FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Exception flow analysis for Kotlin
Student:	Bc. Filip Dolník
Supervisor:	Ing. Jiří Hunka
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

This work aims to develop a static analysis tool that will track exception propagation in a program. The primary purpose of this tool is to help Kotlin backend developers with documenting exceptions in their API.

Steps to cover:
1. Analyze the Kotlin language and its properties that affect exception flow.
2. Design an algorithm for the static analysis of exception flow in Kotlin.
3. Create a working prototype written in Kotlin based on that design.
4. Write acceptance tests to verify the prototype.
5. Assess the prototype usability in a production environment.
6. Propose possible future improvements.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 10 February 2022 in Prague.

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Master's thesis

Exception flow analysis for Kotlin

Bc. Filip Dolník

Supervisor: Ing. Jiří Hunka

May 5, 2022

Acknowledgements

First, I want to thank my supervisor Ing. Jiří Hunka, for his guidance and the advice he gave me. Additionally, I want to thank my friends who helped me with the proofreading of this thesis. I want to thank the people who have prepared and taught the System Programming specialization at FIT CTU. The knowledge I gained from the courses was invaluable for doing this thesis. Finally, I would like to thank my family and friends for their support during my studies, especially while working on this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague on May 5, 2022 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

DOLNÍK, Filip. *Exception flow analysis for Kotlin.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Ošetřování výjimek je nezbytnou součástí vývoje softwaru, je však zároveň také jednou z nejnáročnějších. Chyby ve zpracování výjimek mohou způsobit mnoho problémů – od špatného uživatelského rozhraní až po bezpečnostní zranitelnosti. Tyto chyby se často vyskytují na rozhraní dvou systémů. Důvodem je mimo jiné to, že API těchto systémů obvykle není správně zdokumentováno, zejména pokud jde o výjimky.

Ruční dokumentování výjimek v API je časově náročné, a navíc náchylné na chyby. Tato práce navrhuje řešení v podobě automatické tvorby této dokumentace. Tím řešením je statická analýza propagace výjimek, která vytvoří seznam potenciálně vyhozených výjimek z API.

Výsledkem této práce je návrh této statické analýzy, konkrétně pro programovací jazyk Kotlin. Vytvořený návrh lze použít k implementaci nástroje pro automatické dokumentování API. Navržená analýza je založena na abstraktní interpretaci a klade důraz na vysokou přesnost. Analýza si poradí jak s objektově orientovanými, tak s funkcionálními aspekty Kotlinu. Návrh je modulární a používá dvě různé IR. Díky tomu lze snadno přidat podporu dalších programovacích jazyků.

Součástí této práce je také implementace prototypu navržené analýzy. Vyvinutý prototyp interně používá kompilátor Kotlinu pro převod analyzovaného zdrojového kódu do IR Kotlinu. Tato IR je poté převedena na jednu ze dvou IR statické analýzy. Tento proces převodu je v práci podrobně zdokumentován spolu s významnou částí IR Kotlinu. Vzniklá dokumentace také vysvětluje sémantiku funkcí Kotlinu, zejména těch, které souvisí s propagací výjimek. Tato

dokumentace může být užitečná při implementaci zásuvných modulů pro kompilátor Kotlinu nebo jiných statických analýz.

Vytvořený prototyp lze použít na experimentování s většinou vzniklého návrhu analýzy. Jeho implementace však není úplná, takže jej nelze použít k analýze reálných projektů. Součástí prototypu je mnoho akceptačních testů zaměřených na propagaci výjimek. Prototyp i tyto testy lze použít při implementaci nástroje na dokumentování API.

Klíčová slova statická analýza, tok výjimek, propagace výjimek, doménové výjimky, dokumentace API, Kotlin, kompilátor Kotlinu, Kotlin IR

Abstract

Exception handling is one of the most challenging parts of the software development, yet it is essential. An improper exception handling can cause many problems – from bad UX to security vulnerabilities. Mistakes in exception handling frequently occur at the boundaries of two systems. Part of the reason is that the API of those systems is usually not correctly documented, especially when it comes to exceptions.

Manually documenting API exceptions is a time-consuming and error-prone process. This thesis proposes a solution for creating the documentation automatically. The solution is a static analysis of exception propagation. This static analysis produces a list of potentially thrown exceptions by API endpoints.

The result of this thesis is a design of such static analysis for the Kotlin programming language. The created design can be used to implement a tool for documenting APIs. The designed analysis is based on abstract interpretation, focusing on high precision. The analysis can handle both object-oriented and functional programming aspects of Kotlin. The design is modular and uses two different intermediate representations. As a result, the design can be easily extended to support other programming languages.

This thesis also includes a prototype of the proposed analysis. The prototype internally uses the Kotlin compiler to parse the analyzed source code into the Kotlin IR. The produced Kotlin IR is then converted to one of the intermediate representations. The thesis describes this process in detail and, at the same time, it documents a significant part of the Kotlin IR. The documentation

also explains the semantics of Kotlin features that the analysis handles. This documentation can be helpful when implementing compiler plugins or other static analyses.

The created prototype can be used to experiment with most of the analysis design. However, it does not implement the entire design, so it cannot be used to analyze real-world projects. The prototype contains many acceptance tests focused on the exception propagation. Together the prototype and the tests can be used as a starting point for implementing the tool for documenting APIs.

Keywords static analysis, exception flow, exception propagation, domain exceptions, API documentation, Kotlin, Kotlin compiler, Kotlin IR

Contents

List of Listings

Introduction

I have been working as a mobile and later back-end software developer for the past several years. During that time, I have participated in designing back-end APIs, and architectures for both mobile and back ends on multiple projects. These projects were in different technologies and domains, yet the team always ran into the same problem. The problem always has been to design, implement and document an API such that all domain exception paths are handled properly and cleanly.

Domain exception (or domain exception path) is a different concept from ordinary programming exceptions/errors. Domain exceptions mostly come from incorrect user input. An example of such a situation is a user trying to log in with an incorrect password. A domain exception must be explicitly accounted for and handled by the code. In contrast, programming errors that come from the developer's mistakes are unintentional and should not be directly handled. Null pointer dereference, or division by zero are some examples of such errors.

In my experience, a design of an exception handling has a significant impact on the code quality and the application's UX. Incorrect exception handling can have catastrophic consequences. On the back end, it can cause data corruption or security vulnerabilities. The consequences are usually less severe on the front end, but they are still significant. If the exception is suppressed, the functionality will do nothing, and the app will not tell the user what happened. If the exception is not suppressed, the app will crash. Both situations lead to confused users and bad UX at best.

A promising solution is to use the programming language built-in exceptions with a separate class for each domain exception. Using these single-purpose exceptions has several advantages. They can carry precise information to the user and therefore allow for good UX. They also clearly communicate their domain meaning, making the code more readable. Moreover, they do not significantly interfere with the production code; thus, they do not impact maintainability. However, using single-purpose exceptions is not practical because documentation of these exceptions involves too much manual work.

The primary goal of this thesis is to design a static analysis that tracks exception propagation in Kotlin programs. The static analysis will help developers document domain exceptions thrown by their API. Since the documentation process will be automatic, it will make the use of single-purpose exceptions feasible.

The secondary goal is to implement a prototype of the analysis. The prototype will include automated acceptance tests to verify its correctness. The created prototype's usability will be assessed based on the experience gained during its development. An additional goal is to analyze the Kotlin semantics that affect exception flow. This step is necessary to correctly design the static analysis. The remaining goal then is to propose solutions for future development.

This document is structured into five chapters. The first chapter explains the problem of documenting exceptions in API and contains an example code that uses single-purpose exceptions. The chapter also presents some strategies for documenting exceptions and libraries for implementing static analysis in Kotlin. The end of the chapter describes the current state of the art in the area of static analysis with a focus on exception flow.

The following chapter is about Kotlin and its features related to exception analysis. The chapter also includes a high-level explanation of how the Kotlin compiler implements the features. Chapter three describes the final design of the static analysis created in this thesis. The fourth chapter captures the static analysis development process and the challenges that had to be overcome. It also contains some interesting implementation details. The final chapter assesses the prototype's usability in a production setting and proposes future improvements.

Chapter 1

Initial analysis

This chapter describes my first steps when working on this project. First, I have defined the problem I am trying to solve in this thesis. A good understanding of the target use case was crucial because designing a static analysis is all about making the right tradeoffs. Once I had a good idea of what I wanted to achieve, I assessed options for solving the problem. There were more things to consider than just a static analysis.

After I knew that a static analysis was the right choice, I looked into some options for analyzing a program written in Kotlin. This decision was critical as it dictated the whole direction of the implementation and would not be possible to take back. Furthermore, I have researched existing static analysis solutions and algorithms. Finally, I was able to start with the actual static analysis design and implementation. These two steps are covered in (3) and (4).

1.1 The problem of documenting exceptions

A proper domain exception handling is complex and affects the whole project architecture. The exception handling requires a well-thought-out solution not to impact the codebase negatively. The complexity is primarily caused by the inherent non-locality of any possible implementation. A domain exception is typically thrown by domain logic deep in the back-end code. The back end must inform the front end (mobile or web app) about the exception via the API. The information must be therefore propagated through all the back-end architecture layers. After receiving the information, the front end must also

propagate it throughout its domain logic to the user interface. Moreover, the user interface must understand the domain meaning of the exception path. Only then it can correctly inform the user about the problem.

Additionally, there is usually a large number of different domain exceptions. Each happy path (or functionality) typically has several associated exception paths. Therefore, a project has many more exception paths than actual features. Each exception path needs some code that handles it, which can lead to many *if* statements in the code. Combined with the non-locality, these *ifs* can be spread throughout all code and across all platforms. This inevitably has an impact on code maintainability.

Exceptions always carry some information, and the front end uses that information to communicate with the user. For example, a user tried to sign-up with an already taken username. In this example, the back end responded with an exception, and the front end decoded it into one of the following error messages:

- Username already exists.
- Failed to create an account.
- Something went wrong.

In the first example, the front end can present a proper error message and maybe highlight the *username* field in red. On the other hand, the two remaining messages are examples of bad UX. The second message at least gives the user some information, but it can still hide a different problem like a too weak password. For this reason, the front end must be able to extract necessary information from the exception, and that process must be part of the API. The more relevant information the exception carries, the better the UX can be.

The non-locality makes it challenging to ensure that the front end correctly handles every exception. The front-end developers need to know precisely what exceptions each API endpoint throws and how to present them to the user. Having an API documentation is necessary; otherwise, the developers must rely on reverse engineering and guessing. It is almost impossible to correctly implement the front end of any larger app without its API documentation.

The exact documentation form is not that important. It can be, for example, written in a plain text or defined using classes and annotations. What is important is that there is an accurate list of exceptions associated with each endpoint.

Maintaining such a list is challenging as the project grows in size. Every time a new exception is added/changed/removed, it is necessary to update the documentation of all the affected endpoints. Maintaining the documentation can be a lot of work, as I will demonstrate in the following section.[†]

1.2 Exception handling example

I have prepared a simplified example to demonstrate why it can be challenging to document all the exceptions properly. For the demonstration, I will use a part of a fictional back end written in Spring [33]. The back-end design is based on an Onion architecture [30] and DDD [6]. To keep the example shorter, I intentionally omitted code unrelated to domain exception handling. I am also presenting the code as if it was kept together in a few files. However, these classes would be in separate files spread across multiple modules in a real project.

The example showcases an implementation of a single endpoint that allows the user to book a room. In order for the booking to be successful, the following pre-conditions must be satisfied:

- The user must be logged in.
- The user must be an admin.
- The booked room must exist.
- The booked room must not be already booked.
- The user cannot create a booking in the past.

There is a significant disproportion in the features and exception path count, even in this simple use case. There are already five exception paths, and it would not be that hard to come up with more.

[†]Also, the front-end code must be updated to reflect the change, but that is a concern for an API versioning strategy.

The example starts in the listing (1.1), which shows the declarations of used exceptions. The code uses the single-purpose exceptions strategy, so each exception corresponds to a single exception path. Therefore the front end can easily decode the meaning of each exception. These exceptions could also carry additional information such as entity IDs or debug information.

```
class NotAuthenticated: DomainException()
class NotAuthorizedToCreateBooking: DomainException()
class RoomNotFound: DomainException()
class RoomIsAlreadyBooked: DomainException()
class BookingInPast: DomainException()
```

Listing 1.1: Domain exceptions

The API endpoint is declared in a `BookingController` from listing (1.2). The *controller* maps domain objects to and from DTOs and calls the appropriate method from an `ApplicationService`. The example assumes the presence of a special exception handler that catches all thrown exceptions and maps them accordingly.

```
@Controller
@RequestMapping("/booking")
class BookingController(
    val bookingApplicationService: BookingApplicationService,
): BaseController() {
    @PostMapping
    fun createBooking(dto: NewBookingDto): BookingDto {
        val booking = bookingApplicationService.createBooking(
            user = authenticatedUser,
            roomId = dto.roomId,
            date = dto.date,
        )

        return booking.toDto()
    }

    fun Booking.toDto(): BookingDto = ...
}
```

Listing 1.2: BookingController

In this case, the `BookingController` calls the `authenticatedUser` property declared in the `BaseController` that is shown in the listing (1.3). This property

checks the first requirement (if the user is logged in) and throws an exception if not.

```
abstract class BaseController {
    protected val authenticatedUser: User
       get() {
           val isAuthenticationValid = ...
           if (!isAuthenticationValid) {
               throw NotAuthenticated()
           }
           ...
       }
}
```

Listing 1.3: BaseController

Listing (1.4) then shows `BookingApplicationService` that wires together all the domain logic. It does not directly throw any exception, but it calls multiple methods that do.

```
@ApplicationService
class BookingApplicationService(
    val bookingAuthorizationService: BookingAuthorizationService,
    val bookingRepository: BookingRepository,
    val roomRepository: RoomRepository,
) {
    fun createBooking(user: User, roomId: Room.Id, date: LocalDate):
    ↪  Booking {
        bookingAuthorizationService.verifyUserCanCreateBooking(user)

        val room = roomRepository.get(roomId)

        val booking = Booking.new(user, room, date)

        bookingRepository.add(booking)

        return booking
    }
}
```

Listing 1.4: BookingApplicationService

`BookingAuthorizationService` from listing (1.5) checks if the user is an admin (the second rule) and throws an exception otherwise. More complex services might need direct access to repositories making the exception propagation even more complicated.

```
@Service
class BookingAuthorizationService {
    fun verifyUserCanCreateBooking(user: User) {
        val canCreateBooking = user.role == User.Role.Admin
        if (!canCreateBooking) {
            throw NotAuthorizedToCreateBooking()
        }
    }
}
```

Listing 1.5: BookingAuthorizationService

Then there is `RoomRepository` and `DatabaseRoomRepository`. Both are shown in the listing (1.6). Even though they are presented together, the interface and the implementing class are often located in different modules. According to the architecture, the interface belongs to the *domain* module, while its implementation should be in the *infrastructure* module. Repositories work directly with the database and typically produce many domain exceptions like `NotFound` or `AlreadyExists`. In this case, the repository checks the requirement that the booked room must exist.

```
@Repository
interface RoomRepository {
    fun get(id: Room.Id): Room
}

class DatabaseRoomRepository: RoomRepository {
    override fun get(id: Room.Id): Room =
        find(id) ?: throw RoomNotFound()

    fun find(id: Room.Id): Room? = ...
}
```

Listing 1.6: RoomRepository

The second repository is shown in the listing (1.7). It contains an example of a more complicated domain rule / database query – the requirement that the room is not already booked.

```
@Repository
interface BookingRepository {
    fun add(booking: Booking)
}

class DatabaseBookingRepository: BookingRepository {
    override fun add(booking: Booking) {
        if (isRoomBooked(booking.roomId, booking.date)) {
            throw RoomIsAlreadyBooked()
        }

        ...
    }

    fun isRoomBooked(roomId: Room.Id, date: LocalDate): Boolean =
    ↪  ...
}
```

Listing 1.7: BookingRepository

The last condition is checked during instantiation of the `Booking` entity (listing (1.8)). The check is intentionally not performed in the primary constructor. Doing so would prevent the back end from retrieving past bookings, which is not prohibited by the domain.

The problem with documentation maintenance is that a single code change often influences the behavior of multiple endpoints/features. If that happens, all transitively affected methods must be checked. Adding documentation for new features is also not trivial since it is common practice to reuse existing code (especially repositories and entities).

For example, there is a new domain requirement that the room cannot be booked on weekends. I would personally put this check in the `Booking` entity below the check for no past booking requirement. There is only one endpoint in this case, so it is not that hard to remember to update its documentation. However, if the `new` method is called from multiple places, it becomes much more difficult.

Keeping track of all the possible exceptions for each endpoint is equivalent to keeping track of all the methods they call internally. Doing this entirely

```
@Entity
class Booking internal constructor(
    val id: Id,
    val userId: User.Id,
    var roomId: Room.Id,
    var date: LocalDate,
) {
    companion object {
        fun new(user: User, room: Room, date: LocalDate): Booking {
            val isInPast = date.isBefore(LocalDate.now())
            if (isInPast) {
                throw BookingInPast()
            }

            return Booking(Id(UUID.randomUUID()), user.id, room.id,
            ↪  date)
        }
    }

    data class Id(val value: UUID)
}
```

Listing 1.8: Booking entity

manually is doable on a small scale, but not if there are dozens or hundreds of endpoints.

1.3 How to document exceptions

This section introduces several strategies for maintaining domain exception documentation. I will focus on those that can help developers with the problem presented in the previous two sections. The strategies can be categorized by an agent that makes sure the documentation is correct. It can be manual – done by a human, or automatic with some potential manual intervention.

Maintaining the documentation just manually is not practical, as I have explained in the previous section. The upside is that the developer can make sure the documentation is accurate. The previous statement is, however, valid only in theory. In practice, humans make mistakes.

Automatic solutions usually rely on some form of static or dynamic analysis. Making the process automatic allows the developers to focus on more exciting tasks or tasks that cannot be automated. For this reason, automatic solutions

tend to scale much better with the project size. Computers also do not make the same kind of mistakes as humans. Nonetheless, they have a different problem: Rice's theorem [31] puts a fundamental limit on the accuracy of any automatic analysis. No algorithm will correctly analyze any non-trivial property for every possible program. However, it is possible to design an analysis that will only make a specific type of error. What type of error the analysis makes is related to the concept of soundness and completeness.

Soundness typically means that the analysis catches all errors. Completeness is the dual opposite, and it means that the analysis catches only actual errors (does not produce false warnings). Both terms have slightly subjective meaning that depends on the analysis goal. Typically, the notion of soundness is defined such that its absence has more severe consequences than the absence of completeness.

For the purpose of documenting API exceptions, the analysis is sound if it reports all possible exceptions each endpoint can throw. Complementary, it is complete if it reports only possible exceptions. If the analysis is not sound, then it may miss some exceptions. Those missed exceptions might not get handled by the front end and cause problems.

If the analysis is incomplete, it may report more exceptions than necessary. As a result, the developer would likely implement unnecessary code to handle those impossible exceptions. However, the application would still work – only the unnecessary code would never be executed.

Because of Rice's theorem, no analysis can be sound and complete simultaneously. Only one of those properties can be fully satisfied. Given the definition, soundness is usually preferred over completeness. So the analysis can produce false warnings. The precision of the analysis then states how many false warnings it produces. It might make sense to sacrifice some soundness in exchange for higher precision, at least in some situations.[†]

1.3.1 Checked exceptions

The idea of checked exceptions is to allow the compiler to check if all the thrown exceptions are correctly documented. Each method that directly or

[†]This and the previous two paragraphs are based on the article "Soundness and Completeness: With Precision" [24]. I have only touched the surface of this topic, so I recommend reading the article for more information.

indirectly throws a checked exception must state that fact as part of its signature. The information about thrown exceptions is transitively propagated to the `Controller` and can be used for the API documentation. This approach is generally sound but not complete since methods can declare they throw some exception when they actually do not. Also, even if a method can throw an exception in some situations, it does not mean it will never happen.

Checked exceptions have the advantage of being sound, but they also have some significant downsides. First, they interfere with the whole codebase as every method needs to be aware of them. Because of that, it may be difficult to change the exception logic of code used in multiple places. Adding an exception is fine since the compiler enforces the update of all the calling methods. What causes a problem is when the method no longer throws some exception. The compiler does not check this in all cases, so the developer must manually track down all the affected methods and adjust their signature.

Nevertheless, this is not the primary problem. The primary problem is that checked exceptions are from Java, and have not been adopted by Kotlin. There are good reasons for their absence, as explained in the article "Kotlin and Exceptions" [5]. The primary one is that they do not work well with functional programming.

1.3.2 Typed results

The previous article [5] suggests typed results as a replacement for checked exceptions. Result pattern is a fundamentally different strategy of exception handling that does not rely on built-in exceptions. Each function that uses this pattern returns a `Result` object that holds either the originally returned value or some exception. The information about exceptions is therefore propagated using a function return value.

Both results and built-in exceptions can achieve the same effect. The primary difference is that results are more explicit than built-in exceptions. The explicitness makes results safer (at least in theory), at the cost of more code to be written. How much the pattern interferes with the codebase depends on the used library.

There are two different types of results: typed and untyped. A typed result specifies which exceptions it can hold using the language type system. Untyped

results can contain any exception and, for that reason, are not a suitable replacement for the checked exceptions.

Typed results and checked exceptions are similar concepts because both rely on the compiler to verify their correct usage. They also share the same properties in terms of soundness, completeness, and maintainability. The advantage of typed results is that they work with functional programming. However, in my opinion, they are also not a suitable solution to the problem as they require the developer to write too much boilerplate code. I have yet to see an implementation that would have overhead at most that of checked exceptions.[†]

1.3.3 Automated acceptance tests

Both previous strategies have issues with detecting the removal of an exception. They also introduce a lot of boilerplate to the production code. Automated acceptance tests[‡] do not have any of those problems. They fall somewhere between manual and automatic approaches since the developer still has to write the tests.

Acceptance tests can verify the implementation of both the happy paths and the exception paths. Therefore, they can verify that some exception really occurs. At the same time, they provide an example of the situation in which the exception occurs. Furthermore, they ensure that the exception path is still present as the project evolves.

If the automated tests are written in a readable way, they can serve as implicit API documentation. A front-end developer can look at the acceptance tests and see all the possible exceptions and scenarios in which they happen. Alternatively, it is possible to write some script that will automatically update the documentation based on the code in the tests.

Automated tests used in this way are an example of a complete but not sound analysis. If an exception path test passes, it proves that there is a scenario in which the exception occurs (assuming test correctness). On the other hand absence of an exception path test does not mean the path does not exist. So this

[†]The situation could be significantly improved if Kotlin eventually adds support for union types.

[‡]Acceptance tests do not have a uniform definition. For this text, I assume they test the back end in production-like settings by calling the endpoints the same way as the front end.

strategy is not a complete solution to the documentation problem, but it can be used to complement another strategy. Such a combination can achieve soundness, have high precision, and be maintainable.

1.3.4 Dynamic analysis

Automatic acceptance tests have the weakness of not being able to help the developer with finding all the possible exceptions. Dynamic analysis can be used together with the tests to provide additional assistance. Dynamic analysis, in general, looks for some properties of the program during the program's execution. It requires the program to be actually running with real inputs and in a real environment. One possibility to achieve that is to use the acceptance tests and run the analysis during their execution.

In this case, the analysis can look for exceptions that the tests miss. It builds on the following idea: If a method throws an exception when called from one endpoint, then it likely throws the same exception when called from all other endpoints. So if an exception is observed during the testing of only part of the endpoints that call the method, it is probably a mistake.

The analysis as presented above is neither sound nor complete. It is not sound as it may miss code inside possible execution paths not visited by any test. This problem can be partially mitigated by extending the analysis to check code coverage. The analysis is not complete for a similar reason as in the case of checked exceptions: For some endpoints, it may be impossible to call a method such that it throws an exception, even if other endpoints can achieve that.

The absence of both properties is not necessarily a deal-breaker. The analysis potentially improves the precision of automated testing, and it comes at almost no cost. However, it is not a perfect solution as the analysis quality entirely depends on the quality of the tests.

1.3.5 Static analysis

Static analysis is similar to dynamic analysis in that it also tries to decide some program properties. The difference is that static analysis does so without running the program, so it does not need the acceptance tests. Static analysis can eliminate most problems of dynamic analysis. It is at the cost of introducing new ones, however less significant, as I will explain below.

The checked exceptions presented earlier are in some sense implemented as a static analysis done by the compiler. Their implementation is relatively primitive, so it relies on the developer to provide enough information. The limitation of not being interoperable with a functional programming style comes from this reliance. However, this is not a fundamental problem. If the compiler could automatically infer all the method signatures, the checked exceptions would work well even with features such as high-order functions.

So the simplified idea of the static analysis is to implement checked exceptions but fully automatic. The analysis will produce a list of potentially thrown checked exceptions for each method in the program. The analysis output can be limited to the methods representing the API endpoints which effectively creates the required API documentation.

The static analysis has several key advantages over all previously mentioned strategies:

- It can be sound (at the cost of completeness).
- It does not theoretically interfere with the production code.
- Little to no work is required from the developer.
- Its precision can be incrementally improved.

Depending on the specific implementation, a static analysis has to make a tradeoff between precision, performance, and implementation difficulty. For the use-case of API documentation, high precision is the most desirable property. In fact, it may be desirable to sacrifice some soundness in documented edge case scenarios to achieve higher precision. The reason is that false alarms would lower the analysis usability and perceived reliability, hurting its adoption by the developers.

Performance and implementation difficulties are not that important. The implementation of the analysis is a problem for its author, not for the developer who uses it. As for performance, the acceptance tests presented previously can run for several hours on a sufficiently large project. So it does not matter that much if the analysis is slow. Although fast execution would be advantageous, allowing the developer to run the analysis on their local machine. It just cannot be at the cost of the analysis precision.

1.4 Libraries for static analysis of Kotlin

Static analysis works by examining the program's code similarly to how the compiler does it. It is possible to inspect either the source code or the already compiled native code. Each option offers different advantages and brings unique challenges. Implementing the analysis from scratch is possible regardless of the chosen approach, but that involves much unnecessary work.[†] For that reason using a library is a preferred approach.

There are multiple options for the native code inspection because Kotlin supports multiple platforms: JVM, JavaScript, and native. The native target then supports multiple different architectures and operating systems. The native target uses an LLVM compiler internally, so it can also produce an LLVM bytecode. Any one of the produced representations is viable to use for the analysis. I will consider only the JVM bytecode inspection option since the analysis' primary target is back-end code.

This section will cover the options/libraries I was choosing from for both source code and JVM bytecode analysis. I will also present the advantages and disadvantages that I was considering. At the end of the section, I will summarize and present my choice and its reasoning.

1.4.1 Source code analysis

Kotlin is a young language, and as a result, it does not have that many libraries for static analysis yet. There are essentially only two options to choose from. One possibility is to use the Kotlin compiler [20] itself – which is open-source, and its code can be reused for the analysis. The alternative is to use the KSP library [9].

The KSP library creates a Kotlin compiler plugin that usually generates some additional code during compilation. However, the plugin does not have to generate any new code. Instead, it can just analyze the original code. The problem is that the library is designed only for inspecting class and function level declarations. Since any meaningful exception flow analysis must analyze individual expressions, this library is unsuitable for this project.

[†]Especially in the case of the source code analysis, where it involves the reimplementation of a significant part of the compiler front end.

In summary, the Kotlin compiler is the only option. The compiler has built-in support for plugins that can inspect and modify the program's IR. The plugin is called approximately in the middle of the compilation process. A plugin can specify the time it will be called, but there are some limitations. It always has to be sometime after the front end has already processed the source code, but before the back end compiles it to the native code.

Source code analysis has some significant advantages over compiled code analysis. Source code (or IR in this case) contains more information than the compiled code. The static analysis can use this information to achieve better precision.

For a human, it is generally easier to work with the source code than with the native code. The IR, though less readable than source code, is still significantly better in this regard than bytecode. Better readability of the analyzed code can simplify the analysis development, especially during debugging and testing.

The final mentioned advantage is related to the Kotlin multiplatform support. Because the analysis works with the source code, it is not necessarily bound to a single platform. Even though this thesis focuses only on the JVM target, being able to support Kotlin multiplatform in the future would be an advantage.

However, basing the analysis on the source code comes with a few problems. Some of them are inherent to this approach, and some are related to the necessity of using a compiler plugin. The primary inherent limitation is that any already compiled code cannot be analyzed. The same is true for code written in other languages. For this reason, source code analysis makes it impossible to support closed source libraries. The issue is amplified in Kotlin JVM since many libraries are still written in Java.[†]

As for the compiler, its main problem is a lack of official documentation. The internet also does not contain that much information. The only way to obtain the necessary knowledge is by reverse engineering or contacting someone with the required expertise.

Another problem is caused by the way modules work in Kotlin. By design, the compiler compiles each module independently. So the compiler plugin can

[†]Even the Kotlin standard library makes many calls to the Java standard library.

analyze only one module at a time. Depending on the analysis requirements, this might need to be somehow addressed.

Since a compiler plugin runs before the compiler back end, it does have direct access to the semantics of many advanced Kotlin features. Working directly with the more abstract features of Kotlin is an advantage and disadvantage at the same time. The analysis can extract more information from the IR. However, it comes at the cost of writing considerably more code to implement all those features.

1.4.2 JVM bytecode analysis

As the JVM ecosystem is more established (compared to Kotlin), it offers more libraries. Since I later decided not to use this approach, I have not done extensive research on all of those libraries. Instead, I have picked one and used it as a reference point. The sample library is ASM [29] which is widely used and has extensive documentation. It supports everything necessary to implement the static analysis, and it even has an API for some common analysis tasks.

The JVM platform is standardized and has a public specification [26]. A considerable advantage is that bytecode (and the whole JVM) evolves slower than the internals of the Kotlin compiler and its IR. So from a maintainable standpoint, it might be easier to keep the analysis up-to-date if it depended on the bytecode instead of the IR.

All the disadvantages of bytecode analysis are fundamentally caused by information loss. The source of the information loss is the difference between the abstraction levels of the programming language and the compilation target. The larger the difference is, the more information is lost during the compilation. Bytecode is not as low-level as ordinary native code, but Kotlin is an unusually high-level language, so the difference in abstraction levels is still significant. Less information makes it harder for the analysis to understand the code semantics. As a result, the analysis will be either less precise or harder to implement.

For example, Kotlin supports a feature called *default arguments*. This feature is compiled to the bytecode (will be shown in (2.6.3)) in a way that makes it hard to analyze directly. In order to support this feature, the analysis would probably have to implement some form of function inlining and constant propagation.

Another example of the information loss problems is related to the difference in the control flow representation. JVM bytecode does not have structured control flow statements like loops and *if-else* blocks. Instead, it uses conditional and unconditional jump instructions to implement all branching. The presence of jumps does not matter for flow-insensitive analysis, but it may matter for some flow-sensitive analyses. Flow-sensitive analyses may leverage the restrictions of structured control flow.[†] Implementation of an analysis that requires structured control flow is still possible even in the presence of jumps (be it at the cost of additional work). The most straightforward solution is to convert the jumps back to structured control flow statements.[‡]

1.4.3 Summary

In the end, I have decided to implement the analysis as a compiler plugin and therefore opted for the source code analysis option. There are many reasons for that decision. The primary one is that my focus is on making the analysis as precise as possible. In my opinion, directly analyzing the source code will make that task easier. The secondary reason is that I am keeping the possibility to add support for Kotlin multiplatform in the future. By supporting the Kotlin multiplatform, it would be possible to use the analysis during mobile app development.

Then there are some partially subjective reasons. I have more experience with the Kotlin compiler and its IR than bytecode. Also, I want to implement the analysis prototype in Kotlin. The Kotlin compiler is written in Kotlin, which will make the implementation more pleasant. Not that it is impossible to call Java (in which the ASM library is written) from Kotlin, it is just slightly cumbersome. Furthermore, I want to use this opportunity to learn more about the compiler.

The choice comes with all the problems mentioned in (1.4.1). However, upon closer examination, I was able to come up with solutions for all those problems. Not all of them are perfect, and some will require additional work to implement. Implementation of some of those solutions is beyond the scope of this thesis, but the important fact is that it is possible to solve.

[†]In fact, the analysis proposed in this thesis does rely on the structured control flow.

[‡]When analyzing this option, I used the algorithm from the article "Solving the structured control flow problem once and for all" [12] as a reference point.

1.5 Prior art

Static analysis for exceptions is a well-known and well-researched problem with many existing approaches and solutions. However, none of the existing solutions is directly applicable in the context of this thesis – the reasons are explained later in this section. Therefore, the analysis proposed in this thesis cannot be based on a single existing solution. Instead, I have created a custom design utilizing general static analysis concepts and some ideas from the existing solutions.

This section presents the pre-existing work that influenced this thesis. The goal here is to give an overview, not a detailed explanation of each method. I will highlight the core ideas of those methods and how they influenced this thesis.

All the presented articles in this section assume that their reader is familiar with general concepts of static analysis and its terminology. As such, it is impossible to understand them without prior knowledge. I have gained this necessary knowledge from several courses at FIT CTU, especially: NI-APR (Selected Methods for Program Analysis), NI-MPJ (Modelling of Programming Languages), and NI-RUN (Runtime Systems). I have also read the book "Static Program Analysis"[25]. This general knowledge helped me understand the articles, but more importantly, it was essential for me while designing the custom analysis.

I have used the article "A review on exception analysis" [3] as a starting point of my research of existing methods. The article contains an overview of all existing relevant research on this topic – at least, it includes everything relevant its authors could find at that time. The article is from 2016, so it might miss some more recent development. However, I could not find anything relevant not already covered by this article.

The article presents a total of 87 methods, but this number gets quickly reduced as most of the methods focus on different problems. The article also contains duplicates because many of those methods are just improvements of the previous methods. So sometimes, the same underlining concepts are listed multiple times.

Only 59 of the listed methods are about static analysis (the rest is about dynamic analysis). The static analysis methods are split by their target use case

and language. This split narrows the search even further. For example, only 7 static analyses for Java (including duplications) target the correct use case.

Methods that focus on different use cases are generally not usable. They can make simplifications that might render them useless for different purposes. The article splits the methods by the following use cases:

Exception usage analysis These methods produce statistics about the usage of exception handling constructs. They do not analyze exception propagation, which means they are not suitable for this project.

Exceptional control flow analysis These analyses aim to construct a CFG that accounts for exception handling. The constructed CFG could be theoretically used as a basis of the analysis. However, the CFG must preserve the types of propagated exceptions. This is generally not necessary for the intended purpose, therefore, not done.

