var x: Boolean = True() // infers type True for x
x = False() // type checking fails, False is not a subtype of True
```

This code compiles. The programmer can work around this annotation requirement by providing helper functions to construct types, which would upcast the type to the supertype `Boolean`.

```
fun true(): Boolean { return True() }
fun false(): Boolean { return False() }
var x = true() // infers type Boolean for x
x = false() // type checking passes
```

This pattern is commonly used by Kotlin programmers and the native implementation of the sum types in Scala desugars to a very similar code.

It works for non-generic ADTs, but when the programmer would like to create a sum type with generic parameters, this workaround would be no longer viable. Consider an `Option` type.

```
class Option[T]
class Some[T](value: T): Option[T]
class None[T](): Option[T]
fun some[T](x: T): Option[T] { return Some(x) }
fun none[T](): Option[T] { return None() }
```

This type is either a value or nothing. The program below, that uses this type, fails.

```
// fails, cannot infer the type of the parameter T.
var x = none()
// this has information sufficient to infer the type of the type parameter T
x = some(5)
```

The compiler cannot infer the type of the parameter `T` considering only the context of the first statement. There is not enough type information. The programmer has to either manually provide the type argument to `none`:

```
var x = none<Int>()
x = some(5)
```

Or manually annotate the declaration of the variable `x` with `Option<Int>`:

```
var x: Option<Int> = none()
x = some(5)
```

to make this example work.

However, the information is there, if we have declared the type with a default value of the type *Some*, no annotation would be required.

```
// Infers the type of x as Option<Int>
var x = some(5)
x = none()
```

The example above works in Kotlin.

The problem with type inference for ADTs is the motivating example for our work. We explain the unique problems of type inference for object-oriented languages. We explain how those problems rationalize limitations in the inference algorithm. Later we propose a solution to the presented ADTs problem. Our contributions are:

1. We formally define a small language, Inferable Featherweight Java, which represents statically typed, object-oriented programs with type information inferred partially.
2. We formalize a reference type inference algorithm for Inferable Featherweight Java, based on the existing type inference algorithm in Kotlin.
3. We formalize a property - type inference progression. We prove that two subsets of Inferable Featherweight Java hold this property.
4. We extend the reference inference algorithm, using the type inference progression theory. We refer to the new algorithm as Not Too Local Type inference. The new algorithm is more complete than the original one, can type more programs, including the ADTs example.
5. We encapsulate the research results as a proposal for a feature of the Kotlin programming language. We partially implement it into the Kotlin compiler.
6. We provide a vast list of possible future work based on the foundations we laid.

Chapter 2

Literature Review

In this chapter we introduce the concepts which we refer to in the later chapters. We also discuss the related work.

2.1 The programming language Kotlin

Kotlin is an object-oriented, statically typed programming language targeting Java Virtual Machine (JVM). It is a well-established language used in different domains, such as mobile and web programming. In this section, we give a brief overview of some of the language's interesting features.

2.1.1 The overview

Kotlin supports class inheritance and interfaces with subtyping. Subtyping is a form of polymorphism that allows the use of the value of a particular type as an instance of its supertype. If A is a subtype of B, we write it as $A <: B$. Specifically, it means that one can use the type A in a context where an expression of type B is expected (Pierce, 2002). Inheritance hierarchies of classes and interfaces imply subtyping relations between types. A class can inherit only from one class but can implement multiple interfaces, as presented in fig. 2.1. In addition to classes, the programmer can also define standalone functions which are not class members.

Additionally, Kotlin supports parametric polymorphism. Parametric polymorphism allows a single piece of code to be typed "generically," the programmer can use type parameters instead of actual types. Type parameters, later, can be instantiated with specific types (Pierce, 2002).

```
interface A
interface B
open class C : A, B
class D : C
```

FIGURE 2.1: Inheritance in Kotlin

2.1.2 Method overloading

In Kotlin, multiple functions/methods can be defined with the same name but different type signatures and implementations. The compiler selects the most appropriate based on the type information it has.

```
class Containter {
    // ...
    fun add(val: Int) {
        // ...
    }
    fun add(val: String) {
        // ...
    }
}
```

For the example above, if the programmer calls the method `add` with an argument of type `Int`, the first implementation is executed. However, if they call the method with an argument that is a subtype of `Int` and `String`, then it would be ambiguous, and the compiler would return an error.

Kotlin's compiler can sometimes resolve such conflicts and pick the most appropriate one from multiple candidates. The interested reader can read further on that in the language specification (*Kotlin language specification* n.d.).

Because the method/function is selected based on the type information, the inference algorithm and overloading cut across. The impact of the type inference algorithm changes on the overloading is discussed in the chapter 7.

2.1.3 Mixed-site variance

Variance is the relation between types implied by the relation between its parameters. It is present in languages that support subtyping and parametric polymorphism (Igarashi and Viroli, 2002). The type can be invariant, covariant, contravariant, or bivariant on a type parameter.

1. The type $A[U]$ is covariant on the type parameter U when, if $X <: Y$, then $A[X] <: A[Y]$.
2. The type $A[U]$ is contravariant on the type parameter U when, if $X <: Y$, then $A[Y] <: A[X]$.
3. The type $A[U]$ is bivariant on the type parameter U when, if $X <: Y$, then $A[X] <: A[Y]$ and $A[Y] <: A[X]$.
4. If neither of those implications is true, we say the type is invariant on the type parameter U.

Another important characteristic of the variance is how it is integrated into the language from the programmer's perspective.

The authors of Mixed-site variance (Tate, 2013) propose and integrate the variants into the programming language Kotlin. They base their work on the limitations and problems in the previous approaches to the issue in Java, `C#`, and Scala.

```
class A<L, R> {}

fun main() {
    // The type of A() is invariant on L and R
    // The type of sth is covariant on L, invariant on R
    val sth: A<covariant Int, Int> = A()
}
```

FIGURE 2.2: Use-site variance

```
class A[covariant L, R] {}

fun main() {
    // The type of A() is covariant on L, invariant on R
    // The type of sth is covariant on L, invariant on R
    val sth = A<Int, Int>()
}
```

FIGURE 2.3: Declaration-site variance

Java implements the variance annotations use-site through wildcards fig. 2.2. To avoid introducing syntaxes of multiple languages, we present those different ideas in Kotlin-like pseudocode, using `covariant` and `contravariant` annotations. Variance annotations being use-site means that the programmer provides the variance annotation in the place where they use the type. This approach is very powerful and expressive. However, it often leads to unnecessary complexity and confusing behavior.

Based on this experience, `C#` and Scala adopted a more restrictive approach, declaration-site variance annotations. Declaration-site variance annotations mean that the programmer has to provide the variance annotations for the parameters in the declaration of the type fig. 2.3.

The authors of Mixed-site variance found this approach too restrictive, so they proposed an intermediary approach between the previous two. The mixed-site variance allows the programmer to provide the variance annotations in the declarations. However, those can be overridden use-site, as long as it is consistent with the declared variance fig. 2.4.

```
class A<L, contravariant R> {}

fun main() {
    // The type of A() is invariant on L, contravariant on R
    // The type of sth is covariant on L, invariant on R
    val sth: A<covariant Int, Int> = A() // A is
}
```

FIGURE 2.4: Mixed-site variance in Kotlin

In Kotlin, the programmer uses the keyword `out` to mark a covariant type parameter and `in` to mark a contravariant type parameter. Variance on type parameters in Kotlin imposes certain restrictions. Covariance, represented with the keyword `out`, restricts the programmer to use the type parameter only on output positions. This restriction means the programmer can only use those, for example, as a type of read-only value or a type returned by a function. Contravariance, represented with the keyword `in`, restricts the programmer to use type parameter only on input positions. This restriction means that it can be used, for example, as a function argument type (*Kotlin language specification* n.d.).

```
class A<L, in R> {}

fun main() {
    // The type of A() is invariant on L, contravariant on R
    // The type of sth is covariant on L, invariant on R
    val sth: A<out Int, Int> = A() // A is
}
```

2.1.4 Type inference

Type inference in Kotlin is based on the Local Type Inference (Dunfield and Krishnaswami, 2021; Odersky, Zenger, and Zenger, 2001; Pierce and Turner, 2000). Kotlin's type inference algorithm uses constraint sets. Mapping the algorithm onto the idea of Local Type Inference, all inference rules in the algorithm are type synthesis rules, but additionally to the type, a set of constraints is synthesized. The algorithm performs type-checking as a part of constraint solving. Locality comes from the fact that it does not solve the constraint set for the whole program, but only for local contexts e.g. a single statement.

There might be multiple solutions of a constraint system. In Kotlin's compiler, to help disambiguate it and guide the constraint solver, the type constraint set might include push-up $\uparrow T$ and push-down $\downarrow T$ constraints. Those imply that the resolved type should be respectively largest and smallest, according to the subtyping relation and complying with type constraints.

2.1.5 Smart casts

Kotlin's inference algorithm can derive some additional, flow-specific type information. It allows the programmer to avoid explicit type casts when the compiler can guarantee, based on the data-flow information, that a value conforms to a specific type.

```
class Boolean
class True: Boolean
class False: Boolean

val x: Boolean = True
if (x is True) { // in this scope x is assumed to be true
    val z: True = x // type checking passes
}
if (x is False) { // in this scope x is assumed to be true
```

```
    val z: False = x // type checking passes
}
```

In this case, the compiler assumes the type of `x` is `True` when analyzing the statements inside the if-clause. The programmer can use this mechanism to decompose ADTs.

2.1.6 Builder-style inference

We have mentioned before that the type inference in Kotlin is local. We have also introduced local inference shortcomings regarding ADTs. A similar problem with the local type inference occurs when the programmer tries to build a list, like in the code below.

```
var list = List()
list.add(1)
list.add(2)
```

Such code does not compile and requires an explicit type annotation in the first statement. It is because the type of list is `List<T>`, and in the first statement there is no sufficient type information to infer the type of `T`. The information that the type is `Int` is present in the second and third statements, but the local type inference in Kotlin is not able to infer it, because it only considers the context of a single statement.

Kotlin provides a workaround known as type-safe builders. In general, type-safe builders are a very flexible feature that the programmer can use to create semi-declarative domain-specific languages directly in Kotlin (*Kotlin documentation* n.d.). Due to their flexibility and complexity, we will skip an introduction of all the possibilities they enable and focus only on a short overview of the part relevant to our list example. With a type-safe list builder, the programmer could rewrite the previous code as

```
var list = buildList {
    this.add(1)
    this.add(2)
}
```

Kotlin can infer types for such code. Let's first decode this example a bit, before we explain how exactly it works. If we look at the definition of `buildList`, it is defined as

```
inline fun <E> buildList(
    builderAction: MutableList<E>.() -> Unit
): List<E> {
    var list = mutableListOf<E>()
    builderAction(list)
    return list
}
```

It is just a higher-order function that takes a function as an argument. The function `builderAction` takes a `MutableList<E>` as a receiver argument. Receiver argument is just an argument that binds to `this` keyword. Returned `Unit` type is a special type, which can be interpreted as it is lambda does return nothing.

Kotlin supports lambda expressions, the programmer can declare one as `{ a: Int , b: Int -> a + b }`. So, the code wrapped in the curly braces we pass as an argument in the example is just a lambda expression without any arguments.

Coming back to the original question, why does the type inference for such code work? The statement does not have sufficient information to infer the type parameter of `List`. Kotlin can, in this case, infer the type parameter of lambda, based on the lambda's body. It is unusual because we infer a generic type parameter of the function based on its body, which is supposed to be generic over this argument. It makes sense only in the context of lambda functions. It is known as builder-style inference. It only applies to type-safe builders. Interesting to note, the code below also would compile.

```
var list = buildList {
    this.add(1)
    this.add("example")
}
```

The builder inference resolves the type parameter based on the whole body of a lambda. It builds a shared constraint system. In this case, the type parameter is resolved to the least upper bound of `Int` and `String` (*Kotlin language specification* n.d.).

2.1.7 Algebraic Data Types

In this section, we discuss how programmers usually emulate the Algebraic Data Types in Kotlin. The `Option` type presented in the chapter 1 is a simplification. In Kotlin it usually would be represented as:

```
sealed class Option<out T> {}
class None: Option<Nothing>() {}
class Some<T>(val value: T): Option<T>() {}

fun none(): Option<Nothing> { return None() }
fun <T>some(value: T): Option<T> { return Some(value) }
```

First, the sum type is represented as `sealed` class. It means that the class cannot be instantiated and all of its subclasses have to be defined in the scope of the same library.

Another important difference is the covariance annotation `out`, and the fact that the definition does not include a type parameter for `none` and instead uses the `Nothing` type as a placeholder. It uses `Nothing`, because it is a subtype of every type, combined with the covariance it implies that `None` is a subtype of every instance of `Option`. Let's go over a few examples.

