UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Análisis estático de programas .NET

Tesis presentada para optar al título de Doctor de la
Universidad de Buenos Aires en el área de Ciencias de la Computación

Lic. Edgardo Julio Zoppi

Director de Tesis:	Dr. Diego Garbervetsky
Consejero de Estudios:	Dr. Víctor Braberman
Lugar de Trabajo:	Laboratorio de Fundamentos y Herramientas para la Ingeniería de Software (LaFHIS), FCEyN, UBA
Fecha de Defensa:	22 de mayo de 2019

Buenos Aires, 2019

Análisis estático de programas .NET

Resumen

En esta tesis presentamos el diseño e implementación de una amplia gama de análisis estáticos para la plataforma .NET, con foco en la escalabilidad. Nos concentramos en .NET dada su gran popularidad en la industria y el amplio conjunto de características que provee, pertenecientes a los paradigmas orientado a objetos y funcional, incluyendo programación concurrente y la manipulación de bajo nivel de punteros. La combinación de todas estas características hacen del análisis estático un desafío.

Por un lado, presentamos un *framework* de análisis estático distribuido de programa completo, diseñado para escalar con el tamaño de la entrada. Nuestro enfoque está basado en el modelo de programación con actores para ser ejecutado en la nube. Nuestra decisión de utilizar una red de computadoras en la nube provee un grado de elasticidad para recursos de CPU, memoria y almacenamiento. Para demostrar el potencial de nuestra técnica, mostramos cómo puede ser implementado un análisis de *call graph* típico en una configuración distribuida. Además, extendemos nuestro análisis para soportar actualizaciones incrementales del código fuente y mostramos cómo los resultados computados previamente pueden ser actualizados sin tener que volver a calcularlos de cero.

Por otro lado, presentamos un *framework* de análisis estático de programas y herramientas específicamente diseñados para la plataforma .NET. Este *framework* provee muchas funcionalidades, incluyendo algunas representaciones intermedias como el código de tres direcciones, adecuado para la implementación de un análisis estático, así como también provee una amplia gama de análisis y transformaciones como son la inferencia de tipos, los análisis de *control-flow* y *data-flow*, y la construcción de *call graph* y *points-to graph*, entre otros. No sabemos de ningún otro *framework* de análisis estático de código públicamente disponible para la comuni-

dad .NET que provea este tipo de funcionalidades. Para demostrar las capacidades de nuestro *framework*, presentamos también algunas aplicaciones cliente que aprovechan sus funcionalidades, como un análisis de optimización de consultas *Big Data* para detectar automáticamente columnas no utilizadas y dependencias entre tablas de entrada y salida de operadores definidos por el usuario desarrollados en algún lenguaje de la plataforma .NET como C#.

Palabras clave: análisis, programa, estático, distribuido, incremental, call graph, .NET, framework, bytecode, código de tres direcciones, control-flow, data-flow, nube, big data.

Static analysis of .NET programs

Abstract

In this thesis we present the design and implementation of a wide range of static analyses for the .NET platform, with focus in scalability. We target .NET given its popularity in the industry and the rich set of features it provides, ranging from object-oriented to functional paradigms, including concurrent programming and low-level pointer manipulation. The combination of all these features make static analysis very challenging.

On the one hand, we present a distributed, whole-program static analysis framework that is designed to scale with the size of the input. Our approach is based on the actor programming model and is deployed in the cloud. Our reliance on a cloud cluster provides a degree of elasticity for CPU, memory and storage resources. To demonstrate the potential of our technique, we show how a typical call graph analysis can be implemented in a distributed setting. In addition, we extend our analysis to support incremental source code updates and show how the previously computed results can be updated without having to recompute them from scratch.

On the other hand, we present a static program analysis framework and tools specifically designed for the .NET platform. It provides many features, including a few intermediate code representations such as a three-address code suitable for implementing a static analysis on top of it, and a rich set of analyses and transformations such as type inference, control-flow and data-flow analyses, and call graph and points-to graph construction, among others. We don't know of any other static analysis framework publicly available to the .NET community providing these kind of features. To demonstrate the capabilities of our framework, we also present a few client applications that take advantage of its features, such as a Big Data query optimization analysis to automatically detect unused columns and

dependencies between input and output tables of user-defined operators written in a .NET-based programming language like C#.

Keywords: analysis, program, static, distributed, incremental, call graph, .NET, framework, bytecode, three-address code, control-flow, data-flow, cloud, big data.

Contents

Part I

Prologue

Chapter 1

Introduction

Program analysis [88, 67] is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety or liveness. It often focuses on two major areas: program optimization and program correctness. The former focuses on improving the program's performance while reducing the resource usage. The latter focuses on ensuring that the program does what it is supposed to do.

Program analysis is used by several software engineering tools, including compilers, integrated development environments (IDEs), static checkers and verifiers and code style analyzers, as well as many others [67]. Among its typical uses we can mention code analysis, transformation and optimization, type inference, correctness verification, security assurance and program understanding [67].

Program analysis can be performed without executing the program (static program analysis), during run-time (dynamic program analysis) or in a combination of both [88, 67].

Static analysis [88] is the exhaustive evaluation of an application by examining some static representation of the program, typically the source code, without executing it. It infers at compile-time, properties about the run-time behavior of programs and allows to verify, for instance, the absence of run-time errors or security breaches.

Dynamic analysis [88] is the testing and evaluation of an application during run-time. It is performed by executing the program and can use run-time knowledge to increase the precision of the analysis. In order to be effective, the target program must be executed with sufficient test inputs to produce interesting behaviors.

Many software defects that cause errors can be detected both dynamically and statically. The two approaches are complementary because no single approach can find every error. A static analysis can ensure that a property of interest holds on every possible execution of the program. In contrast, even that dynamic analysis can play a role in correctness assurance, its primary goal is finding and debugging errors since it can only analyze a single execution of the program at a time. However, it can reveal subtle defects or vulnerabilities whose cause is too complex to be discovered by static analysis.

An analysis which depends on information only available at run-time is inherently dynamic [67]. If some amount of imprecision can be tolerated, it may be possible to perform an *approximate* static analysis instead. In order to be sound, static analyses often over-aproximate their results. To avoid false positives, precise algorithms and abstractions are required. There are various design dimensions to fine tune the precision of a static analysis [67].

A whole-program static analysis considers the entire program and admits different assumptions. The closed-world assumption assumes that all relevant information is already known and available to be used by the analysis, in contrast to the open-world assumption that assumes new unknown information can potentially arise in the future.

An intra-procedural static analysis restricts its scope by only considering method bodies in isolation. An inter-procedural static analysis concerns about the interaction between the methods of a program.

Additionally, a static analysis can be classified regarding its sensitivity, which affects its overall precision [67]. Context-sensitivity considers the calling context when analyzing the target of a method call. Object-sensitivity distinguishes objects by considering, for instance, their allocation site. Field-sensitivity distinguishes two fields of the same object. Flow-sensitivity considers the ordering of statements inside a method body. Ideally, a highly precise analysis combines all them, but at the cost of compromising scalability.

In this thesis we present the design and implementation of a wide range of static program analyses for the .NET platform [23]. Our goal is twofold: (i) design techniques that can scale to larger programs and (ii) provide a toolchain that allow researchers and developers to implement their own analyses of .NET programs.

For this reason, in the first part of this thesis we present the design and implementation of a typical inter-procedural call graph static analysis from the literature, but distributed in the cloud with support for incremental source code updates. In this context, our goal is to enable the analysis to scale for very large real-world industrial applications in terms of both memory consumption and analysis time.

In the second part of this thesis we present a static program analysis framework specifically designed for analyzing .NET programs, and a few clients applications that take advantage of its features. In this case, our goal is to provide to the .NET community the tools needed to develop a static analysis targeting .NET programs, in a similar manner as they are available in other comparable platforms such as Java.

1.1 Motivation

We focus on the static analysis of .NET programs for many reasons. First of all, the .NET platform [23] comprises a very mature ecosystem of software development resources and it is widely used in the software industry in general since it was first released in 2002 [9, 5, 20]. While it was originally designed by Microsoft for developing applications for the Windows operating system, during the last couple of years its popularity increased [19, 12] by adopting an open-source philosophy [10] and supporting many other platforms, including operating systems such as Linux, macOS and Android, as well as multiple architectures such as x86, x64 and ARM. This makes the .NET ecosystem competitive with other similar cross-platform systems like Java, as well as an attractive alternative option to be considered and eventually adopted by the educative and academic research community.

Currently, it provides a rich set of interesting features, including multiple programming languages belonging to a mix of object-oriented, functional and concurrent programming paradigms [11]. While C# is the language par excellence for developing .NET applications, Visual Basic counts with a long trajectory in the industry from even before the creation of .NET and is very popular because of its simpler syntax. F# is a relatively new language that brings the benefits of the functional programming paradigm to the industry, unlike other similar languages such as Haskell that are well-known by the academic and research community but rarely used to develop real-world industrial applications [5, 20]. The .NET ecosystem also provides many modern development tools, including Visual Studio, a full-fledged IDE, and Visual Studio Code, a sophisticated text editor. Finally, a wealth of open-source and proprietary libraries and frameworks exist and are publicly available for the .NET community.

The combination of all this features makes static analysis very challenging. However, only a poor offer of program analysis related tools targeting the .NET platform are available [84, 8, 16, 15], and the difficulty of analyzing .NET programs with the few existing ones, puts in evidence an arising necessity of providing better tools. The analysis of programs developed in similar platforms like Java has a long research tradition and many papers and tools already exist for this purpose [105, 70, 18]. On the contrary, the .NET platform have not received much attention from the static analysis community yet. For this reason, our goal is to aid in bringing to the software engineer industry in general, and to the .NET community in particular, the benefits of the program analysis techniques by providing the means to develop scalable static analysis tools that target the .NET platform.

1.2 Introducing .NET

.NET [23] is a free, cross-platform, open-source developer framework for building many different types of applications. .NET was first released in 2002 and since then it provides multiple programming languages, libraries, editors and tools to build for web, mobile, desktop, gaming and Internet of Things (IoT) systems.

Common language infrastructure. Often abbreviated to CLI, is an open specification (i.e., technical standard) developed by Microsoft and standardized by ISO and ECMA [14] that describes executable code and a runtime environment that allows multiple high-level programming languages to be used to develop applications on and for different computer platforms and architectures (i.e., platform agnostic). The .NET Framework, .NET Core and Mono are the most popular implementations of the CLI. Among other things, it describes the following aspects.

Common type system. Often abbreviated to CTS, is a standard that specifies how type definitions and specific values are represented in computer memory. It is intended to allow programs written in different .NET based programming languages to easily share information. A type can be described as a definition of a set of values (e.g., integer numbers), and the allowable operations on those values (e.g., addition and subtraction). It also provides an object-oriented model that supports parametric types, type inheritance, virtual methods and object lifetime. The CTS supports two general categories of types: value types and reference types; and a mechanism to convert one to the other and vice versa, called *boxing* and *unboxing* respectively.

Metadata. Refers to how types are described and how these descriptions are stored. Information about program structure is language-agnostic, so it can be referenced between languages and tools, making it easy to work with code originally written in different programming languages. Metadata is also used to support reflection. .NET programs are physically stored in units called *assemblies* that contains modules, namespaces, type definitions and references to other assemblies. Each type defines its hierarchy information and members like fields, properties and methods. Method bodies contain an array of local variables, a sequence of bytecode instructions and a table with information about exception handlers.

Common intermediate language. Often abbreviated to CIL and previously known as Microsoft Intermediate Language (MSIL), is an object-oriented stack-based assembly language that is abstracted from any particular platform hardware.

All CLI-compatible programming languages compile to CIL. Its bytecode is translated into native code or directly executed by a virtual machine.

Common language specification. Often abbreviated to CLS, is a set of rules intended to promote language interoperability. These rules shall be followed in order to conform to the CLS. Conformance is a characteristic of types that are generated for execution on a CLI implementation. These rules apply only to types that are visible in assemblies other than those in which they are defined, and to the members that are accessible outside the assembly.

Virtual execution system. Often abbreviated to VES, refers to how code is executed and how objects are allocated, interact with each other, and automatically deallocated by the garbage collector. It loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at run-time. When the code is executed, the platform-specific VES will compile the CIL to the machine language according to the host specific hardware and operating system. This process is called just-in-time compilation (JIT). Ahead-of-time compilation (AOT) may also be used, which eliminates this step, but at the cost of loosing portability.

Common language runtime. Often abbreviated to CLR, is a virtual machine that implements the VES and manages the execution of .NET programs. It provides run-time services including memory management, type safety, exception handling, garbage collection, security and thread management.

1.3 Comparison with Java

All .NET programming languages and Java compile into bytecode. However, they use different instructions and virtual machines. Nonetheless, Java bytecode and .NET CIL share strong similarities, being both stack-based object-oriented intermediate languages where objects are stored and shared in the heap; both use an evaluation stack to store temporary values and an array of local variables standing

for original source code variables; both are object-oriented, with instructions for object creation, field access and virtual method dispatch.

Despite these undeniable similarities, both bytecodes differ in the way of performing parameter passing (CIL uses a dedicated array of variables for the formal parameters, while Java bytecode merges them into the array of local variables); they handle object creation differently (CIL creates and initializes an object with a single instruction, while these are distinct operations in Java bytecode); they allocate memory slots differently (in CIL each value uses a single slot, while in Java bytecode depends on the size of values, 32-bit values use one slot while 64-bit values use two subsequent slots); finally, CIL uses pointers explicitly, also in type-unsafe ways (i.e., allowing free pointer operations), while Java bytecode has no notion of pointer at all.

Value types and reference types. Java distinguish between primitive types and objects. Primitive types are passed by value and can be allocated in the stack[1] like `int` and `float`. Objects are always allocated in the heap and passed by reference. Instead, CIL has the notion of *value types* and *reference types*. Value types are like Java's primitive types, but they also include `structs` and `enums`, giving the possibility to developers to create their own new types that behave like primitive types. Reference types are also objects like in Java. However, CIL also allows passing a value type by reference instead of just by value, using a dummy wrapper object to contain the value. This mechanism is called *boxing* and is transparent to the user of high-level .NET based languages like C#. Java overcomes the lack of this feature by explicitly defining wrapper classes for all primitive types, like `Integer` for `int` and `Float` for `float`. In the same way, both C# and Java support returning value types by reference.

Generic types. CIL keeps information about generic types and methods, while Java erases it and uses `Object` as the type of variables, parameters and fields that were originally typed with a generic parameter. For instance, imagine a

[1]Primitive types can also be allocated in the heap if they are used, for instance, as the type of fields.

generic class `List<T>` with a method definition like `T GetElementAt(int)`. At source code level, an invocation to this method with a receiver statically typed as `List<A>` returns an object of static type `A`, given that `A` is used as the argument of the generic type parameter `T`. At bytecode level, this is also true in the case of CIL, but it is not in the case of Java bytecode. All the instantiation information of generic types is erased by the Java compiler since its bytecode format does not support it. So instead, an invocation of the `GetElementAt` method returns an object of static type `Object`, and has to be followed by a dynamic cast to type `A` to preserve semantics.

Delegate types. Another important distinction between both platforms are *delegates*. They consist in a pointer to a method with a particular signature and, in the case of instance methods, a reference to an object that is the receiver of that method. They are used as a type-safe way to invoke statically unknown methods that can only be resolved dynamically at run-time. CIL supports delegates and they are used as the building blocks behind the scenes to implement lambdas, anonymous methods and events in high-level .NET based languages like C#. On the other hand, Java does not support them. To achieve the same kind of functionality Java provides other features like anonymous inner classes and the concept of a functional interface, which is any interface with only one method.

Async and await. In .NET, an `async` method returns a `Task` object that allows the caller to execute the method asynchronously. On the other hand, the `await` keyword waits until the execution of the asynchronous method ends to extract the results of the computation. This pattern is compiled into a complex state machine behind the scenes. Java does not have built-in support for this kind of concurrent programming mechanism.

1.4 Contributions

This thesis includes the following contributions:

- We present a distributed, whole-program static analysis framework that is designed to scale with the size of the input. Our approach is based on the actor programming model and is deployed in the cloud. Our reliance on a cloud cluster provides a degree of elasticity for CPU, memory and storage resources.

- We show how a typical call graph analysis can be implemented in a distributed setting for answering program understanding and code browsing queries. We describe how the analysis is implemented on top of the Orleans [38] actor model [26] and is deployed on legacy hardware in the cloud using Microsoft Azure. We experimentally demonstrate the scalability of our distributed call graph implementation using a combination of both synthetic and real benchmarks.

- We show how to extend our call graph analysis to support incremental updates and show how the previously computed results can be updated without having to recompute them from scratch. Only reanalyzing the modified parts of the program is required. We experimentally demonstrate significant performance improvements in comparison to the full version of the analysis.

- We present a static analysis framework and tools specifically designed for the .NET platform [23]. It provides many features, including a few intermediate code representations such as a three-address code suitable for implementing a static analysis on top of it, and a rich set of analyses and transformations such as type inference, control-flow and data-flow analyses, and call graph and points-to graph construction, among others. We don't know of any other static analysis framework publicly available to the .NET community providing these kind of features.

- We present a few client applications of the framework to demonstrate its capabilities, such as a Big Data query optimization analysis to automatically detect unused columns and dependencies between input and output tables of user-defined operators written in a .NET based programming language like

C#. We experimentally demonstrate the benefits of these analyses using a mix of related-work and real benchmarks.

1.5 Thesis Organization

The rest of the thesis is organized as follows.

Static Analysis in the Cloud

Chapter 2 presents our distributed static analysis approach.

Chapter 3 describes the specifics of the call graph construction algorithm.

Chapter 4 describes the call graph analysis extension to support incremental updates.

Chapter 5 discusses some interesting implementation details on top of the Orleans actor model.

Chapter 6 presents our experimental evaluation and results.

Chapter 7 discusses related work of this part of the thesis.

Static Analysis Framework

Chapter 8 presents our static analysis framework and tools for the .NET platform.

Chapter 9 describes the details of our Big Data query optimization analysis implemented using our framework.

Chapter 10 provides a high-level overview of a few additional clients of the framework.

Chapter 11 discusses related work of this part of the thesis.

Epilogue

Chapter 12 discusses future work and concludes.

Appendix contains additional material for the interested reader.

Resumen

Introducción

El análisis de programas [88, 67] es el proceso de analizar automáticamente el comportamiento de los programas de computadora con respecto a una propiedad como la corrección, la robustez, la seguridad o la vida útil. A menudo se enfoca en dos áreas principales: optimización de programas y corrección de programas. El primero se centra en mejorar el rendimiento del programa mientras reduce el uso de recursos. El segundo se enfoca en asegurar que el programa haga lo que se supone que debe hacer.

El análisis de programas es utilizado por varias herramientas de ingeniería de software, incluidos compiladores, entornos de desarrollo integrado (IDEs), verificadores estáticos y analizadores del estilo del código, entre otros [67]. Entre sus usos típicos podemos mencionar el análisis de código, la transformación y la optimización, la inferencia de tipos, la verificación de la corrección, la garantía de seguridad y la comprensión de programas [67].

El análisis de programas se puede realizar sin ejecutarlo (análisis estático de programas), durante el tiempo de ejecución (análisis dinámico de programas) o en una combinación de ambos [88, 67].

El análisis estático [88] es la evaluación exhaustiva de una aplicación mediante el examen de alguna representación estática del programa, generalmente el código fuente, sin ejecutarlo. Deduce en tiempo de compilación, propiedades sobre el comportamiento en tiempo de ejecución de los programas. Permite verificar, por ejemplo, la ausencia de errores de ejecución o violaciones de seguridad.

El análisis dinámico [88] es la prueba y evaluación de una aplicación durante el tiempo de ejecución. Se realiza ejecutando el programa y puede utilizar el conocimiento obtenido durante el tiempo de ejecución para aumentar la precisión del análisis. Para que sea efectivo, el programa a analizar debe ejecutarse con suficientes entradas de prueba para producir comportamientos interesantes.

Muchos defectos de software que causan errores pueden detectarse de forma dinámica y estática. Los dos enfoques son complementarios porque ningún enfoque individual puede encontrar todos los errores. Un análisis estático puede garantizar que una propiedad de interés sea válida en cada ejecución posible del programa. En contraste, incluso siendo que el análisis dinámico puede ayudar a garantizar corrección, su objetivo principal es encontrar y depurar errores, ya que solo puede analizar una sola ejecución del programa a la vez. Sin embargo, puede revelar defectos sutiles o vulnerabilidades cuya causa es demasiado compleja para ser descubierta por un análisis estático.

Un análisis que depende de información que sólo está disponible en tiempo de ejecución es inherentemente dinámico [67]. Si un grado de imprecisión es tolerable, puede ser posible realizar un análisis estático *aproximado*. Para ser correctos, los análisis estáticos a menudo sobre-aproximan sus resultados. Para evitar falsos positivos, se requiere algoritmos y abstracciones precisos. Hay varias dimensiones de diseño para ajustar la precisión de un análisis estático [67].

Un análisis estático *whole-program* considera todo el programa y admite diferentes suposiciones. La suposición *closed-world* considera todos los módulos y bibliotecas a los que hace referencia un programa, mientras que la suposición *open-world* restringe su alcance considerando sólo el código fuente de un programa, sin analizar los módulos de referencia.

Un análisis estático intra-procedural restringe su alcance considerando sólo los cuerpos de los métodos de forma aislada. Un análisis estático inter-procedural se encarga de la interacción entre los métodos de un programa.

Además, los análisis estáticos se pueden clasificar según su sensibilidad, lo que afecta a su precisión en general. La sensibilidad al contexto considera el contexto de llamada cuando se analiza el destino de una invocación. La sensibilidad a

objetos distingue los objetos considerando, por ejemplo, su lugar de creación. La sensibilidad a campos distingue dos campos de un mismo objetos. La sensibilidad a flujo considera el orden de las instrucciones dentro del cuerpo de un método. Idealmente, un análisis de gran precisión combina todas las opciones, pero al costo de comprometer la escalabilidad.

En esta tesis presentamos el diseño y la implementación de una amplia gama de análisis estáticos de programas para la plataforma .NET [23]. Nuestro objetivo es doble: (i) diseñar técnicas que puedan escalar a programas más grandes y (ii) proporcionar un conjunto de herramientas que permita a los investigadores y desarrolladores implementar sus propios análisis de programas .NET.

Por este motivo, en la primera parte de esta tesis presentamos el diseño y la implementación de un análisis estático de *call graph* inter-procedural típico de la literatura, pero distribuido en la nube con soporte para actualizaciones incrementales del código fuente. En este contexto, nuestro objetivo es permitir que el análisis escale a aplicaciones industriales muy grandes del mundo real en términos de consumo de memoria y tiempo de análisis.

En la segunda parte de esta tesis presentamos un *framework* de análisis estático de programas diseñado específicamente para analizar programas .NET, y algunas aplicaciones cliente que aprovechan sus características. En este caso, nuestro objetivo es proporcionar a la comunidad .NET las herramientas necesarias para desarrollar un análisis estático de programas .NET, de la misma manera que están disponibles en otras plataformas similares, como Java.

Part II

Static Analysis in the Cloud

Chapter 2

Overview

In the last decade, we have seen a number of attempts to build increasingly more scalable whole-program analysis tools. Advances in scalability have often come from improvements in underlying solvers such as SAT and Datalog solvers as well as sometimes improvements to the data representation in the analysis itself; we have seen much of this progress in the space of pointer analysis [63, 62, 115, 83, 28, 82, 79].

Limits of scalability. A typical whole-program analysis is designed to run on a single machine, primarily storing its data structures in memory. Despite the intentions of the analysis designer, this approach ultimately leads to scalability issues as the input program size increases, with even the most lightweight of analyses. Indeed, if the analysis is stateful (i.e., it needs to store data about the program as it progresses, typically in memory), eventually this approach ceases to scale to very large inputs. Memory is frequently a bottleneck even if the processing time is tolerable, despite various memory compression techniques such as BDDs. We believe that the need to develop scalable program analyses is now greater than ever. This is because we see a shift toward developing large projects in centralized source repositories such as GitHub [4], which opens up opportunities for creating powerful and scalable *analysis backends* that go beyond what any developer's machine may be able to accomplish.

Distributed analysis. We explore an alternative approach to build distributed static analysis tools, designed to scale with the input size, with the goal of achieving full elasticity. In other words, no matter how big the input program is, given enough *computing resources* (i.e., machines to execute on) the analysis will complete in a reasonable time. Our analysis architecture assumes that the static analysis runs in the cloud, which gives us elasticity for CPU and memory resources, as well as storage. More specifically, in the context of large-scale code repositories, even code understanding and code browsing tasks are made challenging by the size of the code base. We have seen the emergence of scalable online code browsers such as LXR [6]. These tools often operate in batch mode, and thus have a hard time keeping up with a rapidly changing code repository in real time, especially for repositories with many simultaneous contributors. We aim to show how a nimbler system can be designed, where analysis results are largely stored in memory, spread across multiple machines. This design leads to more responsive queries for obtaining analysis results.