Uncaught exception analysis This use case utilizes the analysis to detect uncaught exceptions that can terminate the program. This use case can be easily modified for documenting API endpoints. Each endpoint can be considered a small separate program, and the thrown exceptions are precisely those that need to be documented.

Analysis working with different languages may or may not be usable. It depends on the similarities and differences between the two languages and also on the analysis approach. Typically, an analysis will not support features that do not exist in a given language. So the question becomes if it is possible to extend it to support the non-overlapping features of the second language. However, this question tends to be hard to answer before actually trying. For example, even if the missing features could be added, the approach may not support them well enough – resulting in low precision. For this reason, the rest of this section is divided by the analysis target language.

1.5.1 Existing solutions for Kotlin

Kotlin was first officially released in 2016 [21]. The article mentioned above was written before this release. Therefore, the article does not mention Kotlin, and I could not find any other research in this area. So the situation is the same as with the absence of libraries for the static analysis.

The only related work on this topic (that I am aware of) is a term work [4] for the course NI-APR done by Václav Málek and me. This work aimed to solve the same problem as this thesis and, in fact, was an inspiration for me to choose this topic. The developed analysis was always meant as a term work and not something that could be later used in practice. The analysis supported only a carefully selected subset of features to make its implementation doable as a term work.

The work uses a constraint system capable of analyzing the object-oriented feature of Kotlin. The constraints rely on type information provided by the compiler's type analysis. The limitation of this approach is that it cannot precisely analyze higher-order functions and, in general, any imprecisely typed expression. The analysis cannot determine which exact function is passed as an argument; it knows only its type/signature. Therefore, calling a function value is approximated as a call to any function with the matching signature. This approximation is too coarse for practical use, and it cannot be easily improved.

In summary, this term work can be seen as a proof of concept for this thesis, but other than that, it is not helpful. I have not reused any code from this work as I had to use a completely different approach. However, I have used the knowledge and experience gained while working on the term work.

1.5.2 Existing solutions for Java

A significant part of the overview article focuses on Java. Java is one of the closest languages to Kotlin in terms of semantics. So, in theory, an analysis for Java might be possible to modify for Kotlin. The problem is that this is true only for Java version 8 (and newer) because this was the version when lambdas were introduced [28]. Java 8 was released in 2014 [27]. Since most of the research was published before, these analyses generally do not support functional programming.

Without proper support for higher-order functions, any analysis will have the same problem as the term work. Even though the research is outdated, it still has some value. It shows how this type of analysis can be implemented for an object-oriented language.

The first example comes from the paper "An uncaught exception analysis for Java" [22] from 2002. The analysis was implemented for Java 1.3, so it supports only the language's core features (from today's point of view). The

goal of this analysis was to improve the precision of checked exceptions in Java. The primary improvement was to make the analysis inter-procedural instead of intra-procedural (which is how checked exceptions work). This analysis is similar to the term work as it also uses constraints. It also depends on the type information, so it has similar limitations. However, it directly supports assignment operation using unification – which the term work did not.

The second showcased paper, "Automatic documentation inference for exceptions" [2], is more intriguing. It deals with the same target use case as this thesis – documenting exceptions. The solution proposed in this paper goes even farther in this regard compared to this thesis. It not only analyzes the types of exceptions but also documents the conditions under which they can happen. The analysis first identifies `throw` statements from where exceptions can propagate unhandled by propagating the exceptions. In the second step, the analysis goes back to each `throw` statement and symbolically executes paths leading to these throws. In this way, the analysis creates the conditions describing when each exception is thrown.

Using symbolic execution for this purpose is interesting but, in my opinion, not necessary to document single-purpose exceptions. Their whole idea is that there is only a single reason for each exception. Therefore, it should not be hard to document this reason manually.

1.5.3 Existing solutions for other languages

Scala is another language semantically close to Kotlin, but like with Kotlin, there is no published research – neither the authors of the overview article nor I have found any. There is some research for C++ and Ada, but these languages significantly differ from Kotlin. However, I was able to find some valuable research for ML.

ML is a functional language, so a good exception analysis for ML must deal with higher-order functions. On the other hand, ML does not support object-oriented programming. Since Kotlin is an object-oriented language, the analysis would have to be extended to cover these features. I have found several papers that correctly handle higher-order functions, and I will present those that I have considered in more detail.

The first one is "Type-based analysis of uncaught exceptions" [23]. Initially, this analysis was based on a CFG analysis that tracked the flow of functional

values between function calls. The authors were not satisfied with the performance of that solution and implemented another one based on type analysis.[†] The type analysis works by defining a custom type system that includes information about propagated exceptions. Additionally, the analysis uses a special type inference algorithm. The inferred type of each function then contains the propagated exceptions, which is a result of the analysis.

The type analysis is really complex (at least for me). I have decided not to use this approach because I was not sure if I could adapt it to support features in Kotlin. Adapting it could be potentially very hard because Kotlin does not have the same type system as ML.

The second paper is named "An abstract interpretation for estimating uncaught exceptions in Standard ML programs" [34]. As the name suggests, it uses a method called abstract interpretation. An interesting property of this analysis is that it uses a custom intermediate language instead of directly analyzing ML code.

Conceptually, abstract interpretation analyzes the program by directly running it. It is a static analysis because it uses an abstract representation of the program state instead of exact values. The use of the abstract state is what causes the approximation. By choosing the right abstract state, it is possible to achieve a high level of precision.

The analysis implementation is similar to an interpreter (hence the name). It keeps track of possible values at each location (for example, variable). The possible states are represented in the form of a finite lattice. A new assignment to a location is implemented using unification which means that the old values are always preserved. With a proper unification and correctly chosen state, the lattice has a fixed point. The presence of a fixed point ensures that the analysis terminates because, at that point, the program state cannot change any further. When that happens, the analysis can return an answer because it has safe approximations for values in all locations.

[†]The CFG algorithm had theoretical quadratic time complexity, and the authors deemed it impractical for large programs. Based on their data, I do not think this type of performance would be a problem for this project. Each endpoint usually calls only a small portion of the whole program. Also, hardware got significantly better since 2000, when this analysis was published.

Chapter 2

Analysis of the Kotlin programming language

This chapter will present my analysis of the Kotlin programing language. I will focus on topics that influenced this thesis, mainly the semantics of features that impact the static analysis design. Furthermore, I will describe how these features are implemented in the Kotlin compiler, specifically in the Kotlin IR. Knowing these implementation details is essential to understanding the static analysis design described later in chapter (3). My goal is also to highlight the language complexity I had to deal with during the static analysis implementation.

Kotlin is an extensive language with many advanced features. Therefore, it is impossible to cover everything precisely without writing a new language specification. For this reason, I will intentionally omit some details, skip less interesting features, and make some simplifications.

All the information in this chapter represents my most up-to-date knowledge of the language. I gained that knowledge incrementally while working on the analysis, and the chapter does not reflect this process. The development and learning process will be described later in chapter (4), together with some of my mistakes.

This entire work is based on Kotlin version 1.6.10. While Kotlin is a relatively stable language, it is in active development, and new features are added with each release. New versions are mostly backward compatible, but incompatible changes are allowed in some specific situations [17]. On the other hand,

the compiler code has no backward compatibility guarantee. The IR is naturally more stable since a significant amount of code in the compiler depends on it. However, even the IR usually has some differences between major versions. To summarize: anything in this chapter may become outdated, especially things related to the compiler implementation.

Some information in this chapter is based on the official Kotlin documentation [16] and the language specification[1]. Nevertheless, I frequently had to rely on my experience with the Kotlin ecosystem and a lot of reverse engineering. The reason is that the Kotlin compiler is undocumented, and the language documentation does not cover everything.

The following section describes the Kotlin IR general properties. The second section presents examples of the reverse engineering methods that I have used while working on this project. The section after that contains information about the Kotlin standard library. All the remaining sections in this chapter are about analyzing individual Kotlin features.

2.1 IR description

The Kotlin IR has a relatively high-level abstraction. That brings it much closer to the source code than to the native code. The IR has direct support for most Kotlin features and therefore is quite complex.

The remaining features (mostly syntax sugar) are lowered by the front end to the features supported by IR. An example of such a feature is a destructuring declaration that the front end transforms into multiple individual assignments. This chapter skips description of most of those features as the static analysis does not directly implement them. Here is a list of the most notable features not covered for this reason:

- value classes
- sealed classes/interfaces
- data classes
- companion objects
- anonymous classes

- functional interfaces
- type aliases
- class delegation
- annotations
- operator functions
- infix function

The IR is implemented as a hierarchy of classes and interfaces in the package `org.jetbrains.kotlin.ir` of the Kotlin compiler [20]. At the top of the hierarchy is the interface `IrElement`. Every IR member/element (like a declaration or an expression) implements this interface.

The IR has a form of a graph with a structure similar to an AST. However, compared to an AST, the IR additionally contains semantic information obtained during the semantic analysis done by the front end. For example, all symbols and types are already analyzed and resolved. Also, in contrast to an AST, the IR graph is not a formal tree since the resolved symbols have back-references to their declarations. There is still a tree at the core of the IR in some sense, as there is a strong parent-child relation between elements. This tree can be traversed using a visitor pattern [8].[†] The visitor pattern can be implemented by inheriting `IrElementVisitor`.

While working with the IR, it is helpful to represent it in a human-readable form. Since the IR contains cycles, it cannot be directly serialized or printed. However, the visitor pattern can be used to print the IR structure while replacing the back-references with the declarations symbol name. This is what a `dump` method of the `IrElement` does.

I will use the dump output through the rest of this chapter to explain how the Kotlin features are represented in the IR. To explain the implementation of each feature, I will present some code and its equivalent IR dump. Even though the dump is technically human-readable, it still becomes quickly overwhelming for larger code segments. For this reason, I will always showcase only the interesting piece of the IR, omitting the rest.

[†]It is also possible to transform the IR using `IrElementTransformer` visitor, but this will probably not be required for the static analysis.

The IR dump does not always match the class structure, especially in terms of names. I will be describing the element classes and their properties as they are declared in the code. So there will be a slight mismatch between the examples and their descriptions, but it should be clear how to match them together.

To keep this chapter shorter, I will leave out an explanation of some IR elements that are not that relevant for the static analysis. The omitted elements include, for example, classes `IrFile` or `IrBlock`. Also, I will not describe the elements and their dump in full detail – again, skipping things that are not that important to this thesis.

2.2 Reverse engineering methods

I have used several reverse engineering methods to learn more about Kotlin and the implementation details of its compiler. It is important to know that I have used these methods in combination with my prior experience and information from the documentation. Just relying on reverse engineering is not a good idea as it is easy to miss some crucial details or edge cases. The presented methods are not necessarily overlapping each other. In fact, I have often used a combination of these methods.

The simplest method I have used was to write some code, run it and observe the result. This approach is suitable for analyzing unexpected semantics of nontrivial constructs. It is also helpful for testing if the compiler even supports such constructs. For example, this way, I have discovered that Kotlin does not support getters and setters for local properties.

To analyze how the compiler implements some functionality, I first downloaded the compiler sources from the GitHub repository [20]. In the case of simpler features, it was enough for me to just look through the compiler code. The main classes to inspect are those related to the IR. A good place to start is the `IrElementVisitor` since it lists all the base IR elements.

To get a sense of more complicated features, I had to inspect the IR dynamically. For that, I have used a standard JVM debugger.[†] There are multiple ways to attach a debugger to the compiler. My preferred approach for this particular use case is to use a library called Kotlin compile testing [32]. This

[†]Kotlin compiler is written in Kotlin JVM.

library is used for testing compiler plugins, and as such, it can call the compiler from Kotlin code. After getting that library, I created an empty plugin with a simple test. Then I set up the library as if I wanted to test the plugin. That way, I could attach a debugger to my code and put a break-point into any part of the compiler.

The debugger makes it possible to traverse and inspect the IR hierarchy. Debugging is a valid technique to understand the IR structure, but it is time-consuming. It is better to dump the IR into the readable form for a quicker high-level overview. To make this process even faster, I have created a special compiler plugin whose only purpose is to print the IR dump representation. I have also used this plugin to create all the IR examples in this chapter.

Another helpful approach is to analyze what the compiler produces as an output, which is the JVM bytecode in the case of Kotlin JVM. As was mentioned previously, Kotlin supports additional targets like JavaScript or the LLVM bytecode. Each platform supports a (different) limited set of basic features. Therefore the compiler musts convert the more complex features into simpler ones, and it does so in a slightly different way on each platform. Knowing how some of these conversions work was essential for me, especially when designing the analysis. `Default arguments` (described later in (2.6.3)) are a great example of a feature where I have used JVM bytecode decompilation.

One way to learn how these conversions work is to inspect the compiled code (either directly or by decompiling it). However, there is a better approach. Kotlin puts a strong emphasis on having great bi-directional interoperability with Java [18]. There must be some mapping between Kotlin and Java features for the interoperability to work. So in the case of Kotlin JVM, most of these conversions are indirectly documented by this mapping. The documentation can be found in the Java interoperability guide [13]. Using this method, it is, for example, possible to learn that properties are internally handled as methods.

2.3 The Kotlin standard library

Every project written in Kotlin must include the standard library. Without it, it is impossible to write any meaningful program. The standard library contains implementation for the most fundamental things like numbers, strings, or

arrays. On the other hand, it also has many high-level concepts like collections and functions for collection transformations like `map` or `filter`.

The standard library is mainly implemented in Kotlin and compiled by the Kotlin compiler. The rest of the implementation is in native code and, therefore platform dependent.[†] It has a special handling from the compiler as some things would not be possible to implement in regular Kotlin code. The standard library delegates some parts of its implementation to existing libraries for the specific platform. For example, on the JVM target, the Kotlin standard library delegates many things to the standard library of Java. This native code in the standard library presents the same problem as all the other native code or closed-source libraries.

All basic types in Kotlin are objects. This is true even for types usually considered as primitives like `Int`, `Double`, `Char`, or `Boolean`. Compared to Java, Kotlin does not expose these types as true primitives. Instead, it performs necessary autoboxing and unboxing (as Java does in some cases).

All primitive types can be represented as constants if their value is known during compilation. The simplest example is a literal expression like `1` or `false`. Another important constant is the `null` expression. IR stores these constants in a class `IrConst` that holds the constant type and value. Example dump representing a `true` constant is shown in the listing (2.1).

```
/*
 * CONST Boolean type=kotlin.Boolean value=true
 */
true
```

Listing 2.1: Boolean

Because numbers are objects, all the basic arithmetic operations are implemented as methods of those objects. For that reason, the Kotlin IR does not have any particular constructs to represent these basic operations. Instead, they are represented as function calls. Some of these calls are then substituted for native instructions by the back end as they cannot be implemented in the library.

Arrays are another important group of classes implemented in the standard library. There is a special array type for each primitive type, for example,

[†]The native code can be implemented in another language, such as Java (on the JVM platform).

`IntArray` or `DoubleArray`. Then there is also a generic array class `Array` that can hold objects of the specified generic type. The IR does not have any direct operations related to Arrays, and the standard library handles everything. However, similarly to the primitive types, some things must be implemented natively.

The last special class that is mentioned here is `String`. Strings can also appear in the form of literals, in which case they are represented as other constants via the `IrConst` class. Strings support a join operation called interpolation. This operation is used to create a new string by joining several string literals and other objects in one expression. Objects that are not strings are first converted to `String` by calling their `toString` method, but the IR does not explicitly do that.

The interpolation is represented as `IrStringConcatenation` class on the IR level. This class has a property `arguments`, which is a list of expressions. Example IR of some interpolation is shown in the listing (2.2). The line starting with `GET\_VAR` is a read from a local variable, which will be explained later in (2.6.1). The important thing in the listing is the absence of any explicit conversion to `String` even though the variable has a type `Int`.

```
val i = 1

/*
 * STRING_CONCATENATION type=kotlin.String
 *   CONST String type=kotlin.String value="i: "
 *   GET_VAR 'val i: kotlin.Int ...' type=kotlin.Int origin=null
 *   CONST String type=kotlin.String value="\n"
 */
"i: $i\n"
```

Listing 2.2: String concatenation

2.4 Exceptions

Exceptions are the primary focus of this project and, as such, will be explained first. Exceptions are an excellent example of a feature that Kotlin almost

entirely took over from Java.[†] There are several key concepts to know about exceptions:

- what is an exception,
- what action results in an exception,
- how to handle an exception,
- and how do exceptions propagate through the program.

2.4.1 What is an exception

An exception is a regular class instance that inherits from `kotlin.Throwable`. Any object that is not an instance of `Throwable` cannot be used as an exception. Apart from this difference, an exception is just a regular object. It supports all the usual operations. For example, it can have methods or be stored in a variable.

Even though using an object of type `Throwable` is valid, more common is to use an object of a different class that only inherits from `Throwable`. There are many other exception classes in the standard library. Developers can also define custom exceptions by inheriting any of those classes.

Each exception contains a stack trace, a description, and an optional cause (in the form of another exception). The exception/class name is usually chosen to describe what has happened. Using the exception name in this way is the whole idea behind the single-purpose domain exceptions. Therefore, the exception type is the only thing the static analysis needs to track to fulfill its use case. The rest of the exception content is not of interest.

2.4.2 Throwing an exception

Exceptions can be thrown either explicitly, using the keyword `throw`, or implicitly by another action. The `throw` keyword takes an expression that represents the thrown exception. In the IR throw has a dedicated class `IrThrow` (example in the listing (2.3)). It is one of the simplest IR elements since it only contains a single child expression and has no other significant properties.

[†]With the notable absence of checked exceptions.

```
/*
 * THROW type=kotlin.Nothing
 *   CONSTRUCTOR_CALL 'public constructor <init> () declared in
↪  kotlin.Throwable' type=kotlin.Throwable origin=null
 */
throw Throwable()
```

Listing 2.3: Throw

Exceptions are thrown implicitly if the program performs some illegal operation. Here are some examples of such operations:[†]

- Dereferencing a null pointer
- Using a negative number as an array index
- Allocating more memory than available

Each of these exceptions represents a whole category of exceptions. This distinction impacts the analysis as each category must be handled differently. Dereferencing a nullable pointer is an explicit operation since Kotlin has a null safety built into its type system. The Kotlin compiler generates guards around potentially unsafe dereferences. The generated guard then throws the null pointer exception explicitly. Listing (2.4) shows the guard in decompiled Java bytecode.

The guard call is represented in the IR as a regular function call to a special function `kotlin.internal.ir.CHECK_NOT_NULL`. I will skip what a call in IR looks like for now, as it will be covered in (2.6). During compilation, the back end replaces this special function call with a call to a standard library function `kotlin.jvm.internal.Intrinsics.checkNotNull`.

`ArrayIndexOutOfBoundsException` represents exceptions thrown directly by the JVM because of some instruction other than throw. This specific exception can occur only when accessing an array, so there is a specific set of instructions that can cause it. Another example of such an exception is `ArithmeticException`. These exceptions are theoretically possible to analyze using static analysis as they depend only on the program semantics.

[†]The presented exceptions are taken from the JVM. Other targets generally use equivalent exceptions from the *kotlin* package.

```kotlin
// Source code
fun add1(i: Int?) = i!! + 1

// Decompiled code (Java)
public static final int add1(@Nullable Integer i) {
    Intrinsics.checkNotNull(i);
    return i + 1;
}

// kotlin.jvm.internal.Intrinsics (Java)
public static void checkNotNull(Object value) {
    if (value == null) {
        throwJavaNpe();
    }
}
```

Listing 2.4: Not-null assertion

On the other hand, the `OutOfMemoryError` is very hard to analyze using static analysis.[†] The analysis of this exception is problematic because it depends on the execution environment and runtime implementation. `IOException` is a similar case.

2.4.3 Exception handling

A thrown exception can be caught using a *try-catch* expression. More specifically, exceptions thrown in a `try` block can be handled by code in a `catch` block (handler). It is possible to have multiple handlers in one *try-catch* expression, but each exception can be handled only by a single handler. The handlers handle only exceptions thrown from the current `try` block.

Each handler specifies a variable and its type. This type is used to determine which exception can be handled by the handler. It handles only exceptions that are instance of this type.[‡] If multiple handlers could be selected, then the first one is prioritized.

A handler can throw a new exception. For example, it can use the caught exception to cause the new exception. Exceptions thrown in a handler are not

[†]Not counting the trivial case when the analysis predicts any allocation can throw this exception.

[‡]This implicitly means that the exception type can also be a subtype of the specified type.

handled by the other handlers of the same *try-catch* expression. A handler can also rethrow the handled exception.

An example of the *try-catch* IR is shown in the listing (2.5). The IR represents this whole expression as a single class `IrTry`. The class has properties `tryResult`, `catches` and `finallyExpression`. `tryResult` is an expression representing the try body. The property `catches` is a list of `IrCatch`, which are the handlers. The `IrCatch` contains a property `catchParameter` (the variable including its type) and a property `result` (the handler body). There are also two integer constants as a placeholder for the content of both bodies.

```
/*
 * TRY type=kotlin.Int
 *   try: BLOCK type=kotlin.Int origin=null
 *     CONST Int type=kotlin.Int value=1
 *   CATCH parameter=val e: kotlin.Throwable [val] ...
 *     VAR CATCH_PARAMETER name:e type:kotlin.Throwable [val]
 *     BLOCK type=kotlin.Int origin=null
 *       CONST Int type=kotlin.Int value=2
 */
try {
    1
} catch (e: Throwable) {
    2
}
```

Listing 2.5: Try-catch

The property `finallyExpression` represents a `finally` block that is optionally a part of the `try-catch`. `Finally` block is always called after all code from the rest of the `try-catch`. `Finally` is called even if an unhandled exception is being propagated. In such a case, the `finally` block gets evaluated, and then the exception continues its propagation. The `finally` block cannot interact with this exception. It can, however, throw another exception. If that happens, the exception from `finally` has a priority over the previously thrown exception. The previous exception is therefore discarded. The dump of `IrTry` with a `finally` block is shown in the listing (2.6).

A *try-catch* being an expression is a significant difference from other languages like Java. To understand how this feature works, it is first necessary to understand the evaluation of blocks. In general, expression blocks evaluate to their last statement if it is an expression. If the last statement is not an

```
/*
 * TRY type=kotlin.Int
 *   try: BLOCK type=kotlin.Int origin=null
 *     CONST Int type=kotlin.Int value=1
 *   finally: BLOCK type=kotlin.Unit origin=null
 *     CONST Int type=kotlin.Int value=2
 */
try {
    1
} finally {
    2
}
```

Listing 2.6: Try-finally

expression (or the block is empty), then the block technically evaluates to `kotlin.Unit`. However, from a practical point of view, it does not have value since it can no longer be used as an expression. Also, if the block throws an exception, the result does not effectively exist and cannot be accessed.

In the case of the *try-catch*, there are multiple blocks. The rule is that the expression evaluates to the block that is evaluated to its end. If the `try` block does not throw an exception, its value is used. Alternatively, the value comes from the handler that handles the exception thrown in try. The content of a `finally` block does not affect this rule, and a `finally` block never produces a value.

2.4.4 Exception propagation

How exception propagation works can be described using an imaginary data structure called a handler stack.[†] The handler stack is a data structure that holds all the *try-catch* expressions that can eventually handle exceptions. The stack structure makes it so the handlers can be later accessed in the correct order.

Every time the program enters a try block, its *try-catch* expression is added to the handler stack. Once the program leaves a `try` block, the handler stack is popped – restoring the state before entering the `try` block. If an exception is thrown, the execution is paused, and the runtime starts popping the handler

[†]Actual runtime implementation is platform-specific.

stack. The runtime continues popping the handler stack until it finds a *try-catch* with a matching handler. If the popped *try-catch* has a `finally` block, this block is executed before another *try-catch* is popped from the stack. If there is a correct handler, the execution continues from the handler block, and the runtime stops popping the handler stack. If the stack is empty and the exception is still not handled, then the program is terminated with that exception as a result.

2.5 Control flow

Control flow dictates which program instruction is executed next. Kotlin has multiple constructs that affect the control flow. These are conditions, loops, jumps, and function calls.

2.5.1 Conditions

Conditions are used to create branches in the control flow. Only one of the branches can be executed in one pass (if any). Which one is executed depends on the condition construct semantics. There are three different condition constructs in Kotlin:

- `if` + `else` keywords
- `when` keyword
- Elvis operator

From the IR perspective, all these constructs are represented as `IrWhen`. The `IrWhen` has a list of branches (`IrBranch`). Each branch contains an `if` expression (condition) and a `then` expression (body). Condition determines which branch body gets executed. It is the first branch in the list whose condition evaluates to `true`. Subsequent evaluation of all remaining branches (including their conditions) is skipped. The branches are in the same order as in the source code.

An *else branch* is an ordinary branch with the condition set to `true`. The *else branch* is not always required, and in fact, it is possible to have only a single branch. If *else branch* is present, it is always in the last position.

Example of a primitive `when` is shown in the listing (2.7). Each branch in this listing has a special `TYPE\_OP` expression wrapping the constant. Static analysis can safely ignore this expression as it only tells the compiler that it should discard the result of the expression it wraps.

```
/*
 * WHEN type=kotlin.Unit origin=WHEN

 *   BRANCH
 *     if: CONST Boolean type=kotlin.Boolean value=true
 *     then: BLOCK type=kotlin.Unit origin=null
 *       TYPE_OP type=kotlin.Unit origin=IMPLICIT_COERCION_TO_UNIT
↪  typeOperand=kotlin.Unit
 *         CONST Int type=kotlin.Int value=0

 *   BRANCH
 *     if: CONST Boolean type=kotlin.Boolean value=false
 *     then: BLOCK type=kotlin.Unit origin=null
 *       TYPE_OP ...
 *         CONST Int type=kotlin.Int value=1

 *   BRANCH
 *     if: CONST Boolean type=kotlin.Boolean value=true
 *     then: BLOCK type=kotlin.Unit origin=null
 *       TYPE_OP ...
 *         CONST Int type=kotlin.Int value=2
 */
when {
    true -> 0
    false -> 1
    else -> 2
}
```

Listing 2.7: When

The compiler converts all types of conditions to the same representation used for `when`. The listing (2.8) shows a semantically equivalent `if` to the `when` from listing (2.7). The `if` IR is practically the same as the `when` IR, except `origin` is `IF` instead of `WHEN`. However, the origin is insignificant for the static analysis.

Kotlin conditions can be used as expressions, but only if they are exhaustive.[†] The condition expression is evaluated to the expression of its body. In

[†]Condition is exhaustive if the compiler can prove that there will always be some branch evaluated for every possible program state.

```
if (true) {
    0
} else if (false) {
    1
} else {
    2
}
```

Listing 2.8: If

the previous listing (2.7), the result of the `when` is not used, so the body result is discarded, and the condition type is `Unit`.

An example of a condition whose value is later used is shown in the listing (2.9). In this case, the type of `IrWhen` is `Int`, and there is no `TYPE\_OP`. At the same time, the listing demonstrates how an elvis operator is transformed into `IrWhen`. The elvis operator returns the alternative expression if the value is null (the first branch). Otherwise, it returns the value itself (the *else branch*). The value is stored in a temporary variable because its expression must be evaluated only once. The `EQEQ` call in the first condition determines if the value is null.[†]

```
fun example(optionalInt: Int?) =
/*
 * BLOCK type=kotlin.Int origin=ELVIS

 *   VAR IR_TEMPORARY_VARIABLE name:tmp0_elvis_lhs ...
 *     GET_VAR 'optionalInt ...

 *   WHEN type=kotlin.Int origin=null
 *     BRANCH
 *       if: CALL 'public final fun EQEQ ...
 *         arg0: GET_VAR 'val tmp0_elvis_lhs ...
 *         arg1: CONST Null ...
 *       then: CONST Int type=kotlin.Int value=0
 *     BRANCH
 *       if: CONST Boolean ... value=true
 *       then: GET_VAR 'val tmp0_elvis_lhs ...
 */
   optionalInt ?: 0
```

Listing 2.9: Elvis operator

[†]This call is a good example of how primitive operations are represented as functions.

2.5.2 Loops

Loops are used to evaluate some block of code (loop body) multiple times. The evaluation repeats as long as some condition is met. Jump instructions (introduced later in (2.5.3)) may also interrupt the loop. There are three different types of loops in Kotlin: `while`, `do-while`, and `for`.

A *while* and *do-while* loops differ only by the moment when the loop condition is tested. The *while* loop tests the condition before each iteration, and *do-while* loop does that after each iteration. For that reason body of *do-while* loop is always executed at least once.

In the IR, they are represented as two different classes: `IrWhileLoop` and `IrDoWhileLoop`, respectively. However, these classes are practically the same as both extend `IrLoop`, which defines their content. The `IrLoop` contains a body, condition, and an optional label. The label is used together with `break` and `continue`, explained later in (2.5.3).

An example of a trivial *while* loop is shown in the listing (2.10). The IR of *do-while* loop is almost the same. There are two notable differences. One is that `WHILE` is replaced with `DO\_WHILE` in the IR dump. The second one is in the wrapping block, which in the case of `do-while`, envelopes the whole loop instead of only the body.[†]

```
/*
 * WHILE label=exampleLabel origin=WHILE_LOOP
 *   condition: CONST Boolean type=kotlin.Boolean value=true
 *   body: BLOCK type=kotlin.Unit origin=null
 *     CONST Int type=kotlin.Int value=1
 */
exampleLabel@while (true) {
    1
}
```

Listing 2.10: While

A *for* loop in Kotlin is not the same as a *for* loop in languages like Java. Instead, it is more similar to a *for-each* loop since it loops over an iterator (`kotlin.collections.Iterator`). Kotlin standard library provides multiple functions that create these iterators. Example of such function is `until`. A loop

[†]The wrapping block limits the scope of variables declared in the loop. The difference is because variables declared in the `do-while` body can be accessed in the loop condition.

`for (i in 0 until 10)` is in this case equivalent to a more standard Java syntax `for (int i = 0; i < 10; i++)`.

The *for* loop does not have a direct representation in the IR. Instead, it is converted to an equivalent `while` loop. Listing (2.11) shows how this conversion would look if done directly in Kotlin instead of the IR. The exact IR is intentionally not disclosed since it is fairly verbose.

```
for (i in 0 until 10) {
    i
}

// Converted approximately to
val tmp0_iterator = (0 until 10).iterator()
while (tmp0_iterator.hasNext()) {
    val i = tmp0_iterator.nextInt()
    i
}
```

Listing 2.11: For

2.5.3 Jumps

A jump instruction transfers execution to a different part of the program. Kotlin specifically does not have any `goto` instruction and supports only the following jumps:

- break
- continue
- return
- throw

The keyword `throw` is related to the exception handling and was discussed in previous section. The keyword `return` stops the execution of the current function. The execution then continues after the function call that has called the current function. The keyword `return` is described in more detail in (2.6.5).

The keywords `break` and `continue` are very similar to each other. They both alter the execution of a loop. The only difference between them is that

`break` jumps right after the whole loop, while `continue` jumps at the end of the loop body. Therefore, `break` terminates the execution of a loop while `continue` starts a new iteration if the loop condition is still met.

Since loops can be nested, there must be a way to determine which loop should be affected by these statements. This loop resolution is implemented by `labels`. Both loop and `break`/`continue` can optionally have a label. If `break`/`continue` does not have a label, then it always belongs to the innermost loop (regardless if the loop has a label or not). If `break`/`continue` does have a label, then it is related to the loop with the same label. If there are multiple loops with the same label, then the innermost one with a matching label is selected.

In the IR, `break`/`continue` is represented as `IrBreak`/`IrContinue` respectively. Both classes inherit `IrBreakContinue`, which contains the optional `label`. An example of the `IrBreak` IR dump is shown in the listing (2.12).

```
exampleLabel@while {
/*
 * BREAK label=null loop.label=exampleLabel
 */
    break
}
```

Listing 2.12: Break

2.5.4 Function calls

A function call transfers the program execution to the beginning of the called function body. Which function is called depends on overloading and dynamic dispatch, which are described in (2.6.6). The function execution is terminated in one of these scenarios:

- execution reaches the end of the function
- `return` instruction is executed
- exception is thrown

The execution resumes right after the function call in the first two cases. If an exception is thrown, the control flow is decided by the exception propagation

rules described in (2.4.4). The called function can call any other function. Therefore, function calls can be arbitrarily nested. To support this feature, the runtime store location of the function call (including the local variables) in a structure named *call stack*.

2.6 Functions

Functions together with classes are the fundamental building blocks of Kotlin programs. In contrast to Java, Kotlin supports proper functions, not just methods. From the IR perspective, both functions and methods are represented the same way as `IrSimpleFunction`. `IrSimpleFunction` inherits most of its properties from `IrFunction`.[†] For example, listing (2.13) shows IR of the simplest possible function with no parameters and without a body.

```
/*
 * FUN name:foo visibility:public modality:FINAL <> ()
↪  returnType:kotlin.Unit
 *   BLOCK_BODY
 */
fun foo() {
}
```

Listing 2.13: Function declaration

`IrCall` is the IR class for all function calls. The listing (2.14) contains an example of a function call that calls the function from the previous listing. `IrCall` holds a symbol to the called function. The class has several other properties related to different features of functions. These features are explained separately in corresponding subsections, and so are the function call properties.

2.6.1 Local variables

Variables (or local variables) can be declared almost anywhere inside a function body. Variable values are isolated between function calls. Therefore, each

[†]The `IrFunction` is also a base class for other special functions, most notably the `IrConstructor` class.

```
/*
 * CALL 'public final fun foo (): kotlin.Unit declared in <root>'
↪  type=kotlin.Unit origin=null
 */
foo()
```

Listing 2.14: Function call

function can access only the variables it directly declares.[†] Variables can be declared and later accessed by a read or write operation. Variables can be potentially initialized with some value at their declaration.

The IR represents variable declaration using the class `IrVariable`. This class contains a symbol of the declared variable, its type, and optionally an initialization expression. There are also some flag properties, but these are not important. Listing (2.15) shows an example of a variable declaration. If there is no initialization expression, the second line in the listing (2.15) is missing.

```
/*
 * VAR name:a type:kotlin.Int [var]
 *   CONST Int type=kotlin.Int value=1
 */
var a = 1
```

Listing 2.15: Local variable declaration

Reading from a variable is represented as `IrGetValue`. A variable write is represented as `IrSetValue`. The IR of both operations is showcased in the listing (2.16). Both of these classes contain a symbol associated with the target variable. `IrSetValue` has an additional `value` expression which is the value written to the variable.