```
var x: Option<Int> = None()
```

It works because `None<Nothing> <: Option<Nothing>` and `Option<Nothing> <: Option<Int>` because of the covariance.

```
var x = none()
```

It works because `none` has no type parameters to infer.

```
var x = none() // x initialized as Option<Nothing>
x = some(1) // Option<Int> is not a subtype of Option<Nothing>
```

Unfortunately, the case from the chapter 1 still fails. The type of the variable is initialized as `Option<Nothing>`. Inference fails for the second statement as `Option<Int>` is not a subtype of `Option<Nothing>`, it is the other way around.

We see that this approach improves the situation slightly but does not solve the problem completely. It is also very complicated and confusing for programmers.

2.1.8 Language specification

Kotlin does not yet have a formalized core calculus akin to Featherweight Java (Igarashi, Pierce, and Wadler, 2001) or Featherweight Scala (Cremet et al., 2006). Strict formalization of the language that considers its distinctive features would require significant work and is outside the scope of this paper. Kotlin has language specification (*Kotlin language specification* n.d.), but it is more of a detailed overview of the language than a formal specification.

2.2 Type parameters categorizations and phantom type parameters

Type parameters were previously categorized to utilize properties of the particular categories.

In Safe zero-cost coercions for Haskell (Breitner et al., 2014), authors provide a method to automatically infer which types in Haskell can coerce into each other without breaking the soundness of the type system. As a foundation of the work authors present a framework of roles for type parameters assigning them into nominal, representational, and phantom roles. Their work is based on Generative type abstraction and type-level computation (Weirich et al., 2011).

Phantom type parameters are a special category of type parameters that are not used anywhere in a declaration of the type except as arguments to phantom type parameters for other type constructors. Those have various interesting applications. Phantom type parameters have been used to build type-safe Domain Specific Languages in Haskell (Leijen and Meijer, 2000) or to provide an additional type safety in Scala (Stucki, Biboudis, and Odersky, 2017).

2.3 Type inference in presence of subtyping

Type inference is a process of automatically deducing type annotations from the context. In this section, we review the development of the type inference algorithms, focusing on the type systems with the notion of subtyping.

2.3.1 Hindley-Milner type inference

Often discussed type inference algorithm is Algorithm W for the Hindley-Milner type system (Milner, Morris, and Newey, 1975). Languages with type systems based on the extended Hindley-Milner type system, like Haskell or Rust, enjoy a more complete type inference algorithm. This comes with a set of limitations. One of them is the support of subtyping, which those languages have very restricted.

There have been approaches to extend the Hindley-Milner type system with the notion of subtyping. One of the approaches was extending the types with an associated set of constraints (Odersky, Sulzmann, and Wehr, 1999; Pottier, 1996, 1998; Trifonov and Smith, 1996). The produced constraint sets in this approach tend to be colossal. The papers in this line of work often contemplate solutions to simplify those constraint sets.

Constraint-based type inference utility is not limited to the type systems with the notion of subtyping. (Heeren, Hage, and Swierstra, 2003) used a constraint-based type inference in the Haskell compiler Helium, to improve the error messages provided to the programmer. (Vytiniotis et al., 2011) used constraint-based approach to extend the type inference with support for GADTs, type classes, and type families.

A more recent approach to integrate subtyping into the Hindley-Milner type system is Polymorphism, subtyping, and type inference in MLsub (Dolan and Mycroft, 2017). The authors present a type system with parametric polymorphism and algebraic subtyping. They present an inference algorithm for the type system. Additionally, the authors map the type simplification problem into automata theory, allowing the usage of any algorithm for automata simplification to simplify the types. The work presented in the paper was heavily theoretical. To encourage the adoption the authors of The Simple Essence of Algebraic Subtyping (Parreaux, 2020) provided a more accessible explanation of the techniques, not relying on the complex concepts from abstract algebra.

Another part of the work to encourage the adoption of this type system is to improve the error messages in the inference algorithm. The authors of Getting into the Flow (Bhanuka et al., 2023) introduce a technique to improve the error messages in the inference algorithm for the type system with algebraic subtyping. The authors take inspiration from the constraint-based inference algorithms with a notion of subtyping.

2.3.2 Bidierctional type inference

Type inference for $F_{\leq}$, the system F with subtyping and parametric polymorphism had been researched, but it did not result in any real-life language applications.

The breakthrough was Local Type Inference by Pierce and Turner (Pierce and Turner, 2000). The authors identified that, in practice, there is no need for a complete type inference algorithm. Type annotations in function signatures serve as a source of documentation. On the other hand, there are certain places in which manual type annotations are the cause of most of the overall inconvenience, namely local variables declarations, anonymous function arguments, and type arguments provided to functions. The authors propose two kinds of inference rules, type synthesis (inference) and type checking. They combine type propagation with constraint solving to infer the type parameters for function calls. This leads to an algorithm which, even though, is not complete, is very comfortable to work with in practice, which authors prove in a quantitative study.

Local Type Inference has been developed further in Colored Local Type Inference (Odersky, Zenger, and Zenger, 2001). Contrary to the previous work, the authors propose type inference rules where the type does not synthesize/check as a whole, instead different parts of the type can either synthesize or check. The authors mark if a part of a type synthesizes or checks using colors, which is the origin of the paper's title.

(Jones et al., 2007) use bidirectional type inference ideas to improve the type inference algorithm in Haskell to support arbitrary-rank types.

The idea of bidirectional typing and the past research on it has been recently summarized in (Dunfield and Krishnaswami, 2021). The idea of bidirectional typing and type information propagation influenced the design of many modern programming languages including Scala and Kotlin.

2.4 Weak type variables

The programming language OCaml introduces a concept of weak type variables (*OCaml Weak Type Variables* n.d.). At first glance, those might seem related to the problem of ADTs described in the introduction. In OCaml, the programmer can encounter weak type variables, among others, when initializing algebraic types under mutable references. In the example below the type `None` has a generic type parameter but it is not able to infer it from the constructor.

```
let x = ref None ;;
(* val x : '_weak1 option ref = {contents = None} *)
x:= Some 1 ;;
(* val x : int option ref = {contents = Some 1} *)
```

This compiles, first `x` is initialized with generic weak type variable `'_weak1`. Then this type variable is fixed greedily when the reference is mutated. That is the difference between weak and normal type variables, as a normal type variable is not fixed when the value is applied, it remains generalized.

This concept was introduced in Relaxing the Value Restriction (Garrigue, 2004). However, the problem the authors deal with is extending the type system, not the type inference algorithm. They extend the type system to relax the value restriction, without breaking the soundness property. Value restriction is a solution to the problem that is introduced when adding mutable references to a language. Mutable reference cannot hold a generalized list as it would mean anything could be written into it, even though the type would remain generalized. Weak type variables are in a way, externally exposed inference variables, which are greedily fixed. (Garrigue, 2004) achieve it by proving that the bottom type (the subtype of every type) on covariant positions, can be recovered into a type variable without compromising the soundness of the type system.

Because of that, even though it at first sight might look like a similar problem to ours, it is not. The Kotlin's type system is, by design, different. It does not allow generic types of variables at all. However, interesting to note that (Garrigue, 2004), akin to us, exploits properties of specific type parameters and serves us as an inspiration for our ideas.

Worth noting is that further in this paper we use the name type variable for a slightly different concept. It is not the same thing as in this section, those should not be confused.

Chapter 3

Inferable Featherweight Java

In this chapter, we introduce a small language that we use throughout the whole thesis. The language is heavily influenced by Generic Featherweight Java (GFJ) (Igarashi, Pierce, and Wadler, 2001), which is Featherweight Java extended with parametric polymorphism. Our language is also influenced by Inherently-typed Featherweight Java (Feitosa et al., 2019), where the program contains the set of predefined classes (a class table), which are assumed to be correct by definition. This work introduces the novel concept of type variables included in programs explicitly. The purpose of the language is the type inference algorithms analysis, which is why we refer to our language as Inferable Featherweight Java (IFJ).

Our language is a somewhat restricted version of GFJ to the set of features that concern us, then extended with imperative programs and explicit type variables. We introduce a new language instead of using GFJ for two main reasons:

- The GFJ does not include imperative programs. In GFJ a body of a method is a single expression.
- Our language makes it much easier to formalize our ideas as we are not concerned with the language's type system, semantics, or program's well-formedness. Additionally, decoupling from elements we do not rely on makes our results much more general.

3.1 The overview

Every program in IFJ consists of two parts, declarations fig. 3.1 and a list of statements fig. 3.2. The declarations part consists of definitions of classes. Those describe all classes available in the scope of the program. We do not define the semantics of the language, it is designed to analyze the type inference algorithms, thus we are only interested in types. However, for a better intuitive understanding, the reader can imagine semantics being very similar to any existing, statically typed, object-oriented language. Another consequence of the fact that we are only interested in types is that declarations of classes do not include the method bodies, only the signatures.

All the classes from the declarations part are available in the scope of the list of statements. Here is an example program in IFJ.

```
// Declarations
class Any {}
```

```
class Pair: Any {
    first: Any
    second: Any

    Pair(first: Any, second: Any) {}

    fun get_first(): Any {}
    fun get_second(): Any {}
}

class Elem: Any {}

// Statements
var x = Pair(Pair(Elem(), Elem()), Elem())
return x.get_first()
```

Classes can inherit from each other. If a class inherits from another class it means that it acquires all the properties and methods of the class it inherits from. Every class inherits from zero or one other class. In the example above classes `Pair` and `Elem` inherit from the class `Any`.

Classes inheritance hierarchies imply subtyping relations between their corresponding types. If a class inherits from another class, then it is a subtype of this class. A subtype can be used in places where its supertype is required. In the example above types `Pair` and `Elem` are subtypes of the type `Any`.

If a class implements the same parameter or method as its parent, the parent's parameter or method is overridden. To override a property its type has to be a subtype of the corresponding parent's property type. To override a method its arguments have to be supertypes of the corresponding parent's method arguments types. Its return type has to be a subtype of the corresponding parent's method return type.

In the second statement, the method `get_first` is called on the `Pair` object. We refer to the expression on which a method is called, or a property accessed as *receiver*. In the second statement, of the example above, `x` is the receiver expression, and `Pair` is the type of the receiver.

IFJ supports parametric polymorphism. When defining a class or a class's method, type parameters can be introduced and used instead of specific types. Those can later be instantiated with concrete types. A type representing pairs can be implemented using parametric polymorphism, as presented in the example below.