The vision that motivates this work is that every large-scale software repository will be able to perform static analysis on a large scale. In this context, we present the design and implementation of a distributed, whole-program static analysis framework that is designed to scale with the size of the input. Our approach is based on the actor programming model and is deployed in the cloud. Our reliance on a cloud cluster provides a degree of elasticity for CPU, memory, and storage resources. To demonstrate the potential of our technique, we show how a typical call graph analysis can be implemented in a distributed setting.

We experimentally validate our implementation of the distributed call graph analysis using a combination of both synthetic and real benchmarks. To show scalability, we demonstrate how the analysis presented is able to handle very large inputs without running out of memory. This work was done in collaboration with researchers from Microsoft Research, and it was published [57, 58] and presented at the international conference on Foundations of Software Engineering (FSE) in 2017.

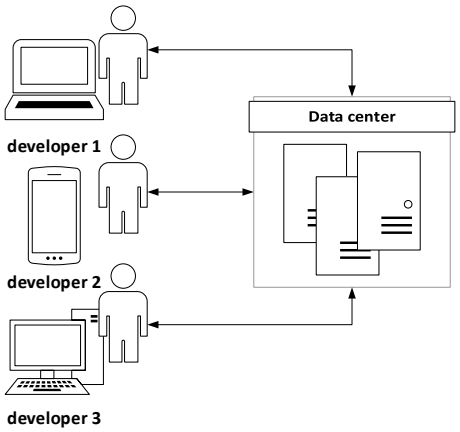


Figure 2.1: Analysis architecture: the analysis is performed using a cloud backend of multiple machines, with developers both querying the results and sending updates.

2.1 Motivation: Static Analysis Backend

Imagine a large project hosted within a centralized source repository such as GitHub [4], BitBucket [2] or Visual Studio Team Services [22]. We see an emerging opportunity to perform server-side analysis in such a setting, as illustrated in Figure 2.1. Indeed, the backends of many such repositories consists of a large collection of machines, not all of which are fully utilized at any given time. During the downtime, some of the available cycles could be used to do static analysis of the code base. This can help developers with both program understanding tasks, such as code browsing, as well as other static analysis applications, such as finding bugs.

The ever-changing code base. As is typical for large projects, multiple developers constantly update the code base, so it is imperative that the server-side analysis be both responsive to read-only user queries and propagate code updates fast. At the same time, within a large code base, many parts of the code, often entire directories remain unchanged for days or months at a time. Often, there

is no reason to access these for analysis purposes. Therefore, to ensure that we do not run out of memory, it is important to have a system that is able to bring analysis nodes into memory on demand and persist them to disk (put them to sleep) when they are no longer needed.

2.2 Call Graph Computation

We advocate the use of the *actor model* as a building block of typical worklist-based analysis approaches. More specifically, we use this approach to implement a typical call graph construction algorithm. While the algorithm itself is quite well-known and is not a contribution of this thesis, the way it is implemented in a distributed setting is. Also note that call graph information is used for interactive tasks such as autocomplete (or Intellisense), as shown in Figure 5.6. For tasks like these, both accuracy and responsiveness are important. Call graph construction is a fundamental step of most whole-program analysis techniques. However, most of the time, call graph analysis computation is a *batch* process: starting with one or more entry points such as `Main`, the call graph is iteratively updated until no more methods are discovered.

Interactive analysis. Our setting in this project is a little different. Our goal is to answer *interactive* user queries quickly. Our queries are the kind that are most frequently posed in the context of code browsing and debugging, and are already supported on a syntactic level by many IDEs. Specifically, our analysis has been developed to provide semantic, analysis-backed answers for the following IDE-based typical tasks:

- **Go to definition.** Given a symbol in the program, find its possible definitions[1].

[1]Note that this process is challenging due to the presence of polymorphism, common in object-oriented languages. Given a call site, it is not always possible to determine which is the actual method implementation being invoked. This problem known as *call site devirtualization* is well-studied in the literature. Therefore, a static analysis can only approximate the target method definitions for a virtual method invocation.

- **Who calls me.** Given a method definition, find all of its callers.
- **Auto-complete.** Auto-completion, invoked when the developer presses a dot, is one of the most common and well-studied tasks within an IDE [72, 89, 87, 86, 40, 80]. If the variable or expressions on the left-hand side of the dot is of a generic interface type, completion suggestions are not particularly useful or too general. It is therefore helpful to know which concrete type flow to a given abstract location.

We have architected our analysis backend to respond to REST calls [13] that correspond to the queries above. These queries constitute an important part of what is collectively known as *language services* and can be issued by both online IDEs, sophisticated code editors such as SublimeText or Visual Studio Code, and full-fledged IDEs such as Eclipse and Visual Studio. Figure 5.6 shows two screenshots of an experimental IDE prototype responding to user interactions and Table 5.1 shows some examples of REST queries.

Given the architecture shown in Figure 2.1, our goal is to have the analysis backend respond to queries quickly, independently of the input size. Of course, we also need to make sure that the backend does not run out of memory or timeout in some unpredictable way. Our requirements force us to rethink some of the typical assumptions of whole-program analysis.

Soundness. Considering the nature of such tasks that focus on *program understanding*, the goal is not to always be absolutely precise, but to be both useful to the end user and responsive. Our analysis judiciously cuts corners in the spirit of soundiness [77]. As the analysis results are used in an advisory role in the context of program understanding in an interactive setting, complete soundness is not the goal. For instance, we do not attempt to model reflective constructs. While we focus on C# as the input language, our work should apply equally well to analyzing large projects in Java and other similar object-oriented languages. It is not, however, our goal to faithfully handle all the tricky language features such as reflection, runtime code generation and `pinvoke`-based native calls.

2.3 Analysis Design Principles

We use a *distributed actor model* [26] as the basis of our distributed static analysis engine. For a program written in an object-oriented language such as Java or C#, a natural fit is to have an actor per *method* within the program. We could choose to have an actor per *class* in a program, or any other well-defined program entity. These actors are responsible for receiving messages from other actors, processing them using local state (a representation of the method body, for instance) and sending information to other methods that depend on it. For example, for a call graph construction analysis, actors representing individual methods may send messages to actors for their callers and callees. Our analysis design adheres to the following distilled principles:

- **Minimal in-memory state per actor.** We want to place as many actors per machine as possible without creating undue memory pressure, leading to swapping, etc.

- **Design for lightweight serialization.** We have designed our analysis so that the updates sent from one actor to another are generally small and easily serialized. There is minimal sharing among actors, as actor holds on to its local state and occasionally sends small updates to others. The same principle applies to persistent per-actor state as well. We only serialize the bare minimum to disk, before the actor is put to sleep. This can happen when the actor runtime decides to page an actor out due to memory pressure or lack of recent use.

- **State can be recomputed on demand.** In a distributed setting, we have to face the reality that processes may die due to hardware and/or software faults. It is therefore imperative to be able to recover in case of state loss. While it is possible to commit local state to persistent store, we eschew the overhead of such an approach and instead choose to recompute per-node state on demand.

- **Locality optimizations to minimize communication.** We attempt to place related actors together on the same machine. In the case of a call graph analysis, this often means that entire strongly connected components co-exist on the same physical box, which minimizes the number of messages that we actually need to dispatch across the network.

2.4 Distributed Worklist Algorithm

We now present a high-level view of a distributed analysis problem as a pair $\langle A, L \rangle$ where:

- A is a set of actors distributed in a network.
- $\langle L, \sqsubseteq, \sqcup \rangle$ is a complete semi-lattice of finite height[2].

Each actor $a \in A$ has the following associated functions:

- $VALUE[a] = v \in L$ is the local state of actor a.
- $TF[a] : L \mapsto L$ is the transfer function for the local computation performed within actor a. We assume all TF are monotone.

The following helper functions are for communicating state changes among actors:

- $DELTA(v, v')$ computes a set U of (global) updates required when switching from local state v to $v' \in L$.
- $PACK(a, u)$ is a function that given an update $u \in U$ at actor $a \in A$ produces one or several messages to communicate to other actors.
- $UNPACK(m)$ is a function that unpacks a message and returns a value $v \in L$.

[2]The finite height requirement can be avoided with the use of a widening operator.

Algorithm 1 Distributed worklist algorithm.

```
1:  while |MQ| > 0 do
2:     ⟨a, m⟩ := MQ.choose()
3:     v := UNPACK(m) ⊔ VALUE[a]
4:     if v ⊑ VALUE[a] then
5:        continue
6:     end if
7:     v′ := TF[a](v)
8:     if v ⊑ v′ then
9:        U := DELTA(v, v′)
10:       for each u in U do
11:          MQ := MQ ∪ PACK(a, u)
12:       end for
13:       VALUE[a] := v′
14:    end if
15: end while
```

Algorithm 1 shows the pseudocode for a distributed worklist algorithm. The algorithm makes use of a global message queue, denoted as MQ[3]. The queue is initialized with a set of starting messages that will depend on the actual analysis instance.

For each message, the algorithm merges its content with the local state of the destination actor and executes the transfer function associated with it to produce a new local state. Then, the algorithm proceeds to compute the information required to update the state of other affected actors and produces new messages to communicate this information to them.

[3]Note that MQ is a *mathematical abstraction*: we do not actually use a global message queue in our implementation. Conceptually, we can think of a (local) worklist maintained on a per-actor basis. Termination is achieved when all the worklists are empty.

2.5 Termination

We would like to show that the presented distributed worklist algorithm terminates.

Let H denote the (finite[4]) height of semi-lattice L and let $N = |A|$. Consider iterations through the loop on line 1. Let's consider two sets of sequences of iterations, I_1 are iterations that lead to a value increase on line 7 and I_2 are those that do not.

We can have at most $H \times N$ iterations in I_1 given the finite size of the lattice. For iterations in I_2, the size of MQ decreases because at least one message is consumed but it does not generate other messages. We consider two possibilities:

- Starting from some iteration i, we only have iterations in I_2. This, however, means that on every iteration the size of MQ decreases, until it eventually becomes empty.

- The other possibility is that we will have an infinite number of iterations in I_1. This is clearly impossible because the size of I_1 is bounded by $H \times N$.

It is important to emphasize the difference between this distributed algorithm and a single-node worklist approach. If a message is in flight, we do not wish the program analysis to terminate. However, detecting the emptiness of MQ is not trivial, so in practice we must have an effective means for detecting termination. For this reason. we make use of a sophisticated mechanism for termination detection, as described in Section 5.2.

While the Algorithm 1 reaches a fixed-point independently of the arrival order of messages, it is natural to ask whether that is the only fixed-point that can be reached. Given that $TF[a]$ is monotone and L is of a finite height the uniqueness of least fixed-point is guaranteed [47, 71].

[4]Note that our approach can also terminate for an infinite hight lattice with the use of a widening operator.

2.6 Possible Analysis Instantiations

In the following chapter we show in detail an instantiation of the framework used to compute call graphs. However, we believe the distributed Algorithm 1 can be instantiated for other program analyses that follow the same design principle. For instance, consider an inclusion-based analysis like Andersen's points-to [31]. A possible instantiation may be as follows:

- Each actor represents a method.
- The transfer function implements Andersen's inclusion rules locally and, in case there is a change in an argument of a method invocation, produces an update message to be sent to the potential callees.

Similarly, by just replacing the inclusion rules with unification rules in the transfer function, we can turn it into a unification based points-to analysis like Steensgaard's [102]. Context-sensitivity can be achieved by representing different $context \times method$ combinations with different actors.

It is worth noticing that our analysis has similar characteristics as standard data-flow analyses, but an ordering on how information flows between the actors cannot be assumed.

We envision future work where our distributed back-end would be combined with a natural front-end for this kind of analysis that uses Datalog, as previously proposed for single-machine analysis [69]. However, as we describe in Chapter 2, our evaluation in Chapter 6 focuses on quickly answering interactive questions related to call graph resolution in the context of an IDE.

Resumen

Información General

En la última década, hemos visto varios intentos de crear herramientas de análisis de programas completos cada vez más escalables. Los avances en la escalabilidad a menudo provienen de mejoras en los solucionadores subyacentes, como los solucionadores para SAT y Datalog, así como a veces mejoras en la representación de los datos en el análisis mismo; hemos visto mucho de este progreso en el espacio de análisis de punteros [63, 62, 115, 83, 28, 82, 79].

Límites de scalabilidad. Un análisis típico de programa completo está diseñado para ejecutarse en una sola máquina, principalmente almacenando sus estructuras de datos en la memoria. A pesar de las intenciones del diseñador del análisis, este enfoque conduce en última instancia a problemas de escalabilidad a medida que aumenta el tamaño de la entrada del programa, incluso con el análisis más liviano. De hecho, si el análisis guarda estado (es decir, necesita almacenar datos sobre el programa a medida que avanza, generalmente en la memoria), eventualmente este enfoque deja de escalar a entradas muy grandes. La memoria es frecuentemente un cuello de botella, incluso si el tiempo de procesamiento es tolerable, a pesar de las diversas técnicas de compresión de memoria, como las BDD. Creemos que la necesidad de desarrollar análisis escalables de programas es ahora mayor que nunca. Esto se debe a que vemos un cambio hacia el desarrollo de grandes proyectos en repositorios de código fuente centralizados como GitHub [4],

lo que abre oportunidades para crear *análisis backends* potentes y escalables que van más allá de lo que cualquier máquina de un desarrollador pueda lograr.

Análisis distribuido. Exploramos un enfoque alternativo para crear herramientas de análisis estático distribuido, diseñadas para escalar con el tamaño de la entrada, con el objetivo de lograr una elasticidad total. En otras palabras, no importa qué tan grande sea el programa de entrada, si se cuenta con suficientes *recursos de computación* (es decir, máquinas para ejecutar), el análisis terminará en un tiempo razonable. Nuestra arquitectura de análisis supone que el análisis estático se ejecuta en la nube, lo cual permite elasticidad para los recursos de CPU y memoria, así como también para el almacenamiento. Más específicamente, en el contexto de los repositorios de código de gran escala, incluso la comprensión de código y las tareas de navegación de código son difíciles debido al tamaño del código fuente. Hemos visto la aparición de navegadores de códigos en línea escalables, como LXR [6]. Estas herramientas a menudo funcionan en modo *batch* y, por lo tanto, tienen dificultades para mantenerse al día con un repositorio de código que cambia rápidamente en tiempo real, especialmente para los repositorios con muchos colaboradores simultáneos. Nuestro objetivo es mostrar cómo se puede diseñar un sistema ágil, donde los resultados del análisis se almacenan en gran parte en la memoria, distribuidos en múltiples máquinas. Este diseño conduce a consultas más receptivas para obtener los resultados del análisis.

La visión que motiva este trabajo es que cada repositorio de software de gran escala pueda realizar análisis estáticos a gran escala. En este contexto, presentamos el diseño y la implementación de un *framework* de análisis estático distribuido de programa completo que está diseñado para escalar con el tamaño de la entrada. Nuestro enfoque se basa en el modelo de programación con actores y se despliega en la nube. Nuestra dependencia en un *cluster* de la nube proporciona un grado de elasticidad para los recursos de CPU, memoria y almacenamiento. Para demostrar el potencial de nuestra técnica, mostramos cómo se puede implementar un análisis de *call graph* típico en un entorno distribuido.

Validamos experimentalmente nuestra implementación del análisis distribuido de *call graph* utilizando una combinación de *benchmarks* sintéticos y reales. Para mostrar la escalabilidad, mostramos cómo el análisis presentado puede manejar entradas muy grandes sin quedarse sin memoria. Este trabajo se realizó en colaboración con investigadores de Microsoft Research y se publicó [57, 58] y presentó en la conferencia internacional sobre Fundamentos de la Ingeniería de Software (FSE) en el año 2017.

Chapter 3

Distributed Call Graph Analysis

In this chapter we present an instantiation of the general framework described in the previous chapter for computing call graphs. Our analysis is a distributed inter-procedural inclusion-based static analysis inspired by the Variable Type Analysis (VTA) presented in [103]. This flow-insensitive analysis computes the set of potential types for each *object reference* (i.e., variables, fields, etc.) by solving a system of inclusion constraints. Because it propagates type constraints from object allocation sites to their uses, this kind of analysis is sometimes referred to as *concrete type* analysis.

3.1 Program Representation

Propagation graphs. At the method level, the inclusion-based analysis is implemented using a data structure we call *propagation graph* (PG) [103]. A PG is a directed graph used to *push* type information to follow the flow of data in the program, as described by analysis rules. Our analysis naturally lands itself to incrementality. A typical change in the program would require often minimal recomputation within the modified code fragment as well as propagation of that information to its *neighbors*. Propagation graphs support incremental updates since the propagation of information is triggered when a new type reaches (or no longer reaches) a node.

Terminology. More formally, let $PG = \langle R, E \rangle$ where R denotes a set of nodes representing *abstract locations* in the method (such as variables and fields) and E refers to a set of edges between them.

An edge $e = (v_1, v_2) \in E$ connects nodes in the PG to model the potential flow of type information from v_1 to v_2. Essentially, an edge represents a rule stating that $\mathit{Types}(v_2) \supseteq \mathit{Types}(v_1)$ (e.g., $v_2 = v_1$). To model inter-procedural interaction, the PG also includes nodes representing method invocations and return values, denoted by inv_{loc} and rv respectively. Finally, $I \subseteq R$ denotes the set of invocations. Let T be the set of all possible types, *DeclaredType* contains declared types (compile-time types) for abstract locations and *Types* denotes concrete types inferred by our analysis.

3.2 Analysis Phases

In the actor model, the choice of granularity is key for performance. We decided to use one actor per method, although other design decisions such as one actor per class are also possible. Each method-level actor contains a PG that captures type information that propagates through the method.

The analysis starts by analyzing an initial set of root methods. We describe both intra- and inter-procedural processing below.

3.2.1 Intra-procedural Analysis

This phase is the responsible of computing the local state of an actor representing a method.

Instantiating the problem. The lattice L for our analysis consists of a mapping from abstract locations to sets of possible types and is defined as shown below.

$$L \stackrel{\text{def}}{=} \langle \mathit{Types} : R \mapsto 2^T,\ \sqsubseteq_{type},\ \sqcup_{type} \rangle$$

$$
\begin{array}{rcl}
v_1 = v_2 & \Longrightarrow & \mathit{Types}(v_1) \supseteq \mathit{Types}(v_2) \\
v_1 = v_2.f & \Longrightarrow & \mathit{Types}(v_1) \supseteq \mathit{Types}(\mathit{DeclaredType}(v_2).f) \\
v_1.f = v_2 & \Longrightarrow & \mathit{Types}(\mathit{DeclaredType}(v_1).f) \supseteq \mathit{Types}(v_2) \\
v = \texttt{new C()} & \Longrightarrow & \texttt{C} \in \mathit{Types}(v) \\
\texttt{return}\ v & \Longrightarrow & \mathit{Types}(rv) \supseteq \mathit{Types}(v) \\
loc : v = v_0.m(v_1, \ldots, v_n) & \Longrightarrow & \mathit{Types}(inv_{loc}) \supseteq \bigcup_{j=0..n} \mathit{Types}(v_j)
\end{array}
$$

Figure 3.1: Variable type analysis rules.

$$
\begin{array}{rcl}
l_1 \sqsubseteq_{type} l_2 & \Longleftrightarrow & l_1.\mathit{Types}(r) \subseteq l_2.\mathit{Types}(r) \ \ \forall r \in R \\
l_1 \sqcup_{type} l_2 & \stackrel{\text{def}}{=} & l_3 \ \text{ where} \\
 & & l_3.\mathit{Types}(r) = l_1.\mathit{Types}(r) \cup l_2.\mathit{Types}(r) \ \ \forall r \in R
\end{array}
$$

Analysis rules to compute the transfer function $TF[a]$ are summarized in Figure 3.1 and cover the typical statement types such as loads, stores, allocations, etc. Object dereferences (i.e., `v.f`) are represented by using the name of the class defining the field. That is, the analysis is field-sensitive but not object-sensitive. In the case of invocations there is an inclusion relation to model the flow of all the arguments to the invocation abstract location $inv_{loc} \in I \subseteq R$. Note that the left-hand side v of the invocation is not updated by the rule since it depends on the result of the invoked method. This will be handled by inter-procedural analysis.

Notice that $TF[a]$ is monotonic because the propagation of newly added types never removes a type and L satisfies the finite-height condition because it is a finite lattice.

Algorithm 2 shows the pseudocode for the local propagation worklist algorithm for adding types. The algorithm makes use of a queue, denoted as RQ, to track modified PG nodes that need to be processed. The queue is initialized with previously modified nodes corresponding to object allocations and method

Algorithm 2 Local propagation algorithm for adding types.

```
1: S := ∅
2: while |RQ| > 0 do
3:    n := RQ.dequeue()
4:    if n ∈ I ∪ {rv} then
5:       S := S ∪ {n}
6:    else
7:       for all { m | (n, m) ∈ E } do
8:          D := Types(n) \ Types(m)
9:          if |D| > 0 then
10:            Types(m) := Types(m) ∪ D
11:            RQ := RQ ∪ {m}
12:         end if
13:      end for
14:   end if
15: end while
16: return S
```

parameters. Modified method invocation and return value nodes (i.e., inv_{loc} and rv respectively) are returned by the algorithm to be further processed by the inter-procedural phase of the analysis. Otherwise, the algorithm propagates the addition of newly reaching concrete types from each modified node to all of its destination nodes by following the associated PG edges.

3.2.2 Inter-procedural Analysis

Once the intra-procedural phase finishes, relevant updates must be communicated to the corresponding methods (callees and callers). As mentioned, the analysis considers invocations using the set $I \subseteq R$. To handle callers' updates, we need to extend the lattice to include the caller's information for the current method. This is a tuple of the form (m, lhs), where $m \in A$ denotes the caller's name and $lhs \in R$ represents the left-hand side of the invocation made by the

$$
\begin{aligned}
\texttt{let } \mathit{diff}_r(v, v') &:= v'.\mathit{Types}(r) \mathbin{\triangle} v.\mathit{Types}(r) \\
\texttt{let } \mathit{deltaInvs}(v, v') &:= \{\ \mathit{inv} \mid \mathit{inv} \in I \wedge \mathit{diff}_{\mathit{inv}}(v, v') \neq \varnothing\ \} \\
\texttt{let } \mathit{deltaRv}(v, v') &:= \begin{cases} \{rv\} & \text{if } \mathit{diff}_{rv}(v, v') \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\mathit{DELTA}(v, v') \stackrel{\text{def}}{=} \mathit{deltaInvs}(v, v') \cup \mathit{deltaRv}(v, v')
$$

Figure 3.2: Definition of *DELTA* function.

caller. The extended lattice is shown below.

$$
L \stackrel{\text{def}}{=} \langle \mathit{Types} : R \mapsto 2^T \times \mathit{Callers} : 2^{A \times R},\ \sqsubseteq,\ \sqcup \rangle
$$

$$
\begin{aligned}
l_1 \sqsubseteq l_2 &\iff l_1 \sqsubseteq_{\mathit{type}} l_2 \ \wedge\ l_1.\mathit{Callers} \subseteq l_2.\mathit{Callers} \\
l_1 \sqcup l_2 &\stackrel{\text{def}}{=} (\mathit{ts},\ \mathit{cs}) \ \text{ where} \\
&\quad \mathit{ts} = l_1 \sqcup_{\mathit{type}} l_2 \ \wedge \\
&\quad \mathit{cs} = l_1.\mathit{Callers} \cup l_2.\mathit{Callers}
\end{aligned}
$$

A message m has the form $\langle \mathit{kind},\ d,\ \mathit{data} \rangle$, where $\mathit{kind} \in \{\texttt{CallMsg}, \texttt{ReturnMsg}\}$ is the kind of message, $d \in A$ is the destination actor and data is a tuple.

Instantiating *DELTA*. In Figure 3.2 we show the definition of the *DELTA* function described in Section 2.4. It computes the set of invocations that were affected by the propagation. An invocation is affected if the set of types flowing to any of its parameters grew. Additionally, we also must consider changes in types that the return value may correspond to, since they need to be communicated to the callers.