It is possible to declare multiple variables with the same name in a single function. However, they cannot be declared in the same scope.[‡] Variables with the same name have their own separate storage and do not affect each other.

[†]Unless some variable is captured, for example, by a lambda expression.

[‡]In some regards, Kotlin scoping is similar to classical block scoping from other languages, but it is more complicated than that. The exact details are not that important for this thesis since the compiler resolves these collisions. The Kotlin language specification [19] describes the scoping in detail.

```
/*
 * GET_VAR 'val a: kotlin.Int [val] declared in <root>.example'
↪  type=kotlin.Int origin=null
 */
a
/*
 * SET_VAR 'var a: kotlin.Int [var] declared in <root>.example'
↪  type=kotlin.Unit origin=EQ
 *   CONST Int type=kotlin.Int value=2
 */
a = 2
```

Listing 2.16: Local variable access

Only a variable from the nearest enclosing scope can be accessed by its name. The rest is shadowed and cannot be accessed directly. An example of shadowing is shown in the listing (2.17). The IR dump does not contain information on which variable the `GET\_VAR` points to. However, this information is present in the IR as an object reference to the specific variable declaration.

2.6.2 Parameters

Function parameters are a mechanism for passing objects between functions. A function call must provide a matching argument for each declared parameter. Arguments are passed by copying a reference (pointer) to the object in the argument.

Inside the function body, a parameter is conceptually like an immutable variable. It cannot be reassigned, but it may be shadowed by a variable. This variable then can be reassigned.

In the IR, a parameter has its dedicated class `IrValueParameter`. It and `IrVariable` both inherit from the same base class `IrValueDeclaration`. Therefore, there is some relation between parameters and variables even in the IR. For example, parameters are also read using the same operation `IrGetValue`. The primary difference is that parameters have their declaration stored in the `IrFunction` directly (and not in the function body). An example of the parameter declaration can be seen in the listing (2.18).

To support calling a function with parameters, `IrCall` contains a list of all the arguments. The listing (2.19) shows an example of such a function call.

```
/*
 * VAR name:a type:kotlin.Int [val]
 *   CONST Int type=kotlin.Int value=1
 */
val a = 1

/*
 * WHILE ...
 *   body: BLOCK type=kotlin.Unit origin=null
 */
while (true) {
/*
 *     VAR name:a type:kotlin.Int [val]
 *       CONST Int type=kotlin.Int value=2
 */
   val a = 2

/*
 *     GET_VAR 'val a: kotlin.Int [val] declared in <root>.example'
↪  type=kotlin.Int origin=null
 */
   a
}

/*
 * GET_VAR 'val a: kotlin.Int [val] declared in <root>.example'
↪  type=kotlin.Int origin=null
 */
a
```

Listing 2.17: Local variable shadowing

The argument `value` in the IR can be a complex expression, not just a constant. These values are evaluated as part of the function call.

An argument can be optionally associated with the matching parameter name. This functionality is known as *named arguments*. Named arguments make it possible to pass the arguments in a different order.[†] The compiler takes care of reordering the arguments back to the order of parameters. For example `foo(j = 2, i = 1)` compiles to the same IR as regular `foo(1, 2)`.

The ordering of arguments is significant because the order of arguments (not parameters) decides the evaluation order. So the Kotlin compiler needs to generate additional code to preserve the evaluation order. The compiler generates the code only if it cannot prove that changing the evaluation order

[†]However, the primary use case is to improve the readability of calls with many arguments.

```
/*
 * FUN name:foo visibility:public modality:FINAL <> (i:kotlin.Int,
↪  j:kotlin.Int) returnType:kotlin.Unit
 *   VALUE_PARAMETER name:i index:0 type:kotlin.Int
 *   VALUE_PARAMETER name:j index:1 type:kotlin.Int
 *   BLOCK_BODY
 */
fun foo(i: Int, j: Int) {
}
```

Listing 2.18: Declaration of function with parameters

```
/*
 * CALL 'public final fun foo (i: kotlin.Int, j: kotlin.Int):
↪  kotlin.Unit declared in <root>' type=kotlin.Unit origin=null
 *   i: CONST Int type=kotlin.Int value=1
 *   j: CONST Int type=kotlin.Int value=2
 */
foo(1, 2)
```

Listing 2.19: Function call with arguments

does not change the program semantics. For this reason, the previous example with constant arguments has the same IR, but the example in listing (2.20) does not. In this example, the arguments are function calls that can have side effects. Therefore, their evaluation order matters and must be preserved. The analysis does not have to explicitly handle named arguments since they are resolved by the compiler front end.

2.6.3 Default arguments

A default argument is declared as a parameter that has an associated expression. If the function call does not provide an argument for the parameter, the default argument is used instead. The default expression is stored in the `defaultValue` property of the `IrValueParameter` class. An example is shown in the listing (2.21).

The default value can be any expression valid in the function context. The expression is not required to be valid at the call site. As a result, the expression can access *this* and private members of the same class (in case of methods).

```
/*
 * BLOCK type=kotlin.Unit origin=ARGUMENTS_REORDERING_FOR_CALL

 *   VAR IR_TEMPORARY_VARIABLE name:tmp0_j type:kotlin.Int [val]
 *     CALL 'public final fun someFunction (x: kotlin.Int) ...
 *       x: CONST Int type=kotlin.Int value=2

 *   VAR IR_TEMPORARY_VARIABLE name:tmp1_i type:kotlin.Int [val]
 *     CALL 'public final fun someFunction (x: kotlin.Int) ...
 *       x: CONST Int type=kotlin.Int value=1

 *   CALL 'public final fun foo (i: kotlin.Int, j: kotlin.Int) ...
 *     i: GET_VAR 'val tmp1_i ...
 *     j: GET_VAR 'val tmp0_j ...
 */
foo(j = someFunction(2), i = someFunction(1))
```

Listing 2.20: Function call with named arguments

```
/*
 * FUN name:foo visibility:public modality:FINAL <> (i:kotlin.Int,
↪  j:kotlin.Int) returnType:kotlin.Unit

 *   VALUE_PARAMETER name:i index:0 type:kotlin.Int
 *     EXPRESSION_BODY
 *       CONST Int type=kotlin.Int value=1

 *   VALUE_PARAMETER name:j index:1 type:kotlin.Int
 *     EXPRESSION_BODY
 *       CONST Int type=kotlin.Int value=2
 */
fun foo(i: Int = 1, j: Int = 2) {
}
```

Listing 2.21: Declaration of function with default arguments

It can also access preceding parameters. An interesting observation (from the analysis perspective) is that the expression may throw an exception. An exception produced by a default argument will only be thrown if the call does not provide a value for the parameter.

The IR can recognize that an argument is missing because its value is not in the `IrCall` list of arguments. The listing (2.22) shows such a function call. As can be deduced from the example, the front end does not resolve default

arguments. Since default values can affect the exception flow, the static analysis must implement support for this feature.

```
/*
 * CALL 'public final fun foo (i: kotlin.Int, j: kotlin.Int) ...
 *   j: CONST Int type=kotlin.Int value=3
 */
foo(j = 3)
```

Listing 2.22: Function call with default arguments

It is worth looking at how the back end implements default arguments to understand better how they work. The listing (2.23) shows a decompiled code generated by the JVM back end. In summary: the back end generates a wrapping function with an additional control parameter. That control parameter is interpreted as a bit field, and it signals which arguments are provided and which are not. The original call is replaced with a call to this wrapping function. All the omitted arguments in the call are replaced with nulls.[†]

```
// Kotlin code
foo(j = 3)

// Generated code
 // Replaced call
foo_default(0, 3, 1, (Object)null);

 // Wrapping function
public static void foo_default(int var0, int var1, int var2, Object
↪  var3) {
    if ((var2 & 1) != 0)
        var0 = 1;
    if ((var2 & 2) != 0)
        var1 = 2;
    foo(var0, var1);
}
```

Listing 2.23: Implementation of default arguments

[†]More precisely, a missing argument is replaced with a default empty value determined by the runtime type of the parameter. What is the default value is in general platform-specific. In JVM, it is, for example, `null` for classes and `0` for whole numbers.

2.6.4 Varargs

Varargs or variable arguments pass multiple values via a single parameter. Varargs are more or less a syntax sugar as the same effect can be achieved with an array. However, the compiler treats them as core functionality since they are not resolved by the front end.

From the function perspective, varargs behave as if the parameter has a type `Array`. This change of type can be seen in the listing (2.24). In the example, the actual parameter type is an `IntArray`, and the parameter is marked as vararg.

```
/*
 * FUN name:foo visibility:public modality:FINAL <>
↪  (i:kotlin.IntArray) returnType:kotlin.Unit

 *   VALUE_PARAMETER name:i index:0 type:kotlin.IntArray
↪  varargElementType:kotlin.Int [vararg]
 */
fun foo(vararg i: Int) {
}
```

Listing 2.24: Declaration of function with vararg parameter

The main difference is from the caller's perspective. All the vararg values are wrapped in the `IrVarargs` object, as is shown by the listing (2.25). Varargs without any value are represented precisely the same way as an omitted value for a default argument.

```
/*
 * CALL 'public final fun foo (vararg i: kotlin.Int) ...
 *   i: VARARG type=kotlin.IntArray varargElementType=kotlin.Int
 *     CONST Int type=kotlin.Int value=1
 *     CONST Int type=kotlin.Int value=2
 */
foo(1, 2)
```

Listing 2.25: Function call with vararg

What makes the implementation of varargs complicated is a spread operator. The spread operator allows passing an array into a vararg parameter. Such an action cannot be done directly as the array size is generally unknown at the call site. The spread operator is represented as the `IrSpreadElement` class. This

class only holds an expression of type array. The spread operator can be used in combination with other values and can even be used multiple times. An example of this situation is shown in the listing (2.26).

```
/*
 * CALL 'public final fun foo (vararg i: kotlin.Int) ...
 *   i: VARARG type=kotlin.IntArray varargElementType=kotlin.Int

 *     SPREAD_ELEMENT
 *       CALL 'public final fun intArrayOf (vararg elements:
↪  kotlin.Int) ...
 *         elements: VARARG type=kotlin.IntArray ...
 *           CONST Int type=kotlin.Int value=1
 *           CONST Int type=kotlin.Int value=2

 *     CONST Int type=kotlin.Int value=3

 *     SPREAD_ELEMENT
 *       CALL 'public final fun intArrayOf (vararg elements:
↪  kotlin.Int) ...
 *         elements: VARARG type=kotlin.IntArray ...
 *           CONST Int type=kotlin.Int value=4
 *           CONST Int type=kotlin.Int value=5
 */
foo(*intArrayOf(1, 2), 3, *intArrayOf(4, 5))
```

Listing 2.26: Spread operator

Varargs can have default values, in which case it is always an array. The default value is used when there is no value for the parameter. As a result, it is no longer possible to directly call the function with an empty vararg.[†]

2.6.5 Return

All functions have a return type and return a value. The returned value is the evaluation result of the function call expression. The return type is stored as a `returnType` property of the `IrFunction` class. The `return` keyword has a dedicated class `IrReturn` (an example is shown in the listing (2.27)). This class has a property `value` representing the returned expression.

A function does not have to return a value explicitly. For these types of functions, the compiler makes sure that the function evaluates to a value of

[†]A possible workaround is to use a spread operator on an empty array.

```
/*
 * FUN name:foo visibility:public modality:FINAL <> ()
↪  returnType:kotlin.Int
 */
fun foo(): Int {
/*
 *   BLOCK_BODY
 *     RETURN type=kotlin.Nothing from='public final fun foo ():
↪  kotlin.Int declared in <root>'
 *       CONST Int type=kotlin.Int value=1
 */
    return 1
}
```

Listing 2.27: Return

`kotlin.Unit`. `Unit` is similar to `void` from other languages as it represents a missing value. However, in Kotlin, the `Unit` is implemented as a regular singleton class / object class.

A function in Kotlin can be declared as a single expression using a = symbol. The compiler front end transforms this form back to an ordinary function declaration. So, for example, `fun foo() = 1` produces exactly the same IR as was in the listing (2.27).

2.6.6 Methods

Functions that are members of some class are called methods. Methods differ from ordinary functions in three fundamental properties:

- Methods have a special parameter called a dispatch receiver.
- A method can override a method from the parent class/interface.
- Functions can be called only using a static call, while methods can also use a virtual call.

The dispatch receiver is always present and must be an instance of the method's class. The dispatch receiver cannot change during the execution of the method (it is not writable). The IR does not have a special class for methods, so it uses the `IrSimpleFunction` class. The only difference for methods is the

presence of a value for the `dispatchReceiverParameter` property. An example is shown in the listing (2.28).

```
class A {
/*
 * FUN name:foo visibility:public modality:FINAL <> ($this:<root>.A)
↪  returnType:kotlin.Unit
 *   $this: VALUE_PARAMETER name:<this> type:<root>.A
 *   BLOCK_BODY
 */
   fun foo() {
   }
}
```

Listing 2.28: Method declaration

A dispatch receiver can be accessed using the `this` keyword. Even though the dispatch receiver has an extra property in the declaration class, it does not have a dedicated access instruction. The dispatch receiver is accessed by `IrGetValue` as a regular variable, but it has a unique symbol `<this>`. The dispatch receiver access is shown in the listing (2.29).

```
/*
 *  GET_VAR '<this>: <root>.A declared in <root>.A.foo'
↪  type=<root>.A origin=null
 */
this
```

Listing 2.29: This keyword

The function declaration IR has a property named `modality`. The modality is related to method overriding. In all previous listings, the property was set to `FINAL`. This property can have two other options: `OPEN` and `ABSTRACT`. The modifier `FINAL` prohibits other methods from overriding the current method. An `OPEN` method is possible to override, and at the same time, it must have a body (the `BLOCK\_BODY` in the IR). An `ABSTRACT` method can also be overridden. However, it cannot have a body, and it must be overridden in the first non-abstract child class. Both `FINAL` and `OPEN` methods can be a target of a static call, but `ABSTRACT` methods cannot be. The following listing (2.30) shows an example of an open and subsequently overridden method.

```
open class A {
/*
 * FUN name:foo visibility:public modality:OPEN <> ($this:<root>.A)
↪  returnType:kotlin.Int
 *   $this: VALUE_PARAMETER name:<this> type:<root>.A
 *   ...
 */
    open fun foo() = 1
}

class B: A() {
/*
 * FUN name:foo visibility:public modality:OPEN <> ($this:<root>.B)
↪  returnType:kotlin.Int
 *   overridden:
 *     public open fun foo (): kotlin.Int declared in <root>.A
 *   $this: VALUE_PARAMETER name:<this> type:<root>.B
 *    ...
 */
    override fun foo() = 2
}
```

Listing 2.30: Method overriding

An example of a method call is shown in the listing (2.31). By default, methods are called using the virtual call. Static and virtual calls differ in the way they are dispatched. A static call uses a static dispatch which means that the exact called function is determined at compile time. A virtual call uses a dynamic dispatch, and in this case, the exact method can be determined only at runtime. Which method is called depends on the dispatch receiver class. In the IR, the call looks the same in both cases since the IR uses the `IrCall` class for both static and virtual calls. Which dispatch is used can be determined only by the class content.

There are several situations where a static call can be or must be used even for methods. A static call is always used when the dispatch receiver is provided using a `super` keyword. A super call always executes the directly overridden method (relative to the currently evaluated method). This overridden method is typically located in the direct parent class.[†] It is impossible to use a virtual call with the `super` keyword because it would call the same method instead of the

[†]However, there are some exceptions since the parent class may inherit this method from its parent without overriding it. Importantly the target method is always resolvable at compile time.

```
class A {
    fun foo() {
    }
/*
 * CALL 'public final fun foo () ...
 *   $this: GET_VAR '<this> ...
 */
    fun callFoo() = foo()
}
```

Listing 2.31: Method call

overridden one. The `super` keyword is represented by a `superQualifierSymbol` in the `IrCall` class. The super qualifier symbol points to the class of the statically called method. An example is shown in the listing (2.32).

```
class B: A() {
    override fun foo() {
/*
 * CALL 'public open fun foo (): kotlin.Unit declared in <root>.A'
↪  superQualifier='CLASS CLASS name:A modality:OPEN
↪  visibility:public superTypes:[kotlin.Any]' ...
 *   $this: GET_VAR '<this> ...
 */
        super.foo()
    }
}
```

Listing 2.32: Super call

For a similar reason, private methods must always be called using a static call. Otherwise, a wrong method could be called if a child declares its own private method with the same signature. On the other hand, final (non-overridable) methods may be called either by static or dynamic dispatch. The dynamic dispatch can be used because final methods cannot be overridden nor shadowed.

Kotlin, in general, does not allow method shadowing. Therefore, a class cannot declare a new method that shares a signature with another method from its parent class. The two exceptions are overridden methods and private methods.

2.6.7 Extension functions

Extensions are used to associate an additional function with a class outside this class declaration. The function does not become a method of that class, and it is always called via a static dispatch. Each extension function has an extension receiver, which is different from the dispatch receiver. In the IR, the extension receiver is stored in a property `extensionReceiverParameter` of the `IrFunction` class. The listing (2.33) shows an example of an extension declaration.

```
class A
/*
 * FUN name:foo visibility:public modality:FINAL <>
↪  ($receiver:<root>.A) returnType:kotlin.Unit
 *   $receiver: VALUE_PARAMETER name:<this> type:<root>.A
 *   BLOCK_BODY
 */
fun A.foo() {
}
```

Listing 2.33: Declaration of extension function

Extensions are a syntax sugar in the sense that they allow calling a function with the same syntax as if it was a method. An example of the call is shown in the listing (2.34). The same function can be implemented by passing the extension receiver as an additional argument. In fact, this is the way the compiler implements the extension functions.

An extension function can be declared inside another class, and in that case, it becomes a method of that class. An extension method has both a dispatch receiver and an extension receiver. This extension then can be overridden and called using a dynamic dispatch resolved with respect to the dispatch receiver. The extension receiver is never used for a dynamic dispatch.

The extension receiver can be accessed by the same `this` keyword as the dispatch receiver. If the extension also has a dispatch receiver, there is a collision, and the extension receiver has a priority. The dispatch receiver can still be accessed by a qualified `this` keyword. The qualified `this` is resolved by the front end, and only the target variable symbol differs in the IR. This difference is shown in the listing (2.35).

```
class A {
    fun callFoo() {
/*
* CALL 'public final fun foo (): kotlin.Unit declared in <root>'
↪  type=kotlin.Unit origin=null
*   $receiver: GET_VAR '<this> ...
*/
        foo()
    }
}

fun A.foo() {
}
```

Listing 2.34: Extension function call

```
class A {
    fun B.fooA() {
/*
 * GET_VAR '<this>: <root>.A declared in <root>.A.fooA'
↪  type=<root>.A origin=null
 */
        this@A
    }

    fun B.fooB() {
/*
 * GET_VAR '<this>: <root>.B declared in <root>.A.fooB'
↪  type=<root>.B origin=null
*/
        this
    }
}
```

Listing 2.35: Qualified this

2.6.8 Overloading

Two functions are overloaded when they share a name but do not have the same signature. In JVM, the function signature is composed of the function name and its parameter types. Which function is called depends on the argument types (and the count of arguments). The overloading is resolved statically, so what matters is the argument type known during compilation (not the exact type in runtime).

Kotlin has a more complicated overloading resolution because it also considers extension receivers and default arguments. Overloading is handled by the compiler front end. At the IR level, all the calls already have the proper symbols resolved. For this reason, it is not necessary to explicitly handle overloading in the static analysis.

2.7 Properties

Kotlin properties are a replacement for Java fields as their primary purpose is to hold class data. However, they are more than just Java fields. They can be declared outside of class in the global scope to represent global variables. They can also have custom getters and setters (together known as accessors) that alter their behavior.

Properties are in some sense similar to local variables. They have the same syntax, and they also share some features. For example, they can both have delegates, and both can be lateinit. However, local variables do not support all the features of properties. Most notably, local variables cannot have custom getters/setters, and they cannot be referenced (function references are explained in (2.9.3)).

A property declaration is implemented by the class `IrProperty`. The most important properties of this class are: `backingField`, `getter` and `setter`. The property `backingField` contains a declaration for what can be considered an equivalent of a Java field. The backing field is the actual entity in which the property value is stored. A declaration of the backing field has a type `IrField` that is almost identical to `IrVariable` (at least from the static analysis perspective).

Accessors are implemented as regular functions via the `IrSimpleFunction` class. A property getter is a function without parameters that returns the property value. A property setter updates the property value and has one parameter that holds the new value. Also, it does not return a value.

The listing (2.36) shows an example of the simplest possible property. It is declared as `val`, meaning it is read-only, and it does not have a custom getter – only an initializer. A property must always have a getter, so the compiler generates a default one if none is explicitly provided. This default getter only returns the field value. The field value is read similarly to how local variable

reads work. The only difference is that the class `IrGetField` is used instead of `IrGetValue`.

```
/*
 * PROPERTY name:foo visibility:public modality:FINAL [val]

 *   FIELD PROPERTY_BACKING_FIELD name:foo type:kotlin.Int ...
 *     EXPRESSION_BODY
 *       CONST Int type=kotlin.Int value=0

 *   FUN DEFAULT_PROPERTY_ACCESSOR name:<get-foo> ...
 *     correspondingProperty: PROPERTY name:foo ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='... fun <get-foo> ...
 *         GET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:foo ...
 */
val foo: Int = 0
```

Listing 2.36: Property declaration

A backing field cannot be accessed directly from outside the property declaration. To read a value from a property, the code must instead call the property getter (example in the listing (2.37)). Since the getter is just a function (though with a special name), the call uses the already introduced class `IrCall`.

```
val foo: Int = 0

/*
 * CALL 'public final fun <get-foo> (): kotlin.Int declared in
↪  <root>' type=kotlin.Int origin=GET_PROPERTY
 */
foo
```

Listing 2.37: Property read

Mutable properties (declared as `var`) work the same way, except they must also have the setter. Setters are implemented similarly to getters, except they have a different name and slightly different default body. The listing (2.38) shows the setter default implementation. The default setter writes to the backing field using a class `IrSetField` (equivalent of class `IrSetValue`).

```
/*
 * FUN DEFAULT_PROPERTY_ACCESSOR name:<set-foo> ...
 *   correspondingProperty: PROPERTY name:foo ...
 *   VALUE_PARAMETER name:<set-?> index:0 type:kotlin.Int
 *   BLOCK_BODY
 *     SET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:foo ...
 *       value: GET_VAR '<set-?> ...
 */
```

Listing 2.38: IR of default property setter

2.7.1 Computed properties

All previous examples declared the properties as if they were just fields – which they effectively are without custom accessors. In fact, the default accessors are equivalent to the getter/setter pattern from Java (only with a better syntax). Providing a custom getter and setter makes it possible to change the property semantics entirely. The property is no longer restricted to just storing a value, and in some cases, the property might not even have a backing field.

In a custom accessor body, the property backing field can be accessed by a soft keyword `field`. An example of such a getter is shown in the listing (2.39). The presented code has the same semantics as the one with the default getter. The IR is also almost the same compared to listing (2.38). The only difference in the IR is that the getter is explicit and is therefore not marked as `DEFAULT\_PROPERTY\_ACCESSOR`.

```
val foo: Int = 0
    get() = field
```

Listing 2.39: Property with a custom getter and a backing field

A property does not need a backing field if it has a custom getter and no accessor accesses the backing field.[†] The compiler does not generate IR for the backing field if it is not necessary. The missing backing field is showcased in the listing (2.40).

[†]The requirement implicitly forbids the presence of a default accessor.

```
/*
 * PROPERTY name:foo visibility:public modality:FINAL [val]
 *   FUN name:<get-foo> ...
 *     correspondingProperty: PROPERTY name:foo ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing ...
 *         CONST Int type=kotlin.Int value=0
 */
val foo: Int
    get() = 0
```

Listing 2.40: Computed property

2.7.2 Member properties

So far, everything in this section was implicitly about the global properties. The member properties differ from the global properties in two significant ways:

- They belong to some object and need a reference to it.
- They are initialized during the construction of the object.

Accessors are implemented as functions, so the reference is passed as their dispatch receiver. The `IrGetField` and `IrSetField` also need access to the object. For that reason, both have a property `receiver`.[†] The listing (2.41) shows an example of a simple read-only member property. A notable change from the previous examples is in the last line of the IR dump. There the object is passed to the `IrGetField`. The setter IR is modified similarly, and its example is therefore omitted.

Since accessors are methods effectively, they support features related to inheritance. They can be overridden, they can be called using the `super` keyword, and they can be shadowed if declared as private. The implementation of those features also works the same way. The only difference is that the whole property must be declared as `override`, so it is not possible to override only one of the accessors. The backing field is not involved in the overriding in any way. Each overridden property declares its own private backing field, meaning it does not reuse the overridden property backing field.

[†]It is not called `dispatchReceiver` since it technically is not used for dynamic dispatch as field access is resolved statically.

```
class A {
 /*
 * PROPERTY name:foo visibility:public modality:FINAL [val]
 *   FIELD PROPERTY_BACKING_FIELD name:foo ...
 *     EXPRESSION_BODY
 *       CONST Int type=kotlin.Int value=0

 *   FUN DEFAULT_PROPERTY_ACCESSOR name:<get-foo> ...
 *     correspondingProperty: PROPERTY name:foo ...
 *     $this: VALUE_PARAMETER name:<this> type:<root>.A
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='... fun <get-foo> ...
 *         GET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:foo ...
 *           receiver: GET_VAR '<this>: <root>.A ...
 */
    val foo: Int = 0
}
```

Listing 2.41: Member property

2.7.3 Initialization of properties

A property initialization only applies to properties with backing fields. Properties without backing fields cannot be initialized as their value depends only on the custom getter. Each property with a backing field must satisfy one of the following requirements:

- have an initializer
- be lateinit (explained in the following subsection)
- have a type with a default value determined by the platform (for example, an `Int`)
- is provably not read before the first write occurs

The initializer is stored in a property `initializer` of the backing field representation `IrField`. The exact time when this initializer is executed depends on the property type. Member properties are initialized by the class constructor, and the exact process is explained together with constructors in (2.8.1). Initialization of global properties is more complicated as it is partially platform-dependent (at least to my knowledge).

Kotlin guarantees (on all platforms) that a global property field will be initialized somewhen before it is first accessed. The initialization may or may not happen if the property is not accessed during the program execution. It is also not required to initialize the property right before its first access (it can happen sooner).

The remaining explanation is based on my experiments with the JVM platform, so it might not be entirely accurate for other platforms. JVM does not support global constructs like properties, variables, or functions. Kotlin compiles these constructs as static members of a special hidden class (preserving their declaration order). The compiler generates one such class for each file with these global constructs. On the JVM, every static member of a class is initialized together in their declaration order. This initialization happens right before the class is first instantiated or before access to any of its static members.

That means accessing a single global property initializes all global properties from the same file. Calling a global function will also have the same effect. The initialization happens in the order in which the properties are declared in the file.

It is important to note that the exact initialization order may affect the static analysis soundness in some rare cases. The problem may happen if the analysis is sensitive to the order in which exceptions are thrown. The initializer can be any expression, including a function call (or even a `throw` keyword), so it can throw an exception. As a result, the analysis could first run into a different exception than the actual program.

2.7.4 Lateinit modifier

The Kotlin compiler guarantees that all variables (including backing fields) are initialized before their first access. The lateinit modifier removes that guarantee. The modifier can be used for both local variables and properties with backing fields.

Reading a lateinit property before it contains any value causes a runtime exception `kotlin.UninitializedPropertyAccessException`. The compiler inserts a guard before each lateinit variable read to implement this behavior. The guard has a form of an `if` statement with a call to a method that throws the exception.

The listing (2.42) shows what the guard looks like when decompiled to Java. It also shows that the check is not in the IR. The explicit check is unnecessary since the IR represents the lateinit modifier differently. For that purpose, both `IrVariable` and `IrProperty` classes have a boolean property `isLateinit`.

```
/*
 * PROPERTY name:foo visibility:public modality:FINAL [lateinit,var]
 *   FIELD PROPERTY_BACKING_FIELD name:foo ...
 *   FUN DEFAULT_PROPERTY_ACCESSOR name:<get-foo> ...
 *     correspondingProperty: PROPERTY name:foo ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='... fun <get-foo> ...
 *         GET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:foo ...
 */
// Source code
lateinit var foo: A

fun callFoo() {
/*
 * FUN name:callFoo visibility:public modality:FINAL ...
 *   BLOCK_BODY
 *     TYPE_OP type=kotlin.Unit origin=IMPLICIT_COERCION_TO_UNIT ...
 *       CALL 'public final fun <get-foo> (): ...
 */
    foo
}

// Decompiled code (Java)
public static final void callFoo() {
    if (foo == null) {
        Intrinsics.throwUninitializedPropertyAccessException("foo");
    }
}
```

Listing 2.42: Lateinit property

2.7.5 Extension properties

Extension properties are very similar to extension functions. Compared to ordinary member properties, they have one notable limitation: they cannot have a backing field. Extension properties are like global properties with custom accessors from the implementation standpoint. The one difference

is that the accessor has IR equivalent to an extension function, not a global function. An example IR of an extension property is shown in the listing (2.43).

```
class A {
}

/*
 * PROPERTY name:foo visibility:public modality:FINAL [val]
 *   FUN name:<get-foo> ...
 *     correspondingProperty: PROPERTY name:foo ...
 *     $receiver: VALUE_PARAMETER name:<this> type:<root>.A
 *     BLOCK_BODY
 *       RETURN ...
 *         CONST Int type=kotlin.Int value=1
 */
val A.foo: Int
    get() = 1
```

Listing 2.43: Extension property

2.7.6 Delegated properties

The implementation of custom property accessors can be delegated to some other object. Since the delegation works fundamentally the same for both accessors, I will focus only on getters in the following examples. The object to which the property is delegated must implement an operator function `getValue`. If the property is mutable, the object must also have a similar function `setValue`. The listing (2.44) shows this delegation and how it is converted by the back end.

```
class Delegate {
    operator fun getValue(thisRef: Any?, property: KProperty<*>):
    ↪  Int = 0
}
val foo: Int by Delegate()

// Converted to
private val foo_delegate = Delegate()
val foo: Int
    get() = foo_delegate.getValue(null, ::foo)
```

Listing 2.44: Delegated property implementation

The listing (2.45) shows IR representation of the truly delegated property from the listing (2.44). The IR from this example contains two elements that were not discussed yet. For that reason, I have removed them from the example. The first one is the constructor call that creates the `Delegate` object. Constructor calls will be explained in (2.8.1). The second element represents a property reference that will be discussed in (2.9.4). Other than that, the IR effectively matches the converted code from the listing (2.44).

```
/*
 * PROPERTY name:foo visibility:public modality:FINAL
↪  [delegated,val]
 *   FIELD PROPERTY_DELEGATE name:foo$delegate type:<root>.Delegate
↪  ...
 *     EXPRESSION_BODY
 *       CONSTRUCTOR_CALL ...

 *   FUN DELEGATED_PROPERTY_ACCESSOR name:<get-foo> ...
 *     correspondingProperty: PROPERTY name:foo ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='... fun <get-foo> () ...

 *         CALL 'public final fun getValue (...): kotlin.Int
↪  [operator] ...
 *           $this: GET_FIELD 'FIELD PROPERTY_DELEGATE
↪  name:foo$delegate ...
 *           thisRef: CONST Null type=kotlin.Nothing? value=null
 *           property: PROPERTY_REFERENCE ...
 */
val foo: Int by Delegate()
```

Listing 2.45: IR of delegated property

The previous example shows a global property, but member properties work almost the same. The only difference is that the associated object is passed to the `getValue` function in the `thisRef` parameter.

Local variables can also be delegated even though they do not support custom accessors. The semantics of the delegation is the same, but the implementation is different. The IR cannot use `IrVariable` as it does not have properties for storing accessors. Instead, `IrLocalDelegatedProperty` is used. This class is similar to `IrProperty`, except it does not have a backing field.

The listing (2.46) shows an example with a delegated local variable. The resulting IR is not that different from the case with ordinary properties, except

the local variable is referenced using another class. Access to a delegated local variable is translated to an accessor call as if the variable was a property.†

```
fun a() {
 /*
 * LOCAL_DELEGATED_PROPERTY name:foo ...
 *   VAR PROPERTY_DELEGATE name:foo$delegate ...
 *     CONSTRUCTOR_CALL ...

 *   FUN DELEGATED_PROPERTY_ACCESSOR name:<get-foo> ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='... fun <get-foo> (): ...

 *         CALL 'public final fun getValue ...
 *           $this: GET_VAR 'val foo$delegate ...
 *           thisRef: CONST Null type=kotlin.Nothing? value=null
 *           property: LOCAL_DELEGATED_PROPERTY_REFERENCE ...
 */
    val foo: Int by Delegate()
}
```

Listing 2.46: Local delegated property

The delegate for a delegated property can be provided by another object. The object providing the delegate must implement the operator function `provideDelegate` to support this feature. An example of this implementation is shown in the listing (2.47). Since it is just syntax sugar, the IR does not fundamentally change compared to the listing (2.45).

```
class DelegateProvider {
    operator fun provideDelegate(thisRef: Any?, prop: KProperty<*>)
    ↪  = Delegate()
}

val foo: Int by DelegateProvider()

val foo: Int by DelegateProvider().provideDelegate(null, ::foo)
```

Listing 2.47: Property delegate provider

†In the compiled code, each access to the local variable is directly replaced by a call to the delegate (so no extra getter is generated).

2.8 Classes

Classes group together functions and data (in the form of properties). These functions and properties are then called class members. The previous two sections already described how this membership changes the semantics of functions and properties. This section focuses on the classes themselves as well as the remaining associated constructs.

Each class in Kotlin must inherit from another class; by default, this class is `kotlin.Any`. The only exception to this rule is the class `Any`. For this reason, `Any` is a supertype of all classes.

At the IR level, a class is represented by the class `IrClass`. An example of an empty class is shown in the listing (2.48). `IrClass` contains the following interesting properties:

kind – is an enum called `ClassKind` that says which type of class the construct represents. It can be one of: class, interface, enum, annotation, or object.

modality – has a similar meaning as the modality of methods (explained in (2.6.6)). One difference is that only abstract classes can have abstract methods, and they cannot be directly instantiated.

superTypes – is a list of all parent classes (and interfaces).

thisReceiver – holds `IrValueParameter` that represents the receiver accessible by the `this` keyword.

declarations – contains all the member declarations inside the class.

```
/*
 * CLASS CLASS name:A modality:FINAL visibility:public
↪  superTypes:[kotlin.Any]
 *   $this: VALUE_PARAMETER INSTANCE_RECEIVER name:<this>
↪  type:<root>.A
 *   ...
 */
class A
```

Listing 2.48: Class declaration

A class, as declared in the previous listing, is never entirely empty. Every class has at least one constructor and the `equals`, `hashCode`, and `toString` methods. If these declarations are not explicitly present, the compiler will generate them. These declarations make every class listing quite long (it has at least four additional methods). For this reason, the listings in this section never include the whole class IR at once.