```
// Declarations
class Pair[L, R] {
    first: L
    second: R

    Pair(first: L, second: R) {}

    fun get_first(): L {}
    fun get_second(): R {}
}
```

```
class Elem {}

// Statements
var x = Pair[Pair[Elem,Elem], Elem](Pair[Elem, Elem](Elem(), Elem()), Elem())
return x.get_first()
```

As we see in the example above, in IFJ type arguments in the constructor and method calls have to be provided explicitly.

3.2 Type variables

In the previous section, we mentioned that type arguments in the constructor and method calls have to be provided explicitly. Though, the language purpose is to analyze the type inference, hence we need to make it possible to omit explicit type parameters, so the inference algorithm can infer them. That is why explicit type variables are introduced. A type variables are different from type parameters. Type parameters appear only in the declarations part, and represent *any* type. Type variables, on the other hand, can only be used in the method and constructor calls in the statements part of the program. Type variable is a specific, fixed type, which is just unknown. Type variables are akin to unification variables in Algorithm W.

The goal of the type inference, which is described later, in the chapter 4, is to replace all type variables in an IFJ program with concrete types, in a way that results in a sound program. We assume that every type variable in the statements part is unique for a single program.

If we would like the type arguments to be inferred by the inference algorithm, the `Pair` constructor from the previous section could be reformulated as:

$$\texttt{var x} = \texttt{Pair}[\alpha_0, \alpha_1](\texttt{Pair}[\alpha_2, \alpha_3](\texttt{Elem}(), \texttt{Elem}()), \texttt{Elem}())$$

For a type used in a statement, to substitute a type parameter, we refer to as a type argument. In the example above type arguments are $\alpha_0, \alpha_1, \alpha_2, \alpha_3$.

Other papers discussing local type inference algorithms (Odersky, Zenger, and Zenger, 2001; Pierce and Turner, 2000) rely on two languages, external and internal, where in the external one, type arguments can be omitted. The type inference is a process of mapping the external language to the internal one. We diverge from this model and make type variables explicit because it allows us to represent partially inferred programs, which proves itself useful in further sections. It captures the type inference as a process, with programs with more and less information inferred, instead of just two states, not inferred and inferred.

3.3 IFJ formulation

We have introduced the language on a high level, in this section we provide a formalization of the previously introduced concepts. In the whole work, we often use $\overline{x}$ shorthand of a sequence $x_1, \ldots, x_n$, where x can be replaced with any object.

Cld ::= class C$[\overline{X}]$: ρ { $\overline{p}$: $\overline{\tau}$ Cod $\overline{Md}$ }	Class Declaration
$Cldni$ ::= class C$[\overline{X}]$ { $\overline{p}$: $\overline{\tau}$ Cod $\overline{Md}$ }	Class Declaration Without Inheritance
Cod ::= C($\overline{p}$: $\overline{\tau}$)	Constructor Declaration
Md ::= fun $[\overline{X}]$ m($\overline{a}$: $\overline{\tau}$): τ	Method Declaration

FIGURE 3.1: Class declaration

3.3.1 Declarations

The declarations part is a set of declared classes fig. 3.1. Each class consists of zero or more properties with explicit type annotations. A class contains a single constructor with arguments equal to the class's properties. It also consists of zero or more methods with explicit type signatures. Each class has an arbitrary number (greater or equal to zero) of generic type parameters. Each method also has an arbitrary number (greater or equal to zero) of generic type parameters. Method bodies are not included in the language. In a well-formed program, if a class is inherited from, then it is declared. Class declarations do not contain type variables in the type signatures.

3.3.2 Statements

A statement fig. 3.2 is either a variable declaration with or without a type annotation, a variable assignment, or a return statement. Variable declaration and assignment statements have an expression whose value binds to the variable. An expression fig. 3.3 is either a property access, a variable access, a method, or a constructor call. If there is a constructor call in the program, then the corresponding class declaration exists. Method and constructor calls require providing type parameters explicitly, though type variables can be used there. Type parameters cannot be used in the statements part of the program. Type variables cannot be used in the variable declaration's explicit type annotation. Every program also includes an expected return type which does not contain type variables or parameters.

3.3.3 Types

Type is either a type parameter, type variable, or class type fig. 3.4. If a type contains type variables we refer to it as a partially inferred type. We refer to a type without type variables as a fully inferred type. If type $A[]$ is a subtype of the type $B[]$, we write it as $A[] <: B[]$. We mentioned in the first section that subtyping relations are implied by inheritance. However, this is only specified to help the reader build an intuitive understanding. We do not rely on any semantics of subtyping relation in this work. Actually, we do not rely on the subtyping relation at all, we just rely on *some relation* that the inference algorithm uses to represent the type constraints. We refer further to the subtyping relation $<:$, though keep in mind it can be substituted with another relation.

3.3.4 Helper functions

We also introduce a set of utility functions that are used in the later chapters.

$Body ::= acc = e; Body$	Assignment
$\mid \; var\ v : \tau = e; Body$	Declaration with annotation
$\mid \; var\ v = e\ ; Body$	Declaration
$\mid$ return e	Method return

$acc ::= v.acc$
$\mid \; v$

FIGURE 3.2: Statements

$e ::= e.m[\overline{\tau_\diamond}](\overline{e})$	Method call
$\mid \; e.p$	Parameter access
$\mid$ `new` $C[\overline{\tau_\diamond}](\overline{e})$	Constructor call
$\mid \; v$	Variable access

FIGURE 3.3: Expressions

$\tau, \gamma, v ::= X$	Type parameter
$\mid \; \rho$	Class type
$\rho ::= C[\overline{\tau}]$	Class type
$\tau_\diamond, \gamma_\diamond, v_\diamond ::= \alpha$	Type variable
$\mid \; \rho_\diamond$	Partial class type
$\rho_\diamond ::= C[\overline{\tau_\diamond}]$	Paritial class type

FIGURE 3.4: Types

1. $TV : \tau \to$ `Set`$[\alpha]$ - returns all type variables in the given type, including type variables in the nested types.
2. `type_params` $: C \to \overline{X}$ - returns the list of the type parameters of the given class
3. `prop_type` $: C, p \to \tau$ - returns the type of the given property of the given class
4. `prop_types` $: C \to \overline{\tau}$ - returns the list of the types of the type properties of the given class
5. `method_type` $: C, m \to \overline{X}, \overline{\tau}, \tau$ - returns the signature of the given method. It returns respectively: type parameters, type arguments, and return type.
6. `return_type` $: () \to \tau$ - returns the return type of the program

3.3.5 Substitutions

Substitution traverses a type or an expression recursively replacing types with defined in the substitution, corresponding types. We represent a substitution as σ and the substitution application as σe. Sometimes we represent a substitution as $[x/y]$, which means that it replaces occurrences of y with x. Substitutions can be composed together using the $\circ$ operator.

3.3.6 Free functions

Even though the core language does not include free functions (functions that are not members of any class), for brevity, we sometimes use those in declarations in further examples. Free functions can be easily desugared to a class without any properties, containing all of them as methods.

Chapter 4

Type inference in statically typed, object-oriented languages

In this chapter, we dive into the characteristics of type inference in statically typed, object-oriented languages. We model a type inference algorithm for IFJ. The described type inference algorithm is based on the existing type inference algorithm in the programming language Kotlin. Even though the algorithm is based on Kotlin, we try to capture the essence of the type inference in statically typed, object-oriented languages, thus the introduced concepts also apply to languages like `C#` and `Scala`. We explain what unique problems for the type inference are associated with this paradigm. We explore those problems and later dive into possible solutions.

4.1 Kotlin inspired type inference overview

The type inference's goal is to map an IFJ program to the corresponding, type correct, IFJ program without type variables. The inference algorithm is defined by the translation relation:

$$Body \rightarrow Body$$

It takes a program and returns a new program with all type variables replaced with fully inferred types. It uses two important components, constraint generation and constraint solving.

Constraint generation is defined by the inference relation:

$$e \Rightarrow \tau_{\diamond}, \mathbb{C}, \sigma$$

For an expression e, it infers a partially inferred type $\tau_{\diamond}$ of the expression, with corresponding set of constraints $\mathbb{C}$. It also includes a substitution σ which might contain information learned during the inference. The substitution maps from the type variables to fully inferred types.

The constraint solver is defined by the function

$$\texttt{solve} : \mathbb{C}, \texttt{Set}[\alpha] \rightarrow \sigma$$

It takes a set of constraints and a set of all type variables that have to be fixed. It returns a substitution.

4.2 Constraint generation

First, we look into constraint generation for a single expression in IFJ. Let's go back to the Pair example from the chapter 3.

$$\texttt{Pair}[\alpha_0, \alpha_1](\texttt{Pair}[\alpha_2, \alpha_3](\texttt{Elem}(), \texttt{Elem}()), \texttt{Elem}())$$

The algorithm traverses the AST bottom-up.

$$\texttt{Elem}() \Rightarrow \texttt{Elem}, \varnothing, \varnothing$$

For the `Elem()` expression the type is known, it is `Elem`, and the same goes for the second one. There are no constraints or substitution generated.

$$\texttt{Elem}() \Rightarrow \texttt{Elem}, \varnothing, \varnothing$$
$$\texttt{Elem}() \Rightarrow \texttt{Elem}, \varnothing, \varnothing$$

$$\texttt{Pair}[\alpha_2, \alpha_3](\texttt{Elem}(), \texttt{Elem}()) \Rightarrow \texttt{Pair}[\alpha_2, \alpha_3], \{\texttt{Elem} <: \alpha_2, \texttt{Elem} <: \alpha_3\}, \varnothing$$

For the `Pair` constructor call, we know the types of the arguments, though we do not know what are the type parameters of the constructor. For this program to be sound, types of arguments have to be subtypes of the expected types of arguments. Thus, the relevant constraints are generated. The algorithm returns the type containing type variables, the constraints set, and an empty substitution.

$$\texttt{Pair}[\alpha_2, \alpha_3](\texttt{Elem}(), \texttt{Elem}()) \Rightarrow \texttt{Pair}[\alpha_2, \alpha_3], \{\texttt{Elem} <: \alpha_2, \texttt{Elem} <: \alpha_3\}, \varnothing$$

$$\texttt{Elem}() \Rightarrow \texttt{Elem}, \varnothing, \varnothing$$

$$\texttt{Pair}[\alpha_0, \alpha_1](\texttt{Pair}[\alpha_2, \alpha_3](\texttt{Elem}(), \texttt{Elem}()), \texttt{Elem}()) \Rightarrow \texttt{Pair}[\alpha_0, \alpha_1] \{\texttt{Elem} <: \alpha_2, \texttt{Elem} <: \alpha_3, \texttt{Elem} <: \alpha_1, \texttt{Pair}[\alpha_2, \alpha_3] <: \alpha_0\}, \varnothing$$

The same idea follows for the top-level constructor call, the constraints from the subexpressions are merged into the returned set with the new constraints.

The algorithm can pass the generated constraints to the constraint solver. The constraint solver receives a set of constraints and, based on it, builds a substitution function, that maps all type variables to fully resolved types. Adherence to the constraints guarantees type correctness. Sometimes constraints might be contradictory or insufficient to build a substitution. In such a case, the constraint solver returns an error and it fails the whole algorithm. If we apply the substitution to the inferred partial type of an expression, the result is a fully inferred type for the expression. If we apply the substitution to the expression, the result is an expression in IFJ with all type information resolved.

4.3 Constraints generation for expressions with receivers

In the previous section, we have shown the high-level idea of the constraint generation for a single expression. However, the example expression in the previous section consists only of constructors and does not include method calls or property accesses. Both make things slightly more complicated. In this section we stay in the context of the constraint generation for a single expression, though this time we discuss one, that includes method calls and properties accesses. The core problem we face is the fact that multiple classes might implement methods or properties with the same name. Let's introduce a new example:

```
class ClassOne {
    fun foo() -> Int {}
}

class ClassTwo {
    fun foo() -> String {}
}

class Functions {
    fun identity[T](x: T): T {}
}
```

We have two classes `ClassOne` and `ClassTwo` implementing the method `foo` but with different type signatures. We also have a class `Function` which holds a helper, free function, `identity`. Given an expression

$$\texttt{Function}().\texttt{identity}[\alpha_0](\texttt{ClassOne}()).foo()$$

Let's try to apply the algorithm from the previous section.

$$\texttt{Function}() \Rightarrow \texttt{Function}, \varnothing, \varnothing$$

The type of the `Functions()` constructor call is trivially inferred without any constraints.

$$\texttt{Function}() \Rightarrow \texttt{Function}, \varnothing, \varnothing$$

$$\texttt{ClassOne}() \Rightarrow \texttt{ClassOne}, \varnothing, \varnothing$$

$$\texttt{Function}().\texttt{identity}[\alpha_0](\texttt{ClassOne}()) \Rightarrow \alpha_0, \{\texttt{ClassOne} <: \alpha_0\}, \varnothing$$

We know the type of the receiver `Functions()` thus, we can deduce the type signature of the method based on the declaration. As `method_type(Functions, identity)` $= T, T, T$ thus, the algorithm progresses forward, including the new type constraint from the method argument

$$\texttt{Function}().\texttt{identity}[\alpha_0](\texttt{ClassOne}()).\texttt{foo}()$$

However, in the next step it runs into a problem, we know that the type of the receiver is α_0. Based on that we cannot determine the signature of the method `foo`. There are multiple classes implementing this method which makes this call ambiguous. As the signatures are different, progressing with each of them later could further lead to completely different constraint sets and results. Some of them could

later turn out contradictory and be rejected, though the complexity of such a process would be exponential, the result often would be ambiguous and the outcome could often be confusing for the programmer who could have different intentions.

We do not need our type inference algorithm to be able to type all IFJ programs. That is why we make the algorithm greedy. It executes the constraint solver on the constraint set inferred from the receiver type, fixing all type variables. The type of the receiver is inferred locally, based only on the information collected from the receiver expression. If it is not possible, such a program is rejected.

$$\sigma = \texttt{solve}(\{\texttt{ClassOne} <: \alpha_0\}, \{\alpha_0\})$$

For this example, constraint solver produces a substitution $\sigma = \{\alpha_0 \to \texttt{ClassOne}\}$, which allows the algorithm to progress forward as

$$\sigma\alpha_0 = \texttt{ClassOne}$$

and

$$\texttt{method\_type}(\texttt{ClassOne}, foo) = () \to \texttt{Int}$$

$$\texttt{Function}().\texttt{identity}[\alpha_0](\texttt{ClassOne}()).\texttt{foo}() \Rightarrow \texttt{Int}, \varnothing, \{\alpha_0 \to \texttt{ClassOne}\}$$

It further does not return the constraints as those are already resolved. Instead, it returns the created substitution.

Looking at this case closer, we do not need all type variables in the receiver fixed to determine the method/parameter type. We just need the inferred type of the receiver to be in a form such top-level class is fixed, though there might be remaining type variables as type parameters. We refer further to such form of a partially inferred type as *fixed-head form*.

4.4 Inference for programs with multiple statements

Now let's apply the constraint generation and constraint solver concepts to formulate the translation relation of the whole inference algorithm. Let's go back again to the `Pair` example, take the program with the return type `Any`.