Instantiating *PACK*. Figure 3.3 shows a definition of *PACK*. This function is in charge of converting local updates to messages that can be serialized and

$$
\begin{aligned}
\texttt{let } \mathit{callees}(\mathit{inv}) &:= \{\ C.m \mid C \in \mathit{Types}(\mathit{args}(\mathit{inv})_0)\ \} \\
\texttt{let } \mathit{callData}(a, \mathit{inv}) &:= (a,\ \mathit{lhs}(\mathit{inv}),\ \overline{\mathit{Types}}(\mathit{args}(\mathit{inv}))) \\
\texttt{let } \mathit{callMsgs}(a, \mathit{inv}) &:= \{\ \langle \texttt{CallMsg},\ d,\ \mathit{callData}(a, \mathit{inv})\rangle \\
&\qquad \mid\ d \in \mathit{callees}(\mathit{inv})\ \} \\
\texttt{let } \mathit{returnData}(a, c) &:= (a,\ \mathit{lhs}(c),\ \mathit{Types}(rv)) \\
\texttt{let } \mathit{returnMsgs}(a) &:= \{\ \langle \texttt{ReturnMsg},\ \mathit{method}(c),\ \mathit{returnData}(a, c)\rangle \\
&\qquad \mid\ c \in \mathit{Callers}\ \}
\end{aligned}
$$

$$
\mathit{PACK}(a, u) \stackrel{\text{def}}{=} \begin{cases} \mathit{callMsgs}(a, u) & \text{if } u \in I \\ \mathit{returnMsgs}(a) & \text{if } u = rv \end{cases}
$$

Figure 3.3: Definition of *PACK* function. $\overline{\mathit{Types}}(\mathit{args})$ is the lifting of *Types* to the list of arguments; it returns a list of sets of types. Given $\mathit{inv} = \langle v = v_0.m(v_1, \ldots, v_n)\rangle$, $\mathit{args}(\mathit{inv}) = [v_0, v_1, \ldots, v_n]$, $\mathit{lhs}(\mathit{inv}) = v$. For a caller $c = (m,\ \mathit{lhs}) \in \mathit{Callers}$, $\mathit{method}(c) = m$, the caller's name and $\mathit{lhs}(c) = \mathit{lhs}$, the left-hand side of the original invocation made by the caller.

sent to other actors. For each invocation, the analysis uses the computed type information of the receiver argument to resolve potential callees.

Then, it builds a caller message including the potential types for each argument. Those types will be added to the set of types of the parameters on the caller actor. In case of an update in return value it builds a message to inform the caller about changes to the return value's types. This message includes the (original) caller's left-hand side, so that the caller can update its types.

Instantiating *UNPACK*. Function *UNPACK* in Figure 3.4 is responsible for processing messages received by an actor. This function converts a message into a value in the lattice of the local analysis that will be then joined into the local

$$
\begin{aligned}
\texttt{let}\ l_1.Types(r) &= \begin{cases} argumentTypes(m)_i & \text{if } r = p_i \\ \varnothing & \text{otherwise} \end{cases} \\
\texttt{let}\ l_1.Callers &= \{\ (sender(m),\ lhs(m))\ \} \\
\texttt{let}\ l_2.Types(r) &= \begin{cases} returnTypes(m) & \text{if } r = lhs(m) \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
UNPACK(m) \stackrel{\text{def}}{=} \begin{cases} l_1 & \text{if } kind(m) = \texttt{CallMsg} \\ l_2 & \text{if } kind(m) = \texttt{ReturnMsg} \end{cases}
$$

Figure 3.4: Definition of *UNPACK* function. For a message $m = \langle \texttt{CallMsg},\ d,\ (a,\ lhs,\ [ts_0, \ldots, ts_n]) \rangle$, $argumentTypes(m)_i = ts_i\ \forall i = 0..n$, the set of potential types for the i^{th} argument of the method invocation (corresponding to the i^{th} parameter p_i of the callee d); $lhs(m) = lhs$ and $sender(m) = a$. For a return message $m' = \langle \texttt{ReturnMsg},\ d,\ (a,\ lhs,\ ts) \rangle$, $returnTypes(m') = ts$, the set of potential types of the method's return value.

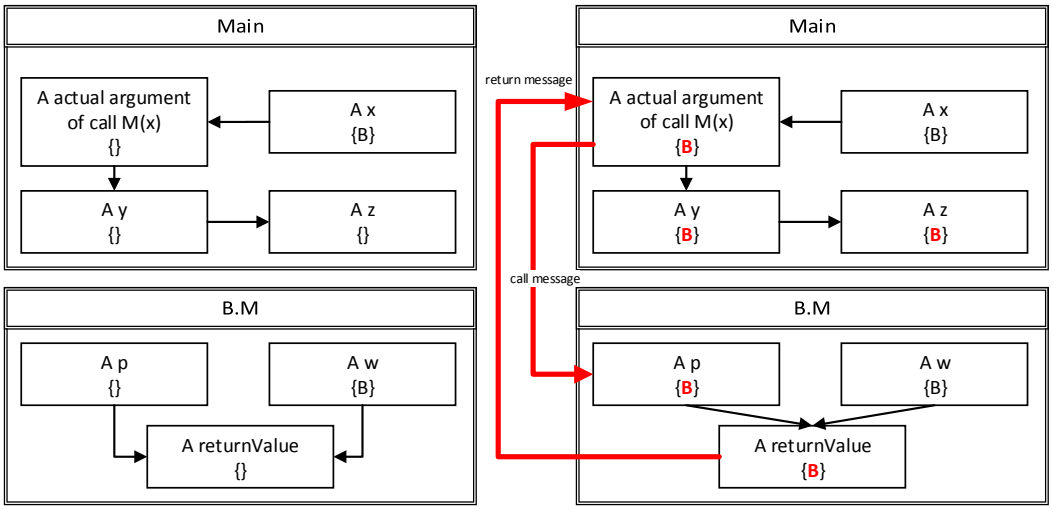
state. A message can be either a *call message* (i.e., an invocation made by a caller) or a *return message* (i.e., to inform a change in the callee's return value). For call messages we produce an element that incorporates the types for each call argument into the method parameters. We also update the set of callers. For return messages we need to update the left-hand side of the invocation with the potential types of the return value.

Example 1. This example illustrates the advantage of using concrete types as opposed to declared types to obtain more precision. Consider the small program of Figure 3.5a. The propagation graphs for both methods are shown in Figure 3.5b.

As the analysis starts, only the left-hand sides of allocations (lines 2 and 13) contain types. During propagation, type `B` flows from variable `x` into an invocation of `M` as an argument. This triggers a message to the actor for method `B.M`. The flow

```
1   public static void Main() {
2     A x = new B();
3     A y = x.M(x);
4     A z = y;
5   }
6
7   class A {
8     public abstract A M(A p);
9   }
10
11  class B : A  {
12    public override A M(A p)  {
13      A w = new B();
14      return (p != null) ? p : w;
15    }
16  }
```

(a) Code example for inter-procedural propagation.



(b) Propagation graphs for methods `Main` and `B.M` before (left) and after (right) the propagation.

Figure 3.5: Code and propagation graphs for Example 1.

```
1   void Main() {
2     A x = new B();
3     x.f = new B();
4     A y = M(x);
5   }
```

```
6   A M(A p) {
7     A z = p.f;
8     return z;
9   }
10
```

Figure 3.6: Rapid Type Analysis scenario for Example 2.

through parameter `p` and `w` makes the return value of `B.M` to contain type `B`. This in turn triggers a return message that adds `B` to the types of `y`. This propagates to `z`. Concrete type analysis produces results that are more accurate for `y`, `z`, etc. than what we can obtain from their declared types. ■

Type approximation. In the inter-procedural stage, our analysis sends information about concrete parameter types to its callees. However, when it comes to complex, nested objects, this information is potentially insufficient, as it only concerns one level of the object hierarchy. Instead of sending information about nested fields, which leads to increased message sizes, we opted to use the types given by a distributed adaptation of the Rapid Type Analysis (RTA) [33] that runs simultaneously on each method-actor. When RTA provides no useful information, we fall back on declared types. We did not observe imprecision caused by this overapproximation.

Example 2. Consider the code fragment shown in Figure 3.6. Suppose there is a class `A` that defines a field `f` of type `A` and another class `B` that inherits from `A`.

Function *PACK* will create a message that propagates the type of `x` into `M` and *UNPACK* will discover the type of `p` to be `B`. However, no information is given for the type of `p.f`, potentially leading to unsoundness. In order to be sound the analysis should conservatively assume that both types `A` and `B` are possible for `p.f`. But, by using RTA, the analysis knows that the code only allocates objects of class `B`, so `A` is not considered as possible type for `p.f`, only `B`. ■

Resumen

Análisis Distribuido de Call Graph

En este capítulo presentamos una instanciación del *framework* general descrito en el capítulo anterior para construir *call graphs*. Nuestro análisis es un análisis estático distribuido inter-procedural basado en inclusión, inspirado en el *Variable Type Analysis* (VTA) presentado en [103]. Este análisis insensible al flujo calcula el conjunto de tipos posibles para cada *referencia de objeto* (es decir, variables, campos, etc.) resolviendo un sistema de restricciones de inclusión. Debido a que propaga las restricciones de tipo desde los lugares de creación de objetos a sus usos, este tipo de análisis a veces se denomina análisis de *tipo concreto*.

Representación del Programa

Grafos de propagación. Al nivel de método, el análisis basado en la inclusión se implementa mediante una estructura de datos que llamamos *grafo de propagación* (PG) [103]. Un PG es un grafo dirigido que se utiliza para *empujar* información de tipos siguiendo el flujo de datos del programa, como es descrito por las reglas de análisis. Nuestro análisis acepta ser incremental de forma natural. Un cambio típico en el programa requeriría, a menudo, una recomputación mínima del fragmento de código modificado, así como la propagación de esa información a sus *vecinos*. Los grafos de propagación admiten actualizaciones incrementales, ya que la propagación de la información se activa cuando un nuevo tipo alcanza (o deja de alcanzar) un nodo.

Chapter 4

Incremental Call Graph Analysis

In this chapter we present an extension of the distributed call graph analysis described in the previous chapter for supporting incremental source code updates. The goal of our incremental analysis is to efficiently update its results when the program source code changes. Intuitively, the key idea that motivates this extension is that computing new results based on previous results should be much less expensive than computing them from scratch [67, 96]. In order to achieve this, the incremental call graph analysis algorithm has to be effective in avoiding work on parts of the program source code that have not changed since the last propagation. An incremental algorithm promises significant performance improvements over the exhaustive call graph analysis because analyzing code changes is often much faster than analyzing the entire code base [96, 100, 65, 76]. This approach is particularly useful, since only a relatively small part of the program needs to be reanalyzed, rather than the whole program [67].

4.1 Analysis Design Principles

In contrast to many other incremental static analyses that typically focus in program modifications made while editing code within an IDE [100, 76, 66], our approach has a different goal. As mentioned before, we care about providing a static analysis backend to be run in a cloud-based source repository. For this reason, our incremental analysis design adheres to the following fundamental principles:

- **Revision-based approach.** Source code modifications are taken from the revisions submitted to source control management systems such as Git [3], Mercurial [7] or SVN [1]. This makes our analysis ideal to be run in background, triggered by user commit events in centralized web-based hosting repositories like GitHub [4], as part of a Continuous Integration (CI) environment. We also restrict our incremental analysis to modifications that can be compiled successfully.
- **Method granularity.** To be consistent with the decision made for the distributed analysis, we consider methods as the minimal analysis unit. This means that the whole method will be reanalyzed even when only one of its statements was changed.
- **Modified methods simplification.** For simplicity we decided to treat modified methods as the removal of the current version followed by the addition of the new version. This means that our analysis only deals with two kinds of modifications: methods removed and methods added.

4.2 Analysis Challenges

Soundness. We want our incremental analysis to remain sound without loosing precision. This is a complex task in the context of arbitrary source code modifications. The analysis has to be carefully designed to avoid removing more concrete types than the ones that should be actually removed and to ensure that all reachable methods are eventually discovered.

Termination. Contrary to the full exhaustive analysis that achieves termination based on the fact that concrete types and methods are always added and never removed by the propagation algorithm, the incremental analysis cannot rely in the same arguments. Supporting removed methods implies the propagation of the removal of its effects to its neighbors (callees and callers). The interleaving of successive additions and deletions of reachable concrete types and methods could lead to termination issues if not carefully consider.

Detecting modified methods. We use source control features to obtain the list of modified documents between two (typically consecutive) revisions. To detect modified methods we consider only source code documents within this list. Each document have potentially two versions, one corresponding to the current (old) revision and another corresponding to the (new) revision with the changes. For some documents there is only one version: documents added and removed in the new revision. Both versions of each document are parsed to get the set of methods defined in each document version[1]. These are potential candidates of modified methods.

By comparing the sets of defined methods for both versions of the same corresponding document we classify each method as added, removed or modified. A method is added when it is defined in the new version of a modified document but it is not present in the old version of the same document. Analogously, a method is removed when it is defined in the old version of a modified document but it is not present in the new version of the same document. Finally, a method is modified when it is defined in both versions (old and new) of the same modified document and its source code is different between those versions.

Removing a method. Consider a method m defined in a source control revision r_1, but removed later on in some following revision r_2. If the source code corresponding to both revisions compiles successfully, the following scenarios are possible.

- The method is unreachable (dead code) at r_1.
- It is a reachable non-virtual method and all of its corresponding caller methods at r_1 are modified at r_2. Otherwise the source code shouldn't compile at r_2 due to the presence of invocations to m made by its callers that cannot be resolved anymore.
- It is a reachable virtual method overriding a base class method implementation m_b at r_1 and all of its corresponding callers' invocations that resolve to m at r_1 resolve to m_b at r_2.

[1]Our implementation makes use of a cache to store this information for the old version of each document, so only the new version needs to be parsed at this point.

Adding a method. Consider a method m added in a source control revision r_2, that was not defined before in some previous revision r_1. If the source code corresponding to both revisions compiles successfully, the following scenarios are possible.

- The method is unreachable (dead code) at r_2.
- It is a reachable non-virtual method and all of its corresponding caller methods at r_2 that are also defined at r_1 are modified at r_2, to include the new invocations to m that cannot be present in r_1 because m was not defined.
- It is a reachable virtual method overriding a base class method implementation m_b at r_2 that is also defined at r_1 and all of its corresponding callers' invocations that resolve to m at r_2 resolve to m_b at r_1.

Identifying root call sites. In order to be effective, the incremental algorithm must only reanalyze the bare minimum part of the source code affected by the changes and no much more than that. For this reason, is necessary to identify the root call sites from which to reanalyze. They are the starting point of the incremental analysis.

As depicted before, in most of the cases is straightforward to detect which are the relevant call sites. When a method is removed, they are just its already known call sites. But when a method is added, it depends on whether it is a virtual method or not. If the method is not virtual, its callers (if any) must be also modified and therefore, they will be reanalyzed in turn. However, if it is a virtual method that overrides a base class method implementation, is a little bit more difficult to detect the relevant call sites. Note that in this case, is not easy to determine which are their corresponding call sites, since these could remain unchanged and not be part of the modifications. Nevertheless, they can be over-approximated instead, by considering the call sites of the overridden base method as candidates to be reanalyzed. If the newly added method is reachable, some of those call sites will end up resolving to it, instead of to the overridden base method they originally resolved to.

Therefore, two kinds of call site modifications can be differentiated:

Explicit. When the modification of a method generates in turn explicit modifications in its call sites. In other words, call sites are either added, removed or the set of possible types for the receiver of each call site invocation is modified. Both the addition and removal of a non-virtual method produce this kind of modification.

Implicit. When the modification of a method does not generate explicit modifications in its call sites, but implicit ones. In other words, the set of possible types for the receiver of each call site invocation is not modified. It causes call site invocations to resolve to different method implementations, but they are not necessarily part of the source code changes. Both the addition and removal of a virtual method produce this kind of modification.

As mentioned, determining the call sites of implicit modifications is more complex in comparison to the explicit ones, just because they are not necessarily included in other modified methods. In the case of removing a virtual method, we know exactly which are its corresponding call sites because it was already analyzed. However, in the case of adding a virtual method that overrides a base class method implementation, is not always possible to determine a priori its call sites, but they can be safely over-approximated by taking as candidates the call sites of the overridden base method.

Example 3. This example illustrates both explicit and implicit call site modifications.

Explicit. Consider the code fragment shown in Figure 4.1a. Suppose later on, a new method `M2` is defined in the source code. Since it is a non-virtual method, in order to be reachable there must be at least some other modified and already reachable method calling it. In this case it is called from the modified and already existing reachable method `M1`, as shown in Figure 4.1b at line 3. Since this call site is explicitly modified (or more precisely, directly added) it is considered an explicit call site modification.

```
1  public void M1() {
2    Log("M1");
3  }
4
5
6
7
8
```

(a) Original version.

```
1  public void M1() {
2    Log("M1");
3    M2();
4  }
5
6  public void M2() {
7    Log("M2");
8  }
```

(b) Modified version.

(i) Explicit call site modification.

```
1  public void M1(B p) {
2    p.M2();
3  }
4
5  class A {
6    virtual void M2() {
7      Log("A.M2");
8    }
9  }
10
11 class B : A {
12 }
13
14
15
```

(c) Original version.

```
1  public void M1(B p) {
2    p.M2();
3  }
4
5  class A {
6    virtual void M2() {
7      Log("A.M2");
8    }
9  }
10
11 class B : A {
12   override void M2() {
13     Log("B.M2");
14   }
15 }
```

(d) Modified version.

(ii) Implicit call site modification.

Figure 4.1: Original and modified code versions for Example 3. The modified lines are highlighted in yellow.

Implicit. Now consider the code fragment shown in Figure 4.1c. Suppose later on, method `M2` is defined in class `B` as shown in Figure 4.1d, thus overriding the implementation inherited from its base class `A`. Even that this is the only modification of the source code and there is no other modified method calling it, the newly added method `B.M2` could still be reachable. This is because it is a virtual method that overrides an already reachable base class method implementation `A.M2`. In this case, its root call sites can be over-approximated by considering the call sites of the overridden base method implementation as candidates. In particular, the call site in method `M1` at line 2 has `B` as the only possible concrete type for the receiver `p` of the invocation to `M2`. Therefore, even that this call site was actually never explicitly modified, it no longer resolves to `A.M2`, but to `B.M2` instead. For this reason it is considered an implicit call site modification. ■

4.3 Supporting Removed Methods

One important difference of the incremental analysis in comparison to the full exhaustive analysis is the necessity of having to support removed methods. When a method is removed, the analysis needs to update its neighbors (callees and callers) to revert the effects produced by the removed method. This in turn leads to having to support the removal of concrete types that might have been originally propagated by the method, both intra- and inter-procedurally.

For this reason, we present another instantiation of the general analysis framework described in Section 2.4 for removing types. It is analogous to the one presented in the previous chapter, but instead of propagating the addition of types, it propagates the deletion of types.

4.3.1 Intra-procedural Analysis

As mentioned, this phase of the analysis is the responsible of updating the local state of an actor representing a method.

Instantiating the problem. Like before, the lattice L for our analysis consists of a mapping from abstract locations to sets of possible types and is defined as shown below.

$$L \stackrel{\text{def}}{=} \langle Types : R \mapsto 2^T,\ \sqsubseteq_{type},\ \sqcup_{type} \rangle$$

$$\begin{array}{rcl} l_1 \sqsubseteq_{type} l_2 & \iff & l_2.Types(r) \subseteq l_1.Types(r) \quad \forall r \in R \\ l_1 \sqcup_{type} l_2 & \stackrel{\text{def}}{=} & l_3 \quad \text{where} \\ & & l_3.Types(r) = l_1.Types(r) \cap l_2.Types(r) \quad \forall r \in R \end{array}$$

The analysis rules to compute the transfer function $TF[a]$ are exactly the same as described in Section 3.2.1. In this case, $TF[a]$ is monotonic because the propagation of removed types never adds a new type and L satisfies the finite-height condition because it is a finite lattice.

Algorithm 3 shows the pseudocode for the local propagation worklist algorithm for removing types. The algorithm makes use of a queue, denoted as RQ, to track modified PG nodes that need to be processed. The queue is initialized with previously modified nodes corresponding to object allocations and method parameters. Modified method invocation and return value nodes (i.e., inv_{loc} and rv respectively) are returned by the algorithm to be further processed by the inter-procedural phase of the analysis. Otherwise, the algorithm propagates the removal of no longer reaching concrete types from each modified node to all of its destination nodes by following the associated PG edges. Is important to note that the same concrete type can reach a node from more than one path in the PG. For this reason, a concrete type can only be removed from the set of possible types of a node if it was previously removed in *all* of its predecessor nodes. Contrary to the propagation of the addition of a newly reaching concrete type, that only requires a single modified predecessor node.

Algorithm 3 Local propagation algorithm for removing types.

```
1:  S := ∅
2:  while |RQ| > 0 do
3:    n := RQ.dequeue()
4:    if n ∈ I ∪ {rv} then
5:      S := S ∪ {n}
6:    else
7:      for all { m | (n, m) ∈ E } do
8:        D := Types(m)
9:        for all { h | (h, m) ∈ E } do
10:         D := D \ Types(h)
11:       end for
12:       if |D| > 0 then
13:         Types(m) := Types(m) \ D
14:         RQ := RQ ∪ {m}
15:       end if
16:     end for
17:   end if
18: end while
19: return S
```

4.3.2 Inter-procedural Analysis

Once the intra-procedural phase finishes, relevant updates must be communicated to the corresponding methods (callees and callers). Like before, to handle callers' updates, we need to extend the lattice to include the caller's information for the current method. The extended lattice is shown below.

$$L \stackrel{\text{def}}{=} \langle \mathit{Types} : R \mapsto 2^T \times \mathit{Callers} : 2^{A \times R},\ \sqsubseteq,\ \sqcup \rangle$$

$$
\begin{array}{rcl}
l_1 \sqsubseteq l_2 & \iff & l_1 \sqsubseteq_{type} l_2 \ \wedge \ l_2.Callers \subseteq l_1.Callers \\
l_1 \sqcup l_2 & \stackrel{\text{def}}{=} & (ts,\ cs) \ \text{ where} \\
 & & ts = l_1 \sqcup_{type} l_2 \ \wedge \\
 & & cs = l_1.Callers \cap l_2.Callers
\end{array}
$$

Instantiating the problem. All helper functions *DELTA*, *PACK* and *UNPACK* remain defined exactly as described in Section 3.2.2, but instead they make use of the extended lattice shown above.

4.4 Incremental Algorithm

Next we describe how our incremental analysis algorithm supports the following two fundamental scenarios.

Removing a method. As mentioned, in this case the algorithm needs to revert the effects produced by the removed method. There are two kind of effects: types propagated to callees through invocations and types propagated to callers through returns.

For each callee, the analysis removes the types propagated from the invocation arguments to the callee parameters and the method itself from the callers collection of the callee.

For each caller, the analysis removes the types propagated from the returned value to the left-hand side of the invocations that originally resolved to the removed method. Is important to note that this in turn could lead to further propagation of the removal of those types, both intra- and inter-procedurally.

Once the effects produced by the removed method are reverted, the analysis repropagates from each call site of each caller that had the removed method as possible callee. This is because those invocations might resolve to a different method now that the removed method is gone. Is important to note that some callees and callers could not exist anymore because they were also removed, or

could be different because they were modified. For those cases the analysis does not have to do anything because it will take care of them in their own turns.

Adding a method. In this case the algorithm has to check if the newly added method is virtual and overrides an already existing base class method or not.

If the added method does not override a base class method, then the analysis does not have to do anything because it could be an unreachable method that is not yet used in the program. However, if the newly added method is reachable, it has to be at least some other modified method that calls it, so it will be processed in turn later on. Remember in this case the method does not override another already existing one, so the only way for it to be reachable is by a newly added invocation made by another reachable method.

If the added method overrides a base class method, then the analysis treats the overridden method as removed. This way, the effects produced by the overridden method are reverted and the analysis repropagates from each call site of each of its callers as described above. Finally, some of those invocations might start resolving to the newly added method, while some others might still resolve to the same overridden method. Depending on this, each of those methods will become reachable or not.

Algorithm 4 shows the pseudocode for the incremental analysis algorithm. It makes use of two previously initialized sets: methods removed and methods added, denoted as MR and MA respectively. Remember that we consider method updates as a removal followed by an addition of the corresponding method. Additionally, S, W and M are sets of messages to be processed by the distributed worklist algorithm presented in Section 2.4.