The example omits two types of declarations: constructor and fake overrides of the three methods. Constructors will be explained in the following subsection. Fake override is either a function (`IrSimpleFunction`) or a property (`IrProperty`) with a flag `isFakeOverride` set to `true`. This flag means the declaration is inherited from a supertype and is not explicitly overridden by the current class. A fake override does not have a body (as shown by the example in listing (2.49)). The compiler ensures that a fake overridden method is never directly called and instead makes a super call to the parent method.

```
/*
 * FUN FAKE_OVERRIDE name:toString visibility:public modality:OPEN
↪  <> ($this:kotlin.Any) returnType:kotlin.String [fake_override]
 *   overridden:
 *     public open fun toString (): kotlin.String declared in
↪  kotlin.Any
 *   $this: VALUE_PARAMETER name:<this> type:kotlin.Any
 */
```

Listing 2.49: IR of fake override method

2.8.1 Constructors

Constructors are a special type of function that create class instances. The IR represents them as `IrConstructor`, which inherits from `IrFunction`. Therefore, `IrConstructor` shares most properties with `IrSimpleFunction`, used for regular functions. In fact, constructors are very similar to statically dispatched functions. There are still some differences; for example, the constructed type is returned implicitly – without a return statement.

One notable property unique to `IrConstructor` is the flag `isPrimary`. This flag decides if the constructor is primary or secondary. Primary constructors always have precisely two statements in their body and cannot have more.

The first statement is a delegating call to a parent constructor represented as `IrDelegatingConstructorCall`. This delegating call is almost the same as a regular function call, except it can only be used to call other constructors.

The second statement is a special instruction that initializes the created object. This instruction has a class `IrInstanceInitializerCall`. The back end initializes the object by replacing this instruction with all blocks that participate in the object initialization. These initialization blocks are explained later.

The listing (2.50) shows an example of the simplest primary constructor. In this case, the compiler generated the constructor because it was not declared explicitly. The default constructor is no different from an explicit constructor without parameters.

```
/*
 * ...
 * CONSTRUCTOR visibility:public <> () returnType:<root>.A [primary]
 *   BLOCK_BODY
 *     DELEGATING_CONSTRUCTOR_CALL 'public constructor <init> ()
↪  [primary] ...
 *     INSTANCE_INITIALIZER_CALL classDescriptor='CLASS CLASS name:A
↪  ...
 */
class A
```

Listing 2.50: IR of default constructor

A primary constructor can have parameters and can optionally declare them as properties. Parameters of the primary constructor (but not secondary) can be used to initialize properties in the class. The listing (2.51) presents this property initialization. In this example, both the constructor and property are the class's direct children (the remaining IR is omitted). The alternative syntax is an example of a property declaration directly in the constructor. The alternative syntax produces identical IR.

A class can have only one primary constructor, but it can have several secondary constructors. A secondary constructor can have a custom body, but its parameters cannot be used in a property initializer. However, the properties can still be initialized via an assignment in the custom body.

A secondary constructor does not always have the initialization instruction. This instruction is present only if there is no primary constructor. The reason is that each secondary constructor must take over the responsibilities of the

```
class A(foo: Int) {
/*
 * ...
 * CONSTRUCTOR visibility:public <> (foo:kotlin.Int) ...
 *   VALUE_PARAMETER name:foo index:0 type:kotlin.Int
 *   BLOCK_BODY
 *     DELEGATING_CONSTRUCTOR_CALL ...
 *     INSTANCE_INITIALIZER_CALL ...
 *
 * PROPERTY name:foo visibility:public modality:FINAL [val]
 *   FIELD PROPERTY_BACKING_FIELD name:foo type:kotlin.Int ...
 *     EXPRESSION_BODY
 *       GET_VAR 'foo: kotlin.Int ...
 */
    val foo: Int = foo
}

// Alternative syntax
class A(val foo: Int)
```

Listing 2.51: Class with a member property initialized in the primary constructor

missing primary constructor. The taken responsibilities include both the object initialization and calling the parent constructor. However, the secondary constructor does not do either if the primary constructor is present. Instead, it must call the primary constructor as the first statement in the body. This type of delegation is showcased in the listing (2.52).

```
class A() {
/*
 * CONSTRUCTOR visibility:public <> (i:kotlin.Int)
↪  returnType:<root>.A
 *   VALUE_PARAMETER name:i index:0 type:kotlin.Int
 *   BLOCK_BODY
 *     DELEGATING_CONSTRUCTOR_CALL 'public constructor <init> ()
↪  [primary] declared in <root>.A'
 */
    constructor(i: Int): this()
}
```

Listing 2.52: Secondary constructor

Since the primary constructor cannot have a body, there must be another way to perform complex initialization. The init block serves this exact purpose.

Its content is implicitly called by the initialization instruction. The backing field initializers also participate in the object initialization, so the initialization instruction also calls them. A single class can have multiple init blocks and properties with backing fields. All init blocks and backing field initializers are called in the order in which they are declared.

In the IR, the init block has a class `IrAnonymousInitializer` and is always stored directly in the class body. The listing (2.53) shows an init block that initializes a read-only property. Interestingly, the init block directly accesses the property field as the property does not have a setter.

```
class A {
    val foo: Int

/*
 * ANONYMOUS_INITIALIZER isStatic=false
 *   BLOCK_BODY
 *     SET_FIELD 'FIELD PROPERTY_BACKING_FIELD name:foo ...
 *       receiver: GET_VAR '<this>: ...
 *       value: CONST Int type=kotlin.Int value=1
 */
    init {
        foo = 1
    }
}
```

Listing 2.53: Anonymous initializer

The initialization instruction targets only constructs declared directly in the current class. It specifically does not call anything from parent classes. On the other hand, each parent class is also initialized because at least one of its constructors must be called. Initialization is never overridden, and it happens in the order from the most base class down to the actually constructed class. This rule has some unexpected side effects. For example, an initialization expression of a base class property is called even if the property is overridden with a new initialization expression.

Creating an object requires calling one of its constructors, for which there is the IR class `IrConstructorCall`. For the purpose of this analysis, the class is almost the same as an ordinary `IrCall`, as shown by the listing (2.54). The only interesting difference is that the constructor call implicitly allocates

memory for the created object. The object allocation is also why there is a distinction between constructor call and delegating constructor call.

```
/*
 * CONSTRUCTOR_CALL 'public constructor <init> (foo: kotlin.Int)
↪  [primary] ...
 *   foo: CONST Int type=kotlin.Int value=1
 */
A(1)
```

Listing 2.54: Constructor call

2.8.2 Interfaces

Each class inherits precisely from one other class, but it can implement any number of interfaces. Interfaces are remotely similar to abstract classes in that they can have methods/properties. These declarations can be either abstract (without implementation) or open (with implementation).

Interfaces are represented as classes (`IrClass`) of kind *interface*, as shown in the listing (2.55). The primary difference from a regular class is that interface does not have a constructor. Interestingly, they still implicitly inherit from `Any` even though interfaces cannot explicitly inherit a class. Interfaces also have all the fake overrides of methods from `Any` (omitted from the example).

```
/*
 * CLASS INTERFACE name:A modality:ABSTRACT visibility:public
↪  superTypes:[kotlin.Any]
 *   $this: VALUE_PARAMETER INSTANCE_RECEIVER ...
 *   ...
 */
interface A
```

Listing 2.55: Interface declaration

Because of interfaces, a single class can have multiple supertypes. Having multiple supertypes can create an interesting situation if some of those super types declare a method with the same signature. In this case, the class inherits the method from multiple sources. The listing (2.56) shows an example in

which this is not a problem because both methods are abstract. From the IR perspective, the only difference is that now there are two overridden methods at once.

```
interface A1 {
    fun foo()
}

interface A2 {
    fun foo()
}

class B: A1, A2 {
/*
 * FUN name:foo ... modality:OPEN <> ($this:<root>.B) ...
 *   overridden:
 *     public abstract fun foo (): kotlin.Unit declared in <root>.A1
 *     public abstract fun foo (): kotlin.Unit declared in <root>.A2
 *   $this: VALUE_PARAMETER name:<this> type:<root>.B
 *   BLOCK_BODY
 */
    override fun foo() {
    }
}
```

Listing 2.56: Overriding the same method from multiple interfaces

What causes a problem is when there are multiple inherited implementations. The method then must be explicitly overridden. The method may want to call one of those inherited implementations. This super call would be, under normal circumstances, done using the `super` keyword. However, since there are multiple options, `super` would not be able to distinguish which implementation to call. The `super` keyword has a qualifier like in the listing (2.57) to solve this problem. The correct *super* symbol for the IR is provided by the front end.

Multiple overridden methods also affect rules for default arguments. Only the base declaration of each method can specify default arguments, which are then the same for all overrides. This rule prevents the method from having multiple different default values for a single parameter. However, the same problem can happen if multiple base declarations of the method have default arguments. For this reason, it is not possible to inherit two interfaces that both have the same function with a default argument for the same parameter. So, for

```
interface A1 {
   fun foo() = 1
}

interface A2 {
   fun foo() = 2
}

class B: A1, A2 {
/*
 * RETURN ...
 *   CALL 'public open fun foo (): kotlin.Int declared in <root>.A2'
↪  superQualifier='CLASS INTERFACE name:A2 ...
 *     $this: GET_VAR '<this> ...
 */
   override fun foo() = super<A2>.foo()
}
```

Listing 2.57: Qualified super call

example, the code in listing (2.58) will not compile – even though the specified default argument is the same in both cases.

```
interface I1 {
    fun foo(i: Int = 1, j: Int)
}

interface I2 {
    fun foo(i: Int = 1, j: Int)
}

class B: I1, I2 {
    override fun foo(i: Int, j: Int) {
    }
}
```

Listing 2.58: Default arguments with multiple base methods

The no collision rule technically allows each base method to declare a default argument, but only for different parameters. So the above example would compile if the method `I2.foo` had this declaration instead: `fun foo(i: Int, j: Int = 1)`. In this case, the method `B.foo` has default values for both parameters.

The fact that this combination of default arguments is allowed has an implication for a caller of such a method. In the IR, the default arguments are kept in the method declaration, not in the call. So to find the default arguments, the back end (and the static analysis) has to search each base declaration of the called method. The front end ensures that each parameter will have at most one default value. However, each default argument may be in a different base declaration.

2.8.3 Objects

An object class is effectively a syntax sugar for a singleton pattern [8]. In contrast to other syntax sugar, this one is not resolved by the front end. In the IR, the object class is implemented as the `IrClass` of a kind `object`. The rest of the IR is similar to other classes, except the object class has a private constructor accessible only by the compiler. The listing (2.59) shows example of object class declaration.

```
/*
 * CLASS OBJECT name:A modality:FINAL visibility:public
↪  superTypes:[kotlin.Any]
 *   $this: VALUE_PARAMETER INSTANCE_RECEIVER name:<this>
↪  type:<root>.A
 *   CONSTRUCTOR visibility:private <> () returnType:<root>.A
↪  [primary]
 *     ...
 */
object A
```

Listing 2.59: Object declaration

The single instance of the object class can be accessed globally. The IR has a special expression `IrGetObjectValue` that returns the object instance. As shown by the listing (2.60), this expression only holds a symbol to the retrieved object class.

2.8.4 Enum classes

An enum class can be seen as an object class with multiple different instances, all globally accessible. Therefore, they can have methods and properties but

```
/*
 *  GET_OBJECT 'CLASS OBJECT name:A ...
 */
A
```

Listing 2.60: Object access

also custom constructors. The constructors are still private, but they can be used explicitly to create each enum entry. Like in the case of regular classes, if the constructor is not explicit, it is automatically generated.

An example of an enum class is shown in the listing (2.61). Implementation-wise, enum classes are once again represented as `IrClass`, this time of kind `ENUM\_CLASS`. By default, enum classes contain even more constructs than regular classes. The listing shows only objects of type `IrEnumEntry` that declare each enum entry. The class contains the property `initializerExpression`. This expression holds the constructor call used to create the associated object. Enum classes have an extra class `IrEnumConstructorCall` to represent the constructor call. This class is not different from the `IrConstructorCall` (at least for the static analysis purpose).

The previous listing omits multiple generated fake override methods declared in the parent class `kotlin.Enum`. These methods are not interesting for the static analysis since the actual implementation is in the standard library. However, there are also two special synthetic methods. The IR of these methods is shown in the listing (2.62). These two methods are used to retrieve the enum entries. The method `values` always returns all entries while the `valueOf` method returns only entry with the matching name. The `SYNTHETIC\_BODY` means it is up to the back end to generate the correct implementation.

Accessing an enum entry is semantically similar to accessing an object class, but the IR implements it by a different class `IrGetEnumValue`. However, the content and meaning of `IrGetEnumValue` and `IrGetObjectValue` are very similar. Listing (2.63) shows the IR dump of this class.

2.8.5 Inner classes

A class declared inside another class can be marked as `inner`. The advantage of doing so is that the inner class members can access the outer class members.

```
/*
 * CLASS ENUM_CLASS name:A modality:FINAL visibility:public
↪  superTypes:[kotlin.Enum<<root>.A>]
 *   $this: VALUE_PARAMETER INSTANCE_RECEIVER ...

 *   CONSTRUCTOR visibility:private <> () returnType:<root>.A
↪  [primary]
 *     BLOCK_BODY
 *       ENUM_CONSTRUCTOR_CALL 'public constructor <init> (...)
↪  [primary] declared in kotlin.Enum'
 *         <E>: <root>.A
 *       INSTANCE_INITIALIZER_CALL ...

 *   ENUM_ENTRY name:X
 *     init: EXPRESSION_BODY
 *       ENUM_CONSTRUCTOR_CALL 'private constructor <init> ()
↪  [primary] declared in <root>.A'

 *   ENUM_ENTRY name:Y
 *     init: EXPRESSION_BODY
 *       ENUM_CONSTRUCTOR_CALL 'private constructor <init> ()
↪  [primary] declared in <root>.A'
 *   ...
 */
enum class A {
    X, Y
}
```

Listing 2.61: Enum declaration

The inner class implicitly captures an object of the outer class to implement this behavior. For this reason, an inner class can be instantiated only in a scope containing the intended object to capture.

The listing (2.64) contains an example of an empty inner class. If a class is inner or not is decided by a flag `isInner` in the class `IrClass`. The class IR itself does not differ in any other way, but all constructors do. Each constructor has an additional parameter `$outer` representing the captured outer class object. An inner class does not explicitly store this parameter even if it is necessary. This task is left to the back end.

The outer object can be used as a dispatch receiver for calling members of the outer class. An example of such a call is shown in the listing (2.65). The IR does not explicitly use the captured outer object. Instead, it acts as if the object was stored as *this*.

```
/*
 *   FUN ENUM_CLASS_SPECIAL_MEMBER name:values visibility:public
↪  modality:FINAL <> () returnType:kotlin.Array<<root>.A>
 *     SYNTHETIC_BODY kind=ENUM_VALUES

 *   FUN ENUM_CLASS_SPECIAL_MEMBER name:valueOf visibility:public
↪  modality:FINAL <> (value:kotlin.String) returnType:<root>.A
 *     VALUE_PARAMETER name:value index:0 type:kotlin.String
 *     SYNTHETIC_BODY kind=ENUM_VALUEOF
 */
```

Listing 2.62: IR of synthetic enum methods

```
/*
 *  GET_ENUM 'ENUM_ENTRY name:X' type=<root>.A
 */
A.X
```

Listing 2.63: Enum access

2.9 Functional programming

Kotlin treats functions as first-class citizens meaning developers can work with functions as with other values. Functions can be referenced, and this reference can be subsequently stored in a variable or passed to a so-called higher-order function.[†] Functions can be declared inside other functions, capturing their local variables. Kotlin also provides a lot of syntax sugar like lambda functions or anonymous classes.

To represent a function as a value, it must first have some type. For this purpose, Kotlin has a special family of generic classes. These classes are named in the form of `kotlin.FunctionN`, where the N suffix is the number of parameters the function has. The parameter and function return types are generic parameters of the `FunctionN` classes. Each `FunctionN` has an operator method `invoke` with the corresponding number of parameters. The `invoke` method is used to call the function.

[†]Higher-order functions are like normal functions, except they either take another function as an argument or return it.

```
class A {
/*
 * CLASS CLASS name:B modality:FINAL visibility:public [inner]
↪  superTypes:[kotlin.Any]

 *   $this: VALUE_PARAMETER INSTANCE_RECEIVER name:<this> ...
 *   CONSTRUCTOR visibility:public <> ($this:<root>.A)
↪  returnType:<root>.A.B [primary]

 *     $outer: VALUE_PARAMETER name:<this> type:<root>.A
 *     BLOCK_BODY
 *       DELEGATING_CONSTRUCTOR_CALL ...
 *       INSTANCE_INITIALIZER_CALL classDescriptor='CLASS CLASS
↪  name:B ...
 *    ...
 */
    inner class B
}
```

Listing 2.64: Inner class declaration

```
class A {
    fun foo() {
    }

    inner class B {
/*
 * FUN name:callFoo ...
 *   $this: VALUE_PARAMETER name:<this> type:<root>.A.B
 *   BLOCK_BODY
 *     CALL 'public final fun foo (): kotlin.Unit ...
 *       $this: GET_VAR '<this>: <root>.A declared in <root>.A' ...
 */
        fun callFoo() {
            foo()
        }
    }
}
```

Listing 2.65: Inner class calling a method from its enclosing class

Developers rarely work with the function type and `invoke` method directly as Kotlin provides syntax sugar for both. For example, a function with one parameter of type `Int` and the return type `String` would have type `Function1<Int, String>`. This function type can be written as `(Int) -> String`. A function of this type stored in a variable `foo` can be called directly using the invoke method

as: `foo.invoke(1)`. An alternative syntax is `foo(1)`, which mimics an ordinary function call. The compiler front end transforms this syntax sugar back to the original form, so the IR does not distinguish between them.

Examples of both of these transformations can be seen in the listing (2.66). At the same time, the listing shows an example of a higher-order function. There is no difference between calling a lambda, an anonymous function, or a function reference – all have the same type. Since `invoke` is a method, the called function is passed as a dispatch receiver. Arguments for the function expression are then passed as arguments to the `invoke` method.

```
/*
 * FUN name:a visibility:public modality:FINAL <>
↪  (foo:kotlin.Function1<kotlin.Int, kotlin.String>)
↪  returnType:kotlin.String

 *   VALUE_PARAMETER name:foo ...
 *   BLOCK_BODY
 *     RETURN ...
 *       CALL 'public abstract fun invoke (p1: P1 of
↪  kotlin.Function1): R ...
 *         $this: GET_VAR 'foo: ...
 *         p1: CONST Int type=kotlin.Int value=1
 */
fun a(foo: (Int) -> String) =
    foo(1)
```

Listing 2.66: Higher-order function calling invoke

2.9.1 Local functions

A function declared inside another function is called a local function. Local functions behave almost the same as ordinary functions. The notable difference is that they cannot be directly called (and referenced) from outside of the scope in which they were declared. Their IR is also almost identical to the other functions (as shown in the listing (2.67)), except they have local visibility.

An example of a local function call is shown in the listing (2.68). As the listing shows, there is also no significant difference in the IR representation compared to regular functions. The same is true for local function references that will be presented later.

```
/*
 * FUN LOCAL_FUNCTION name:foo ...
 *   BLOCK_BODY
 *     RETURN ...
 *       CONST Int type=kotlin.Int value=1
 */
fun a() {
    fun foo(): Int = 1
}
```

Listing 2.67: Local function declaration

```
fun a() {
   fun foo(): Int = 1

/*
 *  CALL 'local final fun foo (): kotlin.Int declared in <root>.a'
↪  ...
 */
   foo()
}
```

Listing 2.68: Local function call

Local functions can capture local variables of the function in which they are declared. Only variables declared before the local function are captured. The dispatch and extension receivers are also captured this way. The capturing is transitive, which plays a role if several local functions are nested inside each other. Because of the transitivity, each local function has access to variables from all functions that enclose it.

The IR does not explicitly express the capturing. Instead, it acts as if the variable belonged to the function, only it is declared elsewhere. Therefore, it is up to the back end to decode and implement the capturing. An example of an IR with capturing function is shown in the listing (2.69).

A class can also be declared inside a function, making it a local class. Local classes have very similar semantics to local functions. They are similar because the property of being local can be generalized to almost all declarations. In general, local declarations can access declarations from their enclosing scope but have local visibility. The local visibility prevents them from being accessed outside the declaration scope. In the case of a local class, its members can

```
/*
 * FUN name:a ...
 *   BLOCK_BODY
 *     VAR name:i type:kotlin.Int [var]
 *       CONST Int type=kotlin.Int value=0
 *     FUN LOCAL_FUNCTION name:foo ...
 *       BLOCK_BODY
 *         SET_VAR 'var i: kotlin.Int [var] declared in <root>.a'
↪  ...
 *           CONST Int type=kotlin.Int value=1
 */
fun a() {
    var i = 0
    fun foo() {
        i = 1
    }
}
```

Listing 2.69: Writing to a captured local variable

access variables from the enclosing functions. For example, a method from a local class captures the enclosing function variables as if the method itself was a local function. Again, the capturing is not explicitly expressed in the IR, and the variable access also looks the same.

2.9.2 Lambda functions

Lambda functions can be described as local functions without a name or, in other words, as anonymous functions. However, Kotlin distinguishes between these two terms. There are some differences in what a lambda function can and cannot do compared to anonymous functions. Though, these differences are only in the code and not at the IR level.

In Kotlin, lambdas can be declared with a pair of curly brackets.[†] The IR represents a lambda function as an ordinary function (`IrSimpleFunction`) that is wrapped inside `IrFunctionExpression`. An example of the lambda declaration and its IR is shown in the listing (2.70).

The following listing (2.71) shows a proper anonymous function with its IR. There is a minor difference compared to the lambda function IR, but it has no particular effect on the static analysis. One difference at the code level is that the

[†]This syntax represents an ordinary block in other languages like Java.

```
fun a() {
/*
 * FUN_EXPR type=kotlin.Function0<kotlin.Int> origin=LAMBDA
 *   FUN LOCAL_FUNCTION_FOR_LAMBDA name:<anonymous> visibility:local
↪  ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='local final fun
↪  <anonymous> (): kotlin.Int declared in <root>.a'
 *         CONST Int type=kotlin.Int value=1
 */
    {
        1
    }
}
```

Listing 2.70: Lambda function

anonymous function uses a regular `return` statement while lambdas implicitly return their last expression. However, the analysis does not have to deal with that since the IR always contains the `return` statement explicitly.

```
fun a() {
/*
 * FUN_EXPR type=kotlin.Function0<kotlin.Int>
↪  origin=ANONYMOUS_FUNCTION

 *   FUN LOCAL_FUNCTION name:<no name provided> visibility:local ...
 *     BLOCK_BODY
 *       RETURN type=kotlin.Nothing from='local final fun <no name
↪  provided> (): kotlin.Int declared in <root>.a'
 *         CONST Int type=kotlin.Int value=1
 */
    fun(): Int = 1
}
```

Listing 2.71: Anonymous function

2.9.3 Function references

A function reference is an expression of one of `FunctionN` types. More precisely, the reference type is actually `kotlin.reflect.KFunctionN` instead of `FunctionN`. `KFunctionN` inherits from `FunctionN`, so it has the `invoke` method, but it also holds additional runtime information about the function.

This runtime information is used for reflection. The `invoke` method of this reference expression calls the referenced function.

In the IR, the function reference is represented as the `IrFunctionReference` class. The class has the property `reflectionTarget`, which is the symbol of the referenced function. `IrFunctionReference` shares a base class with other classes related to calling functions such as `IrCall`. The base class contains all the necessary properties for handling parameters and receivers. An example of a function reference IR is shown in the listing (2.72).

```
fun foo(i: Int) {
}

/*
* FUNCTION_REFERENCE 'public final fun foo ...
↪  type=kotlin.reflect.KFunction1<@[ParameterName(name = 'i')]
↪  kotlin.Int, kotlin.Unit> ...
*/
::foo
```

Listing 2.72: Function reference

Methods can also be referenced via a similar syntax. Method references introduce two complications compared to ordinary functions. First, methods have a dispatch receiver. That receiver can be either captured by the reference or provided later by the reference caller. The second problem is that the method reference must preserve the type of dispatch the method is subjected to.

Combining the two above properties can create unexpected results (depending on the point of view). An example of such a situation is shown in the listing (2.73), where the actually called method is not the one that was referenced. The example shows that the method reference itself may not exactly know which method will be called later. In fact, it can be called multiple times with different dispatch receivers, calling different methods each time. At the same time, not every method can be called using dynamic dispatch (for example, private methods). As a result, the method reference must correctly choose between the two dispatch types.

The IR does not explicitly contain information about the used dispatch type, but it can be inferred from the referenced method. What the IR distinguishes is whether the dispatch receiver is captured or not. The situation

```kotlin
open class A {
    open fun foo() {
        println("A")
    }
}

class B: A() {
    override fun foo() {
        println("B")
    }
}

// References A.foo without capturing the dispatch receiver
val foo = A::foo
// Prints "B"
foo(B()) // or B().foo()
```

Listing 2.73: Virtual method reference

in which the receiver is not captured is shown in the listing (2.74). The later provided dispatch receiver is implemented as an additional parameter of the `invoke` method. For that reason, the function reference type in the example is `KFunction2`, even though the method does have only one regular parameter.

```kotlin
open class A {
    open fun foo(i: Int) {
    }
}

/*
* FUNCTION_REFERENCE 'public open fun foo (i: kotlin.Int) ...
↪  type=kotlin.reflect.KFunction2<<root>.A, @[ParameterName(name =
↪  'i')] kotlin.Int, kotlin.Unit> ...
*/
A::foo
```

Listing 2.74: Method reference without capturing receiver

The following example in the listing (2.75) shows the IR of a call that provides the dispatch receiver externally. There is effectively no difference in the call itself compared to a situation when the object would be a regular parameter.

```
open class A {
    open fun foo() {
    }
}

/*
 * CALL 'public abstract fun invoke (p1: P1 of
↪  kotlin.reflect.KFunction1): R ...

 *   $this: FUNCTION_REFERENCE 'public final fun foo () ...
 *   p1: CONSTRUCTOR_CALL ...
 */
(A::foo)(A())
```

Listing 2.75: Calling a method reference without captured receiver

If a method reference captures the dispatch receiver, it is stored in the property `dispatchReceiver` of the class `IrFunctionReference`. An example of such a situation is shown in the listing (2.76).

```
class A {
    fun foo() {
    }
}

/*
 * FUNCTION_REFERENCE 'public final fun foo () ...
↪  type=kotlin.reflect.KFunction0<kotlin.Unit> ...

 *   $this: CONSTRUCTOR_CALL ...
 */
A()::foo
```

Listing 2.76: Method reference with captured receiver

Function references are also possible to use with extension functions. The extension receiver is handled almost identically to the dispatch receiver. It can also be captured during the reference or provided later as an extra argument. However, there is one notable difference. Extension receivers have their own property `extensionReceiver` in the `IrFunctionReference` class. The distinction is important as extension receivers are not used for dynamic dispatch. An example is shown in the listing (2.77). Since there are two different properties, it would be technically possible to capture

both receivers simultaneously. However, Kotlin does not directly support referencing extension methods, so this cannot happen. The reference itself (without captured receivers) can be obtained only through reflection.

```
fun A.foo() {
}

/*
 * FUNCTION_REFERENCE 'public final fun foo () ...
↪  type=kotlin.reflect.KFunction0<kotlin.Unit> ...

 *   $receiver: CONSTRUCTOR_CALL ...
 */
A()::foo
```

Listing 2.77: Extension function reference with captured receiver

2.9.4 Property references

Properties are like two functions joined together, and therefore they can also be referenced. A property reference has similar semantics to function references, but the implementation is different. Property reference has the class `IrPropertyReference` instead of `IrFunctionReference`. The main difference between these two references is that property references can contain both `getter` and `setter` at the same time.

The property reference type is also different making it possible to call either the `getter` or `setter`. Properties have two families of types: `KPropertyN` and `KMutablePropertyN`, both from package `kotlin.reflect`.[†] `KPropertyN` is used for read-only properties and exposes only the `get` method. Mutable properties utilize `KMutablePropertyN` which has support for both accessors. `KMutablePropertyN` inherits the `get` method from `KPropertyN`. `KPropertyN` inherits from `FunctionN`, so the property reference also behaves like a function reference to the property getter.

Similar to `FunctionN`, the N in `KPropertyN` determines the number of parameters. Properties do not have user-definable parameters, so this number

[†]In contrast to function references, there is no `PropertyN`.

only represents the number of receivers. Since Kotlin has two different types of receivers, this number can go only up to two.[‡]

The listing (2.78) contains an example of a global property reference that is subsequently called. Methods `get` and `set` work similarly to the method `invoke`. They call the referenced property getter, respectively setter, and have corresponding signatures. For example, as shown in the listing, the set method has a parameter `value` that is passed to the property `setter`.

```
var foo: Int = 1

/*
 * VAR name:fooReference
↪  type:kotlin.reflect.KMutableProperty0<kotlin.Int> [val]
 *   PROPERTY_REFERENCE 'public final foo: kotlin.Int [var]' ...
 */
val fooReference = ::foo

/*
 * CALL 'public abstract fun set (value: V of
↪  kotlin.reflect.KMutableProperty0): kotlin.Unit ...
 *   $this: GET_VAR 'val fooReference ...
 *   value: CONST Int type=kotlin.Int value=2
 */
fooReference.set(2)
```

Listing 2.78: Property reference

References to member and extension properties work the same and with the same peculiarities as was the case of function references. However, local variables cannot be referenced yet. There is a class which can be used to reference local variables with delegate. It is the `IrLocalDelegatedPropertyReference`. Nevertheless, it can only be used by the compiler. Implementation-wise, the class is almost the same as `IrPropertyReference`, so the example is omitted.

2.9.5 Function types with receiver

A function type with receiver is a form of syntax sugar. This syntax sugar makes it possible to treat function expressions as extension functions. Compared to

[‡] `FunctionN` can have the number theoretically unlimited. In practice, the number is limited by the platform to some large value.

other syntax sugar features, this one is only partially resolved by the front end. Therefore, it has an impact on the IR.

The IR for function type with receiver is not that different from a regular function type. As the listing (2.79) shows, the difference is only in the parameter type. The receiver is represented as the first parameter in the function type. This is the same approach used for method references without a receiver. In this case, the function type is also marked as `ExtensionFunctionType`.

```
class A

/*
 * FUN name:foo visibility:public modality:FINAL ...
 *   VALUE_PARAMETER name:action index:0
↪  type:@[ExtensionFunctionType] kotlin.Function2<<root>.A,
↪  kotlin.Int, <root>.A>
 *   BLOCK_BODY
 */
fun foo(action: A.(Int) -> A) {
}
```

Listing 2.79: Declaration of a higher-order function with function type with receiver

`ExtensionFunctionType` signalizes that the first type can be provided as a receiver. The name `ExtensionFunctionType` can be misleading, however. The reason is that this feature can be combined with method references that do not capture the dispatch receiver. In this case, the function expression receiver is the dispatch receiver, not the extension receiver. It is possible to pass such method reference as an argument without any conversion, and the IR does not differentiate that directly.

Lambda function can also be used as arguments for parameters with receivers. In this case, the lambda type is inferred to represent an extension function. An example of the inference is presented in the listing (2.80). The example uses a function call to enable the type inference, but an assignment to a variable with the same type also works.

The call from the listing looks complex, but it can be decomposed into several already explained components. The call contains a single `action` argument that holds the lambda function. The lambda has the corresponding type to the parameter of the called function. The function wrapped in the function expression is an extension function, so it has an extension receiver.

```
fun foo(action: A.(Int) -> A) {
}

/*
* CALL 'public final fun foo ...
*   action: FUN_EXPR type=@[ExtensionFunctionType]
↪  kotlin.Function2<<root>.A, kotlin.Int, <root>.A> ...

*     FUN LOCAL_FUNCTION_FOR_LAMBDA name:<anonymous> ...
*       $receiver: VALUE_PARAMETER name:$this$foo type:<root>.A
*       VALUE_PARAMETER name:it index:0 type:kotlin.Int
*       BLOCK_BODY
*         RETURN ... from='local final fun <anonymous> ...
*           GET_VAR '$this$foo: <root>.A ...
*/
foo {
    this
}
```

Listing 2.80: Calling a higher-order function with function type with receiver

The wrapped function also has a single parameter of type `Int`. The parameter is named `it`, which in Kotlin is the default name for a single lambda parameter. Finally, the lambda returns its extension receiver.

Variable holding a function expression with receiver behaves as an extension function declaration. This virtual declaration is accessible in the same scope as the variable itself. The listing (2.81) shows how the function can be called and how it looks on the IR level. The receiver is passed as a regular argument (as with method references). The example also shows an alternative syntax for the call, demonstrating that the receiver truly is a regular argument. The alternate syntax produces the same effect and also the same IR.

2.9.6 Inline functions

A function can be declared as inline using the `inline` keyword. In the IR, the inline function is marked by a flag `isInline` from the `IrFunction` class. The compiler may inline the function at the call site, but it is not always required. The inlining is performed by the back end, which means that the analysis has to somehow account for this feature.

The primary use case for the inlining are functions that take another function as an argument. Inlining a function has two benefits in this case.

```
fun foo(action: A.() -> A) {
    val a = A()

/*
 * CALL 'public abstract fun invoke (p1: P1 of kotlin.Function1): R
↪  ...
 *   $this: GET_VAR 'action: ...
 *   p1: GET_VAR 'val a ...
 */
    a.action() // action(a)
}
```

Listing 2.81: Calling function value with receiver

First, it is a performance optimization if the argument is a lambda expression – it avoids unnecessary allocation of an object. The second advantage is that the inlined lambda becomes part of the caller function body even from the developer's perspective. As a result, inlined lambda functions can do some things that are otherwise not possible. For example, they can directly call `return`, a feature known as *non-local returns*.