```
var x = Pair[α0, α1](Pair[α3, α4](Elem(), Elem()), Elem())
return x.get_first()
```

In the first statement, from the section 4.2 we know that:

$$\texttt{Pair}[\alpha_0, \alpha_1](\texttt{Pair}[\alpha_2, \alpha_3](\texttt{Elem}(), \texttt{Elem}()), \texttt{Elem}()) \Rightarrow \texttt{Pair}[\alpha_0, \alpha_1]$$
$$\{\texttt{Elem} <: \alpha_2, \texttt{Elem} <: \alpha_3, \texttt{Elem} <: \alpha_1, \texttt{Pair}[\alpha_2, \alpha_3] <: \alpha_0\}$$

Variable `x` declaration does not include an expected type annotation, so we pass the constraint set to the constraint solver in the same form as it was generated from the expression. If the statement included an expected type, would be a variable assignment or a return statement we would add a constraint $\texttt{Pair}[\alpha_0, \alpha_1] <: \tau$, where τ is respectively the expected type, a type of the variable or the predefined return type. The constraint solver based on the constraints produces a substitution σ, which maps the type variables to fully inferred types. We set the type of the variable `x` as $\sigma\texttt{Pair}[\alpha_0, \alpha_1]$. We map the statement to $\texttt{var x} : \sigma\texttt{Pair}[\alpha_0, \alpha_1] = \sigma e$. For the second

statement:

$$\texttt{x.get\_first()} \Rightarrow \texttt{Pair[Elem, Elem]}, \varnothing, \varnothing$$

type of x is included in the context so it infers the type `Pair[Elem, Elem]` with an empty constraint set. The algorithm adds a constraint $\{$ `Pair[Elem, Elem]` $<:$ `Any` $\}$ from the return type and passes those to the constraint solver, which returns an empty substitution σ_1. Then maps the statement to `return` σ_1 `x.get_first()`, which in this case is the same as the original. The only thing that the algorithm does for this statement, is that it validates if the expression type is a subtype of the return type.

An important question is why we execute the constraint solver for the statements without explicit expected type annotations. Instead, we could keep the inferred type with constraints in the context, collect more constraints in subsequent statements, and solve those for the whole program. There are three main reasons why this decision was taken for the Kotlin inference algorithm.

1. **Errors far away from the issue** - If in the subsequent statement variable is used in a receiver, the inference algorithm will still force the fix. This might be confusing for the programmer if, for example, a method call or a parameter access would trigger an inference error a few statements above the modified code.

2. **Inflated types of variables** - If a language includes type `Any`, a supertype of every type, variable would often be resolved to it. In practice, it is better to keep the variable type fixed. The programmer often wants type system to keep them in check, validating if the type of value assigned to the variable is correct, instead of accepting everything and resolving the type of the variable to `Any`. An example case is included below.

   ```
   var x = 1
   x = None()
   ```

   In this case, the type of `x` would be resolved to `Any`, which is not desired. The programmer would generally prefer to have the type of `x` resolved to `Int`, and have the second statement rejected.

3. **Computational complexity** - Constraint solving cost can grow exponentially, thus collecting the constraints for the whole function could have a big impact on the performance of the type inference algorithm.

4.5 Inference algorithm rules

Those informal descriptions from this and previous sections lead us to the formal definition of the inference relation for IFJ fig. 4.1, constraint solver and the translation relation fig. 4.2 which encapsulates the whole inference algorithm. The presented algorithm logic is based on the Kotlin type inference algorithm, thus we further refer to this algorithm as Kotlin inspired type inference.

One thing that might seem controversial is the fact that we do not discuss the correctness of the proposed algorithm. The goal of our work is to present a possible adjustment that could be introduced to an existing type inference algorithm in Kotlin, or other similar language. The fact that we formalize this algorithm on our

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v \Rightarrow \tau, \varnothing, \varnothing}\ \textsc{CGen-LocalVar}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau_\diamond, \mathbb{C}, \sigma \qquad \sigma' = \texttt{solve}(\mathbb{C}, TV(\sigma e)) \qquad C[\overline{\tau}] = \sigma'\sigma\tau_\diamond \qquad \texttt{prop\_type}(C, p) = v}{\Gamma \vdash e.p \Rightarrow [\overline{\tau}/\texttt{type\_params}(C)]v,\ \varnothing, \sigma' \circ \sigma}\ \textsc{CGen-Prop}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow \tau_\diamond, \mathbb{C}, \sigma \qquad \sigma' = \texttt{solve}\,(\mathbb{C}, TV(\sigma e)) \\ C[\overline{\tau}] = \sigma'\sigma\tau_\diamond \qquad \texttt{method\_type}(C, m) = \overline{X}, \overline{\gamma}, \gamma \qquad \Gamma \vdash \overline{e} \Rightarrow \overline{\tau_\diamond}, \overline{\mathbb{C}}, \overline{\mathbb{D}}, \overline{\sigma''}\end{array}}{\begin{array}{l}\Gamma \vdash e.m[\overline{v_\diamond}](\overline{e}) \Rightarrow [\overline{v_\diamond}/\overline{X}][\overline{\tau}/\texttt{type\_params}(C)]\gamma, \\ \left(\bigcup_{i \in 1\ldots n} \mathbb{C}_i\right) \cup \{\overline{\tau_\diamond} <: [\overline{v_\diamond}/\overline{X}][\overline{\tau}/\texttt{type\_params}(C)]\overline{\gamma}\},\ \circ_{i \in 1\ldots n}\sigma_i'' \circ \sigma' \circ \sigma\end{array}}\ \textsc{CGen-MCall}$$

$$\frac{\Gamma \vdash \overline{e} \Rightarrow \overline{\tau_\diamond}, \overline{\mathbb{C}}, \overline{\mathbb{D}}, \overline{\sigma}}{\begin{array}{l}\Gamma \vdash \texttt{new}\ C[\overline{v_\diamond}](\overline{e}) \Rightarrow C[\overline{v_\diamond}], \\ \left(\bigcup_{i \in 1\ldots n} \mathbb{C}_i\right) \cup \{\overline{\tau_\diamond} <: [\overline{v_\diamond}/\texttt{type\_params}(C)]\texttt{prop\_types}(C)\}, \circ_{i \in 1\ldots n}\sigma_i\end{array}}\ \textsc{CGen-Constr}$$

FIGURE 4.1: Kotlin inspired type inference, expressions

language serves merely as a reference point for our work, based on which we provide the adjustments. We argue that this lack of proved correctness is not a limitation of our work. It is an advantage, as the results we prove in the next chapter do not rely on semantics of the constraint generation, only on its control flow, thus those can be applied to any object-oriented language with any type system that implements a type inference algorithm with the comparable control flow. If the base algorithm generates the sound constraints, the modified version preserves this property.

We define the type environment as

$$\Gamma : v \to \tau$$

It takes a variable name and returns a fully inferred type. Let's first look into the proposed inference rules fig. 4.1 for expressions.

For a variable access CGEN-LOCALVAR, it just accesses the type from the environment and returns it with an empty set of constraints and empty substitution. We know that all types in the environment are fully inferred.

For a property access CGEN-PROP, it first infers the receiver type with constraints. It passes those constraints to the constraints solver, which we know, results in a substitution which produces fully inferred type. It accesses the head of the type and queries the property type. All constraint have been solved so it returns an empty set, though it returns the inferred type information in the substitution.

For a method call CGEN-MCALL, the logic remains the same as for the property access. It differs only in the fact that after getting the method signature, the argument types are inferred. Those constraints are merged with the new constraints implied by the required method's arguments types. It also returns composed substitution with all inferred type information.

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond, \mathbb{C}, \sigma \qquad \mathbb{C}' = \mathbb{C} \cup \{\tau_\diamond <: \gamma\} \qquad \sigma' = \texttt{solve}(\mathbb{C}', TV(\sigma e_{val})) \qquad \Gamma' = \Gamma; x \to \sigma'\sigma\tau_\diamond \qquad \Gamma' \vdash \overline{St} \to \overline{St}'}{\Gamma \vdash var\ v : \gamma = e_{val} \to var\ v : \gamma = \sigma'\sigma(e_{val}); \overline{St}'} \text{ KI-ANN-DECL}$$

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond, \mathbb{C}, \sigma \qquad \sigma' = \texttt{solve}(\mathbb{C}', TV(\sigma e_{val})) \qquad \Gamma' = \Gamma; x \to \sigma'\sigma\tau_\diamond \qquad \Gamma' \vdash \overline{St} \to \overline{St}'}{\Gamma \vdash var\ v = e_{val} \to var\ v : \sigma'\sigma(\tau_\diamond) = \sigma'\sigma(e_{val}); \overline{St}'} \text{ KI-DECL}$$

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond, \mathbb{C}, \sigma \qquad \Gamma(l_{val}) = \gamma \qquad \mathbb{C}' = \mathbb{C} \cup \{\tau_\diamond <: \gamma\} \qquad \sigma = \texttt{solve}(\mathbb{C}', TV(\sigma e_{val})) \qquad \Gamma \vdash \overline{St} \to \overline{St}'}{\Gamma \vdash l_{val} = e_{val}; \overline{St} \to l_{val} = \sigma'\sigma(e_{val}); \overline{St}'} \text{ KI-ASSIGN}$$

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau, \mathbb{C} \qquad \mathbb{C}' = \mathbb{C} \cup \{\tau <: \texttt{return\_type()}\} \qquad \sigma' = \texttt{solve}(\mathbb{C}', TV(\sigma e_{val}))}{\Gamma \vdash return\ e \to return\ \sigma'\sigma(e)} \text{ KI-RETURN}$$

FIGURE 4.2: Kotlin inspired type inference, statements

Constructor call CGEN-CONSTR, is similar to the method call but without the receiver resolving. It just introduces the constraints implied by the required constructor's arguments types and merges them with the constraints implied by the arguments.

Let's now look into the type inference algorithm fig. 4.2 for the whole program. For a variable declaration with annotation KI-ANN-DECL, it infers the type of the initializer expression, adds the expected type constraint and solves the constraint set. Then returns a new statement with initializer expression with resolved type variables. It adds the variable type to the type context.

For a variable declaration, without annotation KI-DECL, it infers the type of the initializer expression and solves the constraint set. Then returns a new statement with initializer expression with resolved type variables and the type of inferred expression, with applied substitution, as an annotation. It adds the fully resolved type of the variable to the type context.

For a variable assignment KI-ASSIGN, it works exactly the same as for a variable declaration with annotation. The only difference is the expected type is extracted from the type context.

For a return statement KI-RETURN, it infers the type of the initializer expression, adds the expected return type constraint and solves the constraint set.

4.6 Kotlin inspired type inference limitations

We have provided a base algorithm which adheres to the current limitations of the Kotlin inference algorithm. We see that the points of friction, which restrict collection of the information from a broader context are

1. Algorithm fails if it cannot solve all type variables in the receiver, using only the information collected from this receiver

2. Algorithm fails if it cannot solve all type variables in the statement, using only the information collected from this statement

Looking on the `Option` type example, from the chapter 1.

$$\texttt{var x} = \texttt{none}[\alpha_0]()$$
$$\texttt{x} = \texttt{some}[\alpha_1](5)$$

We know that $\texttt{none}[\alpha_0]() \Rightarrow \texttt{Option}[\alpha_0], \varnothing, \varnothing$, though if we execute the constraint solver, it fails. It cannot fix the type variable α_0, because it has no constraints.

If we think about the declarations of the classes `Some` and `Option`, it not seem possible to **leak** the type variable α_0 from the type in a way that it can become the type of a receiver.

If we defer the only such variables that are never inferred as a type of a receiver, then it avoids the problem for the calls on receivers mentioned in the section 4.3. If we limit those variables to unconstrained ones, such approach avoids the problems listed for the statements in the section 4.4 because, respectively

1. **Inflated types of variables** - We do not defer any variables with existing constraints, thus the inflation of types never occurs.

2. **Computational complexity** - We do not defer any variables with existing constraints, thus the accumulation of constraints never occurs.

3. **Errors far away from the issue** - There are never any problems with forced fixes as such deferred type variables are never necessary to decide the method/parameter type, thus this problem never occurs.

In the next chapter we formalize the intuitive concept of type variables that are never inferred as a type of a receiver.

Chapter 5

Type inference progression

In this chapter, we formally capture the problem of determining the type of a method/parameter, described in the section 4.3. We define type inference progression property. Expressions that hold this property have enough type information inferred, so the inference algorithm can always determine the type of a method/parameter.

5.1 Type inference progression property

In this section, we define the type inference progression property $e \gg \tau_\diamond$. It reads as, the expression e progresses to the type $\tau_\diamond$. This property captures only the control flow of the inference algorithm and is not concerned with constraint generation. If an expression progresses to a type, then it means that type inference can decide the types of all method calls/property accesses in the expression, based only on the information that is already provided explicitly as type arguments.

As an example, for declarations and an expression:

```
class ClassOne {
    fun foo() -> Int {}
}

class ClassTwo {
    fun foo() -> String {}
}

fun identity[T](x: T): T {}
```

```
identity[ClassOne](ClassOne()).foo()
```

This expression progresses to the type `Int`

$$\texttt{ClassOne}() \gg \texttt{ClassOne}$$
$$\texttt{identity}[\texttt{ClassOne}](\texttt{ClassOne}()) \gg \texttt{ClassOne}$$
$$\texttt{identity}[\texttt{ClassOne}](\texttt{ClassOne}()).\texttt{foo}() \gg \texttt{Int}$$

However, if the expression was:

$$\texttt{identity}[\alpha_0](\texttt{ClassOne}()).\texttt{foo}()$$

$$\frac{\Gamma(v) = \tau_\diamond}{\Gamma \vdash v \gg \tau_\diamond}\ \textsc{TypeInfProg-LocalVar}$$

$$\frac{e \gg C[\overline{\tau_\diamond}] \qquad \texttt{prop\_type}(C, p) = \gamma}{e.p \gg [\overline{\tau_\diamond}/\texttt{type\_params}(C)]\gamma}\ \textsc{TypeInfProg-Prop}$$

$$\frac{e \gg C[\overline{\tau_\diamond}] \qquad \texttt{method\_type}(C, m) = \overline{X}, \overline{\gamma}, \gamma_r}{e.m[\overline{v_\diamond}](\overline{e}) \gg [\overline{v_\diamond}/\overline{X}][\overline{\tau_\diamond}/\texttt{type\_params}(C)]\gamma_r}\ \textsc{TypeInfProg-MCall}$$

$$\frac{}{\texttt{new}\ C[\overline{\tau_\diamond}](\overline{e}) \gg C[\overline{\tau_\diamond}]}\ \textsc{TypeInfProg-Constr}$$

FIGURE 5.1: Type inference progression rules

It would not progress.

```
ClassOne() ≫ ClassOne
identity[alpha_0](ClassOne()) ≫ alpha_0
identity[alpha_0](ClassOne()).foo() — Does not progress
```

It is because the receiver does not progress to a type in the fixed-head form. We use this property to analyze how much type information is required for the type inference algorithm to progress. Later we will see that there are type variables that do not break the inference progression property.

We present the full set of rules for the type inference progression property fig. 5.1. The crucial requirement that does rules state is that subexpressions that are receivers have to progress to a type in fixed-head form. If every expression in a language progresses, then we say that this language has type inference progression property.

In the next section, we show that the subset of IFJ with all type information inferred holds the type inference property. Later, we show that certain type variables, substituting **not-leakable** type parameters, are unnecessary. We show that a subset of IFJ where those type variables are not fixed still holds type inference progression property.

Additionally, we define a **well-formed accesses** property. This property has low significance. It is present only because of formality reasons. It guarantees that methods/properties called/accessed on a receiver exists or accessed variable is in the context. In a real type inference algorithm implementation we would just reject programs that not hold it.

Definition 1. *If for a sublanguage of IFJ*

1. *All accessed local variables are in the typing environment*
2. *If an expression progresses to a fixed-head form type, and is a receiver then the method or property exists in the head's corresponding class.*

we say that the sublanguage has ***well-formed accesses****.*

If we restrict IFJ to IFJ with well-formed accesses, then the only condition that its sublanguage has to meet to have the type inference progression property is that if an expression is a receiver, then it has to progress to a type in the fixed-head form.

5.2 Fully inferred type parameters

In this section, we show that there are no type variables and everything is inferred then indeed the language holds type inference progression property. It is a proof that the way that the existing, Kotlin inspired type inference algorithm works, implies the inference progression property.

Lemma 2. *Assuming that in the typing context, there are only fully inferred types, in the sublanguage of IFJ with well-formed accesses, with all type arguments being fully resolved, every expression progresses to a fully inferred type.*

Proof. By induction, base cases:

- Case: LOCALVAR: We know that all expressions have well-formed accesses, thus we know that the expression progresses to a type. We know that all types in typing context are fully inferred thus the type the expression progresses to is fully inferred.
- Case: CONSTRUCTORCALL: We know that the expression progresses to a type as there are no preconditions. We know that in the call there are no type variables, thus the type the expression progresses to is fully inferred.

As the inductive hypothesis, we assume that every expression e progresses to a fully inferred type $e \gg \tau_\diamond$

- Case: PROPERTY: We know that $e \gg \tau_\diamond$ from the inductive hypothesis. We know that $\tau_\diamond$ is fully inferred, thus it must be a fully inferred class type $C[\overline{\tau_\diamond}]$. From the well-formedness of accesses, we know that the acessed property exists in the class. The type of the property contains only the type parameters of the associated class. Hence, if we override those properties with fully inferred types $\overline{\tau_\diamond}$, the type is fully inferred.
- Case: MCALL: We know that $e \gg \tau_\diamond$ from the inductive hypothesis. We know that $\tau_\diamond$ is fully inferred, thus it must be a fully inferred class type $C[\overline{\tau_\diamond}]$. From the well-formedness of accesses, we know that the method exists in the class. We know that type arguments of the method are fully inferred types. The type of the property contains only the type parameters of the associated class and the method. Hence, if we override those properties with fully inferred types $\overline{\tau_\diamond}$ and method type arguments, the type is fully inferred.

Thus, by induction the theorem is true. □

Theorem 3. *Assuming that only fully inferred types are in the typing context, the sublanguage of IFJ with well-formed accesses, with all type arguments being fully resolved, has the type inference progression property*

Proof. From the lemma 2 we know that every expression in this language progresses to a fully inferred type. From this fact, the weaker theorem that every expression progresses is also true. Hence, the theorem is true. □

5.3 Not-leakable type parameters

In the previous section, we have shown that if all type variables are resolved then every expression in such language progresses. In this section, we define **not-leakable** and **leakable** type parameters. Later we show that if we do not fix the type variables substituting **not-leakable** type parameters, then such a subset of IFJ still holds type inference progression property.

We split type parameters into two mutually exclusive categories. The first category we name **not-leakable**, type parameters.

Definition 4. *Type parameter is **not-leakable** if*

- *If is a type parameter of a class:*
  - *If used in a inherited type, then it can be used only as argument for **not-leakable** type parameters*
  - *If used in a property type, then it can be used only as argument for **not-leakable** type parameters*
  - *If used in a method type, then:*
    - *In the types of the arguments, it can be used in any way*
    - *In the return type, then it can be used only as argument for **not-leakable** type parameters*
- *If is a type parameter of a method*
  - *In the types of the arguments, it can be used in any way*
  - *In the return type, then it can be used only as argument for **not-leakable** type parameters*

Definition 5. *Type parameter is **leakable** if is not **not-leakable***

The intuition behind not-leakable type parameters is, that those are defined in such a way, that we cannot leak those type parameters from an object to be the type of an expression through any sequence of transformations. So if a type variable is used as a type argument for a not-leakable type parameter, then the expression will never progress to this variable. Thus, we never end up with a type variable as a receiver type. To help the reader understand what being not-leakable means, let's go over a few examples.

```
class Empty[W] {
    Empty()
}
```

In this case, `W` is not-leakable, not used at all.

```
class Containter [X] {
    value: X
    Containter(value: X)
}
```

In this case, `X` is leakable. You can read the property `value`.

```
class Empty[W] {
    Empty()
}
class Containter [X] {
    value: Empty[X]
    Containter(value: Empty[X])
}
```

In this case, `X` is not-leakable. You can read the property `value`, but you cannot read it from the `Empty[X]` object.

```
class Empty[W] {
    Empty()
}
class Containter [X] {
    fun foo(value: X): Int
    fun foo(value: Empty[X]): Int
}
```

In this case, `X` is not-leakable, in method arguments you can use the type parameter in any way. You cannot read if from the object.

```
class Empty[W] {
    Empty()
}
class Containter [X] {
    fun foo(): X
}
class Containter2 [Y] {
    fun foo(): Empty[Y]
}
```

In this case, `X` is leakable and `Y` is not leakable. For method's return type rationale is the same as for properties.

5.4 Not-leakable type parameters and inference progression

In this section, we prove that the intuition from the end of the previous section is correct. We show that if we leave out the type variables used for not-leakable type parameters, such a subset of IFJ will hold the type inference progression property.

Definition 6. *We define a property* $\texttt{not\_leakable\_c}(C, n)$ *in such a way that if for a class* C*, it holds for a given* n *if and only if* n^{th} *type parameter in the class* C *is not-leakable.*

Definition 7. *We define a property* $\texttt{not\_leakable\_m}(C, m, n)$ *in such a way that if for a class* C*, and method* m *it holds for a given* n *if and only if* n^{th} *type parameter in the method* m *is not-leakable.*

Definition 8. *We say* $\tau_\diamond$ *is in* ***inference progressive*** *form if:*

- *it is in fixed-head form* $C[\overline{\gamma_\diamond}]$

- $\forall_{i \in 1...n}$ *in* $C[\overline{\gamma_\diamond}]$*:*
  - *if* $\gamma_{\diamond i}$ *is* `not_leakable_c`(C, n) *(is on not-leakable position), it is in any form*
  - *if* $\gamma_{\diamond i}$ *is not* `not_leakable_c`(C, n) *(is on leakable position), it is in inference progressive form*

Inference progressive form is a stronger guarantee that the fixed-head form, but it is a weaker guarantee than the fully inferred form.

Definition 9. *We say that a type argument is* ***inference sufficient*** *if:*

- *if used on a not-leakable position, then it is in any form*
- *if used on a leakable position, then it is in inference progressive form*

To give a better understanding what a inference sufficient type argument is, we could say that a type is inference progressive if is in the fixed-head form and its type arguments are inference sufficient.

Lemma 10. *Assuming that in the typing context, there are only types in inference progressive form, IFJ with well-formed accesses, with all type arguments being inference sufficient every expression progresses to a type in inference progressive form.*

Proof. By induction, base cases:

- Case: LOCALVAR: From assumptions, we know that all expressions have well-formed accesses, thus we know that the expression progresses to a type. From assumptions, we know that in the typing context, there are only types in inference progressive form, thus the type the expression progresses to is in inference progressive form.
- Case: CONSTRUCTORCALL: We know that the expression progresses to a type as there are no preconditions. From assumptions, we know that all type arguments are inference sufficient. Hence, the returned type is in inference progressive form, because the leakable type parameters will be substituted with types in inference sufficient form.

As the inductive hypothesis we assume that every expression e progresses to a type in inference progressive form $e \gg \tau_\diamond$

- Case: PROP: We know that $e \gg \tau_\diamond$ from the inductive hypothesis, and that $\tau_\diamond$ is inference progressive. Thus, it must be a form $C[\overline{\tau_\diamond}]$. From the well-formedness of accesses, we know that the property exists in the class. The parameter type contains only the class's type parameters. We know that the type arguments $\overline{\tau_\diamond}$ are inference sufficient. Because arguments in the class are inference sufficient, then the leakable type parameters are substituted with types in inference progressive form. Thus the type the expression progresses to is inference progressive.
- Case: MCALL: We know that $e \gg \tau_\diamond$ from the inductive hypothesis, and that $\tau_\diamond$ is inference progressive. Thus, it must be in the form $C[\overline{\tau_\diamond}]$. From the well-formedness of accesses, we know that the method exists in the class. The method return type contains class type parameters and method type parameters. We know that $\overline{\tau_\diamond}$ are inference sufficient. Because arguments of the

class are inference sufficient, then the leakable type parameters of the class are substituted with types in inference progressive form. Because arguments in the method are inference sufficient, then the leakable type parameters of the method are substituted with types in inference progressive form. Thus, if we substitute both sets of type parameters, the type the expression progresses to is in inference progressive form.

By induction, the lemma is true. □

This leads us to our final result.

Theorem 11. *IFJ with well-formed accesses, with all type arguments being inference sufficient has the inference progression property.*

Proof. From the lemma 10 we know that in such a restricted language every expression progresses, thus it has the inference progression property. □

This is an important result, as it says which type variables are unnecessary for the type inference to progress. In the next section, we will devise an algorithm which does not require from the constraint solver to infer those type variables.

Chapter 6

Not Too Local type inference

In this chapter, we use the inference progression theory from the previous chapter to provide an extension of the type inference algorithm for IFJ. We show that it solves the problem of Algebraic Data Types introduced in the first chapter.

The proposed algorithm uses the idea of not-leakable type parameters. Assuming a program in IFJ, we need to fix enough type variables so all type arguments are inference sufficient. Then, the inference algorithm can determine the type of a method/parameter. When passing them to the constraint solver, we mark type variables that are on not-leakable positions as not-required. It communicates to the solver that it does not have to solve them if it can't. We refer to the newly proposed algorithm as Not Too Local Type inference (NTL Type inference).

6.1 The overview

We start with defining the new inference relations. We define the new program translation relation as

$$Body \rightarrow Body, \sigma$$

It maps one set of statements to another. It differs from the original one as now does not guarantee all type variables being resolved. If one implements the algorithm, it has to include validation of that afterward. It also returns a substitution, so the type information collected in the subsequent statements can be applied to the previous statements. The substitution maps from the type variables to partially inferred types. As in the original algorithm, it uses two components, constraint generation and constraint solver.

The new constraint generation is, as its predecessor, defined by the inference relation

$$e \Rightarrow \tau_\diamond, \mathbb{C}, \sigma, \mathbb{ND}$$

For an expression e, it infers a partially inferred type $\tau_\diamond$ of the expression, with corresponding set of constraints $\mathbb{C}$. It also includes a substitution σ which might contain information learned during the inference. The substitution maps from the type variables to partially inferred types. Additionally, it includes a set of not-required type variables from the expression e.

The new constraint solver is defined as:

$$\texttt{solve} : \mathbb{C}, \texttt{Set}[\alpha], \mathbb{ND} \rightarrow \sigma, \mathbb{ND}$$

It takes as arguments a constraint set, a set of type variables to solve, and a set of not-required type variables. It returns derived substitution and a new set of not-required type variables. We explain why the solver returns a new set of not-required type variables in the section 6.2.

We start with the original motivation for the research, the ADTs example presented in the introduction. We demonstrate how the modified algorithm would work. Assume the set of declarations:

```
class Option[T]
class Some[T](value: T): Option[T]
class None[T](): Option[T]

class Functions {
    fun some[T](x: T): Option[T] {}
    fun none[T](): Option[T] {}
}
```

With the statements:

$$\texttt{var x} = \texttt{Functions}().\texttt{none}[\alpha_0]()$$
$$\texttt{x} = \texttt{Functions}().\texttt{some}[\alpha_1](5)$$

Let's show an example of how the modified version of the algorithm works and that it can infer types for this example.

$$\texttt{Functions}() \Rightarrow \texttt{Functions}, \varnothing, \varnothing, \varnothing$$

There are no type variables for the `Functions()` constructor call. Thus, the set of not-required type variables is empty.

$$\texttt{Functions}().\texttt{none}[\alpha_0]() \Rightarrow \texttt{Option}[\alpha_0], \varnothing, \varnothing, \{\alpha_0\}$$

For the `none` method call, it includes α_0 as a not-required type variable. It does so because for the `none` method, the type parameter `T` is not-leakable.

$$\varnothing = \texttt{solve}(\varnothing, \{\alpha_0\}, \{\alpha_0\})$$

$$\texttt{var x} = \texttt{Functions}().\texttt{none}[\alpha_0]() \rightarrow \texttt{var x} : \texttt{Option}[\alpha_0] = \texttt{Functions}().\texttt{none}[\alpha_0](), \varnothing$$

As in the original algorithm, it runs the constraint solver for the single statement, with α_0 as not-required type variable. The constraint solver returns an empty substitution, and the new set of not-required type variables remains empty. The type of `var x` is set to $\texttt{Option}[\alpha_0]$ and is added to the typing context.
For the second statement:

$$\texttt{Functions}().\texttt{some}[\alpha_1](5) \Rightarrow \texttt{Option}[\alpha_1], \{\texttt{Int} <: \alpha_1\}, \varnothing, \varnothing$$

For the `some` method call it does not include α_1 as a not-required type variable. It is because, for the `some` method, the type parameter `T` is leakable.

$$\sigma = \texttt{solve}(\{\texttt{Option}[\alpha_1] <: \texttt{Option}[\alpha_0], \texttt{Int} <: \alpha_1\}, \{\alpha_0, \alpha_1\}, \{\alpha_0\})$$

$$\texttt{x} = \texttt{Functions}().\texttt{some}[\alpha_1](5) \rightarrow \texttt{x} = \texttt{Functions}().\texttt{some}[\texttt{Int}](5), \{\alpha_0 \rightarrow \texttt{Int}, \alpha_1 \rightarrow \texttt{Int}\}$$

For the variable assignment, it adds constraint $\texttt{Option}[\alpha_1] <: \texttt{Option}[\alpha_0]$ to the constraint set, then passes it to the constraint solver. The constraint solver returns a substitution $\{\alpha_0 \rightarrow \texttt{Int}, \alpha_1 \rightarrow \texttt{Int}\}$. Further, we apply the substitutions from the subsequent statements to the previous statements. The final result is the program:

$$\begin{aligned}&\texttt{var x} : \texttt{Option}[\texttt{Int}] = \texttt{Functions}().\texttt{none}[\texttt{Int}]()\\&\texttt{x} = \texttt{Functions}().\texttt{some}[\texttt{Int}](5)\end{aligned}$$

The substitution produced from the second statement has been applied to the first statement.

6.2 The constraint solver

In this section, we discuss in detail the contract that the implementation of constraint solver has to comply with when used in NTL type inference. We keep the contract possibly minimal and general, assuming as little as possibly about the underlying constraint solver implementation.

As mentioned before, we extend the constraint solver interface in a way that we can pass type variables that are not-required. The constraint solver can fix them but the algorithm can progress further if it does not.

Why do we need the constraint solver to return a new set of not-required type variables? Let's first look into how the constraint solver should interpret and implement the set of required variables. If a variable is **required** does it mean it should just be fixed to **anything**? The problematic case might arise, for example, when

$$\texttt{foo}[\alpha_1](\bar{e}) \Rightarrow \alpha_1, \mathbb{C}, \sigma, \{\alpha_0\}$$

Assume that some $\alpha_0 \in TV(\bar{e})$ which in this case is not-required, thus we need it fixed. If we execute $solve(\mathbb{C}, \mathbb{D})$, and it will return a substitution $\{\alpha_1 \rightarrow \alpha_0\}$, were α_0 is not-required. Which would map the example to

$$\texttt{foo}[\alpha_0](\bar{e}) \Rightarrow \alpha_0, \mathbb{C}, \sigma, \{\alpha_0\}$$

It is not in fixed-head form, and thus could not be used as a receiver. Based on the inference progression theory from the previous chapter, we need to define carefully, what a variable being required should exactly mean for the constraint solver.

From the constraint solver perspective this transformation rule reads as follows

If a required type variable α_r is fixed to a type $\tau_\diamond$:

1. If the type $\tau_\diamond$ is a not-required type variable α_{nr}, then α_{nr} becomes required.
2. If the type $\tau_\diamond$ is type in the form $C[\overline{\tau_\diamond}]$ then
   (a) If for $\tau_{\diamond i}$, `not_leakable_c(C, i)`, then we do nothing
   (b) If for $\tau_{\diamond i}$, $\neg$`not_leakable_c(C, i)`
      i. If $\tau_{\diamond i}$ is a type variable α_{nr}, then α_{nr} becomes required.
      ii. If $\tau_{\diamond i}$ is the type in the form $C[\overline{\tau_\diamond}]$, then we apply (2) recursively

Such formulation will guarantee that all type arguments on the leakable positions will be in the inference progressive form. This is the reason why constraint solver returns a new set of not-required type variables, some variables might be removed from the not-required set according to the rules above.

Additionally, we assume that the new constraint solver will only further defer the not-required variables only when those are not constrained in any way. This assumption simplifies the implementation of the algorithm as we do not have to save and track any existing constraints for the local variable's types. This also simplifies the solver definition as it does not have to additionally return a set of remaining constraints.

6.3 Expression inference

As we have provided a new definition of the constraint solver let's now look into the proposed implementation of the constraint generation algorithm fig. 6.1, modifications of the original algorithm are marked with the red color.

$$\texttt{not\_required\_variables\_c}(\overline{v_\diamond}, C) = \bigcup_{i \in 1 \ldots n} \Big(\\ \textit{if } \texttt{not\_leakable\_c(C, i)} \textit{ then} \\ \quad TV(v_{\diamond i}) \\ \textit{else} \\ \quad v_{\diamond i} = \alpha \to \{\alpha\} \\ \quad v_{\diamond i} = C'[\overline{v'_\diamond}] \to \texttt{not\_required\_variables\_c}(\overline{v'_\diamond}, C') \\ \Big)$$

FIGURE 6.2: Not-required variables for a constructor call

$$\frac{\Gamma(p) = \tau_\diamond}{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond,\ \varnothing, \varnothing, TV(\tau_\diamond),} \text{ CGENNTL-LOCALVAR}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow \tau_\diamond, \mathbb{C}, \sigma, \mathbb{D} \\ \sigma', \mathbb{D}' = \texttt{solve}(\mathbb{C}, TV(\sigma e), \mathbb{D}) \qquad C[\overline{\tau}] = \sigma'\sigma\tau_\diamond \qquad \texttt{prop\_type}(C, p) = v\end{array}}{\Gamma \vdash e.p \Rightarrow [\overline{\tau}/\texttt{type\_params}(C)]v,\ \varnothing, \sigma \circ \sigma', \mathbb{D}'} \text{ CGENNTL-PROP}$$

$$\frac{\begin{array}{c}\Gamma \vdash e \Rightarrow \tau_\diamond, \mathbb{C}, \sigma, \mathbb{D} \\ \sigma', \mathbb{D}' = \texttt{solve}(\mathbb{C}, \mathbb{D}) \qquad C[\overline{\tau}] = \sigma'\sigma\tau_\diamond \qquad \texttt{method\_type}(C, m) = \overline{X}, \overline{\gamma}, \gamma \\ \sigma\sigma'\Gamma \vdash e_1 \Rightarrow \tau_{\diamond 1}, \mathbb{C}_1, \sigma''_1, \mathbb{D}_1 \quad \sigma\sigma'\sigma''_1\Gamma \vdash e_2 \Rightarrow \tau_{\diamond 2}, \mathbb{C}_2, \sigma''_2, \mathbb{D}_2, \ldots \\ \mathbb{D}'' = \texttt{not\_required\_variables\_m}(\overline{v_\diamond}, C, m)\end{array}}{\begin{array}{l}\Gamma \vdash e.m[\overline{v_\diamond}](\overline{e}) \Rightarrow [\overline{v_\diamond}/\overline{X}][\overline{\tau}/\texttt{type\_params}(C)]\gamma, \\ \left(\bigcup_{i \in 1 \ldots n} \mathbb{C}_i\right) \cup \{\overline{\tau_\diamond} <: [\overline{v_\diamond}/\overline{X}][\overline{\tau}/\texttt{type\_params}(C)]\overline{\gamma}\}, \\ \circ_{i \in 1 \ldots n}\sigma''_i \circ \sigma' \circ \sigma, \left(\bigcup_{i \in 1 \ldots n} \mathbb{D}_i\right) \cup \mathbb{D}' \cup \mathbb{D}'',\end{array}} \text{ CGENNTL-MCALL}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow \tau_{\diamond 1}, \mathbb{C}_1, \sigma_1, \mathbb{D}_1 \quad \sigma_1\Gamma \vdash e_2 \Rightarrow \tau_{\diamond 2}, \mathbb{C}_2, \sigma_2, \mathbb{D}_2 \ldots \\ \mathbb{D}' = \texttt{not\_required\_variables\_c}(\overline{v_\diamond}, C)\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{new}\ C[\overline{v_\diamond}](\overline{e}) \Rightarrow C[\overline{v_\diamond}], \\ \left(\bigcup_{i \in 1 \ldots n} \mathbb{C}_i\right) \cup \{\overline{\tau_\diamond} <: [\overline{v_\diamond}/\texttt{type\_params}(C)]\texttt{prop\_types}(C)\}, \\ \circ_{i \in 1 \ldots n}\sigma'' \circ \sigma', \left(\bigcup_{i \in 1 \ldots n} \mathbb{D}_i\right) \cup \mathbb{D}'\end{array}} \text{ CGENNTL-CONSTR}$$

FIGURE 6.1: NTL type inference, expressions

For a local variable access CGENNTL-LOCALVAR, to get the set of not-required type variables for the expression we just fetch all type variables that occur in the variable's type. If any of those would be required, the constraint solver would fix it.

$$\texttt{not\_required\_variables\_m}(\overline{v_\diamond}, C, m) = \bigcup_{i \in 1 \ldots n} \Big(\\ \textit{if } \texttt{not\_leakable\_m(C, m, i)} \textit{ then} \\ \quad TV(v_{\diamond i}) \\ \textit{else} \\ \quad v_{\diamond i} = \alpha \to \{\alpha\} \\ \quad v_{\diamond i} = C'[\overline{v'_\diamond}] \to \texttt{not\_required\_variables\_m}(\overline{v'_\diamond}, C') \\ \Big)$$

FIGURE 6.3: Not-required variables for a method call

The substitution is empty in this case.

For a property access CGENNTL-PROP, we get a not-required variables set and substitution from the receiver. We return the new not-required variables set, returned by the solver.

For a method call CGENNTL-MCALL, for the receiver resolution, it goes the same as for the property access. Then we traverse each of the method arguments, accumulating the inference information in substitutions. At the end, we search type arguments of the method call, for type variables that are not-required, formally the process is defined as specified here fig. 6.3. In the result we include the combined not-required type variables from the receiver, arguments and from the type arguments.

For a constructor call CGENNTL-CONSTR, as in a method call, Then we traverse each of the method arguments, accumulating the inference information in substitutions. At the end, we search type arguments of the constructor call, for type variables that are not-required, formally the process is defined as specified here fig. 6.2. In the result we include the combined not-required type variables from the arguments and from the type arguments.

6.4 Type inference for programs

Modifications of the inference algorithm for statements are modest fig. 6.4. We define the new typing context as:

$$\Gamma : v \to \tau_\diamond$$

It may now contain partially inferred types. Though we assume that all types we put into the typing context are inference sufficient, as we did in assumptions of our main theorem in the chapter 5.

We modify the relation to also return a substitution. This is required so we can use information learned in the subsequent statements, to update the current statement. We do need to pass the not-required type variables forward as those can be recovered from types in context. Additionally, we need to apply new substitutions to the context, as in this formulation not all types in context are fully inferred. This formulation does not ensure that all type variables are inferred thus an additional check is required after the inference.

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond,\ \mathbb{C}, \sigma, \mathbb{D} \qquad \mathbb{C}' = \mathbb{C} \cup \{\tau_\diamond <: \gamma\} \qquad \sigma', \mathbb{D}' = solve(\mathbb{C}', TV(\sigma e_{val}), \mathbb{D}) \qquad \Gamma' = \Gamma; x \to \sigma'\tau_\diamond \qquad \sigma'\sigma\Gamma'; \vdash \overline{St} \to \overline{St}', \sigma''}{\Gamma \vdash var\ v : \gamma = e_{val} \to var\ v : \gamma = \sigma''\sigma'\sigma(e_{val}); \overline{St}', \sigma \circ \sigma' \circ \sigma''} \text{ NTL-ANN-DECL}$$

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond, \mathbb{C}, \sigma, \mathbb{D} \qquad \mathbb{C}' = \mathbb{C} \cup \{\tau_\diamond <: \gamma\} \qquad \sigma, \mathbb{D}' = solve(\mathbb{C}', TV(\sigma e_{val}), \mathbb{D}) \qquad \Gamma' = \Gamma; x \to \tau_\diamond \qquad \sigma'\sigma\Gamma'; \vdash \overline{St} \to \overline{St}', \sigma''}{\Gamma \vdash var\ v = e_{val} \to var\ v : \sigma''\sigma'(\tau_\diamond) = \sigma''\sigma'\sigma(e_{val}); \overline{St}', \sigma \circ \sigma' \circ \sigma''} \text{ NTL-DECL}$$

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond, \mathbb{C}, \sigma, \mathbb{D} \qquad \Gamma(l_{val}) = \gamma_\diamond \qquad \mathbb{C}' = \mathbb{C} \cup \{\tau_\diamond <: \gamma_\diamond\} \qquad \sigma, \mathbb{D}' = solve(\mathbb{C}', TV(\sigma e_{val}), \mathbb{D} \cup TV(\gamma_\diamond)) \qquad \sigma'\sigma\Gamma \vdash \overline{St} \to \overline{St}', \sigma''}{\Gamma \vdash l_{val} = e_{val}; \overline{St} \to l_{val} = \sigma''\sigma'\sigma(e_{val}); \overline{St}', \sigma \circ \sigma' \circ \sigma''} \text{ NTL-ASSIGN}$$

$$\frac{\Gamma \vdash e_{val} \Rightarrow \tau_\diamond, \mathbb{C}, \sigma', \mathbb{D}, \quad \mathbb{C}' = \mathbb{C} \cup \{\tau <: \texttt{return\_type()}\} \qquad \sigma, \mathbb{D}' = solve(\mathbb{C}', TV(\sigma e_{val}), \mathbb{D})}{\Gamma \vdash return\ e \to return\ \sigma'\sigma(e), \sigma \circ \sigma'} \text{ NTL-RETURN}$$

FIGURE 6.4: NTL type inference, statements

6.5 Advanced types

The example we presented in the section 6.1 might look like a special case. One might feel that this technique only applies to some very specific types and improves the type inference for a marginal set of edge cases. It turns out this technique is much more general and it might improve how the type inference works for many common patterns. Below we present a mutable list example. Kotlin inspired type inference would not be able to type this example, though NTL type inference can do so. We do not provide explicit type variables for conciseness.