As mentioned before, methods overridden by newly added virtual methods are treated as removed and added to the MR set to be further processed. Then, the algorithm obtains the effects produced by all of the removed methods (types propagated to callees and callers, at lines 13 and 9 respectively) and reverts them by propagating the removal of types at line 17. This is done by calling the *ProcessMessages* function passing as arguments the initial set of messages to process

Algorithm 4 Incremental analysis algorithm.

```
1: S := ∅
2: W := ∅
3: for each a in MA do
4:    if a overrides some method b then
5:       MR := MR ∪ {b}
6:    end if
7: end for
8: for each a in MR do
9:    M := PACK(a, rv)
10:   S := S ∪ M
11:   W := W ∪ M
12:   for each inv in I do
13:      M := PACK(a, inv)
14:      S := S ∪ M
15:   end for
16: end for
17: ProcessMessages(S, RemoveTypes)
18: UpdateSourceCode()
19: S := GenerateCallMessages(W)
20: ProcessMessages(S, AddTypes)
```

and a constant to indicate which instantiation of the general analysis framework has to be invoked. In this case the `RemoveTypes` constant refers to the instantiation for removing types presented in Section 4.3.

Once the effects produced by the removed methods are reverted, the incremental analysis algorithm performs the source code update by calling the *UpdateSourceCode* function to apply the changes made by the new source control revision under analysis. For instance, in the case of analyzing a project hosted on a Git repository, this function performs the pull of the new commit. It also proceeds to compile the updated source code to meet the requirements of our call graph

analysis. Is important to note that this step has to be done at this point and not before. This is, right after reverting the effects produced by the removed methods and before repropagating from the call sites of their callers. The reason for this is to ensure that the information computed by the analysis is always in sync with the analyzed source code version.

In the W set the algorithm keeps track of return messages only, that correspond to the effects propagated to the callers of the removed methods. The *GenerateCallMessages* function creates call messages (of kind `CallMsg`) from the return messages (of kind `ReturnMsg`) stored in W. Finally, at line 20 the algorithm repropagates types from the call sites of the callers that had the removed method as possible callee. This is done by calling the *ProcessMessages* function again, but this time, passing as argument the `AddTypes` constant that refers to the instantiation of the general analysis framework for adding types presented in Section 2.4.

Example 4. This example illustrates the incremental call graph analysis algorithm. The code fragment shown in Figure 4.2a defines two classes `A` and `B`, where `B` inherits from `A`. Method `M1` creates an object of type `B` and calls its virtual method `M2`.

After running the exhaustive call graph analysis, the only possible concrete type reaching the receiver `x` of the invocation of `M2` at line 3 is `B`. Since class `B` does not override the definition of `M2` inherited from class `A` (i.e., it does not provide a more specific implementation), the analysis resolves the virtual method call as an invocation to `A.M2`. As consequence, `A` is the only possible concrete type for the local variable `y` of method `M1` at line 4.

Suppose later on, method `M2` is defined in class `B` as shown in Figure 4.2b, thus overriding the implementation inherited from its base class `A`. Then, the incremental call graph analysis detects the addition of the new method `B.M2` and since it is a virtual method that overrides an already existing reachable base class method implementation, the analysis proceeds to revert the propagation effects produced by `A.M2` (to callees and callers). In this case, the analysis removes `A.M2`

```
1   public A M1() {
2     A x = new B();
3     A y = x.M2();
4     return y;
5   }
6
7   class A {
8     virtual A M2() {
9       return new A();
10    }
11  }
12
13  class B : A {
14  }
15
16
17
```

(a) Original version.

```
1   public A M1() {
2     A x = new B();
3     A y = x.M2();
4     return y;
5   }
6
7   class A {
8     virtual A M2() {
9       return new A();
10    }
11  }
12
13  class B : A {
14    override A M2() {
15      return new B();
16    }
17  }
```

(b) Modified version.

Figure 4.2: Original and modified code versions for Example 4. The modified lines are highlighted in yellow.

from the callers of the constructor of class `A` (the only callee it has) and propagates the removal of the concrete type `A` returned by `A.M2` and assigned to the local variable `y` of method `M1` at line 3 (the only call site it has). As consequence, type `A` no longer reaches variable `y`.

Finally, the incremental analysis updates the source code and repropagates the addition of concrete types from the `M2` call site in method `M1` at line 3. Note that the only possible concrete type of the receiver `x` of the invocation to `M2` was not modified and is still `B`. However, this time the invocation resolves to `B.M2` instead of `A.M2` as it was previously resolving to. As consequence, `B` is now the only possible concrete type for the local variable `y` of method `M1` at line 4. ■

4.5 Termination

A priori, given the possibility of propagating not only the addition but also the deletion of concrete types both intra- and inter-procedurally, it makes sense to wonder whether the incremental algorithm actually terminates. It could be the case that the same concrete type is repeatedly added and removed to the set of possible types of some PG node and thus, preventing the algorithm from reaching a fixed-point. However, we would like to show that this is not the case and therefore, the presented incremental analysis algorithm terminates.

The reason relies on the fact that it consists of two well different and completely separated phases that can be easily identified by taking a look to the algorithm's pseudocode. The first phase is in charge of reverting the effects produced by the removed methods through the general analysis framework instantiation for removing types. The second phase is in charge of repropagating from the call sites of the callers of the removed methods through the general analysis framework instantiation for adding types. Is important to note that while the first phase removes types, it never adds a type. And analogously, while the second phase adds types, it never removes a type. Therefore, given that both phases are executed in sequence and each one terminates, there is no possible interleaving of type additions and deletions that prevents from reaching a fixed-point. Thus, the incremental analysis algorithm always terminates.

Resumen

Análisis Incremental de Call Graph

En este capítulo presentamos una extensión del análisis distribuido de *call graph* descrito en el capítulo anterior para admitir actualizaciones incrementales del código fuente. El objetivo de nuestro análisis incremental es actualizar de manera eficiente sus resultados cuando cambia el código fuente del programa. De manera intuitiva, la idea principal que motiva esta extensión es que computar nuevos resultados basados en resultados anteriores debería ser mucho menos costoso que calcularlos de cero [67, 96]. Para lograr esto, el algoritmo de análisis incremental de *call graph* debe ser eficaz para evitar el trabajo en partes del código fuente del programa que no han cambiado desde la última propagación. Un algoritmo incremental promete mejoras significativas en el rendimiento en comparación con el análisis exhaustivo de *call graph*, ya que el análisis de los cambios del código suele ser mucho más rápido que el análisis de todo el código fuente [96, 100, 65, 76]. Este enfoque es particularmente útil, ya que sólo es necesario volver a analizar una parte relativamente pequeña del programa, en lugar del programa completo [67].

A diferencia de muchos otros análisis estáticos incrementales que normalmente se enfocan en las modificaciones de programas realizadas al editar código dentro de un IDE [100, 76, 66], nuestro enfoque tiene un objetivo diferente. Como se mencionó anteriormente, nos interesa proporcionar un *backend* de análisis estático que sea ejecutado en un repositorio de código fuente ubicado en la nube.

Chapter 5

Implementation

We implemented a prototype of our distributed approach[1] to analyze large-scale projects written in C#. This prototype relies on Roslyn [84], a compiler framework for analyzing C# code and the Orleans framework [38], an implementation of a distributed actor model that can be deployed in the cloud. Although other deployment options such as AWS are possible, we used Azure as a platform for running our experiments.

5.1 Orleans and the Actor Model

Orleans [38] is a framework designed to simplify the development of distributed applications. It is based on the abstraction of virtual *actors*. In Orleans terminology, these actors are called *grains*. Orleans solves a number of the complex distributed systems problems, such as deciding where (i.e., on which machine) to allocate a given actor, sending messages across machines, etc., largely liberating developers from dealing with those concerns. At the same time, the Orleans runtime is designed to enable applications that have high degrees of responsiveness and scalability. Grains are the basic building blocks of Orleans applications and are the units of isolation and distribution. Every grain has a unique global identity

[1] Source code and benchmarks available at: `https://github.com/edgardozoppi/call-graph-orleans`.

that allows the underlying runtime to dispatch messages between actors. An actor encapsulates both behavior and mutable local state. State updates across grains can be initiated by sending messages.

The runtime decides which physical machine (*silo* in Orleans terminology) a given grain should execute on, given concerns such as memory pressure, amount of communication between individual grains, etc. This mechanism is designed to optimize for communication locality because even within the same cluster the amount of cross-machine messages are considerably smaller than the amount of local messages, within the same machine.

We follow a specific strategy in organizing grains at runtime. This strategy is driven by the input structure. The input consists of an MSBuild *solution*, a `.sln` file that can be opened in Visual Studio. Each solution consists of a set of *project files*, `*.csproj`, which may depend on each other. Roslyn allows us to enumerate all project files within a solution, source files within a project, classes within a file, methods within a class, etc. Furthermore, Roslyn can use its built-in C# compiler to compile sources on the fly.

In Figure 5.1 we show how grains are organized to follow this logical hierarchy.

We define grains for solutions, projects and methods. We did not find it necessary to provide grains for classes and other higher-level code artifacts such as `namespace`s.

A solution grain is a singleton responsible for maintaining the list of projects and providing functionality to find methods within projects; A project grain contains the source code of all files for that project and provides functionality to compute the information required by method grains (e.g., to build propagation graphs by parsing the methods' code) as well as type resolution (e.g., method lookup, subtyping queries, etc). Finally, a method grain is responsible for computing the local type propagation and resolving caller/callees queries; it stores type information for abstract locations within the method.

The solution grain reads the `*.sln` file from cloud storage; in our implementation we used Azure Files, but other forms of input that support file-like APIs such as GitHub or Dropbox are also possible. Project grains read `*.csproj` files

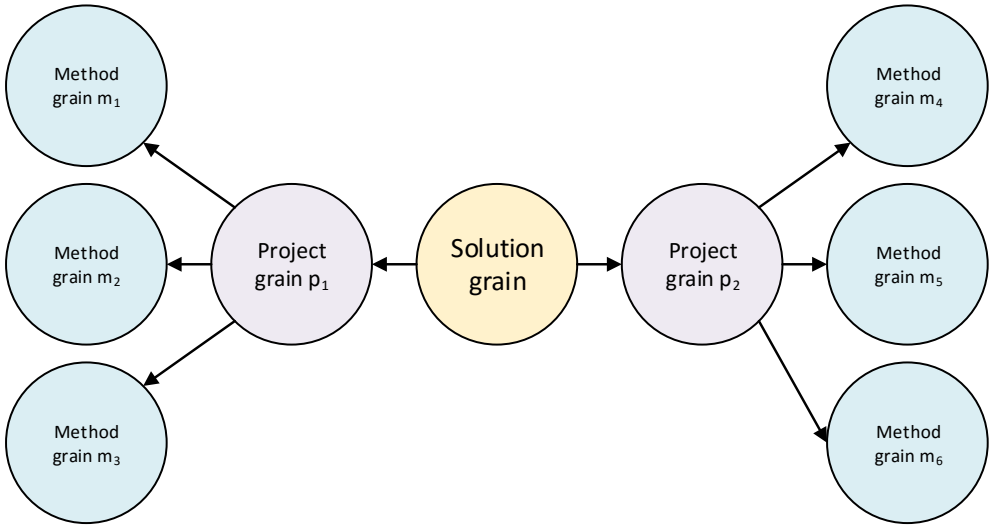


Figure 5.1: Logical organization of grains. The arrows show how grains create each other: solution grains create project grains; project grains create method grains, etc.

and also proceed to compile the sources contained in the project to get a Roslyn `Compilation` object. This information is only contained in the project grain to minimize duplication. To obtain information about the rest of the project, method grains can consult the project grain. We use caching to reduce the number of messages between method and project grains.

Example 5. To illustrate persistent state for a typical method grain, consider the example in Figure 3.5a. The state of both methods is as follows.

Method `Main`:

```
Callers = {}
Types   = {(x, {B}), (y, {B}), (z, {B}), (3, {B})}
```

Method `B.M`:

```
Callers = {(A.Main, y)}
Types   = {(p, {B}), (w, {B}), (returnValue, {B})}
```

This minimal state is easily serialized to disk if the grains are ever deactivated by the Orleans runtime. Orleans deactivates grains when they aren't used for a long time, however, this never happened in our experiments. ■

5.2 Distributed Analysis Details

Implementing a distributed system like ours is fraught with some fundamental challenges.

Reentrancy. Since the call graph can have cycles, a grain can start a propagation which will in turn eventually propagate to the original method. However, since Orleans uses turn-based concurrency this will create a deadlock. Even without recursion it is possible for a method grain that is currently being processed to receive another message (i.e., a return message from a callee).

Termination. In a distributed setting, detecting when we achieve termination is not so easy. This is in part because even if all the local worklists are empty, we may have messages in flight or those that have been delayed.

Timeouts. Similar to other turn-based concurrency systems, in order to detect potential failures and deadlocks, Orleans monitors the duration of calls to other grains and terminates calls that it deems to be timeouts. This has a number of undesirable consequences such as exceptions that propagate throughout the system. Some program analysis tasks, such as compiling a project or creating a propagation graph for a long method, may exceed the timeout that Orleans imposes.

Centralized orchestrator. A naïve implementation is not going to work well because of reentrancy issues: we can block the execution waiting for a message that waits for our response. For this reason, in our initial implementation we used

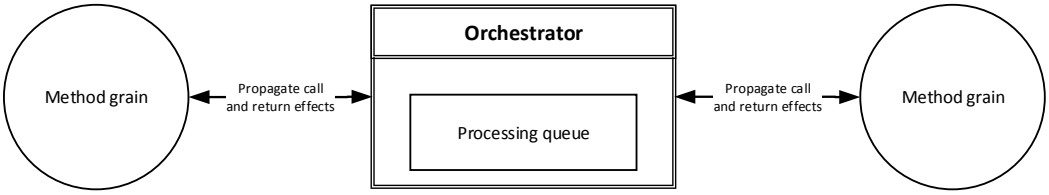


Figure 5.2: Centralized orchestration architecture.

an *orchestrator* to establish some degree of centralized control over the propagation process. Grains communicate with an orchestrator exclusively, instead of communicating with each other peer-to-peer. This avoids the issue of reentrancy by construction; only the orchestrator can send messages to grains via a single *message queue*, as shown in Figure 5.2. The orchestrator keeps track of the outstanding tasks and can therefore detect both termination and prevent reentrant calls from taking place.

The key disadvantage of this design is that it is possible to have a great deal of *contention* when accessing the centralized orchestrator. We observed this in practice, suggesting a different variant of this idea.

Distributed queues. Instead, we use *a collection* of queues placed across the distributed system to reduce contention. Each method grain is a potential producer of *effects* to be processed by other method grains. To avoid reentrancy, this information is not sent directly to the target method grain but it is enqueued in one of the queues in a round robin fashion. The information is then consumed by *dispatchers grains* that pull the data from the queues and deliver it to the corresponding method grains, as illustrated in Figure 5.3.

Using this mechanism we avoid both reentrancy, bottlenecks and single points of failure. The drawback is that detecting termination is more complex. For that, we use timers to determine when a dispatcher becomes idle (i.e., inactive longer than a predetermined threshold), at which point we notify the client. The analysis finishes when the client is sure that all dispatchers are idle[2]. In practice, we set

[2]We have a mechanism to detect when an idle dispatcher becomes active again.

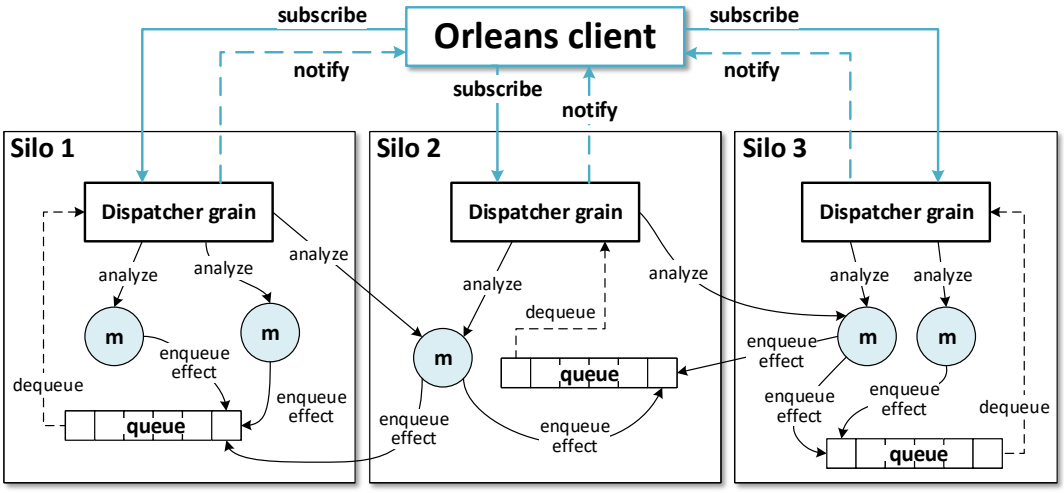


Figure 5.3: The multi-queue approach, illustrated. Method grains are circles shown in light blue. Solid and dashed arrows represent standard invocations and callbacks respectively. Each silo has each own dispatcher grain.

the number of queues to be four times higher than the number of worker VMs (for example, 128 queues for 32 worker VMs) and set the termination threshold to 10 seconds.

5.3 Incremental Analysis Details

In order to be sound, there is an important detail that has to be considered when implementing our incremental analysis algorithm. If a type flows to a propagation graph node from more than one source, and later on, the analysis propagates the removal of that type from only one of those sources, then the type could be incorrectly removed from the set of possible types of the node (depending on how they are stored), thus affecting soundness. So to prevent this from happening, it is mandatory to distinguish types flowing from different paths.

For this reason, the set of possible types for each propagation graph node has to be stored separately per incoming edge. This means to have a different set for each incoming edge of the node. Then, the set of possible types can be easily

```
1  A M1(bool p) {
2    A x = new A();
3    if (p) x = M2();
4    return x;
5  }
6
7  A M2() {
8    return new A();
9  }
```

(a) Original version.

```
1  A M1(bool p) {
2    A x = new A();
3    if (p) x = M2();
4    return x;
5  }
6
7  A M2() {
8    return new B();
9  }
```

(b) Modified version.

Figure 5.4: Original and modified code versions for Example 6. The modified line is highlighted in yellow.

computed by performing the union of all those sets. These are the types that the incremental algorithm will propagate to other nodes. However, when a new type is propagated to a given node through a particular incoming edge, that type will be added only to the set associated to that edge.

Analogous but more interesting is the case when the removal of a type is propagated to a given node through a particular incoming edge. In this case, that type will be removed only from the set associated to that edge. So if the same type does not reach the same node through some other incoming edge, then the type will not be part of the set of possible types of the node. However, if the same type reaches the same node through some other incoming edge, then it will not be removed from the set associated to the other edge, and consequently, the type will still be part of the set of possible types of the node.

Example 6. This example illustrates why it is important to store the set of possible types separately per incoming edge for each node in the propagation graph. Not doing so could lead to unsoundness when propagating the removal of types. Consider the code fragment shown in Figure 5.4a.

After running the analysis, the possible types for `x` at line 4 of method `M1` is {`A`}. Suppose later on, method `M2` is modified as shown in Figure 5.4b, returning an object of type `B` instead of type `A`. When running the incremental analysis to update the call graph with those changes, the first phase of the algorithm propagates the removal of the possible types returned by the original version of `M2`. Let's consider two cases depending on how the set of possible types for each PG node is stored.

- Storing in a single set leads to the removal of type `A` as possible type for `x` at line 4 of method `M1`, which is clearly incorrect.
- Storing in separate sets per incoming edge leads also to the removal of type `A`, but only in the set of `x` corresponding to the edge coming from the invocation of `M2` at line 3 of method `M1`. Note that `A` is still a possible type for `x` since it is stored in the (unchanged) set of `x` corresponding to the edge coming from the allocation at line 2 of `M1`.

The problem comes from the fact that in the original version of the code, type `A` flows to `x` from more than one source. Note that this is not an issue when propagating the addition of possible types. Finally, the second phase of the algorithm repropagates from the invocation of `M2` at line 3 of method `M1`, leading to the analysis of the new version of method `M2`, which in turn leads to the propagation of the addition of type `B` from the return value of `M2` to `x`. ■

5.4 Deployment Details

Our analysis is deployed in Azure as illustrated in Figure 5.5. On the left, there is the analysis client such as an IDE or a code editor like SublimeText. The cluster we used consists on one *front-end VM* and a number of worker VMs. The client used REST requests to communicate to the front-end VM. The job of the front-end VM is to:

- Accept and process external analysis client requests.
- Dispatch jobs to the worker VMs and process the results.
- Provide a Web UI with analysis results and statistics.

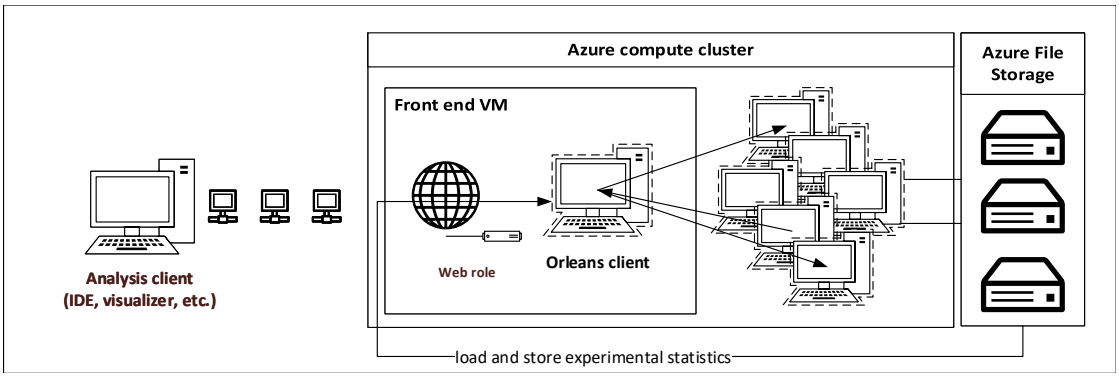
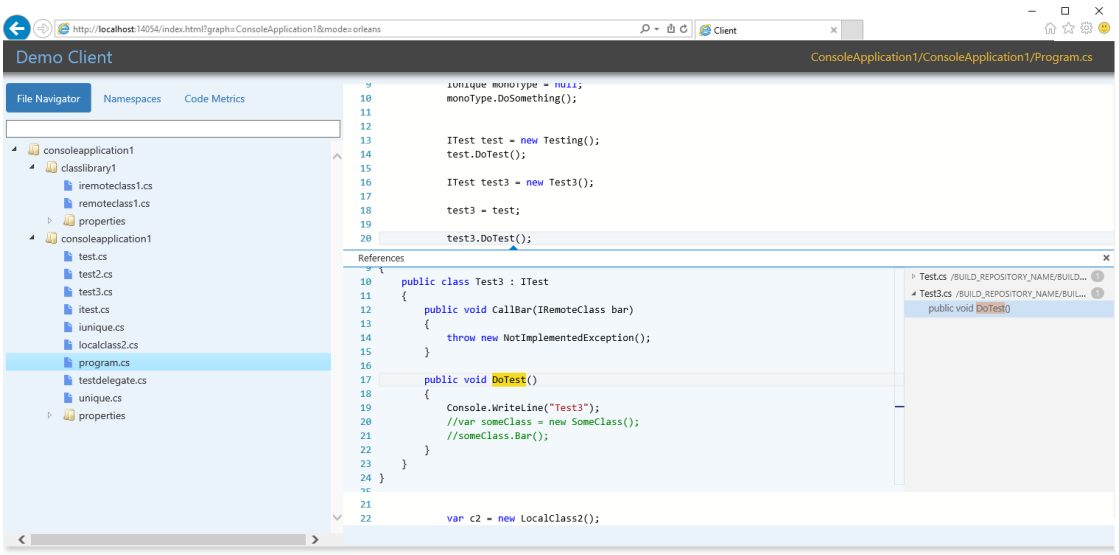


Figure 5.5: Cloud-based deployment of our analysis in Azure. Actual work happens within worker VMs. The analysis client interacts with the cluster via a front-end VM.