By using the non-local return, a lambda can return a value from its enclosing function. The non-local return works as if it was called directly from that enclosing function. Evaluation of the enclosing function ends immediately, returning the result value. The called inline function is also terminated. Therefore, nothing after the lambda call is executed in the inline function that has called the lambda. This situation is demonstrated in the listing (2.82).

If a lambda function can use a non-local return, it can also use the `return` keyword to return from its body explicitly. Anonymous functions also support the explicit return (even if the function is not inlined).[†] The difference is that lambdas need to use a label with the explicit (local) return. An example of this labeled return is shown in the listing (2.83).

The last two listings contain an IR dump of the `IrReturn` class. The first represents the non-local return, and the second one contains the local return. The output is slightly different in each case as the return target is different. The return target is represented as a property `returnTargetSymbol` that points to the function from which the execution should return.

[†]On the other hand, an anonymous function cannot use non-local returns.

```
inline fun foo(action: (Int) -> Int) {
    action(1)
    // Not called
    ...
}

fun callFoo(): String {
    foo {
/*
 * RETURN ... from='... callFoo ... declared in <root>'
 *   CONST String type=kotlin.String value="A"
 */
        return "A"
    }
    // Not called
    ...
}
```

Listing 2.82: Non-local return from inlined lambda

2.10 Other

So far, all the features presented in this chapter are in some way handled by the static analysis proposed in this thesis. This section contains an overview of several Kotlin features for which this is not entirely the case. The static analysis either does not directly support those features, or the implementation is significantly simplified. I explain the inner workings of these features in a very simplified manner compared to the previous explanations. However, a more detailed analysis would be required should these features be implemented appropriately in the future.

2.10.1 Generics

Like Java, Kotlin supports generics, also known as type parameters. Generics allow the developers to write reusable code that is more type-safe. These type parameters can be used together with classes, functions, extensions, and type aliases.

Generics are a compile-time feature and do not add semantic value to the program.[†] Generic types are erased at runtime, which means that all parameter

[†]For this reason, the analysis can safely ignore them, but using them may help with its precision.

```
inline fun foo(action: (Int) -> Int) {
    action(1)
    // Is called
    ...
}

fun callFoo(): String {
    foo {
/*
 * RETURN ... from='... <anonymous> ... declared in <root>.callFoo'
 *   CONST Int type=kotlin.Int value=1
 */
        return@foo 1
        // Not called
        ...
    }
    // Is called
    ...
}
```

Listing 2.83: Local return from inlined lambda

types are replaced with the type `kotlin.Any`. Therefore, the type information is present only at compile time and cannot affect the program at runtime.

In the IR `IrFunction`, `IrClass` and `IrTypeAlias` all implement an interface `IrTypeParametersContainer`. This interface has a property `typeParameters`, which is a list of `IrTypeParameter`. The class contains all necessary properties to implement features like declaration-site variance or constraints.

2.10.2 Coroutines

Kotlin coroutines are a mechanism for asynchronous programming. Coroutines are almost entirely implemented by an external library. For this reason, the analysis can handle them like other libraries. The primary exception is the `suspend` keyword and associated suspend functions that are implemented by the compiler.

If a suspend function needs to wait for some asynchronous event or computation, it can pause its execution. Unlike in classical thread-based multitasking implementations, the thread is not blocked. The thread, therefore, can continue executing some other code. Once the awaited event happens, the

execution can resume from the same place where it was paused (in some cases, using a different thread).

In the IR, a suspend function is marked by a flag `isSuspend` in the class `IrSimpleFunction`. From the IR perspective, there is no other difference. However, the suspend functions do differ a lot from normal functions but only in the compiled code. The primary difference is that the compiler generates an extra `Continuation` parameter for them. At the same time, the compiler performs some transformations of the function body to support the suspending.

2.10.3 Reflection

Reflection is a feature that allows inspecting declarations like classes or functions at runtime. Reflection can be used to reason about the program structure or dynamically call methods by a name created at runtime. Kotlin support for reflection is almost entirely dependent on the target platform and its support for reflection. The most advanced support is on the JVM, where it is even possible to load additional classes dynamically. Static analysis of these features is generally hard or even impossible to implement.

Reflection in Kotlin is implemented similarly to Coroutines. Most of the implementation is in a library, and the IR supports only essential things such as class and function references. These references are used to obtain the object representing that declaration. There are multiple different types of these objects, depending on the declaration. Examples of such types are `KClass` or `KFunction0`.

Class reference expressions are represented in the IR with two different classes: `IrClassReference` and `IrGetClass`. `IrClassReference` is used to obtain the class reference statically, such as `A::class`. `IrGetClass` gets the class reference from an existing object which must be done dynamically.

Chapter 3

Static analysis design

This chapter explains how the proposed static analysis works. Its design was created incrementally together with the prototype implementation, and as such, it changed many times. This chapter captures the final form of the design, not the incremental changes. I will describe some of the changes and their reasons in the following chapter (4).

This chapter is written as a description of a fictional implementation of the static analysis. However, the text is not a documentation of the created prototype. It is meant as a guide for making the implementation. The reason why the prototype differs is explained in the following chapter (4). How the prototype differs from this design is subsequently covered in chapter (5).

The chapter is divided in the following way: The first section introduces the basic concepts of the proposed analysis and explains the reasoning and decisions behind them. The second section is about the static analysis architecture and contains an overview of its core building blocks. The remaining three sections gradually explain the inner workings of the analysis.

3.1 Fundamental design decisions

This section summarizes the key observations made in the first chapter and draws conclusions from them. These conclusions were the starting point for the actual design of the static analysis. Everything stated in this section is so fundamental that it practically has not changed throughout the project.

The section is split into three subsections. The first one lists all the core requirements for the static analysis. The following subsection contains known and intentional limitations that affect the analysis soundness. These limitations can be seen as the opposite of the requirements – things that are not necessary to handle correctly. Based on those stated requirements and non-requirements, I have chosen the appropriate strategy explained in the final subsection.

3.1.1 Requirements

The following requirements have a significant impact on the analysis design as well as some implementation details. The requirements are dictated by the target use case and the state of the Kotlin ecosystem. The analysis focuses on a single narrow use case, and therefore it may not be suitable for other use cases. However, since one of the requirements is `flexibility`, it should be possible to adjust the design to work for many other use cases.

Required output The analysis should return a precise list of all exception types that a given function can throw. The functions of interest represent endpoints in a back-end API. For this reason, they have parameters and can be called at any time while the back end is running. The tracked exceptions are single-purpose domain exceptions resulting from incorrect user input. However, not every user mistake qualifies as a domain exception – it is only those mistakes that violate domain rules. In particular, domain exceptions are not caused by malformed requests (for example, a request with incorrect arguments). To properly analyze these exceptions, the analysis must assume that the function can be called with any possible arguments and at any time.

Soundness In an ideal world, the analysis would be sound to grant that all exceptions are appropriately handled. However, achieving complete soundness is very difficult and not necessary. It is enough for the given use case to list all situations leading to a loss of soundness.[†] Thanks to the list, the analysis users will be aware of these limitations and adjust their code accordingly. This list is in the following subsection (3.1.2).

[†]At least as long as the list does not contain any fundamental or frequently used features.

Accepting any compilable code The analysis must be able to analyze any compilable code to be usable for a real-world project. Therefore, it must support all Kotlin features. On the other hand, the analysis does not have to work with non-compilable code since the compiler checks these problems. So the analysis can assume that any analyzed code is syntactically and semantically valid.

High precision The given use case requires the analysis to have high precision. Frequent false alarms would cause developers to write unnecessary code, which would hurt the adoption of this technology. Some Kotlin features are used more than others – which ones depend on the project type and coding style. It is good to focus on the more generally prevalent features such as virtual method calls and higher-order functions. Especially those two features are necessary to analyze precisely.

Not relying on IR type information This requirement is technically only a consequence of the previous requirement. However, it is so important that I am mentioning it extra. As was explained in (1.5), relying on the type information is not a good strategy for achieving high precision. However, there is another reason not to do this. If the analysis does not require type information, it can easily support constructs without precise types (or with no type). For example, not dealing with types makes support for generics much more straightforward as they can be almost completely ignored. Another benefit is that it will be easier to add support for other languages, including those dynamically typed. Even though multi-language support is not a goal of this thesis, it is nice to have this option available as a potential future improvement.

Execution speed and hardware requirements Analysis performance is not a big priority. It is intended to be run together (in parallel) with acceptance tests on a CI server. Therefore, it is acceptable to have comparable speed and hardware requirements to such acceptance tests.[†] However, it would be beneficial if the analysis could match the performance of unit tests. In that case, developers could use the analysis directly as part of their development process.

[†]Depending on the project size acceptance test can run for minutes or even for hours and may require powerful servers.

Target platform For now, the analysis must support Kotlin JVM as the only target platform. The design should allow adding support for other Kotlin targets in the future. Supporting more targets would make it possible to use this analysis on more projects, including front ends. The problem of domain exception handling on front ends is similar to back ends. So, in theory, this analysis could be used on the front ends with only slight modifications. It is also technically possible to write back ends in Kotlin JS or even Kotlin Native, which could also be supported.

Support for multi-module projects Multi-module projects are common in Kotlin, so they need to be supported. Larger projects are typically split into many modules, and almost all projects use some open-source libraries.[†] The support for multiple modules must be solved by the analysis design. The reason is that the analysis will be implemented as a compiler plugin that works only with a single module at a time.

Flexibility Designing static analysis involves many tradeoffs based not only on these requirements. For making the correct tradeoffs, it is necessary to understand how they will affect the analysis usability for the given use case. It is not always possible to know these effects before actually trying the analysis on different projects. So the design process relies on educated guessing with trial and error. The design must be easy to change to accommodate this process as much as possible.

Maintainability Each release of Kotlin brings new features to the language. In order to support all valid Kotlin code, the analysis will have to be maintained and updated to support new Kotlin versions. The design must make it possible (and ideally easy) to add support for new features.

3.1.2 Intentional simplifications

This subsection lists multiple problems and explains how the analysis will deal with them. The presented solutions have one thing in common: they do not preserve analysis soundness. Better solutions exist for most of the problems.

[†]Libraries with accessible source code can be considered as additional modules (at least for this purpose).

However, incorporating them into this project is not worth it, as not correctly solving those problems makes everything much simpler.

The theoretical loss of soundness is not a big problem. Even though the analysis will not be sound in general, it will still be sound in practice (most of the time). The reason is that these problems do not significantly interfere with the type of code the analysis is intended for. However, it might be necessary to use a better approach for other use cases.

Closed-source libraries Since the analysis will be implemented as a compiler plugin, it requires access to the source code. Libraries for which the source code cannot be obtained are not analyzable this way. The analysis will approximate functions from such libraries as if they returned any possible value. This approach is not sound since a function can throw an exception or mutate a shared state. The analysis will allow its user to specify how the function behaves to address this problem manually. The manual specification is intended as a fallback if the automatic approximation is insufficient. I estimate that the need for manual specification will not be that frequent. The reason is that closed source libraries are not that common, and they are usually not part of the domain logic.

Java code Code written in another language than Kotlin represents the same problem as closed-source libraries. Kotlin makes it easy to integrate Java and Kotlin code. Therefore, it is common to have a project that partially contains Java code either in the form of a module or an external library. The problem with external libraries will eventually become less significant as the Kotlin ecosystem grows. The community has an incentive to reimplement existing libraries in Kotlin because of Kotlin Multiplatform. Projects that directly combine Java and Kotlin are not part of the analysis's intended use case. These projects are typically large legacy systems that are already using some other strategy to handle exceptions and documentation.

Reflection The reflection in Kotlin JVM can do multiple things that affect the analysis. It can obtain runtime information about classes and functions. This information can be safely approximated as any possible value of the given type. This approximation will not reduce the precision that much since this type of

information is typically in the form of a primitive type. The reflection can also call some function that can be approximated by calling any function with the same signature. This approximation is not precise at all, but this is not really a problem. The reflection is usually not used this way in a domain code.

The problematic part of reflection is dynamic class loading. Once again, the analysis cannot analyze code it does not know about. However, this problem has an elegant solution if the loaded class source code is available. In that case, the code can be put into the analysis as another module. From the analysis point of view, the code will be present from the program start. Such approximation is safe and relatively precise. If the code is not available, then the problem is the same as with closed-source libraries.

Implicit exceptions Implicit exceptions are not a theoretical problem. They behave like explicit ones, so the core algorithm for their analysis can be the same. The problem is in the implementation. The analysis would have to know about each operation that can cause an implicit exception. Such operations are then possible to analyze as an `if` block with the operation in one branch and a `throw` instruction in the `else` branch. Identifying and implementing all these operations requires much work, especially because of how Kotlin represents them in the IR. Doing all this work would be practically useless as domain exceptions are always explicit.

Nevertheless, this is an unsound approximation because the presented analysis ignores handlers that are considered unreachable. The approximation is acceptable for the target use case. It will cause problems only if there are handlers for implicit exceptions which interact with domain logic. An example code is a `try` expression catching an implicit exception and rethrowing a domain exception instead. In my opinion, this is a very bad practice as implicit exceptions should handle programming errors. The correct way (that the analysis supports) is to write a guard that directly throws the domain exception.

Coroutines As was mentioned in the (2.10.2), the compiler modifies suspend functions. Most of the time, these modifications can be ignored as they do not change the semantics of the functions. There is one situation (that I know of) in which it matters: When the function suspends itself by obtaining its continuation. The problem here is that this operation completely changes the

function's control flow. The analysis will lose soundness if it is not aware of the suspension. It should be safe to ignore this problem for the current use case as this type of suspension is not common in domain code. However, the problem will eventually have to be adequately solved (which is possible).

3.1.3 The chosen approach

After carefully considering the requirements and options, I have decided to use a method called *abstract interpretation*. This approach was briefly explained at the end of (1.5.3). Because of its inherent advantages and disadvantages, I think that it is a good match for the given problem.

The most notable advantage of abstract interpretation is that it can be sound and have very high precision. The level of precision depends on the chosen program state abstraction and the specific algorithm. Also, the precision can be gradually tuned to meet the requirements of the given use case. The primary disadvantage is that abstract interpretation can be slow compared to other methods.

The proposed algorithm is custom designed, based on general concepts that are shared by all analyses utilizing abstract interpretation. It shares some similarities with the analysis from the previously mentioned article "An abstract interpretation for estimating uncaught exceptions in Standard ML programs" [34]. For example, both methods use an intermediate representation. However, each analysis uses a very different one. My intermediate representation works with dynamic dispatch instead of higher-order functions.

The analysis abstracts the program state in the following way: Each object/value is represented only as a type with fields that hold other objects. Two objects with the same type and the same fields with equal objects are equal. For this reason, the analysis does not distinguish between primitive values of the same type. For example, all integers are considered equal.

In its current form, the algorithm is flow-sensitive but not path-sensitive. This property means that the analysis follows the control flow, and therefore it considers when and what instructions are executed. However, it does not utilize information about the branch conditions. This limitation is a direct consequence of the fact that the analysis does not distinguish primitive values.

Not distinguishing primitive values is not a fundamental limitation of abstract interpretation. The chosen state abstraction naturally reduces the

achievable precision. It is a deliberate decision not to support primitive values because it significantly simplifies the interpretation. Because of how Kotlin represents basic operations, it would be necessary to implement interpretation for dozens, maybe hundreds of methods.[†] Also, there would not be a noticeable benefit for all this work – domain exceptions typically depend on a user input and mutable global state.

The core abstract interpretation algorithm needs some entry point as a starting point. Ideally, this entry point is a function without any parameters. Since endpoints are typically methods with parameters, converting them to entry point functions is necessary. Each endpoint then can be interpreted as an independent program.

The analysis performs the conversion by creating additional wrapping function for each endpoint. The wrapping function calls the endpoint function and uses a special `Any` value as an argument for each parameter. The dispatch receiver of methods is also substituted by this `Any` value. The `Any` value acts as an object of any possible type (restricted by the expected expression type). All fields of the `Any` value are also `Any` values. In summary, the `Any` value behaves like an unknown object that can potentially be anything allowed by the type system.

Even though abstract interpretation does not rely on the type system, having type information can improve the analysis precision. The most obvious (but not only) example is the `Any` value. Without the type information, it would represent all classes in the program (which can be many classes – thousands or even more). The number of possible classes usually goes down to a few once the type information is available.

3.2 Architecture

The architecture described in this section is a fundamental part of the analysis design. It is not just an implementation detail of the prototype because it influences how the analysis conceptually works. As per the architecture, the analysis uses two custom intermediate representations (on top of the Kotlin IR). While converting between all these representations, the analysis

[†]The IR represents everything as methods, in this case, delegated to native code.

gradually lowers all advanced features into the most basic ones. Therefore, the architecture influences the way features are analyzed.[†]

The previous section contained several requirements that are best solved at the architecture level. Namely, these are: *flexibility*, *maintainability*, and *multi-module project support*. All three requirements can be satisfied by using a modular architecture. The modular architecture solves these requirements in the following way:

Flexibility With modular architecture, it is easier to change a single module. The other modules will be affected by these changes less than if everything was implemented together. In an extreme case, a whole module could be dropped and designed/implemented from scratch.

Maintainability Not all modules in the architecture depend on the Kotlin compiler. This separation limits the impact of changes in the Kotlin compiler on the analysis implementation. Implementing new features will also be easier since not every module will have to be changed.

Multi-module project support As explained in more detail below, the analysis works in multiple steps. In the first step, the analysis processes each project module separately. The results of each processed module are merged and used all at once in the remaining steps. The support for this process was the primary driving force of the architecture design.

Since the analysis is modular, it consists of several modules. Each module will be explained in detail in its own subsection.[‡] The following is a list of all the analysis core modules:

FIR – is a custom higher-level IR similar to the Kotlin IR.

BIR – is a custom lower-level IR designed specifically for the abstract interpretation algorithm.

Front end – converts the Kotlin IR into FIR.

Runtime – provides FIR for special functions used (but not provided) by the front end.

[†]Some features are analyzed directly by the abstract interpretation, and some are lowered during the preprocessing.

[‡]The prototype implementation follows the architecture and has the same core modules.

Back end – transforms FIR into BIR.

Interpreter – contains the algorithm that performs the static analysis on the BIR.

Analysis – instruments the interpreter to provide the required outputs.

The module names are based on terminology commonly used in compilers. Kotlin compiler also uses this terminology, so these names might be confused. To avoid this type of confusion, from now on, I will use the following conventions:

- Kotlin compiler components (like the front end) will always be referenced with an appropriate adjective (for example, *compiler front end*).
- Modules of the static analysis will be generally used without an adjective or in another way distinguishing them from the compiler modules.

This naming collision can occur only for the following terms: *front end*, *back end*, and *runtime*. Kotlin compiler also contains its own FIR, but this thesis does not use it. So any future reference to FIR is always related to the static analysis FIR. Similarly, plain IR is only present in the Kotlin compiler.

From a high-level perspective, the analysis works in the following way. First, the front-end module converts the Kotlin IR into FIR and stores it into a file. The conversion is done separately for each module. Therefore, at the end of this step, there are many files with FIR. In the second step, the runtime generates its own FIR (which can be optionally stored in an additional file). Next, the back end loads all the generated FIR, merges it together, and converts it to BIR. As a result, the back end works with the whole program, and the BIR also represents the whole program. Subsequently, the analysis module takes the BIR and generates a unique entry point for each function to be analyzed. The analysis module then uses the interpreter module to interpret each entry point. In the end, the analysis module converts the interpretation results into the expected output format.

3.2.1 FIR

The Kotlin IR is relatively easy to work with during the compilation process. However, it is not specifically designed to be used for static analysis, so it

is unnecessarily complex. It contains a large amount of information necessary for the compiler but irrelevant for static analysis. Having more information is sometimes good, and sometimes it just makes things more complicated. These are the primary reasons why the FIR was created. However, there are also some more practical ones like easier unit testing.

FIR is very similar to the Kotlin IR – it has similar structure and naming conventions. The differences are that it can be serialized and is significantly simpler. It also abstracts away some Kotlin features, which makes it more universal. For this reason, the FIR has a slightly different level of abstraction than the Kotlin IR. The abstraction level is not necessarily higher or lower. FIR just expresses some features more conveniently for the target use case.

Everything in the Kotlin IR can be converted to FIR while preserving all semantics, but the opposite is not true. Not every valid FIR is a valid Kotlin IR. This asymmetry is partially caused by FIR being very dynamic. For example, it has types, but they are not required. The second reason is that FIR is generally less strict than the Kotlin IR.

Even though FIR is optimized for Kotlin, it can express the semantics of other languages. The closer the language semantics is to Kotlin, the easier it will be. So, for example, ideal candidates are languages like Java or Scala. However, it is theoretically possible to also support more diverse languages such as JavaScript.

3.2.2 BIR

BIR is derived from FIR, so it has a similar structure, but it has a much lower level of abstraction. BIR is designed specifically for the purpose of exception static analysis. It supports only a necessary subset of FIR features to simplify the analysis. The remaining features are lowered to the supported features by the back end. BIR also does not preserve all semantics of the program (which FIR does) because the analysis does not utilize the full semantics. Notably, it does not support constants and conditions in branches.

Like FIR, BIR elements also have properly defined semantics. The back end needs to know this semantic to convert FIR into BIR. On the other hand, the analysis does not entirely follow the BIR semantics and makes some approximations.

The reason is that even with all the simplifications, BIR is still Turing-complete. Therefore, some approximations are necessary. Compared to FIR, BIR does not have constants and proper conditions, but both can be theoretically represented differently. For example, each constant can be an extra class, and operations can be implemented by dynamic dispatch. Such representation is not practically usable but still possible.

3.2.3 Front end

The general role of the front-end module is to parse a program source code and create the corresponding FIR. In the case of Kotlin, most of this work is already done by the Kotlin compiler. The front end only needs to translate the Kotlin IR to FIR. Because the front end is implemented as a compiler plugin, it depends on a specific compiler/language version. Multiple versions can be supported, but it requires implementing a front end for each version.

FIR effectively creates an API that isolates the front end from the rest of the analysis. This isolation makes the architecture great at supporting different versions of Kotlin and potentially other languages. The isolation must be total to maintain this property. For this reason, only the front end and runtime can depend on the specific language. All other modules use only FIR and BIR. As a side effect, the front end is the only module depending on the Kotlin compiler.

3.2.4 Runtime

Runtime is a very simple module whose only responsibility is to provide FIR for special functions not defined elsewhere. The provided FIR can be an implementation of functions from the standard library or functions build-in in the actual runtime of the program. The analysis users can also reuse the runtime module to provide definitions for functions from unknown third-party libraries.[†]

The runtime also provides additional helper functions for the front end. For example, the front end can use these helper functions to express some features

[†]These explicit definitions are necessary only for functions that have non-negligible side effects (like throwing an exception). The remaining missing functions are resolved by the back end.

not directly representable in FIR. The front end could generate the helper functions every time it needs them. However, having them only once in the runtime module is simpler. Arrays and iterators are good examples of features implemented this way.

The front end indirectly depends on its runtime because it relies on the presence of runtime FIR. Therefore, front ends for different languages will need different runtimes, each for the specific language. However, one runtime should be able to support different versions of the same language because of backward compatibility.

3.2.5 Back end

The primary goal of the back end is to convert FIR into BIR. The back end is the first module that needs FIR of the whole program at once. Otherwise, it cannot generate the correct BIR. The conversion process involves many consecutive steps that gradually perform the conversion, feature by feature.

For the most part back end directly converts FIR elements into equivalent BIR elements. Some FIR elements are not directly expressible in BIR, and those need to be lowered. These lowerings are explained in more detail later in (3.4).

To support calls to other modules, FIR does not require every used function and class to be declared. However, BIR does not allow that. Therefore, the back end must provide a declaration for each used but missing class and function.[†] The back end generates missing functions such that they do nothing except return the `Any` value. Since the generated function is an unsound approximation of the actual implementation, the back end raises a warning. The analysis user can then decide if it is necessary to provide the correct implementation for the missing function manually.

3.2.6 Interpreter

The interpreter module performs the static analysis of a program represented by BIR. The module's name is derived from the used algorithm – abstract interpretation. Even though the interpreter performs an analysis, it technically does not perform the desired analysis (the analysis module does that). The

[†]Even a perfectly valid program can use some missing functions. For example, if the program contains a library without source code.

interpreter does not know anything about the static analysis use case. For example, it has no notion of API endpoints. It can only analyze a single parameterless function or, in other words, an entry point.

The input for the interpreter is the analyzed program (which includes the entry point). The output is a set of exceptions that can terminate the analyzed program (because they are not handled). Both input and output also contain an environment. The environment contains variables present at the program's start/end. The analysis module uses this environment to modify the interpretation process.

3.2.7 Analysis

Since the interpreter module performs general unhandled-exceptions analysis, it can be used for multiple purposes. The analysis module is there to implement the actual domain API exception analysis. The analysis module does that by instrumenting the interpreter in a specific way. The interpreter is called many times during a single analysis run. Each time the analysis provides different inputs. There are two warm-up stages and an analysis stage.

The warm-up stages are there to initialize the program's global state. API endpoints can be called at any time, so the global state must be initialized accordingly. The interpreter is called during the first warm-up stage with an unmodified program and an empty environment. The returned environment contains the initialized global state of the program. The global state is then passed using the input environment in successive runs of the interpreter.

The API endpoints can also modify the global state. The analysis performs the second warm-up stage to ensure that the order of API calls does not affect the results. In this stage, the analysis creates a special function that calls all endpoints in any possible order with all possible arguments. This function can be constructed as a `When` expression nested in a loop. Each function call is in its own branch, and there is an additional branch with a `Return` statement.[†] The function calls use the `Any` value to simulate all possible arguments.

The environment obtained from the second warm-up stage is then used in the analysis stage. The analysis stage performs the actual analysis of each

[†]The extra return ensures that the loop terminates at all possible times.

endpoint. This part of the analysis is done according to the description at the end of (3.1.3).

3.3 Abstract interpretation algorithm

This section explains the semantics of BIR and how the interpreter analyzes this semantics. BIR consists of many elements that are represented as classes in the implementation. The following text uses code from the implementation to show the structure of each element. The code listings are simplified to contain only the element declarations. In reality, there are additional utility methods.

3.3.1 Symbols

Symbols give names to declarations and, at the same time, can be used as pointers to declarations. There are four basic symbol types, each for a specific declaration. These symbols are: `Variable`, `Function`, `Class` and `Label`. The class symbol is also frequently used as a symbol for types. Both FIR and BIR use the same symbols. Additionally, BIR uses the class `TypeSet` to represent several types at once.

As shown in the listing (3.1), all symbols have a name that gives them identity. The function symbol is more complicated than others because it consists of a signature (name) and an optional dispatch receiver type. The dispatch receiver type is present only for functions that represent methods. There can be multiple functions with the same signature and different dispatch receiver types. This feature is used together with dynamic dispatch to implement method overriding.

Symbol names can have practically any value. However, the back end and interpreter use some special symbols that are reserved. The front end cannot generate these symbols to avoid collisions. These reserved symbols are an implementation detail that can change. So, the back end can introduce new reserved symbols if necessary. All reserved symbols always contain characters "[" and "]". The front end should not use these symbols to ensure there is no collision, even in future versions.[†]

[†]The choice of square brackets is intentional since they cannot be used in any symbol name on the JVM platform. The Kotlin front end uses angle brackets that have the same property.

```
class Variable(override val name: String)

class Function(
    val signature: Signature,
    val dispatchReceiverType: Class?,
) {
    class Signature(override val name: String)
}

class Class(override val name: String)

class Label(override val name: String)
```

Listing 3.1: FIR and BIR symbols

3.3.2 Declarations

BIR has three fundamental declarations: `Function`, `Class` and `Program`. All declarations in BIR are global and cannot appear anywhere else except in the Program element. A program declaration (listing (3.2)) represents the BIR of the whole program, so it contains all other declarations. Additionally, it has a signature of the program entry point function. The entry point is represented only as a function signature because knowing the function dispatch receiver type is unnecessary. The entry point cannot have a dispatch receiver, so its type is implicitly null.

```
class Program(
    val entryPoint: Symbol.Function.Signature,
    val functions: List<Function>,
    val classes: List<Class>,
): Element()
```

Listing 3.2: BIR Program declaration

A function declaration (listing (3.3)) has several properties, and all are used to perform a function call. The function symbol is used to determine which function to call. The function body is the code that gets executed as a result of the call. The parameters have the same symbols as variables because they are represented as local variables. The parameter order in the list is significant because it determines which argument is stored in which variable. All the

remaining function properties are optional and are used only as a source of type information. As a result, a local variable does not have to be declared if it has an unknown type. Since parameters have to be declared, their `allowedTypes` property is optional to signalize the absence of the type information.

```
class Function(
    val symbol: Symbol.Function,
    val allowedDispatchReceiverTypes: TypeSet?,
    val allowedReturnTypes: TypeSet?,
    val parameters: List<Parameter>,
    val variables: List<Variable>,
    val body: Expression,
): Declaration() {
    class Parameter(
        val symbol: Symbol.Variable,
        val allowedTypes: TypeSet?,
    ): Element()
    class Variable(
        val symbol: Symbol.Variable,
        val allowedTypes: TypeSet,
    ): Element()
}
```

Listing 3.3: BIR Function declaration

A class (listing (3.4)) is effectively just a structure in BIR. It contains a symbol and optionally fields. As with local variables in functions, these fields are dynamic, and the declaration is there only to optionally provide the field type. However, Class declarations themselves are required. Otherwise, their symbols cannot be used as types. Classes in BIR do not support inheritance, which is why the `TypeSet` class exists. `TypeSet` is primarily used to represent all possible types castable to the original type.[†]

3.3.3 Basic expressions

Most BIR elements are expressions, which evaluate to some value. In a regularly executed program, the value is an object. In the abstract interpretation, the value is a set of objects that the analysis thinks can be the actual result of the expression.

[†]Which is usually the type and all its subtypes, but it can also be used for union types.

```
class Class(
    val symbol: Symbol.Class,
    val fields: List<Field>,
): Declaration() {
    class Field(
        val symbol: Symbol.Variable,
        val allowedTypes: TypeSet,
    ): Element()
}
```

Listing 3.4: BIR Class declaration

The simplest expressions in BIR are those that directly produce value. These expressions are `CreateObject`, `Any`, and `Nothing` – all shown in the listing (3.5). `CreateObject` evaluates to a new empty object of the given type. `Nothing` is a value with no objects, which means it is an empty set.[†] `Any` is the `Any` value introduced in (3.1.3).[‡] `Any` can optionally be typed to limit the number of objects it represents. `Any` can contain anything meaning all fields also contain `Any`. The analysis cannot make many assumptions about the content of `Any`, only those based on the type information.

```
class CreateObject(val type: Symbol.Class): Expression()

class Any(val allowedTypes: TypeSet?): Expression()

object Nothing: Expression()
```

Listing 3.5: BIR basic expressions

Dereferencing `Nothing` is not possible. If that still happens, the analysis assumes it was caused by an approximation error. In that case, the error is resolved by ignoring the current execution path. Therefore, the analysis effectively acts as if it is impossible to reach the problematic instruction. This assumption is not sound because the program can actually have a null pointer

[†] `Nothing` can be seen in multiple other ways: as a null pointer or as the bottom element of a lattice.

[‡] It is the top element of a lattice.

dereference error. The loss of soundness is justified because implicit exceptions are not analyzed properly.[†]

The other basic expression is the `Block` (in listing (3.6)). The `Block` contains a list of other expressions. Evaluating a block means evaluating all expressions inside it. The `Block` evaluates to the same value as its last expression. If the `Block` does not have any expressions, then it evaluates to `Nothing`.

```
class Block(val expressions: List<Expression>): Expression()
```

Listing 3.6: BIR Block expression

3.3.4 Locations

Locations store values. There are two basic types of locations: variables and fields. Variables (or local variables) belong to a function. Each function execution has its own isolated set of variables. Field values are stored in objects, so they are not isolated between function calls.

Functions with dispatch receiver type in their symbol have access to an additional location. This location stores the function dispatch receiver. The dispatch receiver can be accessed by the `This` expression (listing (3.7)). The dispatch receiver can be used like any other value. The location itself is unique in that it is managed by the interpreter, and the program cannot write to it. The location also cannot be accessed from functions without a dispatch receiver declaration.

```
object This: Expression()
```

Listing 3.7: BIR This expression

Locations can be read from and written to. The corresponding elements for these operations are in listing (3.8). Variables and fields need each their own set of operations because fields cannot be accessed without the receiver

[†]The null pointer dereference can be analyzed by this analysis, but it is not practical. The analysis cannot always decide if the error truly exists or is just a result of an approximation.

(owner object). The interpreter knows which function it is currently executing, so variables can be accessed only by their symbols.

```
class GetVariable(val symbol: Symbol.Variable): Expression()
class SetVariable(
    val symbol: Symbol.Variable,
    val body: Expression,
): Expression()

class GetField(
    val receiver: Expression,
    val symbol: Symbol.Variable,
): Expression()
class SetField(
    val receiver: Expression,
    val symbol: Symbol.Variable,
    val body: Expression,
): Expression()
```

Listing 3.8: BIR location access expressions

The get operations return the value stored in the referenced location. In the case of `GetField`, the receiver is a value, and therefore it can contain multiple objects, not just one. For this reason, `GetField` retrieves the field value of each object separately and merges the values into one result value. Since values are immutable, the merging does not affect the stored values.

The semantics of set/write operations depends on the interpreter mode. The interpreter has two modes called `overwrite` and `unification`. The unification mode works as in other static analyses. In this mode, set operations add the value (the result of the body expression) to the location, preserving the currently stored value. Because of that, once a value is written to a location, it cannot be removed.

The overwrite mode is, to my knowledge, unique to this analysis. In this mode, location writes work as in a regular program which means writing to a location removes the previously stored value. The difference of these two modes can be demonstrated on the following example: `foo = A1(); foo = A2();` After these two statements, the value stored in `foo` depends on the used mode. In the unification mode, the value will be `A1` and `A2`, while in the overwrite mode, the value will be only `A2`.

The example shows that the overwrite mode is more precise than the unification mode. However, there is a reason why other analyses use the unification mode only. The analysis may not terminate when interpreting loops and recursions in the overwrite mode.[†] Both loops and recursions are terminated when the abstract state reaches a fixed point – it can no longer change. The underlying problem of the overwrite mode is that the abstract state lattice is no longer climbed only upwards. Each write can set the abstract state to any lattice element, so there might no longer be a fixed point. For this reason, special measures need to be taken to ensure the termination while using overwrite mode.