```
class List[T] // T is not leakable
class Nil[T](): List[T] // T is not leakable
class Cons[T](value: T, tail: List[T]): List[T]

class MutableList[T] { // T is not leakable
    value: List[T]

    fun add(x: T) {}
    fun remove(): Option[T] {}
}

var list = MutableList(Nil)
list.add(1)
var z = list.remove()
```

The example consists of algebraic data type `List`, which is very similar to the `Option` example from the previous sections. For the sum type `List` and the empty list constructor `Nil` the type parameter `T` is not-leakable.

Using it, we implement `MutableList`. The part that might be unexpected is the fact that `T` is not-leakable for `MutableList`. Such an example would not be typed by Kotlin inspired type inference, though NTL type inference can type it. This example is interesting as it shows the usefulness of the technique beyond only very simple types.

Intuitively, the not-leakable type parameters are type parameters whose values do not have to be provided to the class to construct an object. The type parameters which values are provided to the constructor will usually provide sufficient constraints to be inferred. In this sense, if we enable deferring of the not-leakable type parameters, we solve many problematic cases, because the not-leakable type parameters are those which often cannot be inferred because of insufficient constraints.

Handling such cases would be a great improvement for the programming language Kotlin as currently, this example would have to be transformed into a type-safe builder or require an annotation.

It is worth noting that the proposed technique is different from the builder-style inference as it fixes the type greedily instead of building the constraint set for the whole lambda.

```
var list = MutableList(Nil)
list.add(1)
list.add("example")
```

The example below would not work with NTL type inference. The type parameter of `MutableList` is fixed to `Int` by the NTL type inference algorithm. The second statements produces a contradictory constrain $\{\texttt{Int} <: \texttt{String}\}$.

Chapter 7

Kotlin compiler integration

In this chapter, based on the work from the previous chapters, we propose the syntax and semantics of a Kotlin language feature *imaginary type parameters*. It composes the theoretical ideas into a proposal of the feature for a real-life language. We describe the feature and its semantics from the perspective of the programmer. Later we discuss interactions with features like variance and overloading, which are present in many other statically typed object-oriented languages. We also discuss the partial implementation work we did in Kotlin's compiler to asses if such a feature is feasible to implement.

7.1 Imaginary type parameters

In this section, we propose the syntax and semantics of *imaginary type parameters*, a new feature we propose for the programming language Kotlin. An imaginary type parameter is equivalent to a not-leakable type parameter. If the programmer marks a type parameter as imaginary, then the inference algorithm, extended as described in chapter 6, treats the type parameter as not-leakable. It means the inference of the type parameter can be deferred.

```
sealed class Option<imag T>
class Some<T>(x: T): Option<T> {}
class None<imag T>(): Option<T> {}
```

Let's discuss the proposed syntax. The most important design decision we take is if the programmer should mark imaginary type parameters explicitly, or if the compiler should derive them automatically based on the type declaration if a type parameter fulfills the not-leakable type parameter constraints.

We argue that explicit annotation is preferred as it is easier for the programmer to annotate a type parameter and receive an error if it does not fulfill the constraints. Otherwise, for example, adding a method using the type parameter in not imaginary way, could by accident introduce an unwanted breaking change to the library. It is much easier to go the other way around as the explicit annotations may be suggested by the compiler or a linter if a certain type parameter fulfills the constraints and is not annotated. It is important to emphasize that removing the annotation from a type parameter is a breaking change for a library interface and might require updates in the software that depends on it.

One could argue that explicit annotation might be too much of a burden or too complex for the programmer to provide. We argue that even though it is certainly a limitation, it is not a deal-breaker:

1. Currently for examples like list initialization, the programmer has to use type-safe builders. We argue that providing the annotation in the library is easier than introducing unnecessary complexity to every program that uses the library.

2. Currently the problem is solved (not fully, it does not work for the mutable variables) by marking the parameter contravariant and initializing the type parameter as a bottom type. We argue that this trick is much harder to comprehend than the imaginary type parameters, even though it is commonly used.

3. The type parameters annotations for the variance already exist, providing an additional one is not much of a complexity burden.

Another decision we take is how to refer to such type parameters in the language documentation. Not-leakable is a good name from the perspective of the type inference algorithm, as it emphasizes the rationale for why inference of those type parameters can be deferred. From the programmer's perspective, it is not a very good name as the programmer does not have to understand the why, they should only understand that if an annotation is provided then in some cases the type inference can perform better.

We decide on the name **imaginary** type parameters, with a prefix **imag** used in the type annotations as it captures the nature of type parameters, which do not represent any information about the data being held in the class. We do not use the name **phantom** as it is already commonly used to refer to a subset of imaginary type parameters, which are not used as method argument types.

Another important detail to discuss is the validation of the declarations. We follow the constraints of not-leakable type parameters. The programmer can only use imaginary type parameters in the declaration, as a type of a method argument or as a type argument of another generic class, for type parameters that are also imaginary.

7.1.1 Interaction with variance

Kotlin supports providing the variance on a type parameter in the type declaration. The fact that the programmer marks the type parameter as imaginary does not imply anything about the variance. Imaginary type parameters are by default invariant. It is possible to mark the type covariant/contravariant on an imaginary type parameter by adding respectively `out`/`in` as a prefix, before the `imag` prefix. In the example below we mark the type parameter as covariant.

```
sealed class Option<out imag T>
class Some<out T>(x: T): Option<T> {}
class None<out imag T>(): Option<T> {}
```

Interestingly to note, if we mark the type parameter as contravariant the compiler enforces the same constraints as for the imaginary type parameter. It is still possible to mark an imaginary type parameter covariant only in the case when the type parameter is phantom. It comes from the fact that phantom type parameters are bivariant.

Semantics of variance are not relevant to our work, the algorithm has to generate proper constraints and constraint solver solve then taking the variance into account. Imaginary type parameters semantics only concern the control flow of the algorithm and the inference progression property, which does not rely either on constraints or the constraint solver.

7.1.2 Interaction with overloading

Kotlin supports method overloading. Overloading might introduce problems when intersecting with imaginary type parameters. The example below works.

```
class Collection<imag T> {
        ...
        fun add(x: T) {}
}
val z = Collection().add(1)
```

Assume that the collection is a part of the external library. If authors introduced a new method overload as presented in the example below.

```
class Collection<imag T> {
        ...
        fun add(x: T) {}
        fun add(x: Int) {}
}
val z = Collection().add(1)
```

This might break the code that depends on the library because of the ambiguous overload resolution. However, in this case, the dependent code would have form:

```
class Collection<imag T> {
        ...
        fun add(x: T) {}
        fun add(x: Int) {}
}
val z = Collection<String>().add(1)
```

In this case, this would not result in breaking change on the dependent code side. Not marking the type parameter as imaginary would force the dependent code to always resolve this parameter, which could potentially help avoid a breaking change.

The best route of action to address this issue is to suggest to programmers that they should not use imaginary type parameters in methods they expect might be overloaded in the future. If type parameters are not imaginary then inference of those is always forced, possibly avoiding, at least some, breaking changes. Additionally, it is another argument for the explicit imaginary type parameters annotations.

7.2 Implementation

We partially implement the described feature in the Kotlin compiler. We extended the constraint solver to comply with the semantics defined in the chapter 6, to validate if such an extension is possible. For the declarations of imaginary type parameters we assume the following simplifications:

1. Instead of the `imag` prefix syntax, we assume the imaginary type parameters have such prefix in their names.
2. We do not provide the static analysis verifying the provided constraints, instead it is the role of the programmer to mark those correctly

```
class Option<nrT>() {}
fun test_1() {
    val xz = Option()
    val u: Option<Int> = xz
}
```

Our implementation does not fail for the given example, though it does not properly update the variable type. It also does not work with mutable variables. Still it propagates the types to the previous statements. We focused only on the inference algorithm on Kotlin's intermediate representation FIR. The solution does not work end-to-end. Below we see the inference result as an FIR program.