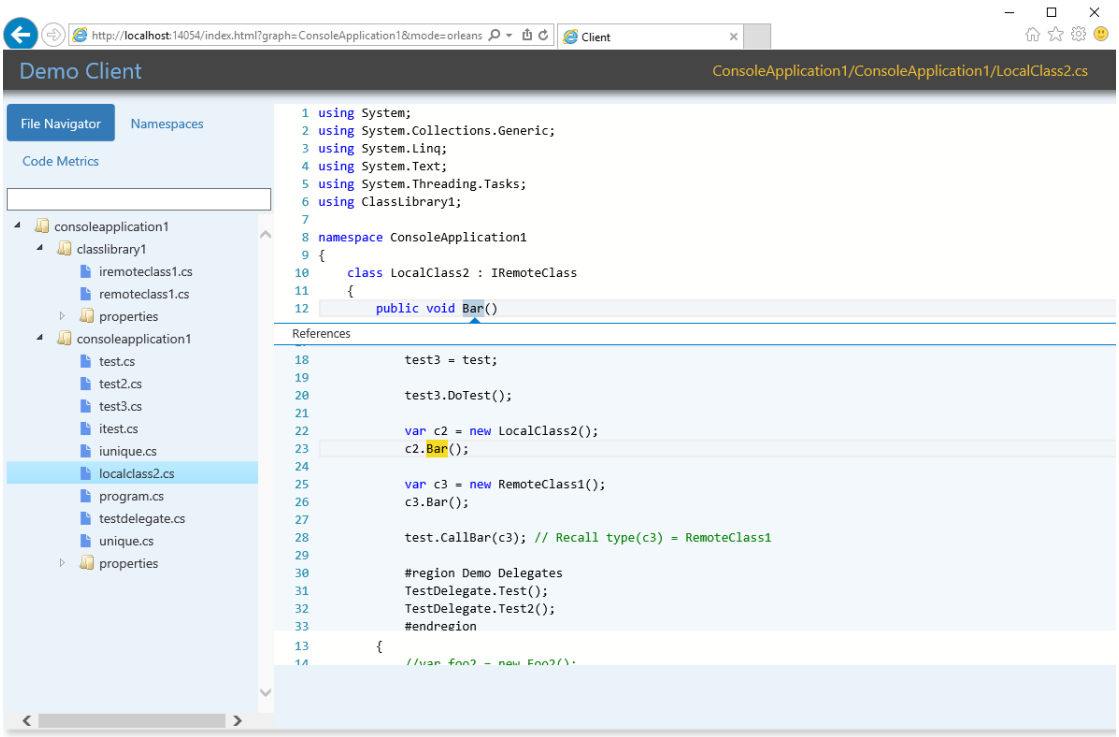
Interactive deployment within an IDE. In Figure 5.6 we show two screenshots of an experimental IDE prototype that uses the API exposed by our analysis to resolve callers/callees queries[3]. We should point out that the precision achieved by our analysis is enough for the autocomplete task.

REST interface. In Table 5.1, we show several typical REST requests for common IDE navigation tasks. The API is designed for use with a variety of clients; for a task such as getting all references to a symbol, we simply package up the name of the symbol into a string and dispatch the request.

[3]Another example of such an IDE can be found at: `http://source.roslyn.io`.



(a) Visualizing callees: call site on line 20 invokes function `DoTest` defined on line 17.



(b) Visualizing callers: method `Bar` defined on line 12 is called on line 23.

Figure 5.6: An experimental online IDE that use our analysis for resolving references for callees and callers.

Client task	REST URL	Server-side request handler

Get all abstract locations in a source code document.	`http://<hostname>:49176/api/Orleans?filePath=program.cs`	`[HttpGet]`
		`async Task<IList<FileResponse>>`
		`GetFileEntitiesAsync(string filePath)`

Get symbol references.	`http://<hostname>:49176/api/Orleans?id=Program.Main`	`[HttpGet]`
		`async Task<IList<SymbolReference>>`
		`GetReferencesAsync(string id)`

Get symbol definitions.	`http://<hostname>:49176/api/Orleans?id=Program.Main@2`	`[HttpGet]`
		`async Task<IList<SymbolReference>>`
		`GetReferencesAsync(string id)`

Table 5.1: Examples of interacting with the analysis backend via REST queries.

Resumen

Implementación

Implementamos un prototipo de nuestro enfoque distribuido[4] para analizar proyectos de gran escala escritos en C#. Ese prototipo se basa en Roslyn [84], un *framework* de compilación para el análisis de código C# y el *framework* Orleans [38], una implementación de un modelo de actores distribuido que se puede desplegar en la nube. Aunque, otras opciones de despliegue como AWS son posibles, utilizamos Azure como plataforma para ejecutar nuestros experimentos.

Orleans y el Modelo de Actores

Orleans [38] es un *framework* diseñado para simplificar el desarrollo de aplicaciones distribuidas. Se basa en la abstracción de los *actores* virtuales. En la terminología de Orleans, estos actores se llaman *granos*. Orleans resuelve una serie de problemas complejos de sistemas distribuidos, como decidir dónde (es decir, en qué máquina) crear un actor determinado, enviar mensajes a través de máquinas, etc., lo que libera en gran medida a los desarrolladores de tener que lidiar con esas preocupaciones. Al mismo tiempo, Orleans está diseñado para permitir aplicaciones que tienen un alto grado de capacidad de respuesta y escalabilidad. Los granos son los componentes básicos de las aplicaciones de Orleans y son las unidades de aislamiento y distribución. Cada grano tiene una identidad global única

[4]Código fuente y *benchmarks* disponibles en: `https://github.com/edgardozoppi/call-graph-orleans`.

que le permite a Orleans enviar mensajes entre actores. Un actor encapsula tanto el comportamiento como el estado local mutable. Actualizaciones de estado de los granos se pueden iniciar enviando mensajes.

Orleans decide en qué máquina física (*silo* en la terminología de Orleans) debe ejecutarse un grano determinado, considerando problemáticas como la presión de memoria, la cantidad de comunicación entre granos individuales, etc. Este mecanismo está diseñado para optimizar la localidad en las comunicaciones porque incluso dentro del mismo *cluster*, la cantidad de mensajes entre máquinas es considerablemente menor que la cantidad de mensajes locales, dentro de la misma máquina.

Chapter 6

Evaluation

We aim to answer the following research questions.

RQ1: Is our analysis capable of handling arbitrary amounts of input (i.e., more lines of code, files, projects, etc.) by increasing the number of worker VMs, without running out of memory?

RQ2: While the communication overhead can become significant, as more worker VMs are added, does an increase in the number of worker VMs significantly increase the overall analysis times?

RQ3: Is the analysis query latency small enough to allow for interactive use[1]?

RQ4: Does our incremental analysis show significant time savings in comparison to the exhaustive version of the analysis?

The focus of our analysis is on being used in an interactive setting. Given the low latency times we can use our analysis interactively as a replacement of source code browsers[2]. This kind of browsers provide code search and basic navigation facilities, but lack more advanced features like actual callers/callees inspection/navigation that we can provide with our analysis. At the same time, we are not as concerned about the completion time for the analysis as a whole as

[1] Generally, query latencies of 10 to 20 ms are considered to be acceptable.

[2] Such as `http://source.roslyn.io`.

we are about its memory requirements on legacy VMs. Even if it takes longer to process, our goal is to engineer an always-on system that responds to messages sent to the cloud to service user requests, in the context of code browsing and other tasks listed in Section 2.2. This work was performed in collaboration with the Roslyn compiler team and while we have not performed user studies, we believe latency numbers (most queries took under 20 ms) to be more than acceptable for interactive use.

6.1 Experimental Setup

All the experiments presented in this chapter were executed in the cloud, on a commercially available Azure cluster. We could also have used an AWS cluster, as our dependency on Azure is small. The Azure cluster we used for the experiments consists on one front-end VM and up to 64 worker role VMs. The front-end VM is an Azure VM with 14 GB of RAM (this is an A4\ExtraLarge VM in Azure parlance[3]). Each worker role is an Azure VM with 7 GB of RAM (called A3\Large in Azure). For benchmarking purposes, we run our analysis with configurations that include 1, 2, 4, 8, 16, 32 and 64 worker VMs. To collect numbers, we used a custom-written experimental controller as our analysis client throughout this section; this setup is illustrated in Figure 5.5. The controller is scripted to issue commands to analyze the next `.sln` file, collect timings, etc.

We heavily instrumented our analysis to collect a set of relevant metrics. We instrumented our analysis code to measure the analysis elapsed time. We introduced wrappers around our grains (solution, project and method grains) to distinguish between local messages (within the same VM) and network messages. Using Orleans-provided statistics, we measured the maximum memory consumption per VM. Lastly, we also have added instrumentation to measure query response times. While these measurements are collected at the level of an individual grain, we generally wanted to report aggregates. To collect these, we post grain-level statistics to a special auxiliary grain.

[3]Up-to-date VM specifications are available at: `https://azure.microsoft.com/en-us/documentation/articles/virtual-workerVMs-size-specs/`.

To evaluate the incremental analysis we made use of 100 already existing and consecutive commits taken from each of the real-world benchmark Git repositories. We wrote a script that runs Git commands in order to checkout the commits, get the list of modified files between the current and previous version of the source code and execute in sequence both the exhaustive and incremental analysis for each commit. To simplify this process and measure the benefits of the incremental analysis in comparison to the full analysis version in a controlled way, we decided to run both analysis on a single machine configuration.

6.2 Benchmarks

For our inputs, we have used two categories of benchmarks, *synthetic* benchmarks we have generated specifically to test the scalability of our call graph analysis and a set of 3 real applications written in C# that push our analysis implementation to be as complete as possible, in terms of handling tricky language features such as `delegate`, `lambdas`, etc. and see the impact of dealing with polymorphic method invocations. In all cases, we start with a solution file (`.sln`) which references several project files (`.csproj`), each of which in turn references a number of C# source files (`.cs`).

Synthetic benchmarks. We designed a set of synthetic benchmarks to test the scalability of our analysis approach. These are solution files generated to have the requisite number of methods (for the experiments, we ranged that number between 1,000 and 1,000,000).

The Table 6.1 summarizes some statistics about the synthetic projects we have used for this evaluation. Synthetic benchmarks were generated to have the requisite number of methods, organized in classes and projects according to a maximum predefined number. Each method invokes between 1–11 other methods, with the only requirement that all methods be reachable. While synthetic programs measure the input size in a controlled way (e.g., LOCs, methods, invocations), the real benchmarks measure the overall complexity (e.g., polymorphism, complex program constructs).

Benchmark	LOC	Projects	Classes	Methods
X1,000	9,196	10	10	1,000
X10,000	92,157	50	50	10,000
X100,000	904,854	100	100	100,000
X1,000,000	9,005,368	100	100	1,000,000

Table 6.1: Information about synthetic benchmarks.

Real-world benchmarks. We have selected several large open-source projects from GitHub for our analysis. A summary of information about these programs is shown in Table 6.2. We tried to focus on projects that are under active development. To illustrate, one of our benchmarks, Azure Powershell is one of the most popular projects written in C# on GitHub. According to the project statistics, over a period of one month, 51 authors have pushed 280 commits to the main branch and 369 commits to all branches. There have been 342,796 additions and 195,366 deletions. Generally, discovering good root methods to serve as starting points for the call graph analysis is not trivial. Because there is no natural `Main` method in several of these projects, we have decided to use as entry points the included *unit tests*, *event handlers* and other public methods within the project to increase the number of methods our analysis reaches[4].

Table 6.3 shows summarized information related to the 100 consecutive commits used to compare the incremental algorithm with the full analysis version, for each real-world benchmark. The numbers correspond to per commit statistics. We only consider source code documents modified by the commits, other kind of files are ignored. For this reason, a couple of commits that do not modify source code files are reported with zero modified documents, and consequently, no modified methods.

6.3 Results

[4]Note that we do not analyze libraries provided as DLLs; our analysis implementation works at the source code level only.

Benchmark	URL https://github.com/	LOC	Projects	Classes	Methods	Main	Test	Event handlers	Public	Total	Reachable methods
Azure-PW	Azure/azure-powershell	416,833	60	2,618	23,617	0	997	1	18,747	18,759	23,663
ShareX	ShareX/ShareX	110,038	11	827	10,177	2	0	1,122	6,257	7,377	10,411
ILSpy	icsharpcode/ILSpy	300,426	14	2,606	25,098	1	0	119	14,343	14,498	21,944

Table 6.2: Summary of information about real-world projects from GitHub. The number of reachable methods include also library methods invoked by the application methods. Note that some application methods might not be reachable.

	Modified documents			Modified methods		
Benchmark	Average	Minimum	Maximum	Average	Minimum	Maximum
Azure-PW	9.49	0	141	37.83	0	876
ShareX	1.74	0	9	6.17	0	158
ILSpy	9.34	0	224	82.26	0	4361

Table 6.3: Information about 100 consecutive commits for each real-world project.

[RQ1]: Scales with input size. To answer RQ1, we measured the memory consumption of each VM and computed the average and maximum memory consumption across all VMs. Figure 6.1 shows the *average* memory consumption for each benchmark during the run, for each experimental configuration (i.e., number of worker VMs used). To give an aggregate perspective of the effect that adding more VMs to the mix has on memory pressure, an additional figure in the Appendix also shows the memory consumption *averaged* across all benchmarks shown for every cloud configuration. As can be observed from the chart, the memory consumption decreases steadily as the number of worker VMs increases. Recall that worker VMs come equipped with 7 GB of memory, so these memory consumption numbers are nowhere near that limit. Looking at Figure 6.1, we can see peaks of about 3.2 GB for a single worker VM while analyzing X1,000,000[5].

[5]Note also that for that benchmark, we needed to use at least 16 worker VMs to fit all the methods into (their shared) memory. We needed at least 4 worker VMs for X100,000.

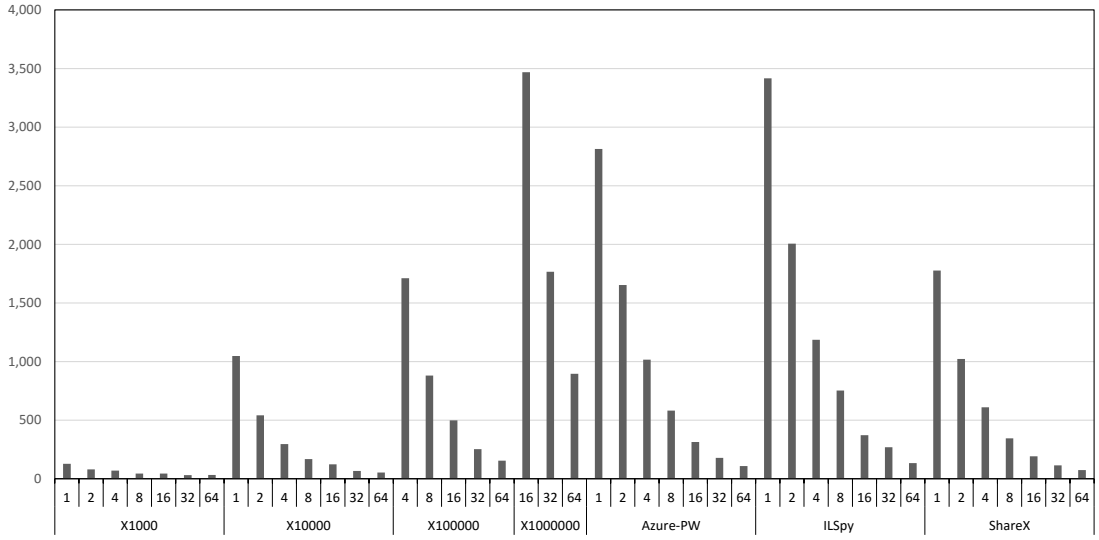


Figure 6.1: Average memory consumption in MB, for each benchmark as a function of the number of worker VMs. We see a steady decrease across the board.

These experiments naturally highlight the notion of analysis *elasticity*. While we run the analysis with different number of VMs set for the sake of measurement, in reality, more machines would be added (or removed) due to memory pressure (or lack thereof) or to respond to how full analysis processing queues get. We can similarly choose to increase (or decrease) the number of queues and dispatchers involved in effect propagation. It is the job of the Orleans runtime to redistribute the grains to update the system with the new configuration.

RQ1: Is capable of handling input size?

The memory consumption per worker VMs steadily decreases as the number of worker VMs increases.

[RQ2]: Scales with the number of worker VMs. To answer RQ2, we proceeded to measure the total elapsed analysis time for each benchmark on all the configurations. Figure 6.2 shows the elapsed analysis time *normalized* by the num-

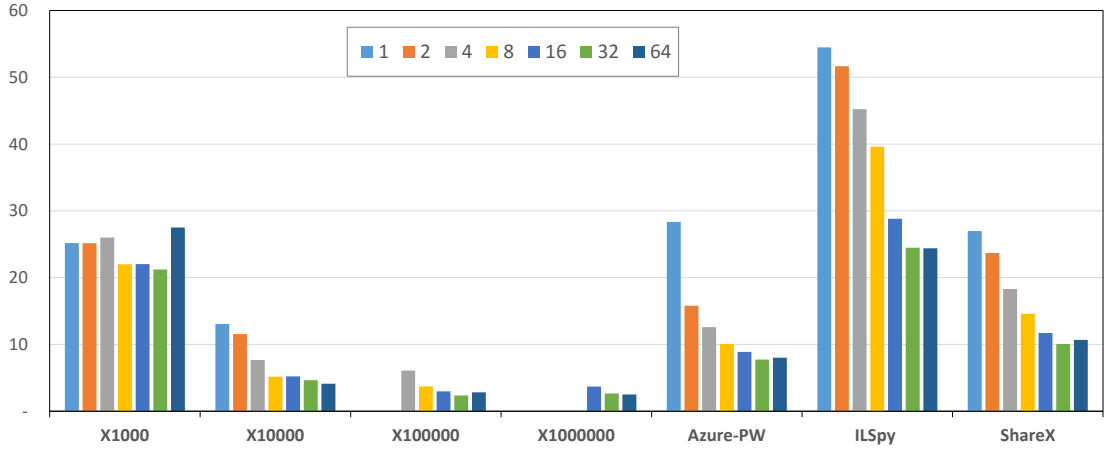
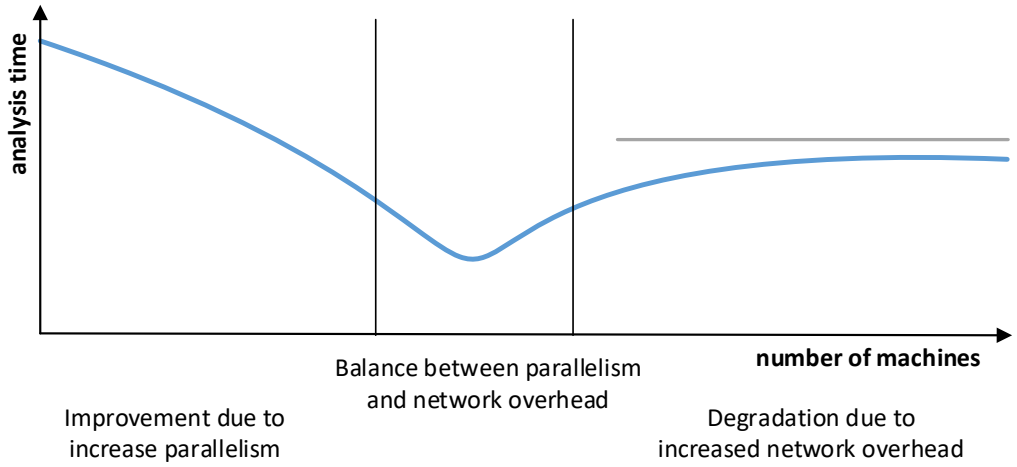


Figure 6.2: Elapsed analysis time in *ms*, as a function of the number of worker VMs per test, normalized by the number of reachable methods. The number of worker VMs is indicated in color in the legend above the figure.

ber of methods in the input[6]. A figure in the Appendix shows the overall analysis and compilation times; the latter can be quite substantial (i.e., about 3 minutes to compile the larger benchmarks such as X100,000 and Azure-PW).

Note that the real-world benchmarks shown on the right-hand side of the chart, despite containing fewer methods, require more time than the synthetic benchmarks with 100,000 methods. This is simply because of the analysis time that goes into analyzing more complex method bodies. Real-world benchmarks allocate more objects per method, involving more type propagation time, and perform more virtual invocations, adding to the method resolution time, while the synthetic benchmarks only perform static invocations and allocate relatively few objects. As the number of worker VMs increases, we see a consistent drop in the normalized analysis times. However, this effect generally diminishes after 16 VMs. This has to do with the tension between more parallel processing power of more machines and the increase in the network overhead, as shown below.

[6]Wall clock times range between less than 1 minute (64 VMs) to about 5 minutes (1 VM) in ShareX and 9 to 20 minutes in ILSpy. For other benchmarks elapsed time is typically less than 5 minutes for 16 VMs, except X1,000,000 that takes about 1 hour (40 minutes in 64 VMs).



It is instructive to focus on the average number of (unprocessed) messages in the analysis queues. If the queues are *too full*, adding more machines will increase the number of queues, reducing the size of each one. More machines will increase the parallelism because of more dispatchers to process the messages in the new queues. As we add more resources, however, when the queues become *mostly empty*, their associated dispatchers will be mostly idle. So the cluster as a whole will have more computing resources than needed. Additionally, if more machines are added, the probability of sending a message to a grain on the same machine as the sender will be reduced, leading to more network overhead. So after reaching a certain cut-off point, adding more machines is not only not helping the analysis, but starts to degrade its performance.

RQ2: Does adding more worker VMs increase analysis time?

Normalized analysis time generally *decreases*, as the number of worker VMs increases, up to a point, where the law of diminishing returns kicks in.

[RQ3]: Fast enough for interactive queries. One of the goals of our approach is to enable interactive queries submitted by an analysis client such as

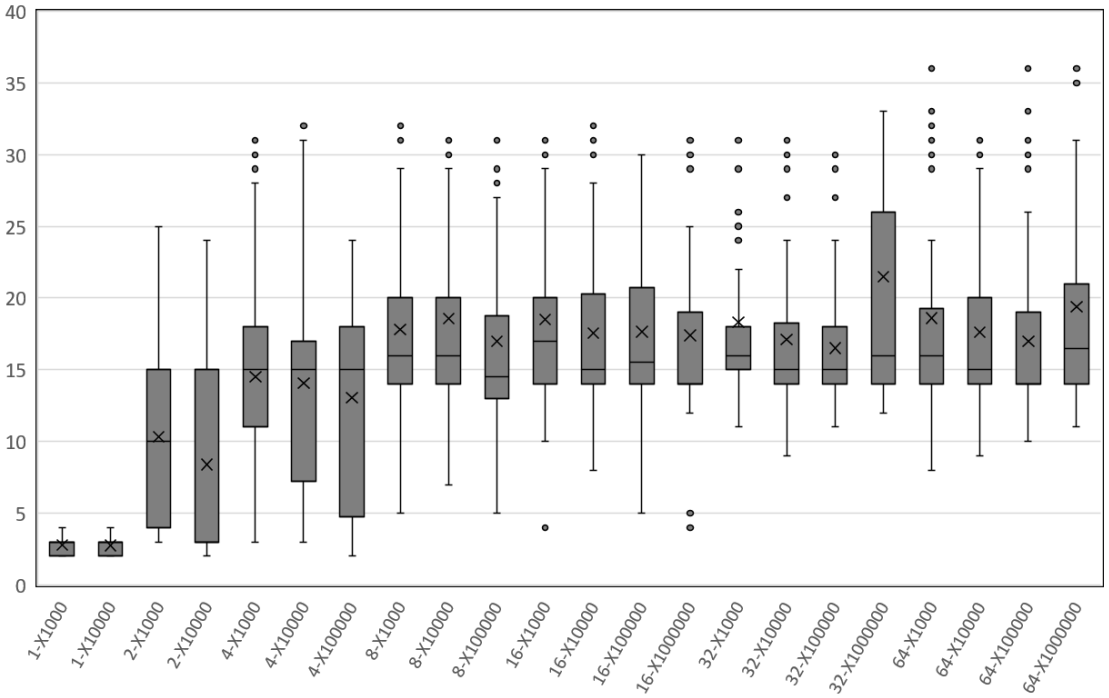


Figure 6.3: Mean and median query time in *ms* for each worker VM and synthetic test.

an IDE or a sophisticated code editor. In such a setting, responsiveness of such queries is paramount [89]. The user is unlikely to be happy with an IDE that takes several seconds to populate a list of auto-complete suggestions. We want to make sure that as the query times remain tolerable (under 20 ms) even as the size of input increases and the number of VMs goes up.

To evaluate query performance, we automatically generated sequences of 100 random queries, by repeating the following process. We would first pick a random method name from the list of all methods. Then we would:

1. Request the solution grain for the corresponding method grain.
2. Select a random invocation from method grain and request the set of potential callees.