Repeated writes of an object already stored in the location do not have any effect (in both modes). The object's identity is determined by its memory address, not by its type or content. Therefore, a single location can store multiple objects of the same type. This definition of identity causes even the unification mode not to terminate. This time the reason is that the lattice becomes infinite. Solutions to both of these problems will be explained later, together with loops and recursion.

3.3.5 Branching

Branching is implemented as a `When` expression (listing (3.9)). `When` has a list of branches, and each branch has its body. Notably, branches do not have conditions, so the interpreter cannot decide which branch should be executed. Instead, it executes all branches simultaneously.

```
class When(val branches: List<Branch>): Expression() {
    class Branch(val body: Expression): Element()
}
```

Listing 3.9: BIR When expression

The interpretation of branching is closely related to the concept of execution paths. An execution path follows the program control flow, and it represents

[†]The static analysis must always terminate, or it would not be practical to use it. The analysis cannot detect that it will run forever, so on its own, it cannot decide to stop if that happens. Therefore, it is necessary to design the analysis so that no program's constructs can cause the analysis to not-terminate.

a single specific path through that control flow. Each execution path has a current program state and a pointer to the following instruction. An execution path can reach its end, in which case it is terminated, and its execution is stopped. The termination may also happen for other reasons like a raised exception, but in this case, the execution path is later resumed in the handler.

The `When` expression splits the current execution path into multiple ones, one for each branch. Each branch is evaluated independently in isolation. Therefore, their execution paths share the past state but do not influence each other. After the branch evaluation is complete, all non-terminated paths are merged into a single one. This merged execution path then continues after `When`. If there is no non-terminated branch, the execution does not continue after `When`.

The example in listing (3.10) highlights the key properties of the previously stated rules. The example is written in a custom DSL that creates BIR.[†] The example assumes that the interpreter uses overwrite mode (unification mode produces the same result but for a different reason). In the example, there is an existing variable `x` with an object `E`. The first branch sets the value of variable `y` to `E` and then changes `x` to a new object `E1`. The second branch also sets the value of variable `y` to `E` because the change to `x` did not happen from this execution path perspective. After `When`, the variable `x` contains either `E` or `E1` because it is not possible to determine which path was taken. The variable `y` contains only a single object `E` because both paths stored the same object in the variable.

```
SetVariable("x") { CreateObject("E") }
When {
    Branch {
        SetVariable("y") { GetVariable("x") }

        SetVariable("x") { CreateObject("E1") }
    }
    Branch {
        SetVariable("y") { GetVariable("x") }
    }
}
```

Listing 3.10: An example of When expression evaluation

[†]The same DSL is used in the prototype tests.

`When` in BIR is always considered as exhaustive, which is a difference compared to the Kotlin IR. Because of this property, the interpretation can assume that exactly one branch is always executed. This assumption is essential. It allows the interpreter to drop the original location value if it was changed in all non-terminated branches.

The requirement for a branch to be non-terminated is also important. For example, a branch can write to some variable and then throw an exception (which terminates the execution path). Anything in that branch cannot affect anything after `When`. So the requirement prevents the changed variable from being visible outside the terminated branch.

Like in Kotlin, `When` is an expression, and therefore it evaluates to a value. The value is obtained by merging result values from every non-terminated branch body.

3.3.6 Exceptions

An object of any type can be used as an exception. Exceptions can be thrown by the `Throw` expression (listing (3.11)). Which exception is thrown is determined by the value of the body expression. A `Throw` expression can throw multiple atomic exceptions at once because the value can contain multiple objects. This multi exception is represented as a single value that is later decomposed if need be.

```
class Throw(val body: Expression): Expression()
```

Listing 3.11: BIR Throw expression

Even though `Throw` is an expression, it effectively does not evaluate to any value. It terminates the current execution path, so the value is not accessible. The terminated execution path state is stored and can be later accessed in the exception handler.

Exception handlers are declared by a `Try` expression (listing (3.12)). The semantics of a `Try` expression is almost identical to that of Kotlin. The primary difference is that there is no `finally` block. The exception handling process is also slightly changed since the exception is decomposed first. From the handler's perspective, the exception may contain both handled and not-handled

exceptions. Handling exceptions in this merged form would unnecessarily reduce precision. The decomposition separates the exception into two parts to solve this problem. The current handler handles the first part of the exception, and the rest is passed to the other handlers.

```
class Try(
    val body: Expression,
    val handlers: List<Handler>,
): Expression() {
    class Handler(
        val variable: Symbol.Variable,
        val handledTypes: TypeSet,
        val body: Expression,
    ): Element()
}
```

Listing 3.12: BIR Try expression

Several execution paths are created during the execution of a `Try` expression. One execution path is for the `Try` body. Additionally, each reachable/activated handler has its own execution path.[†] Any created non-terminated execution path might be the correct one, and it is impossible to determine which one it is. So both the state of these execution paths and their results are merged. The merged result is the result of the whole `Try` expression. The merged state is accessible after the `Try` expression. In this sense, the `Try` expression is evaluated similarly to `When`.

3.3.7 Functions calls

Functions can be called by two different BIR expressions: `StaticCall` and `VirtualCall` (listing (3.13)). The difference between them is in the used dispatch type. `StaticCall` uses static dispatch, while `VirtualCall` uses dynamic dispatch. The dispatch type affects only the function selection. The function execution works the same in both cases. Neither of the two calls supports parameter overloading. Only the function signature and the dispatch receiver type decide which function is called.

`StaticCall` precisely knows which function to call because it has the whole function symbol. The dispatch receiver is optional because `StaticCall` can

[†]The interpreter does not evaluate a handler if it is not necessary.

```
class StaticCall(
    val symbol: Symbol.Function,
    val dispatchReceiver: Expression?,
    val arguments: List<Expression>,
): Expression()

class VirtualCall(
    val symbol: Symbol.Function.Signature,
    val dispatchReceiver: Expression,
    val arguments: List<Expression>,
): Expression()
```

Listing 3.13: BIR function call expressions

also call a function that does not have a dispatch receiver. On the other hand, `VirtualCall` always has a dispatch receiver. It uses the dispatch receiver type to select which function variant to call.

A function call is evaluated slightly differently depending on the call type. In the case of `StaticCall`, the process is the following: First is evaluated the dispatch receiver, but only if its expression is present. If the dispatch receiver expression evaluates to `Nothing`, the current execution path is terminated. Subsequently, all arguments are evaluated in the order in which they are stored in the list. Last, the function body is evaluated in an isolated context. The context contains new local variables for the function and all the arguments. The arguments are stored as local variables under the names of the corresponding parameters (determined by their ordering).[†] Once the called function finishes its execution, the control flow returns to the caller.

`VirtualCall` extends the above process by a function lookup step. This step is necessary because the call does not know which exact function should be executed. The function lookup is performed just before the actual function call. The selected function is the one that has a matching signature and a matching dispatch receiver type (to the evaluated dispatch receiver). The dispatch receiver may be composed of multiple objects of different types. In that case, the dispatch receiver value is grouped by the object type. Each group is then resolved separately. As a result, `VirtualCall` can call multiple functions simultaneously, one for each present dispatch receiver type. The effect of all these calls is then merged as if each was performed in a separate `When` branch.

[†]The number of parameters and arguments must match; otherwise the BIR is not valid.

The dispatch receiver grouping involves several edge cases. First, there might be multiple objects of the same type. If so, they are kept together as a single value, and the function is called only once. Another problem arises if there is no function with matching dispatch receiver type. With no function to call, the dispatch receiver is ignored. If this situation causes the `VirtualCall` not to call any function, the execution path is terminated.

Both function calls are expressions, so they evaluate to the return value of the called function. In the case of `VirtualCall`, it is the merged result of all called functions. A function can return a value using a `Return` expression (listing (3.14)). `Return` itself does not evaluate to anything as it terminates the execution path (same as the `Throw` expression). A function can properly end even without `Return` – when the execution reaches the end of the function body. This execution path does not have a return value, and therefore the call evaluates to `Nothing`.

```
class Return(val body: Expression): Expression()
```

Listing 3.14: BIR Return expression

The execution paths inside the called function can branch. Because of that, there might be multiple ways in which the function was terminated. For example, some execution paths can end with an exception while others with a return. Execution paths that end with an exception do not properly end the function execution. For this reason, they do not contribute to the function call result value. On the other hand, return values from all other execution paths are merged into the result value. If there is no properly ended execution path, then the execution does not continue after the function call.

3.3.8 Loops

Repetition is implemented in BIR via the `Loop` expression and two jumps: `Break` and `Continue`. All these expressions are shown in the listing (3.15). `Loop` represents an infinite repetition. The loop can end *properly* (like a function) only if it is terminated by a `Break` expression. The loop also ends (but not properly) when the function returns, or an exception is raised. To end properly means that the execution continues right after the loop.

```
class Loop(
    val label: Symbol.Label?,
    val body: Expression,
): Expression()

class Break(val label: Symbol.Label?): Expression()

class Continue(val label: Symbol.Label?): Expression()
```

Listing 3.15: Loop related BIR expressions

From the loop perspective, there are three different types of execution paths. The first type terminates the loop properly (`Break`). The second type terminates the loop improperly (`Return` and `Throw`). The last type repeats the loop because it either reaches the `Continue` expression or the end of the loop body. After the loop evaluation, the state from all properly terminated execution paths is merged together.[†] This merged state is the program state after the loop. Improperly terminated execution paths are handled the same as in previous cases. The state of the remaining paths from the previous iteration is merged and used as the initial state for the next iteration. The first iteration uses the program state from before the loop.

For a loop to have any meaning, its body must have multiple branches. Some of those branches must terminate the loop, while others must allow the loop to repeat. Without any terminating execution path, the loop is infinite and effectively does nothing of interest. Without any repeating path, the loop is executed only once, so using a loop is not necessary. The `Loop` expression evaluates to `Nothing`, but only if the execution continues after it. Otherwise, it does not have an accessible result value. Both jumps terminate the current execution path, and therefore they never evaluate to anything.

All three expressions have a label. The purpose of these labels and their semantics is the same as in the Kotlin IR. `Break` and `Continue` can terminate an inner loop improperly (almost like an exception) if they target the outer loop.

The loop evaluation works by repeatedly evaluating the loop body until the next iteration cannot do anything new. For this reason, the loop evaluation may stop just after a few iterations, even though it could repeat thousands of

[†]These terminated execution paths can come from any iteration.

times in actual code. This early stopping occurs once the analysis can prove that it can safely approximate any number of iterations.

Deciding when to stop is what makes the loop evaluation difficult. The interpreter cannot precisely determine if the next iteration will do something different (it is just another type of static analysis). Always predicting that it can do something different is sound but not practical because it results in an infinite cycle. A valid approach is to repeat the loop until the program state no longer changes. The comparison is performed before each iteration by comparing the current program state with the initial state of the previous iteration. In other words, the evaluation continues until the program state reaches a fixed point. This approach is still imperfect because sometimes, the loop body does not do anything new, even with a different program state. However, this problem does not affect the analysis soundness or precision (only performance).

This loop termination algorithm requires two things. First, the program state must always reach a fixed point (after finitely many iterations). Second, there must be a way to compare the two program states.

Both location write modes have a problem because, on their own, they do not guarantee the existence of a fixed point. The overwrite mode deals with this problem by switching to the unification mode after a predetermined number of iterations. Therefore, in the overwrite mode, the loop is evaluated several times to see if there is a fixed point. If the fixed point exists, then everything is all right. Otherwise, the loop is evaluated as if the interpreter used the unification mode the whole time. Once the loop is evaluated, the interpreter can switch back to the overwrite mode.

The overwrite mode can reach a fixed point for many common types of loops. For example, it can be used with any loop that does not change the program state and only iterates over some array. An example of a loop that creates a problem is a builder for an infinite structure like a linked list. If each loop iteration adds another element to the list, then there is no fixed point.

The linked list builder is a problem even for the unification mode. The unification mode does nothing to prevent the list from infinitely growing. For this reason, the unification mode must be extended to ensure that the fixed point exists in all cases. In order for a fixed point to always exist, the following requirement must be satisfied: There must be only finitely many locations to which can be written only finitely many times. There are only finitely many

symbols in the program, limiting the number of variables and fields in each object. There are also only finitely many existing objects. Since the unification mode preserves written values, all existing objects will eventually be present in a given location.[†] What breaks the requirement is the possibility of creating new objects (infinitely many of them). A new object means new locations and a new value to store in the old locations.

The solution is to limit the number of objects that can be created in the loop. Alternatively, limit the number of objects preserved after each loop iteration, which improves precision. There are multiple ways how to achieve this objective. I have chosen one that unifies all objects of the same type, but only if they were created in the loop. Objects that existed before the loop are never unified.[‡] The unification of multiple objects replaces every reference to those objects with a reference to a new object. This new object contains the merged content of all unified objects. Now the loop can create only finitely many new objects.[††]

The unification of objects happens only in the unification mode. Doing it in the overwrite mode is not sound, which is why the mode is changed to unification. The reason is that the program can recognize that the unification happened. The distinction is possible in both modes, but the program can alter its behavior in an unsound way in the overwrite mode.

An example of this situation is shown in the listing (3.16). After the unification, both `a` and `b` point to the same object. The object has a field `i` that contains `E1` because it was present in the object from `a`. What happens after `b.i = E2()` depends on the mode. The unification mode adds `E2` to the shared field, so the read `a.i` returns both `E1` and `E2`. This value is a safe (but imprecise) approximation of the correct value `E1`. However, in overwrite mode, the field's value is overwritten with `E2`. Therefore, the read returns only `E2`, which is not a sound approximation.

The second requirement for the termination algorithm was that there must be a way to compare two program states. For the purpose of the loop termination, the following would be enough: Two program states are equal if the code inside the loop body cannot tell them apart. In other words,

[†]Once that happens, no future writes can alter this location.

[‡]It is unnecessary, reduces precision, and most importantly, it would not be possible to switch back to the overwrite mode.

[††]The program contains only finitely many classes.

```
var a = X()
a.i = E1()
var b = X()
// unification occurs
b.i = E2()
a.i == ?
```

Listing 3.16: An example of problem with object unification in the overwrite mode

the program cannot do anything to detect that the program state is different. Once again, this property is not possible to decide precisely. A more general statement is that two program states are equal if no program can tell them apart. However, given how the abstract state works, this is also not trivial to prove. So the final algorithm only compares the structure of both program states.

The structure comparison begins from objects stored in local variables. Each object is compared by identity and recursively by its fields. Both must match, otherwise the program states are not considered equal. The identity comparison can be used only for objects that existed before the loop. New objects do not have a counterpart with the same identity in the other program state. Instead, the algorithm must establish an identity pairing for each new object. For two objects to be paired, they must have the same type and be stored in equivalent locations. The algorithm has to look in all locations for every occurrence of those objects to prove the pairing is valid. If some object does not have any possible pairing, the program states are not equal.

The algorithm above is the basic idea of the comparison, but the actual implementation is significantly more complex. The primary difficulty comes from values that contain multiple objects. The algorithm must find the correct pairings for objects from the two values (objects in values are not ordered).

3.3.9 Recursion

When evaluating a function call, it is also necessary to ensure that the analysis terminates. The interpretation is guaranteed to terminate as long as each nested function call calls a different function. There are only finitely many functions in the program, so there is a very long but finite chain of function calls in the worst case. A problem happens only when function execution eventually leads to a call of itself. In that case, the program contains a recursion.

Recursion is conceptually similar to loop as it causes the interpreter to execute the same code many times. However, a fundamental difference in recursion makes it harder to deal with. What makes the recursion difficult is demonstrated by the example in the listing (3.17). The function from the example either calls itself or throws an exception. There are multiple handlers that each handle a different exception and rethrow another exception. What exception the function actually throws depends on how many times the recursive call is performed. The function can immediately throw `E`. Alternatively, it throws `E1` if it calls itself exactly once. If the recursion is performed twice, the result is `E2`. In all other cases, it is `E3`.

```
Function("foo") {
    Try({
        When {
            Branch { StaticCall("foo") }
            Branch { Throw("E") }
        }
    }) {
        Handler("_", "E") {
            Throw("E1")
        }
        Handler("_", "E1") {
            Throw("E2")
        }
        Handler("_", "E2") {
            Throw("E3")
        }
    }
}
```

Listing 3.17: An example of a problematic recursion

In the example, the function does not change the program state in any way and has no parameters.[†] However, it is still necessary to call it multiple times. Additionally, the example illustrates that the number of recursive calls matters. It is impossible to estimate how many recursive calls are necessary to observe all possible results of the call.[‡] For this reason, recursion cannot be implemented just by ignoring the recursive call after some predetermined amount of time.

[†]A more real-world example would have a control variable to control the recursion. This variable is omitted because the analysis cannot use it in any useful way.

[‡]The presented example can have many more handlers.

The recursion must eventually end, otherwise the analyzed program will never terminate during actual execution. A program that runs into this infinite recursion does not throw any additional exceptions.[†] The interpreter can use this fact and safely assume that eventually, the branch that leads to the recursive call is not taken.

The basic idea of how to analyze the recursion is as follows. At first, the interpreter ignores all calls that would cause a recursion by terminating the current execution path. At the same time, the interpreter marks the function to know which one was called recursively. The remaining execution paths of the function are interpreted as usual. Once the interpretation returns to the first call of the marked function, the interpreter can start evaluating the recursion.

The interpreter already knows how the function behaves if it does not call itself. In other words, it knows the effect of the last iteration in the actual recursion. Now it can evaluate the same function again. However, this time when it reaches the recursive call, it applies the known effect of the function call instead of evaluating it directly. This process simulates that the recursion was called exactly once. The process can be repeated many times, each time simulating one more recursive call. Therefore, the interpreter evaluates the recursion inside out.

The key thing in the whole process is how to record and apply the function's effect. A function's effect is everything observable from outside after the function was called. Namely, it is the return value, thrown exceptions, and mutated program state. In some sense, recording the function effect caches results of the function call. The effect application means that the function call is substituted by the recorded effect. How exactly it is done depends on the interpreter implementation, so it is hard to explain without going into the implementation details.

The recursion evaluation can end once the effect of the function does not change from the previous iteration. The effect comparison works similarly to the comparison performed by loops but involves comparing more program states. The reason is that the program state captured by all exceptions must also be compared. The exceptions, as well as the returned values, must also match.

[†]Eventually, it will end with a stack overflow error, but the analysis does not report implicit exceptions.

Once the recursive evaluation can end, the interpreter uses the last recorded effect as a result of the original function call.

There are two more problems with the process: It completely ignores the parameters of the recursive function, and it might not terminate because of new objects. A function's effect can depend on its arguments. For example, the function can return the argument. So the approximation does not work if the recursive call has different arguments.

Solving both of these problems involves using the object unification. For this reason, the interpreter must switch to the unification mode to evaluate recursion. At the end of each repetition, all newly created objects are unified. Unification itself does not solve the problem with different arguments. Instead, the interpreter merges and remembers all arguments from each recursive function call. Every time the function evaluation is repeated, it gets all previously encountered arguments. Therefore, the function evaluation can simulate multiple different calls at once. The last change is that the recursion evaluation cannot end while the arguments of the recursive function are changing.

3.3.10 Type coercion

Some BIR elements have required types because they are required for the program to have a meaning. Examples include the function dispatch receiver type, the exception type in handlers, or the object types. When it is not required, the types are optional but not useless. They can be used to improve the analysis precision. Whenever the analysis encounters an object with an incorrect type, it ignores it.

Such approximation is sound because the analysis assumes that the analyzed program is compilable. Being compilable also means that the program had to pass type checking. Type checking ensures that the situation mentioned above cannot happen in an actual program. The only other option is that it happened because of some previous imprecise approximation.

This process, named *Type coercion*, occurs everywhere the type information is available. Most notably, it is performed before a value is written to a location. Type coercion is particularly useful for `Any` because it significantly limits the number of possible types it can represent. Usually, the type coercion removes

only some objects from the value. However, it may happen that the whole value is discarded, in which case the value becomes `Nothing`.

3.4 Translation from FIR to BIR

This section focuses on the translation process from FIR to BIR that is done by the back end. FIR and BIR share many core ideas, and most of their semantics have almost one-to-one mapping. The rest of FIR can be seen as an extension to BIR. This section describes FIR with this fact in mind, focusing on the differences between FIR and BIR. Semantics that is the same (or almost the same) is not explicitly explained again. Therefore, if something is omitted, it implicitly means it works the same as in BIR.

When designing both FIR and BIR, I had to decide which features to support directly and which lower in the front end. There is no strict rule, but there are a few guidelines. Everything supported in BIR must be implemented in the interpreter. It is preferable to limit the number of supported features by the interpreter since it is the most complicated module. At the same time, features that are not supported directly must be possible to lower without a significant loss of precision. Everything supported by FIR must be either supported by BIR or lowered by the back end. The rest of the Kotlin features are handled by the front end. Features that could be useful for other languages are generally supported by FIR. Because of that, other implementations of the front end do not have to duplicate that much code. However, some features are far easier to translate in the front end. In that case, doing the translation directly in the front end is preferred.

FIR is composed of many elements, just like BIR. There are three main types of elements: declarations, expressions, and statements. Both declarations and expressions are statements. Having statements is necessary because FIR allows declarations to be nested and mixed with expressions. The concept of statements is taken over from the Kotlin IR as it supports this nesting.

FIR uses the same symbols as BIR, but it additionally has the `Environment` symbol. This environment symbol is either a `Class` or a `Function`. The symbol is used to reference environments. More on environments is in the last subsection (3.4.8).

Types in FIR are represented by class symbols, like in BIR. Types and classes in FIR work more like in Kotlin because there is support for inheritance. The FIR is dynamic, so many declarations do not require a type, but some do. The back end must convert most of these types because BIR elements primarily work with a `TypeSet` instead of a single type. If necessary, a type is converted to a `TypeSet` that contains that type and all its subtypes.[†] Missing types have the same meaning in both FIR and BIR – they represent any possible type.

FIR does not directly support union types since Kotlin also does not have them. In theory, it is possible to represent union types by creating a synthetic class that all classes from the union type inherit. This workaround can be used even for classes from other modules because classes in FIR can be declared multiple times. The back end then merges duplicate class declarations together.

3.4.1 Declarations

FIR introduces several new declarations compared to BIR. A `DeclarationBlock` from the listing (3.18) is one of them. A declaration block is like a regular block, but it contains declarations instead of expressions. It is a replacement for a program declaration that FIR does not have. Since FIR is meant to be generated per module, the program declaration is unnecessary. On the other hand, FIR needs to represent the modules somehow, so this is the primary use case for the declaration block. It is also used as a class body because classes (compared to functions) cannot directly contain expressions.

```
class DeclarationBlock(
    val declarations: List<Declaration>,
): Declaration()
```

Listing 3.18: FIR DeclarationBlock

Local variables in FIR are declared directly in the function body as this is how the Kotlin IR declares them. Similarly, fields can be declared anywhere in the class body. Additionally, FIR has global variables which can be declared anywhere. As in BIR, all variable declarations are optional because all variables

[†]This conversion is one of the reasons why the back end needs to work with the entire program.

are dynamic. FIR has different declarations for each variable type, as is shown in the listing (3.19). This distinction is necessary, otherwise it would not be possible to declare global variables in functions and classes. The primary purpose of this non-typical feature is to make the front-end implementation easier. Conversion of local variables and fields is straightforward. The back end only needs to find them and move them to the BIR declarations.

BIR does not have global variables, so these are represented as fields of a global object. The global object is created at the program start and then passed as an argument in each function call. Therefore, the back end adds a special global parameter to each function.

```
sealed class Variable: Declaration() {
    class Local(
        val symbol: Symbol.Variable,
        val type: Symbol.Class,
    ): Variable()

    class Global(
        val symbol: Symbol.Variable,
        val type: Symbol.Class,
    ): Variable()

    class Field(
        val symbol: Symbol.Variable,
        val type: Symbol.Class,
    ): Variable()
}
```

Listing 3.19: FIR variable declarations

Functions in FIR (listing (3.20)) have a similar declaration as in BIR. Most of the differences are a consequence of things already explained above. It does not have any semantic meaning where a function is declared. This is a significant change compared to the Kotlin IR. For example, in the Kotlin IR, nested functions capture enclosing function variables. FIR represents this feature differently by explicitly capturing environments which will be explained in the subsection (3.4.8). As a side effect of the isolated declarations, methods can be declared anywhere – outside the class body or even in another module. Therefore, their location does not determine the class they belong to. Instead, this is determined by the dispatch receiver type in the function symbol.

```
class Function(
    val symbol: Symbol.Function,
    val returnType: Symbol.Class?,
    val referenceType: Symbol.Class,
    val parameters: List<Parameter>,
    val capturedEnvironments: List<Symbol.Environment>,
    val body: Statement,
): Declaration() {
    class Parameter(
        val symbol: Symbol.Variable,
        val type: Symbol.Class?,
    ): Element()
}
```

Listing 3.20: FIR Function declaration

Classes (listing (3.21)) in FIR are much closer to the Kotlin IR than to BIR. They support inheritance by the `superType` property and have a body. FIR does not have interfaces – they are represented the same as abstract classes. As a result, FIR technically supports multi-class inheritance. However, it does not resolve collisions of fields, so it is up to the front end to make sure this is not a problem.

```
class Class(
    val symbol: Symbol.Class,
    val superTypes: List<Symbol.Class>,
    val capturedEnvironments: List<Symbol.Environment>,
    val modality: Modality,
    val body: Declaration,
): Declaration() {
    enum class Modality {
        Concrete, Abstract,
    }
}
```

Listing 3.21: FIR Class declaration

Classes have modality, which can be either *concrete* or *abstract*. Abstract classes cannot be instantiated, and the back end does not include them anywhere in the BIR. Abstract classes are necessary for the back end to understand the inheritance hierarchy properly. On the other hand, the interpreter does not want to know about them. The reason is `Any`. If abstract classes were present

in BIR, then `Any` would be less precise since instances of abstract classes cannot exist in reality. An example where this imprecision would manifest is shown in the listing (3.22).

```
interface A {
    @Throws(E1::class)
    fun foo() {
        throw E()
    }
}

class B: A {
    @Throws(E1::class)
    override fun foo() {
        throw E1()
    }
}
```

Listing 3.22: An example of acceptance test that utilizes class modality

The example code is taken from the acceptance tests. The `Throws` annotation contains classes of exceptions reported by the analysis. In the example, the method `foo` in interface `A` throws an exception `E`. However, the analysis reports that it throws `E1`. This behavior is intended because that is exactly what would happen if someone called the method `foo` on an object with a static type `A`. The analysis knows that such an object can only have a type `B` because it is the only concrete class in the `A` inheritance hierarchy. Therefore, any virtual call of the method `foo` can execute only the method declared in `B`.

Since FIR has global variables, it must also have a way to initialize them at the program start. In contrast to the Kotlin IR, FIR does not have initialization expressions for the variables. Instead, they are initialized during the first write. The `Init` declaration (in listing (3.23)) is the primary way to do this initialization. `Init` can be declared anywhere, and it is always executed before the rest of the program.

```
class Init(val body: Statement): Declaration()
```

Listing 3.23: FIR Init declaration

The order in which `Init` declarations are executed is unspecified as it cannot be decided at compile time. The problem is caused by dependencies between init blocks. For example, an initial value of one global variable can depend on another global variable. The back end resolves this dependency by generating code that simulates all possible evaluation orders at once. The code is one big loop with a single `When` inside it. Each `Init` block is in its own branch. Then there is an extra branch with `Break`. The semantics of the BIR `Loop` expression takes care of the rest.

This initialization code is put in the entry point function. The back end generates this function during the construction of the BIR `Program` declaration. The global object is also created in this function (at its start) and stored in a local variable. Therefore, code in `Init` can access the global variables the same way as other functions.

3.4.2 Basic expressions

Basic expressions from BIR have their direct counterparts in FIR (listing (3.24)). There are some minor differences: The `Block` expression has statements instead of expressions. `Nothing` is renamed to `Null` to bring the name closer to Kotlin. `Any` has only a single optional type, so it can either represent everything or a type and all its subtypes.

```
class Block(val statements: List<Statement>): Expression()

class CreateObject(val type: Symbol.Class): Expression()

object Null: Expression()

class Any(val type: Symbol.Class?): Expression()
```

Listing 3.24: FIR basic expressions

FIR is designed to preserve the program semantics, and therefore it has constants. Each constant requires a type in addition to its value. The back end converts any constant to the `CreateObject` expression that creates an object of the corresponding type. The constant value is lost in the process. The constant type is necessary because the back end cannot depend on Kotlin types, so it

would not know which object to create. FIR supports the same type of constants as Kotlin (and other JVM languages), as shown by the listing (3.25).

```
sealed class Constant: Expression()

class Boolean(val value: kotlin.Boolean, val type: Symbol.Class):
↪  Constant()
class Char(val value: kotlin.Char, val type: Symbol.Class):
↪  Constant()
class Byte(val value: kotlin.Byte, val type: Symbol.Class):
↪  Constant()
class Short(val value: kotlin.Short, val type: Symbol.Class):
↪  Constant()
class Int(val value: kotlin.Int, val type: Symbol.Class): Constant()
class Long(val value: kotlin.Long, val type: Symbol.Class):
↪  Constant()
class String(val value: kotlin.String, val type: Symbol.Class):
↪  Constant()
class Float(val value: kotlin.Float, val type: Symbol.Class):
↪  Constant()
class Double(val value: kotlin.Double, val type: Symbol.Class):
↪  Constant()
```

Listing 3.25: FIR constant expressions

3.4.3 Locations

Local variables and fields are accessed the same way as in BIR. These access expressions are in the listing (3.26). The access expressions for local variables have different names compared to BIR because FIR distinguishes between local and global variables.

Global variables are accessed by their own access expressions (listing (3.27)). The back end converts these to field access expressions. The receiver for the field access is the global object. The global object is in the special function parameter explained before. Therefore, the object is internally obtained by the `GetLocalVariable` expression.

FIR does support arrays but in a very specific way. There are two access expressions (listing (3.28)) that access dynamic fields. Fields in FIR do not have

```
class GetLocalVariable(val symbol: Symbol.Variable): Expression()
class SetLocalVariable(
    val symbol: Symbol.Variable,
    val body: Expression,
): Expression()

class GetField(
    val receiver: Expression,
    val symbol: Symbol.Variable,
): Expression()
class SetField(
    val receiver: Expression,
    val symbol: Symbol.Variable,
    val body: Expression,
): Expression()
```

Listing 3.26: FIR local variable and field accessors

```
class GetGlobalVariable(val symbol: Symbol.Variable): Expression()
class SetGlobalVariable(
    val symbol: Symbol.Variable,
    val body: Expression,
): Expression()
```

Listing 3.27: FIR global variables accessors

to be declared, so this feature can be used to implement a map. In this case, an array is just a map that uses numbers as keys.[†]

```
class GetDynamicField(
    val receiver: Expression,
    val field: Expression,
): Expression()

class SetDynamicField(
    val receiver: Expression,
    val field: Expression,
    val body: Expression,
): Expression()
```

Listing 3.28: FIR dynamic field accessors

[†]Such implementation is not practical for an actual programming language, but it is sufficient for this type of static analysis.

The `field` expression is used to determine the accessed field name. The `field` expression can evaluate to an object of any type. The string representation of the evaluated object is the field name, or at least it would be in an actual program. The interpreter cannot precisely evaluate the expression because it has no notion of primitive values. Instead, the back end assumes that all `field` expressions evaluate to the same constant. With that assumption, the back end replaces the dynamic accessors with the ordinary static ones that work with a single special field.

This conversion is sound as long as any write to the field happens using the unification mode. However, the back end has no way to force the interpreter to switch to the unification mode. The solution is to wrap each write operation into a `When` expression with two branches. One branch performs the write operation, and the other one does nothing – preserving the original value even in the overwrite mode.

3.4.4 Control flow

Control flow in FIR works almost the same as in BIR. There are the same expressions as shown in the listing (3.29). The only notable difference is that branches do have conditions. Thanks to that, FIR can preserve the program branching semantics.

The back end cannot just discard the conditions because they can have side effects. The result of the condition evaluation can be discarded, but the evaluation must happen. At the same time, it is necessary to preserve the semantics of which conditions are evaluated when. In an example where a second branch is taken, the first branch condition must have evaluated to `false` and the second to `true`. The key observation is that the first condition was evaluated even though the execution continues in the second branch. This means that the executed branch also sees effects caused by all conditions from the previous branches. The back end simulates this property by copying the appropriate conditions at the beginning of each branch. So the first branch starts with the first condition, the second branch has the first and second condition, and so on.

The `When` expression in both FIR and BIR is exhaustive. The back end and interpreter depend on this property to capture the branching semantics correctly. The back end cannot verify if `When` is genuinely exhaustive, so it is up

```
class When(val branches: List<Branch>): Expression() {
    class Branch(
        val condition: Expression,
        val body: Statement,
    ): Element()
}

class Loop(
    val label: Symbol.Label?,
    val body: Statement,
): Expression()

class Break(val label: Symbol.Label?): Expression()

class Continue(val label: Symbol.Label?): Expression()

class Return(val body: Expression): Expression()
```

Listing 3.29: FIR control flow expressions

to the front end to ensure that. The problem of ensuring exhaustiveness does not exist in BIR because it does not have the conditions. It is impossible to create `When` in BIR that semantically is not exhaustive. On the other hand, in FIR, it is possible to construct `When` with conditions that sometimes all evaluate to `false`.

3.4.5 Exceptions

FIR implements exception handling almost identically to BIR. Both have the same exception expressions (listing (3.30)). Additionally, the FIR `Try` expression has an optional `finally` block with the exact semantics as Kotlin. BIR does not need the `finally` block as it can be expressed via other constructs.

Lowering a `finally` block is a little more complicated because it must preserve the semantics of the exception propagation. As a reminder, the `finally` block is always called at the end of the `Try`, and it propagates thrown exceptions unless it itself throws an exception. A pseudocode of the lowering result is shown in the listing (3.31) and will be gradually explained in the following text.

The lowering is performed by nesting the original `Try` expression into two another `Try` expressions. The middle `Try` has a handler that catches all

```
class Throw(val body: Expression): Expression()

class Try(
    val body: Statement,
    val handlers: List<Handler>,
    val finally: Statement?,
): Expression() {
    class Handler(
        val exceptionVariable: Variable.Local,
        val body: Statement,
    ): Element()
}
```

Listing 3.30: FIR exception handling expressions

exceptions and has the content of the `finally` block. As a final instruction, this handler rethrows the caught exception. This construction preserves the propagation property. If the `finally` block throws an exception, the control flow never reaches the inserted rethrow, so the original exception is ignored.