```
public final class Option<nrT> : R|kotlin/Any| {
    public constructor<nrT>(): R|Option<nrT>| {
        super<R|kotlin/Any|>()
    }

}
public final fun test_1(): R|kotlin/Unit| {
    lval xz: R|Option<TypeVariable(nrT)>| = R?C|/Option.Option|<R|kotlin/Int|>()
    lval u: R|Option<kotlin/Int>| = R?C|<local>/xz|
}]
```

We can see that the type of the variable `xz` is not properly updated with type information, however, the initializer expression is.

Those limitations could be easily addressed in the proper implementation. Our implementation serves mostly as verification of feasibility to implement the algorithm not as a working solution. An obstacle in implementing the solution to Kotlin's compiler is the fact that many places in the codebase rely on the assumption that variables always have fully inferred types.

Chapter 8

Summary

We successfully managed to propose an improvement to the Kotlin type inference algorithm. We achieved so by identifying the property of type parameters **not-leakable**. We prove the type inference progression theorem for the programs with type variables on not-leakable positions. We use the theorem to improve the reference type inference algorithm. The new algorithm can type some common scenarios without explicit annotations. It specifically helps in cases when the type is built in multiple statements, like working with ADTs or building a list.

8.1 Limitations and future work

In this section, we provide a list of limitations of our solution and possibilities for future work.

8.1.1 Production grade implementation of NTL type inference

We have not implemented the technique into the Kotlin compiler properly. An interesting opportunity for future work is completing this implementation. Such implementation should consider more interactions with other features like builder-style inference or smart casts.

8.1.2 NTL type inference correctness validation

We defined the NTL type inference as an application of the main result of our work, the type inference progression. We do not provide any formal proof of NTL type inference's correctness. Future work could include investigating the correctness properly. The algorithm itself is destined to be used in a practical setting, thus such proof itself might not hold that much value if we extended the type system.

Another possible approach to validate the algorithm is to build an extensive test set for a reference implementation.

8.1.3 Downcasting

To extract the value of an algebraic data type we need to downcast it. In the subtyping-based representation of algebraic data types, downcasting has a role akin to pattern

matching. Kotlin supports downcasting through smart casts. This opens a possibility to leak a type variable. If `Option`$[\alpha]$, gets downcasted to `Some`$[\alpha]$, it is possible to leak α and use it as a receiver.

The existence of this case does not make the technique useless. For the downcasting, the full type resolution can still be enforced. In practice, types are composed in different functions/methods than decomposed, it is not usual that the programmer builds the type and immediately decompose it. Our improvements help mostly with the composition.

An idea of future work is to investigate if this could be avoided, instead of enforcing full resolution of a type used in smart cast.

8.1.4 More properties of imaginary type parameters

We have found that imaginary type parameters have an interesting property, non-leakability, which we can use to improve the type inference algorithm in Kotlin. However, we believe that those could have more properties worth studying. As an example:

```
class BlackBox[T] {}

class ClassName {
    fun foo[T](v: BlackBox[T]): T {}
}
```

We believe that the method `foo` could not be implemented, as for `BlackBox[T]`, `T` is also not-leakable, thus we cannot implement a function that returns this type in a leakable way if it is only provided in a non-leakable way. An interesting opportunity is to explore such property further, validate if that is true, and look for interesting applications in static analysis.

8.1.5 Partial order on IFJ

We have mentioned that IFJ encapsulates the idea of type inference as a process, there might be programs that have more and less information inferred. However, what more and less means in this context is merely intuition. An interesting opportunity for future work is formalizing such relation of partial order on IFJ programs. As an intuitive example of two types that could be comparable in such ordering:

$$Pair[\alpha, Int] \leq Pair[Int, Int]$$

It allow saying which inference result is better, which could be used for example in inference algorithm to decide which of ambiguous inference results is better.

Bibliography

Bhanuka, Ishan et al. (Oct. 16, 2023). "Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference". In: *Reproduction Package for "Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference"* 7 (OOPSLA2), 237:431–237:459. DOI: 10.1145/3622812. URL: https://doi.org/10.1145/3622812 (visited on 07/12/2024).

Breitner, Joachim et al. (Aug. 19, 2014). "Safe Zero-Cost Coercions for Haskell". In: *ACM SIGPLAN Notices* 49.9, pp. 189–202. ISSN: 0362-1340. DOI: 10.1145/2692915.2628141. URL: https://doi.org/10.1145/2692915.2628141 (visited on 02/22/2024).

Cremet, Vincent et al. (2006). "A Core Calculus for Scala Type Checking". In: *Mathematical Foundations of Computer Science 2006*. Ed. by Rastislav Královič and Paweł Urzyczyn. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 1–23. ISBN: 978-3-540-37793-1. DOI: 10.1007/11821069_1.

Dolan, Stephen and Alan Mycroft (Jan. 1, 2017). "Polymorphism, Subtyping, and Type Inference in MLsub". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. New York, NY, USA: Association for Computing Machinery, pp. 60–72. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009882. URL: https://doi.org/10.1145/3009837.3009882 (visited on 01/14/2024).

Dunfield, Jana and Neel Krishnaswami (May 25, 2021). "Bidirectional Typing". In: *ACM Computing Surveys* 54.5, 98:1–98:38. ISSN: 0360-0300. DOI: 10.1145/3450952. URL: https://dl.acm.org/doi/10.1145/3450952 (visited on 02/03/2024).

Feitosa, Samuel da Silva et al. (Sept. 23, 2019). "An Inherently-Typed Formalization for Featherweight Java". In: *Proceedings of the XXIII Brazilian Symposium on Programming Languages*. SBLP '19. New York, NY, USA: Association for Computing Machinery, pp. 11–18. ISBN: 978-1-4503-7638-9. DOI: 10.1145/3355378.3355385. URL: https://doi.org/10.1145/3355378.3355385 (visited on 06/26/2024).

Garrigue, Jacques (2004). "Relaxing the Value Restriction". In: *Functional and Logic Programming*. Ed. by Yukiyoshi Kameyama and Peter J. Stuckey. Berlin, Heidelberg: Springer, pp. 196–213. ISBN: 978-3-540-24754-8. DOI: 10.1007/978-3-540-24754-8_15.

Heeren, Bastiaan, Jurriaan Hage, and S. Swierstra (Jan. 1, 2003). "Constraint Based Type Inferencing in Helium". In.

Hoang, My and John C. Mitchell (Jan. 25, 1995). "Lower Bounds on Type Inference with Subtypes". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. New York, NY, USA: Association for Computing Machinery, pp. 176–185. ISBN: 978-0-89791-692-9. DOI: 10.1145/199448.199481. URL: https://dl.acm.org/doi/10.1145/199448.199481 (visited on 01/28/2024).

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler (May 1, 2001). "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems* 23.3, pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505. URL: https://dl.acm.org/doi/10.1145/503502.503505 (visited on 02/22/2024).

Igarashi, Atsushi and Mirko Viroli (June 10, 2002). "On Variance-Based Subtyping for Parametric Types". In: *Proceedings of the 16th European Conference on Object-Oriented Programming*. ECOOP '02. Berlin, Heidelberg: Springer-Verlag, pp. 441–469. ISBN: 978-3-540-43759-8.

Jones, Simon Peyton et al. (Jan. 2007). "Practical Type Inference for Arbitrary-Rank Types". In: *Journal of Functional Programming* 17.1, pp. 1–82. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796806006034. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/practical-type-inference-for-arbitraryrank-types/5339FB9DAB968768874D4C20FA6F8CB6 (visited on 07/19/2024).

Kotlin documentation (n.d.). URL: https://kotlinlang.org/docs (visited on 02/21/2024).

Kotlin language specification (n.d.). URL: https://kotlinlang.org/spec/introduction.html (visited on 01/21/2024).

Leijen, Daan and Erik Meijer (Dec. 31, 2000). "Domain Specific Embedded Compilers". In: *ACM SIGPLAN Notices* 35.1, pp. 109–122. ISSN: 0362-1340. DOI: 10.1145/331963.331977. URL: https://dl.acm.org/doi/10.1145/331963.331977 (visited on 02/21/2024).

Milner, R., L. Morris, and M. Newey (1975). "A Logic for Computable Functions with Reflexive and Polymorphic Types". In: *Proceedings of the Conference on Proving and Improving Programs*. IRIA-Laboria, pp. 371–394. URL: https://www.research.ed.ac.uk/en/publications/a-logic-for-computable-functions-with-reflexive-and-polymorphic-t (visited on 02/04/2024).

OCaml Weak Type Variables (n.d.). URL: http://ocamlverse.net/content/weak_type_variables.html (visited on 07/12/2024).

Odersky, Martin, Martin Sulzmann, and Martin Wehr (1999). "Type Inference with Constrained Types". In: *Theory and Practice of Object Systems* 5.1, pp. 35–55. ISSN: 1096-9942. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9942%28199901/03%295%3A1%3C35%3A%3AAID-TAPO4%3E3.0.CO%3B2-4 (visited on 07/12/2024).

Odersky, Martin, Christoph Zenger, and Matthias Zenger (Jan. 1, 2001). "Colored Local Type Inference". In: *ACM SIGPLAN Notices* 36.3, pp. 41–53. ISSN: 0362-1340. DOI: 10.1145/373243.360207. URL: https://doi.org/10.1145/373243.360207 (visited on 02/03/2024).

Parreaux, Lionel (Aug. 3, 2020). "The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl)". In: *Simple-sub source code, including comparison tests with MLsub* 4 (ICFP), 124:1–124:28. DOI: 10.1145/3409006. URL: https://doi.org/10.1145/3409006 (visited on 07/12/2024).

Pierce, Benjamin C. (2002). *Types and Programming Languages*. 1st. The MIT Press. ISBN: 0262162091.

Pierce, Benjamin C. and David N. Turner (Jan. 1, 2000). "Local Type Inference". In: *ACM Transactions on Programming Languages and Systems* 22.1, pp. 1–44. ISSN:

0164-0925. DOI: 10.1145/345099.345100. URL: https://dl.acm.org/doi/10.1145/345099.345100 (visited on 02/16/2024).

Pottier, François (June 15, 1996). "Simplifying Subtyping Constraints". In: *SIGPLAN Not.* 31.6, pp. 122–133. ISSN: 0362-1340. DOI: 10.1145/232629.232642. URL: https://doi.org/10.1145/232629.232642 (visited on 07/14/2024).

— (Sept. 29, 1998). "A Framework for Type Inference with Subtyping". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. New York, NY, USA: Association for Computing Machinery, pp. 228–238. ISBN: 978-1-58113-024-9. DOI: 10.1145/289423.289448. URL: https://doi.org/10.1145/289423.289448 (visited on 07/14/2024).

Stucki, Nicolas Alexander, Aggelos Biboudis, and Martin Odersky (2017). "Dotty Phantom Types". In: p. 5. URL: http://infoscience.epfl.ch/record/273667.

Tate, Ross (2013). "Mixed-Site Variance". In: *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages*. Indianapolis, IN, USA. URL: http://www.cs.cornell.edu/~ross/publications/mixedsite/.

Trifonov, Valery and Scott F. Smith (Sept. 24, 1996). "Subtyping Constrained Types". In: *Proceedings of the Third International Symposium on Static Analysis*. SAS '96. Berlin, Heidelberg: Springer-Verlag, pp. 349–365. ISBN: 978-3-540-61739-6.

Vytiniotis, Dimitrios et al. (Sept. 2011). "OutsideIn(X) Modular Type Inference with Local Assumptions". In: *Journal of Functional Programming* 21.4-5, pp. 333–412. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796811000098. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/outsideinx-modular-type-inference-with-local-assumptions/65110D74CF75563F91F9C68010604329 (visited on 07/19/2024).

Weirich, Stephanie et al. (Jan. 26, 2011). "Generative Type Abstraction and Type-Level Computation". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. New York, NY, USA: Association for Computing Machinery, pp. 227–240. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926411. URL: https://doi.org/10.1145/1926385.1926411 (visited on 02/18/2024).