In Figure 6.3 we show the mean and median query times (the latency of the two steps above) for each benchmark and worker VM configuration. Approxi-

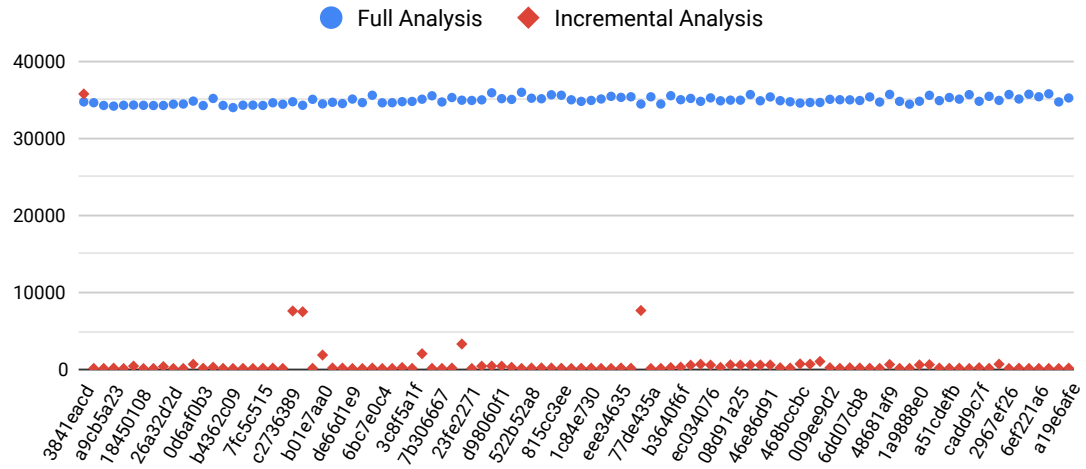
mately 70% of queries took under 20 ms, 97% under 35 ms, 99.5% under 60 ms. Proper system warm-up may reduce the outliers.

RQ3: Is response latency small enough?

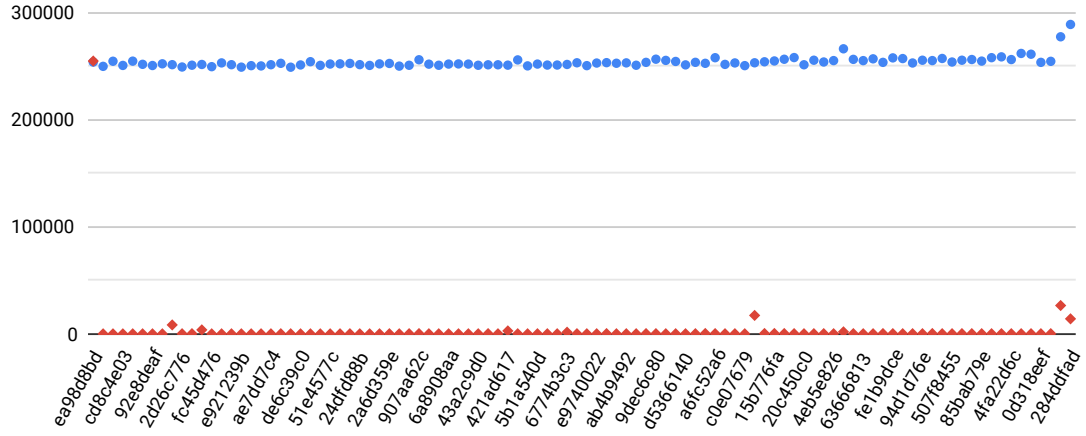
The query median response time is consistently between 10 and 20 ms. Increasing the number of worker VMs and the input size does not negatively affect the query response times.

[RQ4]: Efficient incremental analysis. To answer RQ4, we measured the total elapsed analysis time for both exhaustive and incremental analysis versions on 100 consecutive commits taken from the GitHub repository of each real-world benchmark. In this case we decided to run the experiments on a single machine configuration to isolate the evaluation from other unrelated factors and completely focus in the comparison of both versions of the call graph analysis. In Figure 6.4 we show in a single chart per benchmark, the elapsed times of running the incremental and the full version of the analysis for each commit. Note that the elapsed times shown in the charts do not include the compilation times.

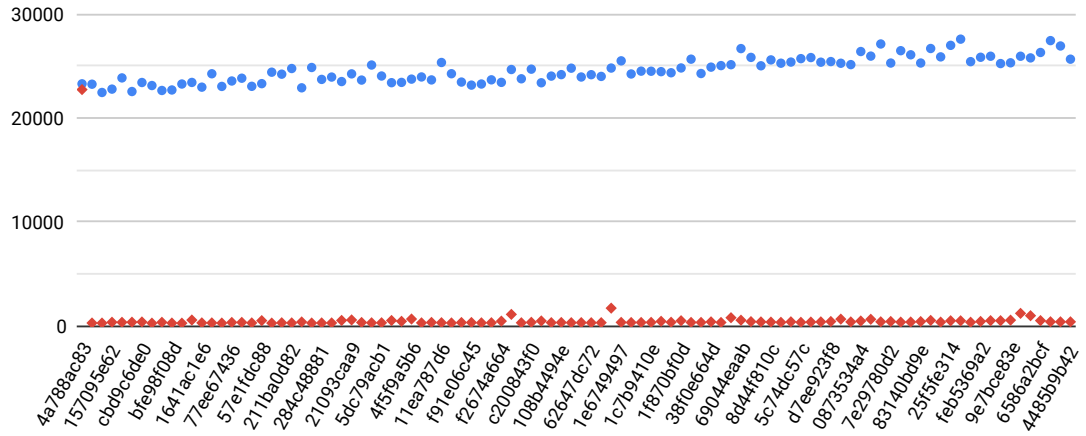
The results show significant time savings for the incremental analysis over the exhaustive version. This can be observed for each commit of each benchmark. The speedup averaged across all real-world benchmarks is about 250X. In the case of ShareX is about 147X, for ILSpy about 543X and for Azure-PW about 61X. However, depending on the amount of methods modified by a particular commit, the incremental analysis propagates more or less information, thus affecting the overall analysis time. This explains the incremental analysis elapsed time outliers present for a few particular commits in all the benchmarks, specially ShareX. For instance, while the average amount of methods modified by ShareX analyzed commits is only 6, there are a few exceptional cases that modify 44 or even 158 methods in the same commit, leading to increased incremental analysis times. But even in those exceptional cases, the time savings are still remarkable.



(a) ShareX.



(b) ILSpy.



(c) Azure-PW.

Figure 6.4: Exhaustive and incremental analysis time in *ms* for 100 consecutive commits of the real-world benchmarks.

RQ4: Is the incremental analysis faster than the full version?

The incremental algorithm shows significant time savings of about 250X average speedup in comparison to the exhaustive analysis.

Resumen

Evaluación

Nos interesa responder las siguientes preguntas de investigación.

RQ1: ¿Nuestro análisis es capaz de manejar cantidades arbitrarias de entrada (es decir, más líneas de código, archivos, proyectos, etc.) al aumentar el número de máquinas virtuales de trabajo, sin quedarse sin memoria?

RQ2: Si bien la sobrecarga de comunicación puede llegar a ser significativa, a medida que se agregan más máquinas virtuales de trabajo, ¿un aumento en el número de máquinas virtuales de trabajo aumenta significativamente los tiempos generales de análisis?

RQ3: ¿Es la latencia de las consultas del análisis lo suficientemente pequeña como para permitir el uso interactivo [7]?

RQ4: ¿Nuestro análisis incremental muestra ahorros de tiempo significativos en comparación con la versión exhaustiva del análisis?

El enfoque de nuestro análisis está en ser utilizado en un entorno interactivo. Dados los bajos tiempos de latencia, podemos usar nuestro análisis de manera interactiva como reemplazo de los navegadores de código fuente[8]. Este tipo

[7] En general, las latencias de consulta de 10 a 20 ms se consideran aceptables.

[8] Tales como `http://source.roslyn.io`.

de navegadores proporciona búsqueda de código y facilidades básicas de navegación, pero carecen de funciones más avanzadas como la inspección/navegación de métodos llamados/métodos llamadores reales que podemos proporcionar con nuestro análisis. Al mismo tiempo, no estamos tan preocupados por el tiempo de finalización del análisis en su conjunto, como sí lo estamos por los requisitos de memoria en máquinas virtuales. Incluso si el proceso toma más tiempo, nuestro objetivo es diseñar un sistema que esté siempre activo y que responda a los mensajes enviados a la nube para atender las solicitudes de los usuarios, en el contexto de la búsqueda de código y otras tareas listadas en la Sección 2.2. Este trabajo se realizó en colaboración con el equipo del compilador Roslyn y aunque no hemos realizado estudios de usuario, creemos que los números de latencia (la mayoría de las consultas son atendidas antes de 20 ms) son más que aceptables para el uso interactivo.

Chapter 7

Related Work

There exists a wealth of related work on traditional static analysis algorithms such as call graph construction [60, 104, 59]. A comparison of analysis precision is presented in Lhoták *et al.* [75]. As mentioned, our implementation is inspired in VTA [103]. While we have seen dedicated attempts to scale up important analyses such as points-to in the literature, we are unaware of projects that aim to bring call graph analysis to the cloud.

Many projects focus on speeding up the analysis through parallel computation (usually on one machine). Instead, we largely focus on handling memory pressure when analyzing large programs. There are two orthogonal ways to do that:

- Develop a compositional analysis using specs/summaries [42, 108], abstractions, compact representations [111], demand-driven [101] and other techniques to scale-up.
- Partition analysis memory among several machines. Our analysis focuses on the latter by presenting an approach designed to run on a standard cluster. The engineering challenges are quite different, including state partitioning, decentralized control, number/size of messages sent, termination and network latency.

Concrete types. Most of the work in concrete type inference for object-oriented programs goes back to the early 1990s [92, 25, 24, 45, 90]. Many of these techniques

are now considered standard. Concrete type inference scales better and generally does not require the same complexity as a content-sensitive points-to analysis [74].

Call graph construction. Call graph construction for object-oriented code was explored in the 1990s, with standard algorithms such CHA and VTA proposed at that time [60, 104, 59]. A comparison of analysis precision is presented in Lhoták *et al.* [75]. Some of the recent work focuses on call graph construction in the presence of frameworks and libraries [80, 30]. We largely skirt that issue in this work, focusing on input being provided to us in the form of source code.

Scaling points-to analysis. Hardekopf *et al.* [63] show how to scale up a points-to analysis using a staged approach. Their flow-sensitive algorithm is based on a sparse representation of program code created by a staged, flow-insensitive pointer analysis. They can analyze 1.9M LOC programs in under 14 minutes. The focus (as alleged by the authors) is in obtaining speedups, not in reducing memory pressure. In fact, their largest benchmark required a machine with 100 GB of memory, which is generally beyond the reach of most people. In contrast, we aim at analyzing large programs in clusters of low-cost hardware.

Hardekopf *et al.* [62] introduce novel techniques for inclusion-based pointer analysis that significantly improve scalability without negatively impacting precision. These techniques focus on the problem of online cycle detection, a critical optimization for scaling such analyses. The combination of their techniques is on average 3.2× faster than Heintze and Tardieu's algorithm [64], 6.4× faster than Pearce *et al.*'s algorithm [91] and 20.6× faster than Berndl [37].

Yu *et al.* [115] propose a method for analyzing pointers in a program level by level in terms of their points-to levels. This strategy enhances the scalability of a context- and flow-sensitive pointer analysis and can handle some programs with over a million lines of C code in minutes. The approach is neither parallel non-distributed, the focus is on speedups but some memory is saved by the use of BDDs.

Mendez-Lojo *et al.* [83] propose a parallel analysis algorithm for inclusion-based points-to and show a speedup of up to 3× on an 8-core machine on code

bases with size varying from 53K LOC to 0.5M LOC. Our focus is on bringing our approach to the cloud using legacy machines and going beyond multi-core, to ultimately support code bases of arbitrary size, not being limited by the size of main memory.

Voung *et al.* [108] propose a technique that uses the notion of a *relative lockset*, which allows functions to be summarized independent of the calling context. This, in turn, allows them to perform a modular, bottom-up analysis that is easy to parallelize. They have analyzed 4.5 million lines of C code in 5 hours, and, after applying some filters, found several dozen races. Knowing which methods to group together ahead of time would help our actor-machine allocation as well.

Sridharan [101] presents an interesting approach for providing query-specific results under time/memory pressure. This idea is orthogonal to our approach and could be interesting to see if the approach can be distributed.

Frameworks. Albarghouthi *et al.* [28] present a generic framework to distribute top-down algorithms using a map-reduce strategy. Their focus is in obtaining speed ups in analysis elapsed times; they admit that a limiting scaling factor is memory consumption and propose distributing their algorithm as future work.

McPeak *et al.* [82] propose a multi-core analysis that allows them to handle millions LOC in several hours on an 8-core machine. In contrast, our approach focuses on a distributed analysis within a cloud cluster on often less powerful hardware.

Xie *et al.* [114] propose a bottom-up analysis that benefits from parallel processing on a multi-core cluster. They rely on a central scheduler/server, while we use several orchestrators. They use method summaries while we flow the data from one method to another. Finally, we do not rely on a centralized DB, we use grains, which can be persisted or recomputed on-the-fly as needed.

Rodriguez *et al.* [94] use an actor model approach in Scala to solve interprocedural distributive subset data-flow problems and evaluate it on an 8-core machine. Our work shares the idea of using actors for analysis but they focused on speed-ups, not memory pressure. Their approach leverages on the use of one

computer to implement a global counter to monitor the size of a (virtual) global worklist. In contrast, we run in a cloud setting and must deal with network latency and serialization due to distribution.

Boa [52, 54, 53] is a domain-specific language for mining large code repositories like GitHub to answer questions such as *"how many Java projects use SVN?"* or *"how many projects use a specific Java language feature over the years?"*. It runs these queries on a map-reduce cluster. However, while it uses a distributed backend, Boa is not a static analysis.

Pregel [81] is a system for large-scale graph processing that uses an asynchronous message passing model similar to actors, but execution on vertices happens in lockstep; the approach is illustrated for algorithms such as PageRank and shortest path computation.

Graspan [109] is a single-machine, disk-based parallel graph processing system for inter-procedural static analyses. It offers two major performance and scalability benefits: the core computation of the analysis is automatically parallelized and out-of-core disk support is exploited if the graph is too big to fit in memory. Our approach focuses on a cloud-based computation, in contrast.

Tricorder [97] is a cloud-based tool from Google, designed for scaling program analysis. However, it is meant for simple, intra-procedural analyses, not distributed whole-program analyses.

Incremental analysis. Souter *et al.* [100] presents an incremental call graph analysis for object-oriented languages like Java. Their approach is based in the Cartesian Product Algorithm (CPA) instead of the Variable Type Analysis (VTA) in which we based our algorithm. For this reason, they avoid propagating types by using a *template* call graph, that has the property that at a given node, every incoming edge carries exactly the same type information. When needed, a traditional call graph can be obtained from the template call graph. A template can be viewed as an instance of a method, with exact dynamic type information for each formal parameter, and representing a particular combination of actual parameter types. In other words, a template is an exact dynamic type signature

of a method. This means that their algorithm creates as many method templates as combinations of actual parameter types are discovered by the analysis. Additionally, in a similar manner to our approach they also distinguish between direct and indirect edits that can affect the call graph of a program. However, they focus in fine-grain edits at the statement level that typically occur during development within an IDE, such as the insertion and deletion of a call site inside a method body. Instead, we opted methods to be the minimal modification unit, leading to a more coarse-grained analysis suitable for handling several edits at once, like typically happens in source control revisions. Finally, their analysis is not built on top of a distributed approach like ours.

Hirzel *et al.* [65] present an online pointer analysis for Java-like programming languages based on Andersen's pointer algorithm. They distinguish between offline, online, incremental and demand-driven analyses. Traditional static analyses like Andersen's pointer algorithm are good offline analysis examples. An online analysis incrementally analyzes new code when it is dynamically loaded into the running program. An incremental analysis updates its results efficiently when the program changes. A demand-driven analysis attempts to compute just the part of the solution that the client is interested in, rather than the exhaustive solution. Their approach focuses in performing pointer analysis online during program execution and differs from our approach in that it uses incrementality to deal with dynamically loaded code, while ours focuses on modifications to the source code during development. Additionally, we concentrate on different problems (pointer analysis and call graph construction).

Liu *et al.* [76] present a parallel incremental algorithm for computing points-to information upon statement insertion, deletion and modification. Their analysis implementation is parallel within each iteration of the fixed-point computation. However, it is not a distributed analysis. Contrary to many other existing Andersen-style incremental pointer analyses for Java-like programming languages, their approach does not assume a pre-built call graph. For the empirical evaluation, incremental updates were artificially simulated by removing and adding statements in each method of the target program. Instead, to demonstrate our

approach we used several consecutive real commits directly taken from the original Git repository of each benchmark. Additionally, our incremental analysis is designed with the slightly different goal of providing a static analysis backend to be run by centralized code repositories such as GitHub. For this reason, our approach recomputes information upon method modifications, in contrast to individual statements.

Resumen

Trabajo Relacionado

Existe una gran cantidad de trabajos relacionados con los algoritmos de análisis estáticos tradicionales, como la construcción de *call graphs* [60, 104, 59]. En Lhoták *et al.* [75] se presenta una comparación de la precisión de los análisis. Como se mencionó, nuestra implementación está inspirada en VTA [103]. Si bien hemos visto en la literatura intentos dedicados a scalar análisis importantes, como *points-to*, no tenemos conocimiento de proyectos que tengan como objetivo llevar el análisis de *call graphs* a la nube.

Muchos proyectos se centran en acelerar el análisis a través de cómputo paralelo (generalmente en una sola máquina). En cambio, nos centramos principalmente en manejar la presión de memoria al analizar programas grandes. Hay dos formas ortogonales de hacer eso:

- Dessarrollar análisis composicionales utilizando especificaciones/resúmenes [42, 108], abstracciones, representaciones compactas [111], dirigido por demanda y otras técnicas para escalar.
- Particionar la memoria del análisis entre varias máquinas. Nuestro análisis se centra en este último presentando un enfoque diseñado para ejecutarse en un *cluster* estándar. Los desafíos de ingeniería son bastante diferentes, incluyendo la partición de estados, el control descentralizado, el número/tamaño de los mensajes enviados, la terminación y la latencia de la red.

Part III

Static Analysis Framework

Chapter 8

Overview

Analysis.NET[1] is an open-source static program analysis framework targeting the .NET platform [23]. It allows the analysis and transformation of CIL bytecode directly from any .NET available programming language and is completely written in C#. It comprises a large set of APIs and a few additional GUI tools that are useful to explore its capabilities without having to write a single line of code. Among other things, the framework includes many classical and well-known static analyses built on top of different kinds of intermediate languages that provide different levels of abstraction. It is designed to be easily extended to support many other user-defined custom analyses and code transformations. The key features of Analysis.NET include a simplified and typed register-based intermediate representation of CIL bytecode, a .NET type system and class hierarchy analysis, a flow-sensitive pointer analysis and call graph construction algorithms and several intra-procedural analyses based on control-flow and data-flow computed information.

Our work is closely inspired by the Java optimization framework Soot [105, 70], but we focus in the .NET platform instead. However, our framework is designed in a more general way to also allow the possibility of porting other similar platforms in the future[2]. Our goal is to enable the static analysis of .NET programs by providing to the .NET community a framework somewhat similar to Soot.

[1] Source code available at: `https://github.com/edgardozoppi/analysis-net`.

[2] We are currently working in a proof-of-concept port that also supports Java.

8.1 Motivation

The creation of this framework was motivated by the poor offer of related tools, and the difficulty of analyzing .NET programs with the few existing available ones, such as Roslyn [84], CCI [15], Cecil [8] and ILSpy [16]. These tools have many flaws regarding the static analysis point of view. Some of them are deprecated, not up to date with the latest language features and currently unsupported. Others focus in a somewhat related but different task, such as compiling source code or decompiling bytecode. Non of them were designed with the goal of allowing client applications to implement their own custom static analyses. As consequence, all of them work either at the abstract syntax-tree level or at the bytecode level. Non of them provide an intermediate code representation suitable for implementing a static analysis [67, 27].

On the one hand, using a tree-based representation which is close to the original source code forces static analysis developers to consider and handle high-level language constructs with complex semantics (including syntactic sugar). Moreover, the nested nature of trees allows many possible combinations of those features, adding an unnecessary extra degree of complexity.

On the other hand, directly analyzing .NET CIL bytecode is a complex task. Although it is possible to construct a control-flow graph for CIL bytecode, the implicit stack masks the flow of data and thus makes the bytecode quite difficult to analyze. In particular, at a given bytecode instruction x, it is not at all obvious which previous instructions produced the stack-based inputs of x. Storing data in named local variables, rather than on the implicit stack, makes the local flow of data explicit and consequently, much more obvious.

For these reasons, we decided to develop a static analysis and code transformation framework for programs targeting the .NET platform. It is completely written in C# and its source code is publicly available under a very permissive open-source license. One of the main features provided by our framework is the implementation of a three-address code intermediate representation that greatly simplifies the task of developing a static program analysis [67, 27].

8.2 Code Representations

There are different ways to represent code. Each one designed for different purposes. Some of them are easier to understand by developers because they are written in a human-readable form, while others are specifically meant to be easily processed by a computer automatically. According to the criteria of how easy is for a developer to understand a particular representation, they can be classified in three categories: high, intermediate and low-level code representations. From the static analysis point of view, some are more suitable than others [67, 27]. Next we list the most commonly used code representations, with a more detailed pros and cons discussion from the static analysis perspective for the intermediate ones.

8.2.1 High-level Representations

Source code. Is the representation of software as it is originally written by a developer using a human-readable programming language, usually in plain text [27]. It can be directly interpreted, and thus immediately executed, by an *interpreter* or *virtual machine*, or translated by a *compiler* into one of the following lower-level code representations to be further processed and possibly optimized.

Concrete syntax tree. Also known as parse tree [27], is a tree representation of the syntactic structure of source code written in a specific programming language. Each node of the tree denotes a construct occurring in the source code. They are typically built by a parser during the source code translation and compiling process. They are considered high-level representations because they are still very coupled to the specific syntax of a particular programming language. For instance, they include all the syntactic symbols, delimiters and even white-spaces present in the original source code, so it can be completely reconstructed without any loss by just traversing the tree. Good examples of this code representation are the `CSharpSyntaxTree` and `VisualBasicSyntaxTree` data structures used by the Roslyn [84] C# and Visual Basic compiler.

8.2.2 Intermediate Representations (IR)

Abstract syntax tree. Often abbreviated to AST [27], is very similar to the concrete syntax tree but with one important difference: it is *abstract*, in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural, content-relevant details. For instance, grouping parentheses are implicit in the tree structure and a syntactic construct like an `if-then-else` statement may be denoted by means of a single node with just three children (condition expression, then block and else block), ignoring syntactic delimiters like curly braces, semicolons, etc. Sometimes, repeating nodes (e.g., different nodes denoting the same symbol) are reused to save memory, leading to a directed acyclic graph (DAG) instead of a tree.

From the program analysis point of view, this kind of IR has several aspects that makes it inadequate for this purpose. Since its structure follows very closely the original source code, they usually have several nodes with complex high-level semantics, that can be arbitrarily nested because of the tree nature of this IR. The result is a very complex combination of different kinds of nodes that the developer of a static analysis algorithm will have to consider. Moreover, ASTs usually represent language constructs of very similar semantics with different nodes (e.g., specific nodes for each kind of loop, like `while` and `for` loops, that are essentially the same), adding an unnecessary extra degree of complexity. Finally, control-flow is represented implicitly using the semantics associated to each particular kind of node, without explicit jumps to explicitly defined targets. A good example of this IR is the `IOperation` data structure used by the Roslyn [84] C# and Visual Basic compiler to represent the statements and expressions of both programming languages with the same data type.

Advantages	Disadvantages
• Close to original source code.	• Nodes with complex, high-level semantics. • Many different nodes with similar semantics. • Arbitrarily nested tree structure. • Implicit control-flow.

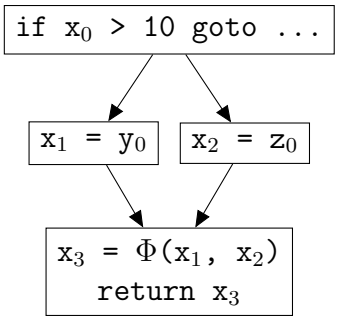
Three-address code. Often abbreviated to TAC [27], is a register-based sequential representation of a program. Its name derives from the fact that each instruction has at most three operands (or addresses) and is typically a combination of an assignment and a binary operator, like in `a = b + c`. It is usually automatically generated from a higher-level code representation like AST by optimizing compilers to aid in the implementation of code-improving transformations. However, it can also be automatically constructed from lower-level code representations like bytecode, making this IR suitable for scenarios where the original source code is not completely available. Complex higher-level constructs are decomposed into a sequence of simpler lower-level instructions during the translation process. Every intermediate expression result is assigned to a temporal variable. Control-flow is explicitly represented using jumps to target labels, and is the only way to alter the sequential flow.