Right now, the handler is not called if the original `Try` expression does not throw an exception. This problem is solved by throwing a helping exception at the end of every preserver body of the original `Try` expression. Therefore, the exception is thrown from the `Try` expression body and all handlers (but not from the `finally` block). The back end uses a special class to represent the helping exception to avoid collisions with other exceptions.

Now the `finally` block inside the handler is called every time. However, the new problem is that the helping exception escapes the handler because it rethrows all exceptions. For that, there is the outer `Try` expression. It has a handler that only catches this helping exception, stopping the helping exception without affecting other exceptions.

The last problem is that the result value of the original `Try` expression is lost. The result value must be accessible if the original `Try` expression does not propagate an exception. This situation happens exactly when the helping exception is thrown and not suppressed in the middle handler. So the solution is to put the result value in a special field of the helping exception. The result value is stored in the field in all places that subsequently throw the exception. The outer handler that catches the helping exception then reads the value from

```
 // Original try
try {
    // A
} catch (e: E) {
    // B
} finally {
    // C
}

 // After lowering
try {
    try {
        try {
            // A without last expression
            throw CallFinally(
                result = // last expression from A
            )
        } catch (e: E) {
            // B without last expression
            throw CallFinally(
                result = // last expression from B
            )
        }
    } catch ([e]: Any) {
        // C
        throw [e]
    }
} catch ([e]: CallFinally) {
    [e].result
}
```

Listing 3.31: Conversion of finally block

the field. The semantics of the BIR `Try` expression ensures that this value is the result of the outer `Try` expression.

3.4.6 Function calls

FIR offers more options than BIR when it comes to calling functions. It supports three types of calls: static, virtual, and newly dynamic. The dynamic call is used to call function values and therefore is explained in the following subsection about function references. `StaticCall` and `VirtualCall` from FIR (in listing (3.32)) are structurally identical to their equivalents in BIR. However, the back end still needs to do some transformations. Namely, it must provide additional arguments to match previously made changes to the

function parameters. The back end adds additional parameters to support global variables and captured environments.

```
class StaticCall(
    val symbol: Symbol.Function,
    val dispatchReceiver: Expression?,
    val arguments: List<Expression>,
): Expression()

class VirtualCall(
    val symbol: Symbol.Function.Signature,
    val dispatchReceiver: Expression,
    val arguments: List<Expression>,
): Expression()
```

Listing 3.32: FIR StaticCall and VirtualCall expressions

`StaticCall` from FIR has the same semantics as in BIR. On the other hand, `VirtualCall` has one difference that the back end must resolve. Since FIR supports inheritance, `VirtualCall` works more like in object-oriented languages. `VirtualCall` calls the appropriate method based on the inheritance hierarchy. More specifically, if an object's class does not contain the called method, the dynamic dispatch will search in the object's parent class. This search does not happen in BIR as `VirtualCall` always selects the function with the matching type. Therefore, the back end needs to provide an explicit implementation for these implicitly inherited methods.[†] The generated methods are simple: they return the result of a super call with forwarded arguments. The super call is implemented as a static call to the parent class method.

3.4.7 Function references

A function reference returns a value that represents the referenced function. The function value acts as any other value. For example, it can be stored in a variable. The function value can be passed to `DynamicCall` (listing (3.33)) that executes the referenced function. The dynamic call is similar to the other calls. It differs in that the called function is decided dynamically based on the `function` expression value. The call arguments and dispatch receiver must

[†]In the Kotlin IR, these implicitly inherited methods are the fake overrides.

match with the called function. Otherwise, the back end will generate invalid code that will later cause a crash in the interpreter. The back end cannot verify that the arguments match, so it does not warn about this problem.

```
class DynamicCall(
    val function: Expression,
    val dispatchReceiver: Expression?,
    val arguments: List<Expression>,
): Expression()
```

Listing 3.33: FIR DynamicCall expression

There are two expressions that can be used to obtain the function reference: `FunctionStaticReference` and `FunctionVirtualReference`. Both are shown in the listing (3.34). The difference between them is how the function is selected during the dynamic call. A dynamic call of a static reference acts as a static call to the referenced function. A dynamic call of a virtual reference is translated to a virtual call, and therefore it uses a dynamic dispatch.

```
class FunctionStaticReference(
    val symbol: Symbol.Function,
): Expression()

class FunctionVirtualReference(
    val symbol: Symbol.Function.Signature,
): Expression()
```

Listing 3.34: FIR function reference expressions

BIR does not have a dynamic call nor function references, so the back end needs to lower them to other constructs. Since a function reference returns a value, it must produce some object. This object encodes the called function and the call type. This information is then decoded at the place of the dynamic call. The back end implements this lowering similarly to the Kotlin compiler – by using anonymous classes with a special method.

The back end generates a special reference class for each used combination of functions and reference types. The function reference is then converted to the `CreateObject` expression of that reference class. The reference class has one special method called `invoke`. This method performs the actual function call

and returns its result. The type of used function call depends on the reference type for which the class was generated. The dynamic call can then be replaced with a virtual call to the `invoke` method. The virtual call uses the function value as the dispatch receiver. In other words, the called function and reference type is encoded in the object type, and a virtual call is used to decode it back. A simple example is shown in the listing (3.35).

```
// FIR
Function("a") {
    DynamicCall({ FunctionStaticReference("b") })
}
Function("b")

// BIR
Function("a") {
    VirtualCall(
        "[invoke]",
        dispatchReceiver = { CreateObject("[static-reference-b]") }
    )
}
Function("b")
Class("[static-reference-b]")
Function("[invoke]", dispatchReceiverType = "[static-reference-b]")
↪ {
    Return { StaticCall("b") }
}
```

Listing 3.35: Conversion of dynamic call

The previous example does not involve function arguments and a dispatch receiver. Both must be forwarded from the dynamic call to the actual call inside the `invoke` method. The function arguments are dealt with easily because they can be directly passed as arguments of the `invoke` method. The problem is with the dispatch receiver. It cannot be directly passed because the `invoke` call already has a different dispatch receiver. Instead, the back end stores the original dispatch receiver in a special field `this` inside the function value. Since the function value is passed as the `invoke` dispatch receiver, the `this` field can be therefore accessed inside the method and forwarded.

3.4.8 Environments

Environments are the most complex feature in FIR. They are only present in FIR and can be used to implement several seemingly unrelated features from Kotlin. They implement extension functions, local functions, and inner classes. Environments are very flexible and can even be used to implement context receivers once they are added to Kotlin [14].

Both functions and classes have their own environment with slightly different semantics. Also, both functions and classes can capture environments. A function environment contains all local variables (including parameters) and the function dispatch receiver. A class environment contains all fields of that class. All environments also contain environments captured by the function/class.

The captured environments are explicitly declared in the function/class declaration property `capturedEnvironments`. Additionally, classes inherit captured environments from their supertypes. By capturing an environment, a function can access the environment content. In some sense, captured environments are implicit parameters. Classes themselves cannot interact with a captured environment. Instead, they store it in a special field so it can be accessed later by the class methods.

These are the operations that capture environments: class instantiation, static and virtual function calls, and all function references.[†] These operations cannot be performed without access to the captured environments. Otherwise, the environments could not be captured. Environments that are currently accessible are also called active. The only active environments at any moment are the following ones:[‡]

- The environment of the currently executed function.
- Environments captured by the currently executed function.
- Environments provided by the `ExtendEnvironment` expression.

What active environments are captured is determined by their type. Each declaration can capture multiple environments, but only one of each type. In

[†]Therefore, function references immediately capture environments, and no environments are captured during the dynamic call.

[‡]The dispatch receiver is not added to the active environments. This is important to note because it must be explicitly added during the conversion of some Kotlin features.

the same way, there can be only one active environment of each type. Therefore, there is no ambiguity about how to capture the environment.

The `ExtendEnvironment` expression (listing (3.36)) adds a new active environment. The environment type is determined by the `symbol` property. The activated environment is the evaluation result of the `environment` expression. The environment is active only inside the `body` expression. The `environment` expression is always evaluated once, no matter how many times the environment is later accessed.

```
class ExtendEnvironment(
    val symbol: Symbol.Environment,
    val environment: Expression,
    val body: Statement,
): Expression()
```

Listing 3.36: FIR ExtendEnvironment expression

It is possible to nest multiple `ExtendEnvironment` expressions to activate several environments. However, if there is already an active environment with the same type, the previous environment is shadowed and deactivated. Therefore, the new environment has a priority. The `ExtendEnvironment` expressions can also shadow environments activated for the other reasons from the previous list. The list is ordered by the environment's priority (the first rule having the lowest priority). For example, the environment of the currently executed function is always shadowed by any other environment of the same type.[†]

A captured environment content can be accessed as other object fields using expressions `GetField` and `SetField`. In fact, an environment is just an object. In the case of classes, the environment is the class instance, so it can also be used as a dispatch receiver. A function environment is an object of a special class, and therefore it does not have an additional meaning.

A function environment also stores the function dispatch receiver that can be accessed similarly to a regular dispatch receiver. That means using the `This` expression (listing (3.37)). `This` in FIR has an optional `environment` property. This property specifies from which environment to obtain the dispatch receiver.

[†]In practice, shadowing happens only for class environments because there is currently no use case for shadowing function environments.

If the property is `null`, then there is no difference to the `This` expression from BIR. In other words, `null` means this function environment.

```
class This(val environment: Environment?): EnvironmentParent()
```

Listing 3.37: FIR This expression

The captured environment object can be obtained by the `Environment` expression (listing (3.38)). The expression has a property `symbol` that specifies the type of the accessed environment. Both `This` and `Environment` expressions can be used as an environment's parent. Only active environments can be accessed without providing a parent. Providing a parent makes it possible to access captured environments and a dispatched receiver from the parent's environment. So, for example, a function can access a class captured by its dispatch receiver.

```
sealed class EnvironmentParent: Expression()

class Environment(
    val symbol: Symbol.Environment,
    val parent: EnvironmentParent?,
): EnvironmentParent()
```

Listing 3.38: FIR Environment expression

Conceptually, the conversion of environments to BIR is relatively simple. On the other hand, the implementation is very complex because this feature interacts with several other features. As a result, there are many combinations to cover, which means many edge cases. There are several relatively isolated constructs that the back end needs to transform, each described below.

Captured functions body A function environment is a special object created at the beginning of the function and stored in a special variable. All function parameters are stored in the object fields under the same name. The dispatch receiver is also saved in the object in an additional field. The back end replaces all local variable reads and writes with reads and writes to fields of this object. As a result, the function no longer directly uses any local variables. Everything is in the environment object.

Function declarations of capturing functions Captured environments are passed to the capturing functions by extra arguments. Therefore, the back end adds additional parameters to each capturing function. Each captured environment has its own parameter. These parameters are also potentially stored in the function environment object.

Function calls of capturing functions These calls must be adjusted accordingly to pass the captured environments. Since each environment has its unique type, it is easy to decide which environment goes into which argument. The back end obtains the environments by performing implicit parent-less environment access. In other words, it works as if the environment was accessed by the `Environment` expression without a parent.

Function references of capturing functions The implementation of function references is probably the most complex step. Conceptually, the captured environments use a similar idea as the referenced function dispatch receiver. They are all stored in the function reference object when the object is created. The `invoke` method then extracts the captured environments from the object and passes them to the function. In contrast to the dispatch receiver, the captured environments are stored there the whole time (and not just before the `invoke` call). There is also another (equivalent) way to look at the implementation. The function reference class can be declared to capture the same environments as the referenced function. Instantiating the reference class performs the capturing automatically. The `invoke` method can then access the captured environments from its dispatch receiver.

Instantiation of capturing classes Classes store the captured environments in fields. These fields must be initialized during the class instantiation. A good place to do this initialization would be in a class constructor. The problem is that FIR does not have any notion of constructors. Objects are created by the expression `CreateObject`, which does not support direct initialization of fields. However, constructors are just functions. Therefore, the back end can generate a special constructor with a parameter for each captured environment. The constructor creates the object using the `CreateObject` expression, stores all its parameters in the object, and finally returns the object. The back end

then replaces all original `CreateObject` expressions with a static call to this constructor.

Environment access Environments are always stored in some variable or field. So any environment access can be replaced either with the `GetVariable` or `GetField` expression. Environments have parents, and therefore they form a hierarchy. Each parent stores the child's environments in its fields. So an environment access with a parent is transformed into several nested location reads. The back end needs to determine which field to access in each environment. This task is relatively easy because there is a predetermined field naming scheme based on the environment type. The only complicated environment to resolve is the active one (the first accessed). The active environment can be stored in multiple places depending on how it was activated. For this reason, the back end needs to track how environments were activated.

Environment extension The environment extension must be stored in a local variable because it must be evaluated exactly once. Each environment extension must be stored in a separate variable with a unique name to support the environment shadowing. The back end needs to remember which variable stores which environment. Otherwise, it would not be possible to access it later.

3.5 Translation from Kotlin IR to FIR

This section explains how the front end translates the Kotlin IR into FIR. Most of the translation is straightforward because FIR was designed with that goal in mind. FIR preserves a significant part of Kotlin features and their semantics, so these can be directly translated. Features that are not directly supported by FIR are easily lowered to supported features.

At least that is the theory. In practice, the actual implementation still has some complexity because it needs to extract the needed information from the Kotlin IR. For this reason, the second chapter (2) went into great detail about the Kotlin IR. The combined knowledge from the second chapter and this chapter is enough to implement a significant part of the front end. This section

intentionally omits the description of this part since it would be just a repetition of what was already explained. The lowering of the features missing from FIR is covered in the rest of this section.

3.5.1 Locations

Variables in FIR do not have initializers. Instead, the initialization is expressed as a regular assignment right after the variable declaration. In the case of global variables, the assignment must be wrapped in the `Init` declaration. Otherwise, it would not be performed at the program start.

FIR also does not explicitly support properties. The front end converts them the same way as the Kotlin back end. The backing field is converted to a class field (or a global variable). The two accessors are represented as regular functions with special names. The conversion of delegated properties is also taken from the Kotlin back end.[†] The same strategy is also used for local delegated properties.

Arrays are the last missing feature related to locations. Kotlin treats arrays as regular objects implemented in the Kotlin standard library. Therefore, it is the standard library that actually implements them. Since the implementation is in native code, there is not much the front end can do. Instead, the implementation is done in the runtime module. The runtime uses the `GetDynamicField` and `SetDynamicField` expressions to implement the get and set operations of arrays. Additionally, the runtime provides a similar implementation for array iterators. Implementing just these methods is sufficient for a sound and precise approximation of the actual array implementation.[‡]

3.5.2 Control flow

The representation of both loops and branching in FIR is slightly different from the Kotlin IR. Kotlin has separate `while` and `do-while` loops. Additionally, the FIR `loop` does not have a stopping condition. The condition can be represented by `When` with two branches inserted into the loop body. The first branch contains the loop condition and has an empty body. The second branch is an

[†]How the translation looks like was already shown in (2.7.6).

[‡]Assuming the remaining methods are handled the same way as the rest of the standard library.

else branch and contains `Break`. Therefore, if the condition is satisfied, the loop continues because nothing happens. Otherwise, the loop ends. The `while` loop has `When` at the beginning of its body and `do-while` at the end.

The `When` expression can be directly converted with one modification. The front end must add an empty else branch to each non-exhaustive `When` to ensure it is exhaustive. The created else branch is a regular branch with a constant `true` as its condition.

The Kotlin IR has one control-flow-related feature that cannot be directly expressed in FIR. This feature is the non-local return of inline functions.[†] The analysis can safely ignore the performance aspects of function inlining, but it must support the non-local returns. The straightforward solution is to perform the function inlining. The concept of function inlining is not that complicated, but the implementation is. Nevertheless, there is another solution that is easier to implement.

The idea is to use the same strategy utilized in the back end to convert the `finally` block. In other words, to implement the non-local return by an exception. First, the front end creates a unique exception.[‡] All non-local returns are then replaced with throwing this exception. Before the exception is thrown, the returned value is stored in the exception field. The front end wraps the inlined function call in a `Try` block with a handler for that exception. Finally, the handler uses a local return to return the value stored in the exception field.

3.5.3 Functions

The Kotlin IR uses one class for both static and virtual calls. During the call translation, the front end needs to decide which type of function call to use. A safe way is to use a static call for every function that is not an override and has the `final` modality. Additionally, the static call is used for super calls. The virtual call is used in all other instances. The front end also needs to resolve function overloading, which is not supported by FIR. The front end does that by name mangling, meaning it adds the parameter types to the function signature. Therefore, each overloaded function gets its unique signature.

[†]In the future versions of Kotlin, the list might be extended by non-local breaks and continues [15]. In that case, these constructs can be implemented almost identically as non-local returns.

[‡]Each inline function call must have its own exception to prevent collisions.

FIR intentionally does not have default arguments because of their associated complexity. Values of default arguments cannot be just inserted in the function call. The problem is that the expression must be performed in the function owner context, not in the caller context.

The workaround is to create a wrapping function with the same dispatch receiver type as the called function. The wrapping function internally calls the originally called function. As a result, the default argument expressions are executed in the correct context. The original call is replaced with a call to this wrapping function. The original arguments are passed as parameters of the wrapping function. The missing arguments are provided by the wrapping function by evaluating the corresponding default argument expressions. Each missing argument is stored in a variable before it is passed to the original function.[†]

FIR also does not support vararg parameters for a similar reason as with the default arguments. Instead, a varagr parameter is implemented as an array. It is up to the caller to create this array and put all vararg arguments into it. The spread operator complicates this process because the array inside the operator does not have a known size. The runtime solves this problem by providing an artificial method that adds all elements from one array to another array.[‡] The caller then can use this method to move all the elements from the spread operator into the vararg array.

Extension functions are an example of a feature that FIR accounts for but does not directly support. Extension functions (and similarly extension properties) are implemented by capturing the environment of the extension receiver. The semantics of captured environments solves almost all problems of extension functions. The front end needs to make only two modifications. The call site must be adjusted by wrapping the function call in the `ExtendEnvironment` expression.[††] The other necessary modification is in the extension function body. The front end must replace all accesses to the extension receiver with the `Environment` expression.

[†]This step is necessary to allow the default argument expressions to access the values of previous arguments.

[‡]The standard library does not have such a method for arrays because they have a fixed size. However, this is not a problem for the runtime. The runtime does not have to consider array indices so that it can store all elements in a single dynamic field. Therefore, the runtime provides arrays with effectively infinite size.

[††]The extension receiver is put into the `ExtendEnvironment` expression as the environment.

The remaining feature discussed in this subsection is string interpolation. String interpolation is technically a syntax sugar that hides several function calls. A string interpolation joints together several values that are converted to a string. The front-end implementation does precisely this by calling functions from the Kotlin standard library.[†] The only difficulty comes from making sure that the calls are chained in the correct order.

3.5.4 Classes

The absence of constructors in FIR is the main difference when it comes to supporting classes. The front end represents constructors as regular functions. This converted constructor initializes the object, but it cannot create it directly.[‡] The initialized object is passed as a dispatch receiver, so the constructor function is technically a method. However, it is always called statically. The dispatch receiver must be provided by the constructor caller. It is either `This` in the case of a delegating constructor call or `CreateObject` for the initial constructor call.

Several constructs participate in the object initialization: a call to another constructor, a secondary constructor body, an init block, and a property initializer. As per the previous paragraph, the delegated constructor call is directly replaced with a function call. The secondary constructor body is left intact, but primary constructor body is changed. The `IrInstanceInitializerCall` is replaced with code from all init blocks and property initializers. The initialization code is merged in the order of declaration.

Kotlin has many special types of classes. All of them are implemented as regular classes in FIR. However, two types need additional handling. These are the object classes and enum classes.

The object class needs an associated global variable that stores the single object of this class. Access to the object class is converted to a read from this global variable. The variable is initialized in the `Init` declaration. The initialization is split into two steps. In the first step, the object is created by the `CreateObject` expression and stored in the variable. In the second step, the

[†] The Kotlin back end does the conversion in a similar but more optimized way.

[‡] The reason is that there are two different situations in which a constructor is called. It can be called with the intent to create the object by `IrConstructorCall`. The other option is that it was called from another constructor by `IrDelegatingConstructorCall`. Since both types of calls are replaced with a regular function call, it is not possible to distinguish these two cases. The problem is that the object must be created only in the first case.

object is initialized by calling its default constructor. This two-step initialization is important because the constructor code can access the object class before it is fully initialized.

The translation of enum classes is based on the same idea as for object classes. The difference is that there are more global variables, one for each enum entry. The front end also needs to generate the enum class synthetic methods. The `values` method is generated such that it returns an array with all the enum entries. Implementing the `valueOf` method can be simplified because the interpreter will not preserve the passed entry name. Therefore, a sound and the best possible approximation is to return a value that represents all the enum entries simultaneously.

3.5.5 Function references

Function references are implemented partially by the front end and partially by the runtime. The runtime provides a special function reference class that implements all the `FunctionN` interfaces.[†] The class implements many `invoke` methods, one for each possible number of parameters. The method implementation uses the `DynamicCall` expression to call the function value. The referenced function value is stored in a field. The method passes all its arguments to the `DynamicCall` and returns the call result.

A code that references a function does so by instantiating the class and storing the function reference in its field. Whether to use a static or virtual function reference is determined by the same rules as for calling the function directly. The advantage of this approach is that the `invoke` call site does not have to be changed.

Additional steps are required to support dispatch receivers properly. There are two situations that need to be dealt with separately. If the dispatch receiver is captured immediately, it is stored in an additional field of the function reference class. The `invoke` method uses the value of this field as the dispatch receiver for the inner call.

The other option is that the dispatch receiver is provided at the `invoke` call site. In this case, the dispatch receiver is passed as a regular argument. However, there is no way to distinguish the dispatch receiver from a regular

[†]Alternatively, it can be implemented as a separate class per interface.

argument at the call site. This information is present only when the function is referenced. There are two different implementations of the function reference class to solve this problem. The first implementation works as presented above and is used in the previous case. The second implementation solves the latter case. It uses its first parameter as a dispatch receiver, and only the remaining parameters are passed as regular arguments. Since the information is encoded during instantiation, it is no longer required at the `invoke` call site.

The implementation of property references works almost identical to the function references. The primary difference is that the property reference class needs to store up to two functions instead of one.[†] The referenced accessors are called in the two inherited methods (`get` and `set`). Both classes also extend the function reference classes and make the `invoke` method an alias for the `get` method.

3.5.6 Local declarations

All declarations inside a function utilize the environment feature. They capture the environment of the directly enclosing function. In contrast to extension functions, the call site does not have to be adjusted. However, the local declaration body conversion is more complicated. The front end needs to replace all accesses to the captured variables. Each such access is substituted by access to a corresponding field in the correct environment. Determining the correct environment can be difficult. The reason is that local declarations can be nested. Therefore, it might be necessary to inspect the declaration hierarchy and construct the environment access accordingly.

Inner classes are similar to local classes, and they are similarly implemented. In this case, the captured environment belongs to the enclosing class. The instantiation of an inner class does not have to be further modified. The access to the outer class object is replaced as in the implementation of extension functions. The only difference is that the replacement is performed in all declarations inside the inner class body.

[†]The same class can be used for both `val` and `var` properties. This is possible because the Kotlin compiler ensures that the setter is not called if it does not exist.

Chapter 4

Implementation

In the introduction of the first chapter (1), I have already discussed my very first steps while working on this thesis. However, the real work really started afterward, and this chapter focuses on that. In this chapter, I will describe my approach to creating this thesis. Subsequently, I will explain how I have tested both the design and the created prototype. The rest of the chapter is dedicated to the challenges I have encountered and had to solve as well as the mistakes I have made.

4.1 The implementation procedure

I have used several software engineering techniques to deal with the complexity and size of this project. This section describes these techniques and how they helped me. The used techniques can be split into two categories: project management and software development.

4.1.1 Project management

Even though this project was done by a single-member team, project management was critical to its success. Without project management, it would be impossible to prioritize and react to problems correctly. The project had a strictly fixed deadline and resources but only a partially fixed scope. The goals of this

project put restrictions on the scope, but they had some flexibility. I have prioritized the goals in the following way (from the most important to the least important):

- the static analysis design
- this document
- the prototype

I have set the primary goal to be the static analysis design. As a result, the design had to be excellent and complete. Therefore, there was little to no change possible to the scope of this goal. The rest of this project was driven by the primary goal, so its scope was adapted to ensure it would be met. The role of project management was to constantly monitor the progress and adjust this flexible part of the scope.[†] However, putting enough effort into this document was also fairly important. Having a good design without proper documentation has little value.

In order to continuously adjust the scope, it was necessary to monitor the progress and estimate the remaining work. At the beginning of the project, I did this evaluation once a week. The frequency slowly increased as the project went on, and till the end, it was once per day. Having some data is critical for the monitoring and estimating process. For this reason, I have kept track of how much time each task took to complete. Additionally, I have used a TODO list to track all remaining tasks. I have also used my bachelor's thesis timesheets as a reference point for the estimates.

Estimating how much time is remaining (in terms of hours) was relatively easy. I estimated the actual number with a margin of error of approximately 5 % two months before the project completion. Correctly estimating the tasks was significantly more difficult. Especially since implementing the interpreter module required much more time than expected. Errors in these time estimates posed a danger to the project because the timeline was relatively strict. There was a risk that some less important tasks would take so long that there would be no time left for the more important ones. Therefore, correct and frequent prioritization was the other job of project management.

[†]This part of the scope was the only possible thing to change since everything else was fixed.

4.1.2 Software development

I have approached the project in an iterative manner and created the design incrementally. The process involved many experiments, including a lot of trial and error. In some sense, it was closer to research than software development. In the beginning, I had an approximate idea of how to structure the architecture modules. However, the overall design emerged gradually through the development. I have used the prototype to guide the design process. In other words, the primary purpose of the prototype was to make experiments and discoveries.

For the project management to work reasonably, the development speed must be predictable and sustainable. To ensure these two properties are achieved, I have used an approach called TDD [7]. The core idea behind TDD is to write tests before the actual implementation (and also to work in small steps). I have used this approach from the project's start to finish. As a result, the prototype has many tests covering almost all functionality. With the TDD, my development process looked something like this:

- First, I have decided what feature to implement.
- Then, I have written acceptance tests for that feature.[†]
- Since the feature was not implemented yet, those tests were failing at this point.
- Subsequently, I have localized in what module the feature must be implemented.
- Sometimes, the feature had to be implemented in multiple modules. In that case, I did the following three steps separately for each module, one module after another.
- In the selected module, I have implemented more low-level tests (typically unit tests).
- Only now, I started with the actual implementation.
- When the implementation was done with all tests passing, I moved to another module.

[†]The different types of tests will be explained in the following section.

- The acceptance should pass once the implementation is done in every module. If not, it is necessary to start debugging and writing unit tests to cover this bug.[†]
- As the last step, I went back to each module to decide if to invest more time into refactoring or not.

The above process hides one crucial concept: The use of so-called *tracer bullets*. This idea is described in the book "The Pragmatic Programmer" [11]. It is about incrementally implementing small incomplete features but always having something that works as intended (the explanation is oversimplified). In order words, it says that each feature should be implemented in the whole system at once (in this case, in all modules). The idea is that development is guided by features and not project structure.[‡] This approach has at least two advantages. It allows using the acceptance tests since they test the whole system at once. Additionally, it makes the incremental design evolution easier by reducing the opportunities to make mistakes. The mistakes that this approach mitigates are those that result from an incomplete understanding of the problem at the given time.

There were occasions when I had to deviate slightly from the above process because it made the development easier. In the beginning, I wrote a large number of acceptance tests for many features without actually implementing them. There were two reasons for that. First, I needed to find a good way to write them.[††] The second reason is that I needed a high-level list of all Kotlin features to make the initial development plan.

The other notable instance where I did things differently is related to the interpreter module.[‡‡] The problem with the interpreter module is that its features interact with each other. Therefore, it is hard to implement them separately. For this reason, I have decided first to implement a fake interpreter. The fake interpreter could do only things necessary for the acceptance tests to pass and nothing more. Therefore, the fake interpreter was a subset of the real

[†]The bug's presence means that the unit tests did not initially cover everything necessary.

[‡]The opposite extreme is implementing a whole module and only then starting the implementation of another one.

[††]It is not possible to decide if some syntax is good or not without writing many tests. If I have to write many tests, why not use them later.

[‡‡]This approach was technically according to the *tracer bullets* idea, but it differs from the procedure I have described.

interpreter. The real interpreter was implemented all at once, but only after the BIR was stable enough.

4.2 Testing

Almost all the prototype testing was done using automated tests.[†] As was mentioned in the previous section, I have used the TDD approach. Therefore, the tests were primarily created gradually together with the implementation. As a result, I have not run into that many bugs during the development. Once a feature was implemented, it usually worked as intended. For this reason, I rarely had to use a debugger, only for challenging bugs.

I would not say that the prototype does not have any bugs. There is no way of knowing if the automated tests cover everything, and I am aware of many not adequately tested features. The problem is not with testing individual features; there are not that many of them. The problem is a combination of two or even more features. For example, an overridden extension method combines two individual features: extension properties and method overriding.

Just because two features are tested separately does not mean that their combination is also tested. In fact, my experience was that these combined features are the hardest to implement and test. There are so many combinations to check that it becomes hard to keep track of what is and is not tested. At the same time, not every combination needs extra handling from the code. Therefore, many newly created tests passed even though nothing new was implemented. Having tests that pass automatically is not good because it is hard to ensure they actually work.[‡] In summary, these tests take more time and add less value per test than other types of tests. However, not writing these tests is also not a good option. I estimate that around 20 % of these tests have found actual bugs and missing features. As a result, these are good exploratory tests. Their downside is that they significantly increase the number of written tests.

The project contains three different types of tests: acceptance tests, integration tests, and unit tests. Each has its advantages and disadvantages, so they are

[†]This type of project would be practically impossible to implement without automated tests.

[‡]The test might contain a bug and pass even when the tested feature does not work.

used accordingly. These terms do not have a precise meaning. For this reason, I will explain how these tests are used in this project and how they work.

The acceptance tests are used to test the whole project together. Their advantage is that they are easy to write and understand. Also, they are the only tests that verify the interactions of multiple modules. They have several downsides, so other types of tests are also necessary. First, they cannot test everything as they only compare the analysis results with the list of expected exceptions. Additionally, they are not that good at explaining why some feature is not working because the problem can be anywhere. A significant downside also is that they are the slowest tests in terms of the execution speed.

The listing (4.1) shows an example of the first written acceptance test. Each test is written as a file that contains a regular Kotlin code. The tests use a `Throws` annotation to tell which functions should be analyzed. This annotation contains a list of expected exceptions. A test passes if the analysis reports that the annotated functions can throw exactly the exceptions from the annotation.

```
@Throws(E::class)
fun main() {
    throw E()
}
```

Listing 4.1: An example of an acceptance test

The integration and unit tests are used to test individual modules. They are faster than acceptance tests, and they can also better isolate the tested behavior. Their downside is that they cannot detect problems caused by the incorrectly used API of another module. This type of problem occurs if the test is designed incorrectly and does not follow the API semantics. Consequently, the implementation is also incorrect even though all tests pass.

Tests from each module look slightly different because each module utilizes a different DSL. I have created these DSLs to make the tests more maintainable. Since each module is different, its DSL must also be different. Not every module in the project has a DSL because some modules have only a few tests. Modules that have a DSL are the following: front end, back end, and interpreter.

The front-end module uses tests like the one in the listing (4.2). This test consists of a name, a string with Kotlin code, and an expected result. The expected result is declared in the body of `matches`, representing the FIR

generated by the front end. The tests use the Kotlin compiler to convert the Kotlin code to the Kotlin IR. Since the front end is a compiler plugin, it is called as part of the compilation process.

```
test("variable declaration", """
    fun foo() {
        var i = 1
    }
""") matches {
    Function("foo") {
        LocalVariable("i", Type.Int)
        SetLocalVariable("i") { Constant(1) }
    }
}
```

Listing 4.2: An example of a front end integration test

The front-end module is the only module that uses the integration tests. From a certain point of view, the integration tests are just like unit tests in other modules. I have made this distinction because there is one important but hidden difference. These tests use the Kotlin compiler, so they actually test the integration of two modules. Most of the time, the difference is only in the test speed.[†] However, the test also tests the Kotlin compiler as a side effect. Therefore, it may happen (and it did) that a test does not work because of the Kotlin compiler and not the front-end module.

The back-end unit tests are very similar in structure to the front-end tests. The difference is that the back-end tests convert FIR to BIR instead of Kotlin code to FIR. The unit tests in the back end reuse part of the DSL that constructs FIR elements. Additionally, they use another DSL builder for BIR elements. The listing (4.3) shows an example of a back-end test.

The tests in the interpreter module once again follow the same idea. The tests reuse the DSL builder for BIR to declare the test. Compared to the previous two modules, declaring the expected answer is more straightforward. The interpreter's goal is to analyze exceptions, and therefore the tested answer is a list of exceptions.

The interpreter has two different modes. Both modes are necessary to test under the same conditions. However, these two modes produce different

[†]Running the Kotlin compiler adds a noticeable overhead compared to the tests from other modules.

```
test("variable declaration") {
    Function("a") {
        LocalVariable("i", Type.Int)
        SetLocalVariable("i") { Constant(1) }
    }
} matches {
    Function("a", variables = {
        Variable("i", Type.Int)
    }) {
        SetVariable("i") { CreateObject(Type.Int) }
    }
}
```

Listing 4.3: An example of a back end unit test

answers in many scenarios. Therefore, one test cannot be used for both modes (at least not in every case). Writing a different set of tests for each mode means duplicating much code – the tests will differ only in the expected answer. The solution is to declare the test once but with two different answers. The DSL then creates two separate tests, one for each mode. An example of such a test is shown in the listing (4.4).

```
test("variable reassignment") {
    SetVariable("a") { CreateObject("E1") }
    SetVariable("a") { CreateObject("E2") }
    Throw { GetVariable("a") }
} expect {
    overwrite toThrow "E2"
    unification toThrow listOf("E1", "E2")
}
```

Listing 4.4: An example of an interpreter unit test

The two modes do not always produce different results. Therefore, the DSL provides a simpler syntax for those cases to save some typing. An example of this syntax is shown in the listing (4.5). Even though the expected answer is declared only once, the test is still run separately for each mode.

```
test("throw") {
    Throw { CreateObject("E") }
} expect "E"
```

Listing 4.5: Simplified syntax of interpreter unit tests

4.3 Challenges and mistakes

Challenges and mistakes have an interesting relation. Often one leads to the other and vice versa. Making a mistake in the form of a wrong decision can significantly complicate things and, as a result, creates a challenge. On the other hand, it is easy to make costly mistakes while dealing with a difficult challenge. Both situations occurred many times on this project. I wrote down the ones that I either consider interesting or important to avoid in the future development of this project.