Static single assignment form. Often abbreviated to SSA [95, 49], is a more restricted variant of TAC that is very popular because of its additional benefits that greatly simplify many static analyses. It requires that each variable is assigned exactly once, and every variable is defined before it is used. Existing variables in the original TAC are split into versions (i.e., new variables typically indicated by the original name with a subscript), so that every definition gets its own version. In SSA form, *use-def chains* are explicit and each one contains a single element. An special instruction called Φ (*Phi*) instruction is added whenever multiple versions of a variable reach the same use. This instruction generates a new variable definition by *choosing* the corresponding reaching version, depending on the control-flow taken. The main benefit provided by this IR is that variables become immutable: once a value is assigned to a variable, it cannot be changed (remember only one definition for each variable is allowed in this form).

Example 7. Consider the code fragment shown in Figure 8.1a. Depending on a condition the variable `x` is assigned different values and returned. Is easy to see that both definitions of `x` at lines 2 and 4 reach the use at line 7.

```
1   if (x > 10) {
2     x = y;
3   } else {
4     x = z;
5   }
6
7   return x;
```

(a) Original source code.



(b) Static single assignment form.

Figure 8.1: Original source code and corresponding static single assignment form for Example 7.

Since SSA form requires each variable to be defined only once, different versions of variable `x` are created for those assignments, x_1 and x_2 respectively. This has the consequence of not having a unique version of `x` to use in the return at line 7. To solve this problem a Φ instruction $x_3 = \Phi(x_1, x_2)$ is inserted before that line, defining a new version x_3 that can be safely returned. Intuitively, the idea behind the semantics of a Φ instruction is to automatically select the correct version depending on the control-flow taken. In this case, if the execution follows the left branch, x_1 will be assigned to x_3, but if the execution follows the right branch, x_2 will be assigned instead. Figure 8.1b shows the resulting code in SSA form. ■

From a static analysis perspective, TAC provides a lot of benefits making it the most adequate for this purpose [67, 27]. It usually has a few number different kinds of instructions, each one with simple and very specific low-level semantics. Since it has a simpler sequential structure in comparison to tree based IRs like AST, there is no complex nested combination of different kinds of instructions. Since it is a register-based IR there is no evaluation stack. Operands are always explicit and can only be variables, with the exception of the `load` and `store` instructions that are the only ones that allow other kind of values like literals, field references and indexed arrays. This restriction on operands makes *def-use and use-def chains* computation straight forward. Control-flow analysis is also quite simple due to the

sequential flow and the presence of explicit branching instructions. The additional restrictions imposed by the SSA form simplify several static analyses even further since there is no need to keep track of variables' values for each program location thanks to the immutability guarantee provided by this form.

Advantages	Disadvantages
• Instructions with simple, low-level semantics • Few instructions with different semantics. • Sequential structure. • Explicit control-flow. • Explicit data-flow. • Register-based. • Close to generated bytecode.	• Far from original source code.

Bytecode. Is a compact machine-independent cross-platform assembly language, used as target of the compilation of high-level programming languages [27]. Its name derives from instruction sets that have one-byte opcodes followed by optional parameters, akin to traditional hardware instructions. It may often be either directly executed on a virtual machine, or it may be further compiled into machine code for better performance. Bytecode is typically stack-based, but it can also be register-based or a mix of both. In the case of CIL, it is entirely stack-based. Even though bytecode instructions are usually typed, the evaluation stack is not, so the same slot can store values of different types during the execution of a method.

From the program analysis point of view, bytecode presents similar characteristics to TAC but with some distinct aspects that makes it also inadequate for this purpose. Since it is designed to be a compact code representation, inspired in traditional hardware instruction sets, each instruction has a very specific and simple low-level semantics. Like most assembly languages, bytecode also has a sequential control-flow with explicit branching instructions to labeled targets. This makes static analyzers easy to follow the program's control-flow. However, the usage of an evaluation stack makes following the program's data-flow very difficult, since instructions' operands are often implicit (i.e., indirectly pushed into the stack by

some other instructions, not necessarily located near the one that consumes the values). Even more, given that stack slots are not typed, in order to ensure type-safety operations it is often the case that different variants of the same instruction exists, each one supporting a different combination of operands' data types.

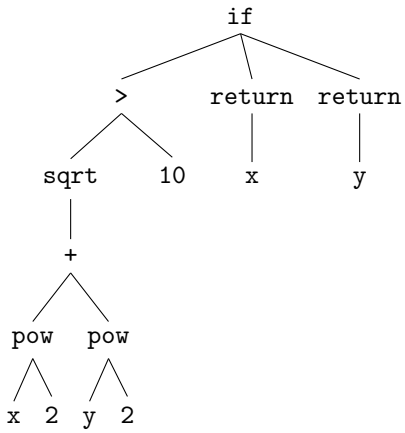
Advantages	Disadvantages
• Instructions with simple, low-level semantics. • Sequential structure. • Explicit control-flow. • No need of original source code.	• Far from original source code. • Many instructions for different types. • Stack-based. • Non-typed stack slots. • Implicit data-flow.

Example 8. Consider the source code fragment shown in Figure 8.2a. It computes a complex algebraic condition that performs nested operations on the results of other complex expressions.

Abstract syntax tree. As we can see in Figure 8.2b, its corresponding AST closely follows the syntactic structure of the original source code while ignoring the unimportant specific syntax details (e.g., parentheses, curly braces and semicolons), preserving only the relevant parts of the code. However, its arbitrarily nested tree structure allows to combine different kinds of nodes in many ways, making very difficult for the developer of a static analysis algorithm to consider all possible combinations. For instance, the operands of operators like `+` can be either literal numbers, variables or even the result of method invocations (in a more complex scenario they can also be field access paths and indexed arrays). The same happens for method invocation arguments, or more generally, in any place were an expression is expected. Another problem is the implicit control-flow given by the `if` AST node. There are no explicit jumps to defined labels, so flow has to be simulated by a static analysis algorithm depending on the kind of node currently processed. In this particular case, the condition node `>` has to be analyzed first, followed by the two other remaining child nodes. Analyzing tree nodes in the right order is very important because they could produce side-effects.

```
1
2 if (sqrt(pow(x,2)+pow(y,2))>10)
3 {
4   return x;
5 }
6 else
7 {
8   return y;
9 }
10
11
12
13
```

(a) Original source code.



(b) Abstract syntax tree.

```
1      t0 = 2
2      t1 = pow(x,t0)
3      t2 = pow(y,t0)
4      t0 = t1 + t2
5      t0 = sqrt(t0)
6      t1 = 10
7      if t0 <= t1 goto L1
8      return x
9  L1: return y
10
11
12
13
14
```

(c) Three-address code.

```
1      ldc.r4 2
2      ldloc.0
3      call   pow(float,float)
4      ldc.r4 2
5      ldloc.1
6      call   pow(float,float)
7      add
8      call   sqrt(float)
9      ldc.r4 10
10     ble.s  L1
11     ldloc.0
12     ret
13 L1: ldloc.1
14     ret
```

(d) CIL bytecode.

Figure 8.2: Original source code and corresponding abstract syntax tree, three-address code and CIL bytecode for Example 8.

Three-address code. Figure 8.2c shows its corresponding TAC representation. The complex nested algebraic expression is decomposed into a sequence of very simple instructions (lines 1 to 5), each having at most three operands. Is important to note that each intermediate expression result is assigned to a temporal variable to be used as operand of other instructions. So both expression results and operands are explicit. The control-flow is also explicitly represented with a conditional jump at line 7.

Bytecode. Its corresponding bytecode is shown in Figure 8.2d. At first sight it seems similar to the TAC version: a sequence of simple instructions with an explicit jump to a target label at line 10. However, after looking in more detail is easy to see some important differences. Local variables and parameters are referenced by their corresponding indexes in the respective dedicated tables. There are different kinds of load instructions, depending if we are loading onto the evaluation stack a literal value (lines 1, 4 and 9), the content of a local variable (lines 2, 5, 11 and 13) or a method parameter. Even more, there are different instruction versions for loading literals of different data types (the modifier `r4` of the `ldc` instruction at lines 1, 4 and 9 is used to load 32-bits floating point numbers). The same happens for other kinds of instructions. Since it is a compact stack-based IR, operations like addition (line 7), conditional branching (line 10) and method calls (lines 3, 6 and 8) implicitly take their operands and store their results from and onto the top of the evaluation stack, respectively. So is not easy to realize which are the operands of a particular instruction. For instance, consider the case of the `add` instruction at line 7. In order to determine the values that are being added together is necessary to first follow the stack behavior of all the instructions that will be executed at run-time before the addition. ■

8.2.3 Low-level Representations

Assembly. Consist in a set of symbolic processor instructions and meta-statements, such as assembler directives, macros and symbolic labels of both code and memory locations. Assembly language uses *mnemonics* [27] to represent low-level machine

instructions or opcodes. Many operations require one or more operands in order to form a complete instruction. It may also be called symbolic machine code since there is a very strong correspondence between the assembly language instructions and the hardware architecture's machine code instructions. Each assembly language is specific to a particular computer architecture and operating system. Assembly code is converted into executable machine code by an *assembler* utility program that is similar to a very simple compiler. Nowadays, it is typically used to implement programs with very demanding performance requirements that need to be optimized for speed or size.

Machine code. Also known as binary code [27], is a strictly numerical code format which is designed to be executed as fast as possible by a particular family of processors, and may be regarded as the lowest-level representation of a compiled or assembled hardware-dependent computer program. While it is possible to write programs directly in machine code, it is tedious and error prone to manage individual bits and calculate numerical addresses and constants manually. For this reason, programs are very rarely written directly in machine code in modern contexts.

8.3 Framework Design Principles

Our framework implementation adheres to the following design principles.

- **Bytecode-based approach.** To allow the analysis of programs written in any programming language available for the .NET platform, and at the same time, automatically support new high-level language constructs and features (typically, newly added *syntactic sugar*[3]). An additional advantage of using CIL bytecode as input is to enable the analysis of programs for which their source code is partially or completely unavailable.

[3]For instance, a lot of new features were added to C# since its very first version, but CIL bytecode almost always remained unchanged. The only exception was the compatibility break introduced when adding support for generics in version 2.0.

- **General extensible design.** To allow extensibility in many ways. Not only the addition of new static analyses and transformations, but also the possibility to support other similar languages like Java in the future. For this reason, our goal is to decouple the framework as much as possible from any .NET specific details, so the same infrastructure can be reused. A base layer of abstraction is required as the building block of all framework features. This means a general and language agnostic metadata model and code representations.

- **Simplicity over performance.** Usability is very important for the framework to be useful and widely adopted. Code optimizations are not performed when they compromise code legibility, maintainability or extensibility. We only consider to optimize code when it is absolutely necessary to make it functional.

8.4 Features

The framework provides the following features, including a few different kinds of intermediate representations, static analyses and transformations. All of them ready to be used out-of-the-box or extended as needed.

8.4.1 Intermediate Representations

Simplified bytecode. A general and concise stack-based bytecode representation. Useful for performing low-level optimizations and transformations. Specially designed to be independent of any programming language. While the original CIL bytecode has about 230 instructions, our simplified bytecode only has about 20 different kinds of instructions to represent all of them. Among other things, the key for achieving this simplification is grouping together all the different variants of the same CIL instruction that only differ on the operands data types. For instance, there is only one conversion instruction for both casts and coercions with a flag to differentiate them if needed. The same happens for branch instructions, loads, stores and many other kinds of opcodes.

Three-address code. A convenient typed and register-based representation for performing static analyses and transformations. This is the main intermediate representation provided with the framework and used by most of the analyses. It only consist in about 30 different kinds of instructions. To simplify their usage, operands are restricted to variables only, except for the `load` and `store` instructions that also allow constant values and more complex constructions like fields and array element references.

Static single assignment form. It is basically an extension of the TAC with an extra Φ instruction. We provide different variations of this form. *Minimal* SSA inserts the minimal number of Φ instructions required to ensure that each variable is assigned a value exactly once and that each use of a variable in the original program can still refer to a unique variable definition. *Pruned* SSA inserts Φ instructions only for variables that are *live* after the Φ instruction. We meant *live* in the sense of *live variables* analysis, where the variable is used along some path that begins at the Φ instruction in question.

Aggregated expressions. Sometimes is useful to also have more complex instructions, for instance to represent nested algebraic expressions with a single `load` instruction. For those cases we provide aggregated expressions. They are a combination of TAC instructions with AST expressions. All TAC instructions that produce a value have a corresponding analogous expression that represents it.

8.4.2 Code Transformations

Disassembler. Converts simplified bytecode to TAC intermediate representation. Since we are converting from a stack-based IR to a register-based IR, this process involves simulating the stack behavior of the bytecode instructions explicitly with temporal variables. Each temporal variable corresponds to a slot in the stack and they are used as the operands of the generated TAC instructions. Since stack slots are not typed, temporal variables does not have a defined type at this point. *Webs analysis* and *type inference* are required for this purpose. Our implementation follows the algorithm described in [106].

```
1   load 2
2   load 3
3   add
4   store v
```

(a) Simplified bytecode.

```
1   $s0 = 2
2   $s1 = 3
3   $s0 = $s0 + $s1
4   v = $s0
```

(b) Three-address code.

Figure 8.3: Simplified bytecode and equivalent three-address code for Example 9.

Example 9. To illustrate this process consider the simplified bytecode fragment shown in Figure 8.3a. The code adds two numbers and stores the result in variable `v`. This is done by (i) pushing both arguments into the stack, (ii) performing the addition of the first two values from the top of the stack while removing them, (iii) pushing the result back into the stack and finally, (iv) popping it from the stack to store it at variable `v`.

By using temporal variables to represent stack slots, this process can be simulated as follows. Pushing values into the stack becomes assigning them to the temporal variables associated with their corresponding destination slots. Which variable depends on the current height of the stack. Popping values from the stack becomes reading their corresponding temporal variables. The resulting equivalent TAC is shown in Figure 8.3b. We use the symbol `$` to prefix the names of temporal variables to avoid clashing with already defined variables. ■

SSA construction. Converts TAC intermediate representation to SSA *minimal* form. *Pruned* SSA form is also possible but requires the information provided by *live variables* analysis. This process has two phases: insertion of Φ instructions and renaming of local variables. The first phase adds the minimal number of Φ instructions needed to ensure that each variable is assigned only once. *Dominance frontier* analysis computes exactly the locations where a Φ instruction should be added. The second phase is in charge of renaming all definitions and usages of local variables with their corresponding version of the variable. Our implementation follows the algorithm described in [32].

```
1  $s0 = 5
2  num = $s0
3  $s0 = "hello world!"
4  str = $s0
```

(a) Typeless three-address code.

```
1  $r0 = 5
2  num = $r0
3  $r1 = "hello world!"
4  str = $r1
```

(b) Typable three-address code.

Figure 8.4: Original and equivalent renamed three-address code for Example 10.

Webs analysis. As mentioned before, *disassembler* generates typeless temporal variables to explicitly represent stack slots. Since these slots are used to store different types of values pushed by simplified bytecode instructions, their corresponding temporal variables also store values of different types. So before typing can take place, it is required to perform *webs analysis* [85] first. The idea is to split each temporal variable into multiple fresh variables that can be successfully typed without having a type conflict. The number of fresh variables created when splitting the original temporal variable depends on how many reuses that variable has. A reuse of a variable means using the same variable to store values of different types. The intuitive notion of multiple uses of the same temporal variable is captured by the notion of *webs* [85]. A web is the minimal set of variable's references (both definitions and uses in the regular sense) that contains for every definition all its uses, and for every use all its definitions. *Def-use and use-def chains* are used to build webs for each temporal variable. The splitting is performed by assigning a fresh variable to each web and renaming all its references. The result of this transformation is typable TAC ready to be processed by the *type inference* analysis.

Example 10. To illustrate this process consider the typeless TAC fragment shown in Figure 8.4a. The code makes use of the same temporal variable `$s0` to store two values of different types (`int` and `string`). So trying to type variable `$s0` leads to a type conflict because of the reuse of the same variable. This problem can be avoided by splitting and renaming the variable `$s0` so that two different fresh variables `$r0` and `$r1` are actually used in each case.

By applying the *webs analysis* transformation, two webs are constructed. One for each def-use chain of the `$s0` variable. The resulting equivalent code after applying this transformation is shown in Figure 8.4b. Now types can be easily assigned to each variable without leading to type conflicts. In this case, `$r0` can be successfully typed as `int` and `$r1` as `string`. ■

Copy propagation. Its a typical forward data-flow analysis and transformation to reduce the number of local variables by removing unnecessary uses [27, 67]. It consist in replacing the occurrences of targets of direct assignments with their values. A direct assignment (also known as *copy*, deriving the name of this transformation) is an instruction of the form `x = y`, which simply assigns the value of `y` to `x`. The idea is to replace each direct use of `x` with `y` as long as its value does not change. It often makes use of *reaching definitions* and *use-def and def-use chains* when computing which occurrences of the target variable may be safely replaced. If all upwards exposed uses of the target may be safely modified, the copy assignment instruction may be eliminated. We have also implemented a backward version of this transformation that does exactly the opposite, replaces the targets of the definitions of sources of direct assignments with their targets. Copy propagation is a useful *clean up* optimization frequently used after other transformations have already been run. It is specially useful to remove unnecessary temporal variables introduced by the *disassembler* and *webs analysis* transformations.

Example 11. To illustrate this optimization consider the code fragment shown in Figure 8.5a. It shows a typical code resulting from running the *disassembler* transformation to convert simplified bytecode into tree-address code. Variables prefixed with `$s` are temporal local variables introduced during the conversion.

By applying the forward *copy propagation* transformation, the code is simplified by removing the first two copy instructions and replacing the corresponding variables as shown in Figure 8.5b. The backward version of this transformation has to be used to remove the last copy instruction. The resulting equivalent code after applying this transformation is shown in Figure 8.5c. Is easy to see that the code is drastically simplified by removing all unnecessary temporal variables. ■

```
1   $s0 = x
2   $s1 = y
3   $s0 = $s0 + $s1
4   num = $s0
```

(a) Original three-address code.

```
1   $s0 = x + y
2   num = $s0
3
4
```

(b) After forward copy propagation.

```
1   num = x + y
2
3
4
```

(c) After backward copy propagation.

Figure 8.5: Original and equivalent optimized code fragments for Example 11.

Inlining. This transformation replaces a method call site with the body of the called method [27]. Inline expansion itself is an optimization, since it eliminates method calls overhead, but it is much more important as an enabling transformation. That is, once a method body is expanded in the context of its call site, a variety of transformations that were not possible before may become available. Optimizations that cross method boundaries can be done without requiring a more complex inter-procedural analysis. For instance, in the case of inlining generic methods, knowing the specific type argument of a generic type parameter enables many optimization possibilities. However, complete inline expansion is not always possible, due to recursion: recursively expanding method calls may not terminate. Additionally, it is not always possible to determine statically the correct method implementation that a call site will resolve at run-time. Moreover, exactly the same call site may resolve to different implementations depending on some data. To overcome this problem, our implementation requires the user to specify which particular method implementation should be inlined.

8.4.3 Intra-procedural Analyses

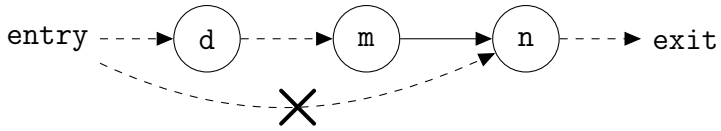
Control-flow analysis. Determines the execution order of program instructions by constructing a control-flow graph (CFG) [27] that explicitly specifies all possible execution paths. Nodes in the CFG are called *basic blocks*. They contain the maximal sequence of consecutive instructions with a single entry point, a single exit point and no internal branches. Edges are created by following branch instructions

and connecting their corresponding basic blocks with the ones associated with the targets of the branch. Program loops are represented explicitly with cycles in the graph. CFGs are commonly used by data-flow analyses to propagate information between basic blocks along their edges. They are a fundamental building block of any flow-sensitive static analysis.

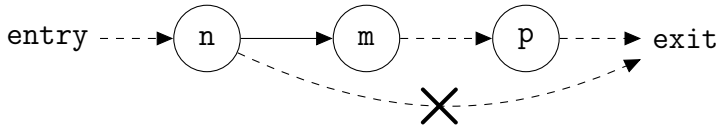
The framework supports the construction of CFGs for all the available intermediate representations in two different variations: normal control-flow and exceptional control-flow. The difference relies on if the exception handling information is considered or simply ignored when constructing the graph. Almost every instruction may throw an exception at run-time, resulting in the presence of many exceptional edges in the CFG, often leading to precision loss. So to overcome this issue whenever possible, each particular static analysis can make use of the CFG version that suits better. Blocks belonging to protected blocks and exception handlers, as well as loops, are contained within a CFG *region*. They group together many related basic blocks in a hierarchical structure. So a region can be nested inside another region and so on.

Dominance analyses. In a CFG, a node d dominates a node n if every path from the entry node to n passes through d, noted as $d\ dom\ n$ [27, 67]. Note that, by definition, every node dominates itself. Additionally, d strictly dominates n if $d\ dom\ n$ and $d \neq n$. The immediate dominator m of n, noted as $m\ idom\ n$, is the last strict dominator of n on any path from the entry node to n. Every node, except the entry node, has a unique immediate dominator. The dominator tree is a tree where each node's parent is its immediate dominator and the entry node is the root.

Analogously, a node p post-dominates node n if every path from n to the exit node passes through p, noted as $p\ pdom\ n$. Similarly, p strictly post-dominates n if $p\ pdom\ n$ and $p \neq n$. Finally, the immediate post-dominator m of n, noted as $m\ pidom\ n$, is the post-dominator of n that does not strictly post-dominate any other strict post-dominators of n. Post-dominance can be obtained from the dominance relationship by reversing CFG edges and exchanging the entry and exit nodes.



(a) Dominance. *d dom n* and *m idom n*.



(b) Post-dominance. *p pdom n* and *m pidom n*.

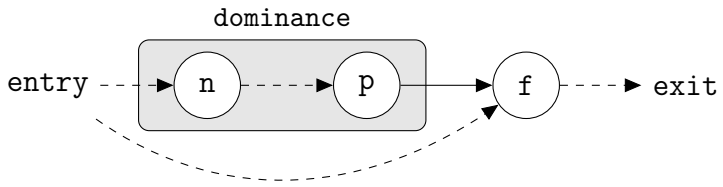
Figure 8.6: Dominance and post-dominance relations in control-flow graphs. A solid line means a direct edge while a dashed line means a path (sequence of edges).

Figure 8.6 illustrates these concepts. Dominance analysis have many uses, including *dominance frontier*, *control dependence* and *loop* analyses. Our implementation follows the algorithm described in [46].

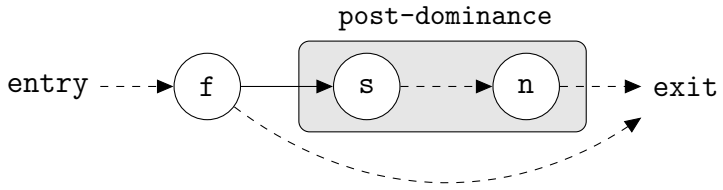
Dominance frontier analyses. In a CFG, the dominance frontier of a node n is the set of all nodes f such that n dominates an immediate predecessor of f, but n does not strictly dominates f [67]. Analogously, the post-dominance frontier of a node n is the set of all nodes f such that n post-dominates an immediate successor of f, but n does not strictly post-dominates f. Intuitively, is the set of nodes where the (post-)dominance of n stops.

Figure 8.7 illustrates these concepts. Dominance frontier is used to compute the exact locations where a Φ instruction has to be inserted when converting TAC to SSA form, while post-dominance frontier is used to compute the *control dependence* graph, among other usages. Our implementation follows the algorithm described in [46].

Control dependence analysis. A CFG node y is control-dependent on another node x if x determines whether y should be executed or not [27, 67]. In other words, if exists a path from x to y such that every node z in the path other than x and y



(a) Dominance frontier. n *dom* p, $\neg(n$ *dom* $f)$ and $f \in DF(n)$.



(b) Post-dominance frontier. n *pdom* s, $\neg(n$ *pdom* $f)$ and $f \in PDF(n)$.

Figure 8.7: Dominance and post-dominance frontier relations in control-flow graphs. A solid line means a direct edge while a dashed line means a path (sequence of edges).

$(z \neq x \wedge z \neq y)$ is post-dominated by y, but x itself is not. The control dependency graph (CDG) is a graph where each node is control dependent on its predecessors. Control dependencies are essentially the dominance frontier in the reverse CFG, and can be easily computed for programs with arbitrary control-flow using the reverse dominance frontier algorithm proposed in [49]. Basically, it computes the post-dominance frontier and reverses it to obtain the control dependency graph.