Many of my mistakes resulted from my faulty assumptions about how the Kotlin compiler works. These mistakes caused the analysis to interpret the semantics of some Kotlin features incorrectly. I have explicitly documented the correct semantics in the second chapter so as not to repeat these mistakes.[†]

4.3.1 Incorrectly tested exception flow

This mistake is related to the fake implementation of the interpreter module. This fake implementation was used for a large part of the prototype development. Most of the acceptance tests were created during that time. From the beginning, I intended to implement the real interpreter later and just swap the implementations. In theory, exchanging the implementations should not affect the tests in any way. However, the deployment of the real implementation broke most of the tests.

The new implementation was correct; the problem was with the tests. More precisely, the problem was that the fake interpreter did not correctly analyze the exception flow. I was aware of this limitation. What I was not aware of was the fact that many tests depended on this incorrect behavior of the interpreter.

[†]However, given my experience with this project, I would say that the documentation is still not entirely correct.

Since the tests passed until that point, it never occurred to me that they were wrong from the beginning.

The problem with the tests is demonstrated by the listing (4.6). This example test verifies that the analysis correctly merges exceptions produced by two different function calls. However, the test is written in such a way that the function `b` is never called since `a` always throws an exception. The real implementation of the interpreter follows the control flow, including the exception flow. Therefore, it detects that the second function call is unreachable and reports that only the first exception is thrown. As a result, all tests that tested more than one exception had to be changed to throw each exception only in some execution paths.

```
// Actual: @Throws(E1::class)
@Throws(E1::class, E2::class)
fun main() {
    a()
    b()
}

fun a() {
    throw E1()
}

fun b() {
    throw E2()
}
```

Listing 4.6: Acceptance test that was broken by the exchange of interpreter implementations

4.3.2 Null pointer dereferences

The final interpreter design terminates the current execution path if it dereferences a `Nothing` value. However, this was not always the case. For a long time, I thought that the analysis could not reach such a state if the dereference were not present in the actual program. For this reason, I have decided not to implement the execution path termination. Then I accidentally discovered a test where this can happen and where it affects the analysis precision.

The test was relatively complicated, but it can be reduced to the version shown in the listing (4.7). If this was an actual program, the handler should

observe an object of type `Y` in the variable `x`. The reason is that the exception can be thrown only after the first iteration of the loop. Therefore, the variable `x` is set to the value of `y`, which can be only `Y`.

The original implementation incorrectly reported that the value of `x` can be either `X` or `Y`. The problem was in the first iteration of the loop, specifically in the upper branch. The variable `y` is `Nothing` during that branch execution. As a result, the original implementation skipped the `Throw` because there was no exception to throw. However, the interpretation continued in this now-empty branch. So from the interpreter's point of view, two paths reached the end of the loop body.[†] One path changed the value of `x`, and the other did not. The exception is correctly thrown in the second iteration, but the variable `x` already contained the incorrect value. This problem is solved by terminating the upper branch path when it tries to throw the `Nothing` value.

```
SetVariable("x") { CreateObject("X") }
Try({
    Loop {
        When {
            // if (y != null)
            Branch {
                // null check call is omitted
                Throw { GetVariable("y") }
            }
            // else
            Branch {
                SetVariable("y") { CreateObject("Y") }
                SetVariable("x") { GetVariable("y") }
            }
        }
    }
}) {
    Handler("_", "Y") {
        // Should be "Y" but was "X" and "Y"
        GetVariable("x")
    }
}
```

Listing 4.7: A test that is not analyzed properly without execution path termination after null pointer dereference

[†]This problem took me way more time to figure out than it should have.

4.3.3 Implementation of Any

The interpreter implementation turned out to be significantly more difficult than I had expected. There was one particularly bad decision that contributed to the implementation difficulty. It was to implement the `Any` value properly. I have also made another less impactful but still bad decision to save time and not implement custom garbage collection.[†]

There are multiple ways to implement the `Any`. The most straightforward one is to create an object for every existing class and merge them into a single value. These objects would have to be marked so that access to their fields would also return `Any`. This solution could have been implemented in an hour. However, this solution has several problems related to the analysis performance. Each new `Any` would cause an allocation of thousands of objects. Additionally, accessing the fields of this value would be equivalent to thousands of operations. A virtual call has a similar effect because it calls one method for each type.

The other option is implementing the `Any` as a special type of object. The `Any` is represented only by a single class instance in this implementation. This solution still has the problem with virtual calls, but it solves allocations and field access overhead. So, in theory, the performance should be much better.

My original estimate was that this implementation would be slightly more complicated but still manageable. Additionally, I thought it would be a fundamental part of the `Any` design. Both assumptions were completely wrong. It took me around a week to implement, and the result depends on the remaining implementation of the interpreter.[‡] Therefore, it is an implementation detail, not describable without an understanding of the remaining implementation.

Overall, this mistake was an exemplary case of premature optimization. For the prototype's purpose, the performance difference does not matter. This is especially true since the interpreter is tested on very small programs with a few classes. Also, I am not sure there would be a difference even in real programs. Without proper performance testing, it is impossible to determine precisely. Most of the time, the type coercion would reduce the thousand of objects only to a few. The problem might be more pronounced for analysis

[†]At least, I thought that I would save time.

[‡]The implementation involved solving surprisingly many edge cases. It also made the remaining implementation of other features harder.

of dynamic languages without type information. However, this was not the intended use case of the prototype.

4.3.4 Implementation of garbage collection

The interpreter internally allocates Kotlin objects to represent the analysis objects. These objects are stored in a map, which holds the object together with its fields. As a result, the Kotlin garbage collector will not deallocate these objects even if they are no longer used by the analysis. Therefore, it is up to the interpreter to ensure that the objects are deallocated.

There are two main options for how to do the deallocation. The easiest one (which I have used) is to put the objects in a `WeakHashMap`. This map holds its keys as weak references allowing the Kotlin garbage collector to deallocate them. The other way is to implement a custom garbage collection. Making a custom garbage collection is not that difficult because even the simplest tracing algorithm would be sufficient.

However, I wanted to save time and opted for the first solution. The first solution has one well-hidden problem. The `WeakHashMap` does not guarantee that all its keys are reachable objects. Therefore, its content can change at any moment because of the garbage collector. This instability is a problem for several algorithms used in the interpreter.

One of them is the algorithm that unifies objects created in the loop. Because of the map instability, the algorithm cannot just unify all new objects in the map. The map could contain no longer accessible objects that would also be unified. In some cases, the unification would alter the analysis output, but only sometimes – depending on the Kotlin garbage collector. As a consequence, the analysis would not behave deterministically.

For this reason, several algorithms need to be implemented in a different way which involves more work. The implementation of these algorithms is very similar to that of the custom garbage collector. In some sense, my initial decision forced me to make multiple implementations of the custom garbage collector instead of one.

4.3.5 Virtual and dynamic dispatch

The original interpreter and BIR did not support virtual dispatch. Instead, it was represented by a dynamic dispatch. In some sense, these two dispatches are equivalent because one can be used to implement the other. The dynamic dispatch was implemented using a VMT generated by the back end. In other words, each object carried a reference to its class object. The class object stored the methods in fields. Therefore, a method call was translated into two field accesses and a dynamic call.

The VMT implementation turned out to be worse than the current solution in all aspects. It took way more time to implement and introduced a runtime overhead. It also made the interpreter more complicated because the interpreter had to distinguish two types of values: objects and functions.

Overall, I did not initially correctly assess what the differences in implementations would look like. Taking back the decision did not cost almost any time, but it meant discarding a lot of no longer necessary code. Therefore, making the right choice at the beginning would save some time.

4.3.6 Changes in FIR and BIR

Both FIR and BIR have changed multiple times during the development. Most of these changes did not cost that much time compared to the previously stated problems. I have expected that there would be some changes because this is the nature of the iterative process. I am making a list of these changes to document the dead ends that I have already explored. The following list contains only the most notable changes, but there were many other ones that had little impact.

FIR loops Initially, loops in FIR were implemented the same as in the Kotlin IR: with conditions and two different types of loops. Since the BIR loop was the same as now, the back end was performing the conversion. I later decided to remove this feature and move the loop conversion from the back end to the front end. The reason was that this feature had not added any significant value to FIR. At the same time, it complicated the front end more than the actual conversion. The problem was mainly with testing because the condition was present in every loop even though it was unnecessary for the test. This decision has one downside: it forces the other front ends to implement the

conversion also. However, the conversion is trivial, so the advantages outweigh the disadvantages.

Function references The original design had only one type of function reference, which is equivalent to the static reference.[†] The reason was that I did not know that virtual reference is necessary. I have incorrectly assumed that function references are only static. This mistake is an excellent example of the types of problems I have revealed through exploratory testing.

Environments The following feature was technically never implemented but was considered as part of the initial design. In the initial design, environment access was transitive, and the `Environment` expression did not have a parent. Instead, the back end searched the environment hierarchy to construct the access automatically. The new design moves this responsibility to the front end. The original design has two problems related to the fact that multiple environments of the same type might be active at once. Therefore, the back end would have to implement the environment precedence rules to match the Kotlin semantics. The first problem is that this requires much work to do correctly. The second problem is that front ends for languages with different semantics would be hard to implement.

Function overloading The initial design of functions supported the function overloading. It was implemented by including parameter types as a list in the function signature (next to the signature name). The signature was later changed to encode the parameters directly in its name. The reason was similar to loops: it made testing easier.

Type information The analysis originally did not use any non-essential type information. Therefore, the type information was not preserved in the intermediate representations. The type information was introduced together with `Any` as it was suddenly very useful.

[†]The current prototype still has only this static reference.

Chapter 5

Evaluation

This chapter summarizes the results of this thesis. The designed static analysis supports all Kotlin features necessary for the analysis's intended use case.[†] The analysis precision is also sufficient, and the created prototype implements most of the static analysis.

The prototype is great for experimenting with the analysis but has several problems that make it unsuitable for production use. These problems are caused by the fact that the prototype was never meant to be used in production. However, the ultimate goal of this thesis was to have a real-world benefit. I still want to achieve this goal, so I intend to continue working on this project. Therefore, understanding these problems and why they exist will help eliminate them.

There is a long way from the prototype phase to the real-world deployment. Just finishing the prototype will not be enough to make it a good product. For this reason, I have proposed several necessary improvements for the analysis design as well as its implementation. Those improvements and some other less significant ones are explained in the second section. Before that, I will describe the current state of the prototype.

5.1 Assessment of the prototype

The prototype was created to explore the static analysis's possible designs and verify the design decisions. I would not be able to make this design without it.

[†]At least to my knowledge. Given the complexity of Kotlin, it is hard to know for sure.

The prototype was created with this purpose in mind. On the other hand, the prototype was not meant to be an actual usable product. Namely, the prototype does not implement all features and does not handle correctly every edge case.[†] Additionally, the prototype implementation is not always as clean as I would like.

The implementation quality of each module varies greatly. The reason is that it was worth investing more time in maintaining the more complex modules. I estimated a return on investment of each refactoring to decide what to refactor and when. The more complicated the module is, the harder it is to implement if the existing implementation is not good. On the other hand, a simple code that is isolated from the rest might be better to ignore in the short term. For example, the most complex module is the interpreter, and therefore it is in a relatively good shape. Another example is the front-end module which is relatively simple but has changed frequently. Therefore, the code in this module is not that good as the constant refactoring was not justified.

Even though the implemented project is meant as a prototype, it is not meant to be entirely discarded.[‡] I have designed the prototype such that it would be possible to reuse some of its parts, especially the tests. Also, the prototype is split into many relatively isolated modules and has extensive test coverage. Therefore, it should be possible to refactor the problematic parts.

Writing tests represented a significant part of the development. In fact, the acceptance tests represent around 27 % of the all code in the project, and the remaining tests are approximately 43 %. In total, all tests combined constitute 70 % of the code. An additional 10 % of the code is in tooling that makes the development easier (by simplifying tasks like testing and debugging). Therefore, only 20 % of the code is the actual implementation.

Currently, the prototype has around 1,750 tests. Of these tests, there are approximately 300 acceptance tests. Since the interpreter is the most complex module, it is also the most tested with 1,000 unit tests. However, not all tests in the interpreter are unique, as was explained in (4.2). I do not know how many unique tests are in the whole project, but my estimate is around 1,000.

[†]Especially edge cases that arise from interactions of multiple features and the inner workings of the Kotlin compiler.

[‡]The code of prototypes like this is usually not used for subsequent development.

The number of tests is not excessive. Many of these tests actually found bugs and prevented regression. Also, I am aware of features that are still not tested thoroughly enough. Therefore, more tests will be required in the future.

The prototype does not implement the complete analysis, and there are two reasons for that. The primary one is that it was not necessary to finish the design. During the design phase, I needed the prototype for exploration. What is possible to explore is always limited by what the prototype currently supports. For example, it is impossible to experiment with function references if the prototype does not implement regular function calls. The prototype is now in a state that allows experimenting with all the missing features separately. Therefore, these features do not have to be fully implemented to discover how they should be handled.

The second reason the prototype is incomplete is that it is impossible to do everything in the given time frame. The project is large, and even if it supported all the features, it would still not be ready for production use.[†] I implemented all I could in the available time, prioritizing the fundamental features.

The core of the analysis is almost fully implemented. The only thing missing is the support for recursion.[‡] For this reason, a recursive call causes the analysis to crash because it runs out of memory. Other than that, the whole interpreter and BIR are implemented correctly. The back end is also almost done and can convert all currently implemented FIR features. The FIR is missing some features because they are not used by the front end yet.

Most of the missing features come from the front end. Analyzing these features will usually result in a crash, occasionally an incorrect output. It is hard to state what features are not correctly implemented. What works and what does not can be determined by looking at the acceptance tests. There are many *TODO* comments in the project that explain what might be implemented incorrectly. Additionally, a more high-level list of the missing features is in the file "TODO.md". Some of the more important missing features are:

- access to transitively captured variables (but directly captured variables work)
- inner classes

[†]For reference, the project has approximately 30,000 lines of code, not counting empty lines.

[‡]The recursion is almost done since the core algorithm can be reused from loops.

- virtual function references (only static references are supported)
- property references
- delegated properties

The prototype also does not support most of the Kotlin standard library. This limitation is not directly addressed in the analysis design, so technically, it is not a missing feature. However, it is important to remember this fact when writing acceptance tests and trying the analysis in general.[†]

5.2 Suggestions for future improvements

This section contains some of my ideas on how to proceed with the project implementation. The theme of this section is to make the analysis usable in practice. Therefore, each suggestion simultaneously describes the potential weaknesses of the analysis.

The most important suggestion is simple: finish the analysis implementation according to the design. All the remaining suggestions can be split into four different categories depending on what they try to improve. These categories are usability, maintainability, performance, and precision. Each category has a dedicated subsection that contains a curated list of ideas from that category. However, many more things need to be done or would be nice to do.

5.2.1 Usability

The primary goal of usability improvements is to motivate developers to use the analysis. There are two different ways to achieve that: improve the added value of the analysis or make it easier to use. One of the biggest problems right now is that the initial analysis setup would take too much work. This problem is addressed by the first two suggestions. The remaining two suggestions focus on new features of the analysis, thus improving the added value.

[†]For example, a mistake that I have made several times was using exceptions from the standard library. These exceptions are not explicitly declared, and therefore they do not have a proper inheritance hierarchy. The missing declaration causes a problem when the exception is passed into a function that expects its supertype. Because of the missing inheritance hierarchy, the type coercion discards the exception.

Kotlin standard library As of now, the Kotlin standard library is handled as a closed-source library. Therefore, developers/users need to write manual FIR declarations for each non-trivial function they use. These declarations can be shared between different projects, but that is not a good solution on its own. It is necessary to generate the FIR declarations like for other libraries – from the Kotlin source code. These declarations then can be put in the runtime module, so this process would not have to be repeated on each project.

The standard library is open source, so the code is available. The problem is that not everything is implemented in Kotlin. There is one trick to mitigate this problem. Analyze the standard library on all platforms, especially Kotlin native, since that is built from scratch in Kotlin. This approach does not solve everything because the native standard library does not contain everything the JVM standard library does. However, having the declarations for a part of the library is still better than nothing.

Gradle plugin The analysis currently requires a complex configuration. For example, it requires paths to all modules with source code, and paths for output files. However, the most complicated step is to register the compiler plugin. Manually running the analysis is not possible, not only because it should be used on the CI server. Therefore, the users will have to write some script to orchestrate the analysis. Writing such a script involves work and requires some knowledge about the internal implementation of the analysis.

A much better solution is to use some build automation tool, for example, Gradle [10]. The solution involves creating a Gradle plugin that wraps up the analysis. The users will only have to add the plugin to their project and maybe do some simple configuration. The Gradle plugin will then perform the appropriate configuration and orchestration of the analysis.

Warnings Given how the analysis works, it can be used to detect several potential problems. The analysis can report these problems as additional warnings. Having these warnings would be helpful because they can uncover bugs in the code. Additionally, they can warn the developers about places where the analysis loses precision or even soundness. Adding these warnings is not very difficult because the analysis already knows about them; it just does not report them. Here are a few examples of problems that could be reported:

- Unreachable handlers – handlers that handle an exception that can never be thrown
- Dereference of the `Nothing` value – could warn about potential null pointer dereference
- Loop without terminating path – will cause the program not to terminate
- Non-analyzable constructs – operations that the analysis knows cause loss of soundness (for example, call to unknown function)

Exception propagation path The analysis knows where each exception originated and how it propagated through the program. Therefore, it can present this information to the user. Having this information is very useful for several use cases. For example, the API documentation could link to the source code where the exception is thrown. This link acts as additional documentation about the exception. The reason is that the code around the exception also explains what the exception means.

5.2.2 Maintainability

Maintainability is about ensuring that the project development continues at a stable pace. This aspect of development becomes more pronounced as the project grows and gets more complicated. The prototype is already a relatively large project and will get much larger before being ready for the production use. Therefore, the following suggestions focus on making the future development easier.

Refactoring As mentioned in the previous section, the current code quality of some modules is not high. This problem should be addressed relatively soon. However, it is not necessary to refactor everything at once. Instead, I would suggest refactoring continuously along with code changes.

Logging This idea addresses a problem with debugging acceptance tests. Sometimes it is difficult to determine why an acceptance test does not work. Using a debugger is possible but takes too much time. The problem with the debugger is that acceptance tests contain much code. Therefore, stepping

through the interpreter is a long process. The proposed solution is to add logging to the interpreter to record all the performed operations. From my experience, this log will be enough to determine the cause of most problems.

Unused declarations The debugging of acceptance tests is difficult because of another related problem. Since there is much code, it is hard to inspect the generated BIR. The only reasonable way to improve the situation is to reduce the amount of code. Most of the code in acceptance tests is there because of the automatically included utility code and code from the runtime module.[†] However, no test actually uses all the included code at once. Therefore, the solution is to remove functions that are not (transitively) called from any analyzed function.

5.2.3 Performance

This subsection contains some proposals for improving the analysis performance. It is important to note that I have not done any performance testing of the analysis. Therefore, I do not know if the prototype does not contain any bottlenecks caused by non-optimal implementation. Also, I am not sure if the performance needs to be improved at all. For this reason, these proposals focus on the analysis design and ideas that are not implementation-dependent. These ideas might be worth considering if the analysis performance will have to be improved. However, this consideration should be guided by performance testing and profiling.

Parallelization Modern computers (and especially servers) have many processor cores. Utilizing more of them can sometimes dramatically improve performance. The analysis has several places that can be easily parallelized. The first one is in the last phase of the analysis module. In this phase, the analysis runs the interpreter for each endpoint separately. Therefore, this phase can be completely parallelized.

However, only parallelizing this phase will not overall have that significant impact. The problem is in the second warm-up phase, which also calls all the endpoints (to perform the initialization). This phase evaluates all the endpoints

[†]This included code is important because the tests frequently use it. Without the included code, the test would have to duplicate it.

in a single interpreter run, so the previous solution is insufficient. Nevertheless, even this phase can be parallelized. The interpreter evaluates each branch in complete isolation. Therefore, the evaluation of any `When` is also parallelizable.

Deduplication A single value in the interpreter can contain multiple objects of the same type. Sometimes these objects are not possible to access separately in any way. In that case, the program cannot tell how many objects there are in the value. From the program's point of view, the value acts as if it contained only one object. However, there are still multiple objects from the interpreter's point of view. As a result, the interpreter performs more work when reading and writing to these objects.

Improving the performance can be done by unifying these objects. The unification comes without the loss of precision or soundness (even in the overwrite mode). The interpreter can detect if the unification can be done, but it requires an additional calculation. Both the cost of the calculation and the saved time by the optimization depend on the analyzed program. Therefore, it is impossible to determine the optimization benefits without trying it on real-world programs.

BIR optimizations The BIR generated by the back end contains many expressions that do not affect the interpretation in any way. In other words, they do not have any observable side effects, and their evaluation value is not used. These expressions are often a result of the conversion from FIR to BIR. For example, the conversion of conditions in `When` moves the conditions in the branch bodies. If the condition expression does not have any side effects, then it is not necessary to evaluate it. However, the interpreter does not recognize that and therefore evaluates the condition.

Overhead from this evaluation can be significant, for example, if the expression calls a function that does not have a side effect. The solution is to implement simple optimizations similar to the optimizations done by compilers. The most useful optimizations are function inlining, constant propagation, and dead code elimination.

5.2.4 Precision

The remaining suggestions are those that improve the analysis precision. The analysis focuses on high precision, so I did consider including these improvements in the current design. However, in all these cases, I have decided against that. The reason is that all these suggestions are hard to implement and bring unclear benefits. Therefore, it is necessary first to evaluate if their implementation will add enough value to justify the cost. This evaluation needs to be done on real-world projects as with the performance improvements. I could not make this evaluation yet, since running the analysis on actual projects is currently impossible.

Arrays All array accesses in the current design are converted into access to a single location. The reason is that the interpreter does not support primitive values. It is possible to improve this approximation for accesses that use constant indices. This improvement does not require the interpreter to support any new features as it can be implemented in the back end. The idea is that the array will have one shared location and then additional locations for constant indices. Writing to the shared location will have to write to all locations. Writing to a specific location will also have to write to the shared location but not to the other specific locations. However, the back end needs to implement all the optimizations mentioned in the previous subsection for this idea to work. The reason is that currently, all array accesses are hidden behind a function call. Therefore, without the optimizations, the indices are never constant.

Unification The unification mode can have a significant negative impact on precision. The interpreter cannot always avoid using the unification mode, but recovering some of that lost precision is possible. The unification mode must be used to ensure the analysis termination. The unification mode generally cannot be mixed with the overwrite mode, otherwise it would lose soundness. However, there are some instances where the soundness is not affected. In those instances, the interpreter can carefully use the overwrite mode. An example of such a situation is a newly created object inside a loop. This object can safely use the overwrite mode until the end of the loop iteration in which it was created.

The other improvement of the unification mode is specific to the object unification algorithm. The algorithm's goal is to set a finite limit on how many

objects can be created in a loop. Right now, the algorithm does that by unifying all new objects of the same type. The algorithm is built on the idea that there are only finitely many classes. However, there are also finitely many `CreateObject` expressions inside each loop. So the algorithm can use this fact instead.

This new algorithm is more precise for loops that create the same type of object using multiple `CreateObject` expressions. The problem with implementing this strategy is that the `CreateObject` expressions are currently indistinguishable. To make them distinguishable, they would, for example, have to contain some ID.

Generics At the moment, generics are implemented by erasure. In other words, the analysis ignores them. Therefore, they do not contribute to the optional type information.

There are two options for how to utilize the type information of generics. The first one is easier since it can be implemented in the front end, but it is less precise. The idea is to replace each generic type with its upper bound type. This approach utilizes only part of the generics since the actual generic types of each object are still erased.

The second option improves that but requires a support from FIR and the back end. In this version, the back end creates many child classes for each generic class. Specifically, it creates one child class for each used combination of generic types – similarly to how C++ templates work. Each object is then instantiated from one of these child classes depending on its generic types. The difficulty of this approach is to correctly create the inheritance hierarchy to match the generic variance rules.

Conclusion

The primary goal of this thesis was to design a static analysis that tracks exception propagation. The intended use case for the analysis is to automatically document domain exceptions in APIs of back ends written in Kotlin. Using this analysis will simplify exception handling in the back ends, and therefore it will save development time and money.

The intended use case puts several requirements on the analysis. The most important ones are that the analysis must support all Kotlin features and, at the same time, have high precision. High precision is necessary, especially for frequently used features like dynamic dispatch and lambda functions.

I researched several existing solutions for the static analysis of exception propagation. However, I did not find any solution that would satisfy the requirements mentioned above. The problem is that all those solutions focus either on purely object-oriented languages or purely functional programming languages. None of the considered solutions combine these two aspects – which is necessary to analyze Kotlin programs. Therefore, I based my design only partially on the existing research. Instead, I primarily relied on general concepts of static analysis.

The designed static analysis uses an approach called abstract interpretation. The analysis utilizes the Kotlin compiler to parse the source code into Kotlin IR. This Kotlin IR is gradually converted into two custom intermediate representations. The first intermediate representation is more high-level, and its primary purpose is to make the design more modular. The second intermediate representation is used to perform the abstract interpretation.

In order to create the design, I had to understand the Kotlin semantics well. Since the analysis uses Kotlin IR, I also had to learn how it works. These two things are related, and I documented them in this thesis.

As part of the analysis design process, I implemented a prototype which was the secondary goal of this thesis. I used the prototype to experiment with different design decisions and ensure the design's correctness. The created prototype has extensive test coverage that ensures the prototype's correctness. The prototype implements most of the design but not all of it. It also has some additional limitations, and therefore it cannot be used in a production environment. I proposed solutions to these limitations as part of the suggested future improvements. In summary, all goals of this thesis were met.

The completion of this project represented a great challenge for me. The project was challenging because of its size, complexity, and time constraints. I learned many new things while working on the project, not only about Kotlin. My vision for this project was (and still is) that it would become a practically usable tool. The prototype is not there yet, but I intend to continue improving it until it is ready.

Bibliography

1. AKHIN, Marat; BELYAEV, Mikhail et al. *Kotlin language specification: Kotlin/Core*. JetBrains / JetBrains Research, 2020.
2. BUSE, Raymond P.L.; WEIMER, Westley R. Automatic Documentation Inference for Exceptions. In: New York, NY, USA: Association for Computing Machinery, 2008. ISBN 9781605580500. Available from DOI: 10.1145/1390630.1390664.
3. CHANG, Byeong-Mo; CHOI, Kwanghoon. A review on exception analysis. *Information and Software Technology*. 2016, vol. 77, pp. 1–16. ISSN 0950-5849. Available from DOI: https://doi.org/10.1016/j.infsof.2016.05.003.
4. DOLNÍK, Filip; MÁLEK, Václav. *NI-APR term project*. 2021. Semestral project. Faculty of Information Technology, Czech Technical University in Prague.
5. ELIZAROV, Roman. *Kotlin and exceptions* [online]. Medium, 2020 [visited on 2022-05-04]. Available from: https://elizarov.medium.com/kotlin-and-exceptions-8062f589d07.
6. EVANS, Eric. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321125215.
7. FOWLER, Martin. *Bliki: Testdrivendevelopment* [online]. 2005. [visited on 2022-05-04]. Available from: https://martinfowler.com/bliki/TestDrivenDevelopment.html.

8. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN 0201633612.

9. GOOGLE. *KSP* [comp. software]. 2022. [visited on 2022-05-04]. Available from: `https://github.com/google/ksp`.

10. GRADLE. *Gradle* [comp. software]. 2022. [visited on 2022-05-04]. Available from: `https://gradle.org`.

11. HUNT, Andrew; THOMAS, David. *The pragmatic programmer: From journey to mastery*. Addison-Wesley, 2020.

12. IOZZELLI, Yuri. *Solving the structured control flow problem once and for all* [online]. leaningtech, 2019 [visited on 2022-05-04]. Available from: `https://medium.com/leaningtech/solving-the-structured-control-flow-problem-once-and-for-all-5123117b1ee2`.

13. JETBRAINS. *Calling Kotlin from Java* [online]. 2022. [visited on 2022-05-04]. Available from: `https://kotlinlang.org/docs/java-to-kotlin-interop.html`.

14. JETBRAINS. *Kotlin – Context receivers* [online]. 2022. [visited on 2022-05-04]. Available from: `https://github.com/Kotlin/KEEP/blob/master/proposals/context-receivers.md`.

15. JETBRAINS. *Kotlin – Support non-local break and continue* [online]. 2022. [visited on 2022-05-04]. Available from: `https://youtrack.jetbrains.com/issue/KT-1436`.

16. JETBRAINS. *Kotlin docs: Kotlin* [online]. 2022. [visited on 2022-05-04]. Available from: `https://kotlinlang.org/docs/home.html`.

17. JETBRAINS. *Kotlin evolution: Kotlin* [online]. 2022. [visited on 2022-05-04]. Available from: `https://kotlinlang.org/docs/kotlin-evolution.html`.

18. JETBRAINS. *Kotlin FAQ* [online]. 2022. [visited on 2022-05-04]. Available from: `https://kotlinlang.org/docs/faq.html#is-kotlin-compatible-with-the-java-programming-language`.

19. JETBRAINS. *Kotlin language specification – Scopes and identifiers* [online]. 2022. [visited on 2022-05-04]. Available from: `https://kotlinlang.org/spec/scopes-and-identifiers.html#scopes-and-identifiers`.

20. JETBRAINS. *Kotlin Programming Language* [comp. software]. 2022. [visited on 2022-05-04]. Available from: https://github.com/JetBrains/kotlin.

21. JETBRAINS. *What is Kotlin* [online]. 2022. [visited on 2022-05-04]. Available from: https://kotlinlang.org/docs/faq.html#what-is-kotlin.

22. JO, Jang-Wu; CHANG, Byeong-Mo; YI, Kwangkeun; CHOE, Kwang-Moo. An uncaught exception analysis for Java. *Journal of Systems and Software*. 2004, vol. 72, no. 1, pp. 59–69. ISSN 0164-1212. Available from DOI: https://doi.org/10.1016/S0164-1212(03)00057-8.

23. LEROY, Xavier; PESSAUX, François. Type-Based Analysis of Uncaught Exceptions. 2000, vol. 22, no. 2. ISSN 0164-0925. Available from DOI: 10.1145/349214.349230.

24. MEYER, Bertrand. *Soundness and completeness: With precision* [online]. 2019. [visited on 2022-05-04]. Available from: https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext.

25. MØLLER, Anders; SCHWARTZBACH, Michael I. *Static Program Analysis*. Department of Computer Science, Aarhus University, 2020. Available also from: https://cs.au.dk/~amoeller/spa.

26. ORACLE. *Java language and Virtual Machine Specifications* [online]. 2022. [visited on 2022-05-04]. Available from: https://docs.oracle.com/javase/specs.

27. ORACLE. *Java releases* [online]. 2022. [visited on 2022-05-04]. Available from: https://www.java.com/releases.

28. ORACLE. *What's new in JDK 8* [online]. [visited on 2022-05-04]. Available from: https://www.oracle.com/java/technologies/javase/8-whats-new.html.

29. OW2. *ASM* [comp. software]. 2022. [visited on 2022-05-04]. Available from: https://asm.ow2.io.

30. PALERMO, Jeffrey. *The Onion Architecture : part 1* [online]. 2008. [visited on 2022-05-04]. Available from: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1.

31. RICE, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society* [online]. 1953, vol. 74, no. 2, pp. 358–366 [visited on 2022-05-04]. ISSN 00029947. Available from: http://www.jstor.org/stable/1990888.

32. SCHUCHORT, Thilo. *Kotlin Compile Testing* [comp. software]. 2022. [visited on 2022-05-04]. Available from: https://github.com/tschuchortdev/kotlin-compile-testing.

33. VMWARE. *Spring* [comp. software]. 2022. [visited on 2022-05-04]. Available from: https://spring.io.

34. YI, Kwangkeun. An abstract interpretation for estimating uncaught exceptions in Standard ML programs. *Science of Computer Programming*. 1998, vol. 31, no. 1, pp. 147–173. ISSN 0167-6423. Available from DOI: https://doi.org/10.1016/S0167-6423(96)00044-5. Selected Papers of the First International Static Analysis Symposium.

Appendix A

Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
BIR	Back end Intermediate Representation
CFG	Control-Flow Graph
CI	Continuous Integration
DDD	Domain-Driven Design
DSL	Domain-Specific Language
DTO	Data Transfer Object
FIR	Front end Intermediate Representation
ID	Identifier
IR	Intermediate Representation
JVM	Java Virtual Machine
KSP	Kotlin Symbol Processing
TDD	Test-Driven Development
UX	User Experience
VMT	Virtual Method Table

Appendix B

Glossary

domain exception	An exception that is expected as a result of a domain rule violation.
single-purpose domain exception	A domain exception that can be produced by only one type of domain rule violation.
analysis soundness	The analysis presented in this thesis is sound if it reports all possible exceptions a function can throw. The precise meaning depends on the specific analysis.
analysis correctness	Complementary attribute to the analysis soundness. The analysis presented in this thesis is complete if it reports only possible exceptions a function can throw.
analysis precision	A relative expression of how many mistakes the analysis makes.
front end	In the compiler context, it is the compiler part that analyzes a source code and converts it to IR. In this static analysis context, it is the module that converts the Kotlin IR to FIR.
back end	In the compiler context, it is the compiler part that generates a native code from IR. In this static analysis context, it is the module that converts the FIR into BIR.

intermediate representation	A data structure used in compilers and this static analysis to represent the processed program.
Kotlin intermediate representation	IR used in the Kotlin compiler.
front end intermediate representation	A custom higher-level IR similar to the Kotlin IR and used mainly by the analysis front end and back end.
back end intermediate representation	A custom lower-level IR used mainly by the analysis back end and interpreter.
runtime	In the context of Kotlin language, it is a set of libraries that provide some special functionality to the running program. In this static analysis context, it is a module that provides FIR declarations for a subset of the Kotlin runtime and standard library.
interpreter	The module of this static analysis that performs the abstract interpretation.

Appendix C

Contents of the enclosed CD

- `MT_Dolnik_Filip_2022.pdf` ..... the thesis in PDF
- `readme.md` ..... description of the CD content
- `src`
  - `prototype` ..... the prototype source code
  - `thesis` ..... the thesis source code in LaTeX