Figure 8.8 illustrates these concepts. Control dependence analysis have many uses, including the construction of the *program dependency graph* (PDG) used in program *slicing*.

Loop analysis. Identifies loops in *reducible* (well-structured) CFGs [27]. A reducible CFG is one with edges that can be partitioned into two disjoint sets: forward-edges and back-edges. Forward-edges form an acyclic graph in which every node can be reached from the entry node. A back-edge is one whose target, called the loop header, dominates its source. The loop header is the only entry

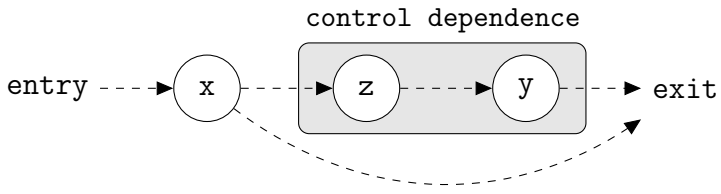


Figure 8.8: Control dependence relation in control-flow graphs, where nodes y and z are control dependent on node x; and y *pdom* z but $\neg(y$ *pdom* $x)$. A dashed line means a path (sequence of edges).

point to the loop and dominates all basic blocks in its body. Loops can be identified by their associated back-edges. The natural loop of a back-edge $n \to h$, where h dominates n, is given by its header h and the set of all nodes dominated by h that can reach n without going through h. Figure 8.9 shows the structure of loops in CFGs. Common structured programming statements such as `if`, `for`, `while`, `break` and `continue` produce reducible graphs. To produce irreducible graphs, statements such as `goto` are needed. A CFG is reducible if it can be converted into a single node using the T1 and T2 transformations shown in Figure 8.10, where T1 simply removes self-cycles and T2 merges nodes with their unique predecessors.

General data-flow analysis. It is a theoretical framework for collecting facts about programs [27, 67, 88]. It attempts to obtain particular information at each point in a method's body. The information gathered can be used to verify a property of interest, for instance, that a program will never throw a particular kind of exception at run-time. For this reason it is often used by optimizing compilers and static checkers. It uses the CFG to determine how this information flows between method instructions, so it is inherently a flow-sensitive static analysis. A program, from a static point of view is a finite sequence of instructions, but from a dynamic perspective it can have infinitely many possible execution paths (when containing a loop). So data-flow analysis overcomes this problem by working with typically finite[4] abstract domains called *lattices* instead of directly dealing

[4]Infinite abstract domains are also possible as long as they are finite in height or by using a widening or narrowing operator.

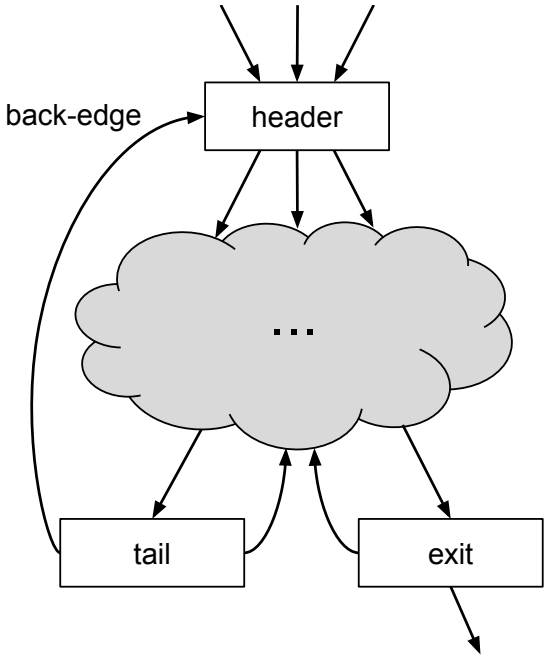
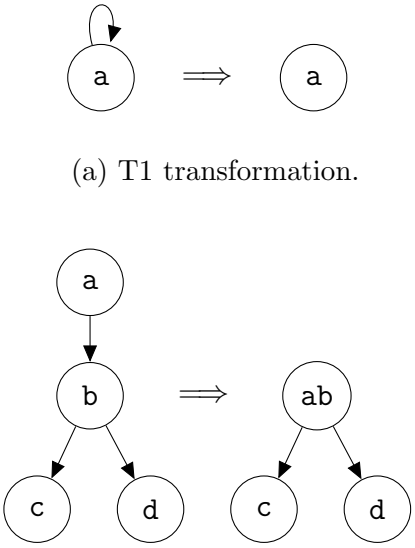


Figure 8.9: Structure of loops in control-flow graphs.



(a) T1 transformation.

(b) T2 transformation.

Figure 8.10: T1-T2 control-flow graph transformations.

with potentially infinite concrete values to represent the state of a program. For each program point, it combines information of all the possible executions of that particular location with an abstract state. In *forward flow analysis*, the exit state of a basic block is a function of the block's entry state. This function called *transfer function*, is the composition of the effects of the instructions in the block. Additionally, the entry state of a block is the result of applying a *join operator* that combines the exit states of its predecessors. This yields a set of *data-flow equations*. After solving this set of equations, the entry and exit states of the blocks can be used to derive properties of the program at the block boundaries. The transfer function of each instruction can be applied to get information at a point inside a basic block. If the CFG does not contain cycles, solving the equations is straightforward, otherwise they can be solved by repeatedly calculating the output from the input locally at each CFG node until the whole system stabilizes (i.e., it reaches a fixed-point). Each particular data-flow analysis instance defines its own specific transfer function and join operation over an abstract domain. Some data-

flow problems require *backward flow analysis*. This follows the same plan, except that the transfer function is applied to the exit state to compute the entry state, and the join operation works on the entry states of the successors to compute the exit state of the block.

Our implementation makes use of a worklist iterative algorithm to compute the fixed-point and provides two general abstract classes that can be inherited to implement specific instances of forward/backward data-flow analyses. They require only the definition of the transfer function and join operation, as well as the initial abstract state and a method that compares two abstract values to detect when the fixed-point is reached.

Reaching definitions. This classic static analysis determines which definitions may reach a given point in the code [27, 67]. Every variable assignment is a definition of that variable. A definition d reaches a point p if exists a path from the point immediately following d to p in the CFG, such that d is not modified (overwritten) along that path. It is important to note that statically, more than one definition of the same variable may reach a given program point.

Because of its simplicity, it is often used as the canonical forward data-flow analysis and serves as a good example of how to implement this kind of analyses in our framework. Our implementation uses bit vectors to efficiently track which variable definitions reach each program point. Reaching definitions is typically used to compute many analyses, including semantic checks to detect uses of uninitialized variables, *def-use and use-def chains* and *loop invariant computations*, among others.

Def-use and use-def chains. A definition-use chain consists of a definition d of a variable and all its uses u_i reachable from d without any other intervening definitions. Its counterpart is a use-definition chain, which consists of a use u of a variable and all its definitions d_j that can reach u without any other intervening definitions. In SSA form, use-def chains are explicit because each one contains a single element [27, 67]. They provide a direct data-flow representation useful for code navigation.

It can be easily computed using the results from *reaching definitions* or *live variable* analyses and are typically used by many code optimizations, including *constant propagation* and *common subexpression elimination*, among others. Our *webs analysis* implementation makes use of these chains to construct webs for each variable. Program *slicing* also takes advantage of the data dependency information provided by this analysis.

Live variables. This classic static analysis determines which variables are live at each point in the code [27, 67]. A variable v is live at point p if the value of v is used along some path in the CFG starting at p, before v is modified (overwritten). Otherwise, the variable is dead. It is important to note that to detect if a variable is live, we need to look at the future uses of it by propagating facts backwards over the CFG.

Because of its simplicity, it is often used as the canonical backward data-flow analysis and serves as a good example of how to implement this kind of analyses in our framework. Our implementation uses bit vectors to efficiently track which variables are live at each program point. Live variable analysis is typically used by many code optimizations like *dead code elimination* to remove statements that assign to a variable whose value is not used afterwards. The information provided by this analysis is also used to simplify the SSA intermediate representation form by pruning unnecessary Φ instructions.

Type inference. The goal of this analysis it to automatically detect the data types of temporal variables introduced after the *disassembler* process, which converts simplified bytecode to TAC [27, 67, 88]. Remember temporal variables are created to explicitly represent stack slots, so initially they don't have a specific data type assigned.

Our implementation takes the TAC intermediate representation as input and repeatedly propagates types from the right-hand side of assignments to its destination variable. Algebraic operations and type conversions are considered when inferring the types of its results. Note that type conflicts cannot occur after performing the *webs analysis* transformation, since it ensures that temporal variables are never reused (i.e., used in more than one context).

However, there are some particular cases that need to be specially handled. Since CIL bytecode is a compact code format, boolean values are represented with numerical values like in C (zero means `false` and any other value means `true`). This leads to ambiguities when loading those values. A priori it is not possible to detect by just looking the `load` instruction if its loading a boolean value or not. There is a similar ambiguity with `null`, where the same conditional branch opcode is used to jump when the value at the top of the stack is (or is not) zero, `false` or `null`. Even more, since `null` is a polymorphic constant, it's not possible to immediately determine the data type of a temporal variable that has that value assigned. For these cases, the analysis delays the type detection until the temporal variable is used, so the right data type can be inferred from its use context instead of its definition context.

Slicing. A program slice consists of all program instructions that may affect the values at some point of interest [110]. It is defined for a slicing criterion (x, v) where x is a program instruction and v is a variable appearing in x. Therefore, a static slice includes all the instructions that can affect the value of variable v at instruction x for any possible input. Static slices are computed by backtracking dependencies between instructions. An instruction x depends on another instruction y if (i) y defines whether x is executed or not (control dependence) or (ii) y defines a variable that is used by x (data dependence). For this reason, this analysis combines the *control dependency graph* (CDG) with the *def-use and use-def chains* to construct the more general *program dependency graph* (PDG) used to generate the slices.

Other forms of slicing exist, for instance dynamic slicing, which works on a specific execution of a program (for a given execution trace and input). It is often used in debugging to locate source of errors more easily.

8.4.4 Inter-procedural Analyses

Class hierarchy. It defines the inheritance relationship between classes [51]. In .NET, the root class of the hierarchy is the `object` class and every other class di-

rectly or indirectly extends (inherits from) it. Interface implementations are also included in the class hierarchy. Multiple inheritance is not supported in .NET, but multiple interfaces can be implemented by the same class, providing similar benefits. This analysis constructs the inheritance tree, where each class (or interface) has all of its subclasses (or implementing classes respectively) as children.

The information provided by this analysis is used by the *type inference* analysis to obtain the least common ancestor of two given classes and by a very simple *call graph* analysis to compute the set of possible targets of a method invocation.

Call graph analyses. The goal of these analyses is to statically construct the call graph of a program, which describes calling relationships between methods [51, 60]. Each node represents a method and each edge $f \rightarrow g$ indicates that method f calls method g. A cycle in the graph denotes recursive method calls. It is a fundamental building block for any other inter-procedural analysis. In languages that feature dynamic dispatch, such as object-oriented programming languages like C# and Java, computing a static call graph precisely requires alias analysis results [60]. Conversely, computing precise aliasing requires a call graph. For this reason, many static analyses solve this interdependence by computing both simultaneously. This is the case of the call graph produced by our *points-to* analysis implementation. In addition, our framework also provides a much simpler approach known as class hierarchy analysis (CHA) [51]. This analysis uses the declared static type of a variable, together with the *class hierarchy*, to determine which are the possible dynamic types of the receiver object at each call site. This analysis is more imprecise, but can be computed efficiently.

Points-to analysis. Pointer analysis is a fundamental static program analysis that determines information on the values of pointer variables [31, 102]. Such information offers a static model of a program's heap. Its goal is to compute an approximation of the set of objects that a program variable may point to at runtime. Alias analysis algorithms focus on the closely related problem of detecting if a given set of variables can be aliases (i.e., point to the same object). Points-to analysis typically construct a points-to graph (PTG), which is a heap abstraction

that models reference relationships between objects, including the variables and object fields that may points to them. The conventional approach is to relate objects with their allocation sites (i.e., to consider a single abstract object to stand for each run-time object allocated by the same instruction).

The framework provides a flow-sensitive and context-insensitive data-flow implementation with support for simulating the effects of non-analyzable methods by providing a mock method body that mimics the behavior of the real one. Additionally, our approach simultaneously builds a more precise call graph than the one obtained with CHA, by querying to which objects (and thereby indirectly to which dynamic types) the receiver of a method call could point to. It also enables further inter-procedural analyses, such us *escape* and *purity* analyses.

Escape analysis. It is a static analysis that determines whether the lifetime of an object exceeds its static scope [39, 112]. This often means whether an object may escape its allocating method, in which case the object is not local to it. The goal of this analysis is to keep track of objects created during the execution of a method. The objects may be created directly by the method or indirectly by the methods it calls (callees). An object escapes the scope of a method if it is still reachable after the execution of the method. This can only happen if a reference to the object is returned from the method, assigned to a field of an external (previously existing) object (such as a method parameter or global static object) or to a newly created one that also escapes. This analysis is typically built on top of the points-to graph of a method.

Our implementation follows a compositional summary-based approach, that uses the already computed escape information from callees to produce the summary of each method. It distinguish between method external and internal objects, in which case the later can be further classified as captured or escaping.

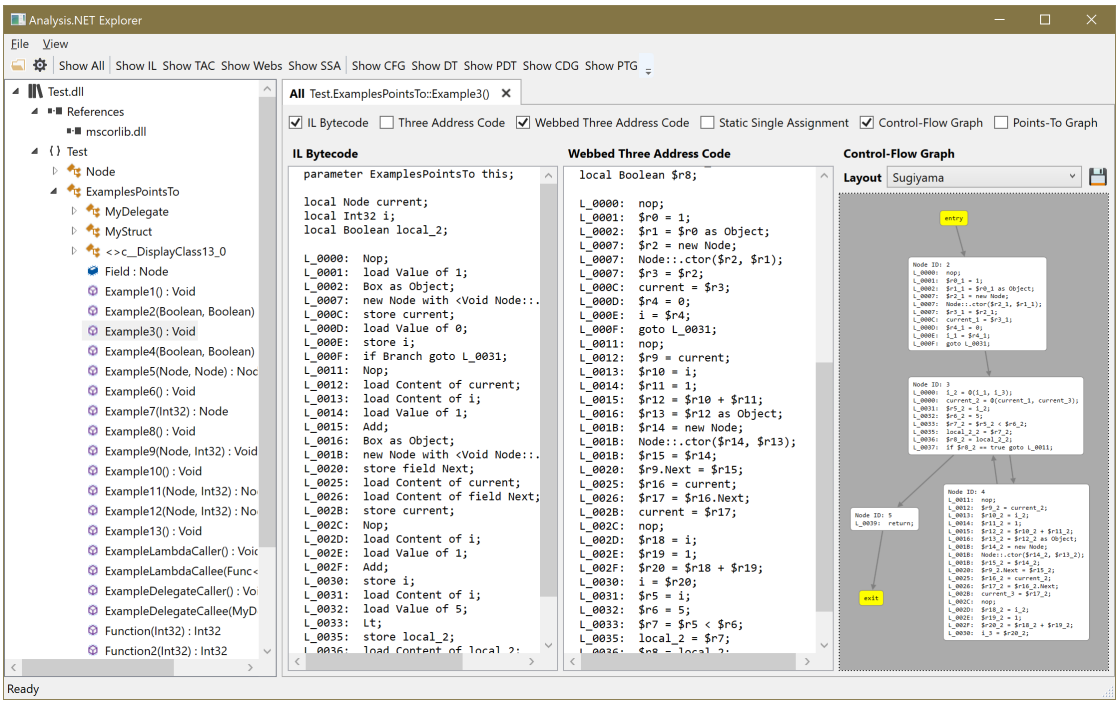


Figure 8.11: Analysis Explorer: a graphical user interface for the analysis framework.

8.5 Tools

In addition to the framework itself, we have implemented a few open-source tools[5] that make use of its features and are publicly available to the community. They provide an interactive way to explore many code analyses and transformations directly without having to write a single line of code. They are particularly useful for debugging purposes when using or extending the framework.

Analysis Explorer. A standalone application with a simple graphical user interface (GUI) for inspecting programs metadata and method instructions. Figure 8.11 shows a screenshot of this tool. The left panel displays the metadata in a tree-based layout for quick navigation between all defined types and methods. The right panel contains tabs to visualize different kinds of contents, from

[5]Source code available at: `https://github.com/edgardozoppi/analysis-explorer`.

method bodies to interactive diagrams of all the data structures provided by the framework, including control-flow, points-to, class hierarchy, dominance and control dependency graphs, among others. Different visualization layouts are possible with an option to export them to the Directed Graph Markup Language (DGML). Perhaps the most interesting feature is the ability to compare side-by-side all the intermediate representations, allowing the user to select which ones are relevant for inspection. Loading metadata from Java Archive files (JARs) is also possible.

Visual Studio extension. A plug-in that completely integrates the framework into Visual Studio IDE. It adds a dedicated context menu when selecting specific elements (typically methods) in the Class View tool window. It supports the same options that the Analysis Explorer provides, but directly inside the most popular IDE for developing .NET applications. In this case, method bodies are shown as text documents in the editor panel and diagrams like trees and graphs are displayed using the already integrated DGML visualization tool.

8.6 Extensibility

As mentioned before, one of the main requirements considered when designing the framework is the ability to extend it to support new features. Next we present the most relevant extensibility scenarios we have considered.

Supporting other languages. Even though the framework was originally created to enable the static analysis of .NET programs, it was designed in a more general way to also allow other similar platforms in the future. In order to extend the framework to support the analysis of programs written in a different non-.NET-based programming language like Java, a new *code provider* has to be implemented for that particular language. Basically, a code provider is a library that contains a class implementing the public interface `ILoader` defined by the framework. This is actually exactly how .NET CIL is supported, so after all, there is nothing special about it that makes it different from other platforms regarding

the framework treatment. This interface only requires the implementation of the `LoadAssembly` method, that given a file path it returns an object of type `Assembly` which represents a language agnostic library to be analyzed. It includes both the metadata and methods' instructions.

This general design allows multiple scenarios. Different code providers can be implemented to load not only already compiled programs in the form of executable files, but also the possibility to load them directly from their source code. Additionally, depending on the kind of language the user wants to support, the code provider can return instructions in any of the supported intermediate representations. For stack-based languages like the .NET CIL or the JVM bytecode the simplified bytecode is a natural choice, but for register-based languages like the Dalvik bytecode (Android's bytecode format) the TAC is a more straight forward option. This way, all the already existing features become available for the new language, since the TAC is the main code representation used by the framework and all the provided analysis and transformations are built on top of it. For example, we already have implemented a proof-of-concept code provider that loads metadata from Java executable Archive files (JARs) and we plan to extend it to provide a simplified bytecode version of the JVM bytecode instructions.

Adding new features. Of course it is also possible to extend the framework by implementing additional analyses and transformations, either from scratch or by combining already existing ones. A good example of this scenario is the instantiation of one of the two general data-flow templates (forward or backward) provided by the framework to implement a new custom flow-sensitive intra-procedural analysis. All the logic to compute the fixed-point is already implemented, greatly simplifying this task.

8.7 Limitations

Perhaps the most important limitation of the framework is the current impossibility to generate executable code. A typical round-trip user case like instru-

menting an already existing program by performing some code transformation or optimization is not possible without this feature. Another useful scenario that we would like to support is to allow users to create a program from scratch. Ideally, a code generation component that converts back the TAC to simplified bytecode and then (or directly) to CIL is required for this purpose. Other executable forms, like the JVM bytecode, are also possible targets, allowing programs originally written in one platform to run in a different one[6]. An alternative related feature that could also be interesting to consider is to implement a TAC (or simplified bytecode) interpreter.

Another feature we would like to improve is the current support for implementing inter-procedural data-flow analyses in a flow-sensitive, fully context-sensitive manner. We would like to provide a more general approach based on the well-known IFDS [93] and IDE [98] algorithms specially designed for this purpose. An analysis able to create inter-procedural control-flow graphs (ICFG) is also currently missing.

[6]Assuming that all the external references to standard and third-party libraries are also translated or replaced by similar versions available in the new target platform.

Resumen

Información General

Analysis.NET[7] es un *framework* de análisis estático de programas de código abierto para a la plataforma .NET [23]. Permite el análisis y la transformación de CIL *bytecode* directamente desde cualquier lenguaje de programación .NET disponible y está completamente escrito en C#. Comprende un gran conjunto de APIs y algunas herramientas GUI adicionales que son útiles para explorar sus capacidades sin tener que escribir una sola línea de código. Entre otras cosas, el *framework* incluye varios análisis estáticos clásicos y bien conocidos desarrollados sobre diferentes tipos de lenguajes intermedios que proporcionan diferentes niveles de abstracción. Está diseñado para ser extendido fácilmente para soportar otros análisis personalizados y transformaciones de código definidos por el usuario. Las características principales de Analysis.NET incluyen una representación intermedia de CIL *bytecode* simplificada y tipada basada en registros, un análisis de jerarquía de clases y sistema de tipos de .NET, un análisis de puntero sensible al flujo y algoritmos de construcción de *call graphs* y varios análisis intra-procedurales basados en la información computada de flujo de control y flujo de datos.

Nuestro trabajo está bastante inspirado en el *framework* de optimización de Java Soot [105, 70], pero en cambio nos enfocamos en la plataforma .NET. Sin embargo, nuestro *framework* está diseñado de una manera más general para permitir

[7]Código fuente disponible en: `https://github.com/edgardozoppi/analysis-net`.

también la posibilidad de portar otras plataformas similares en el futuro[8]. Nuestro objetivo es permitir el análisis estático de programas .NET proporcionando a la comunidad .NET un *framework* similar a Soot.

[8]Actualmente estamos trabajando en una prueba de concepto para también soportar Java.

Chapter 9

Big Data Queries Optimization Analysis

In this chapter we present the details of two static analyses built on top of our program analysis framework presented in the previous chapter.

9.1 Overview

Programming Big Data applications is often done using data processing languages that combine relational-style constructs with imperative user-defined operators. Examples of systems relying on this paradigm are, for instance, Spark [116], SCOPE [44, 117] and U-SQL [21]. An important component of such systems are *query optimizers* that work only over the relational skeleton of a program. The user-defined operators (UDOs) are opaque and not analyzed during optimization. Hence, query optimizers often miss opportunities to significantly improve resource savings for Big Data applications.

The goal of this project is to automatically infer useful information about UDOs during compile-time that can be used to optimize query processing. In particular, we focus on Big Data programs written in SCOPE, a query processing language developed at Microsoft. SCOPE scripts receive, analyze, and return tables of data, similar to SQL. In practice, user-defined operators can operate on tables that can

have several hundreds of columns and be hundreds of GB large. According to SCOPE team experts, most of the network and computational resources spent during execution of a SCOPE query are in fact irrelevant for the query result. For instance, the runtime will pass all of the table columns to a UDO although the operator might use only a small fraction of input columns to produce the output table. This happens since the runtime does not have detailed knowledge on UDO inner-workings and hence must conservatively assume that all input columns are used. The techniques we develop in this work automatically analyze UDOs and provide the query optimizer with such valuable pieces of information.

We use static analysis techniques to detect specific column access patterns induced by a UDO: which columns it reads from and the dependencies between columns. This information can be used to drastically optimize query execution:

- Columns in input tables not read by a UDO can be *pruned* away, i.e., filtered out earlier in the query plan. This can result in less data transferred between nodes, often measured in hundreds of GB.

- Columns that are passed unmodified through a UDO are *pass-through columns*. The runtime can directly copy values of such columns from the input table to the output table without expensive data marshalling through the UDO. Knowledge about pass-through columns can also help the query optimizer understand the distributed partitioning of the output table.

Additionally, unused and pass-through input columns can be used to validate user-provided optimization declarations. The primary constraint for the analyses we present is total soundness: we cannot produce incorrect analysis result because that can lead to invalid query results.

The first static analysis we introduce aims at quickly providing a conservative approximation of the input columns accessed by a UDO. The analysis performs a simple, yet effective escape and constant propagation alike analyses. The second analysis is more ambitious, but also more costly. It computes precise data and control dependencies in a UDO and relies on range and points-to analysis. We require both of the analyses to graciously handle complex .NET constructs such

as loop iterators and closures. In general, we give great attention to making our analyses sound and robust.

We evaluated our analyses on thousands of SCOPE query scripts that are executed internally at Microsoft on a daily basis. We corroborate that our analyses implementation achieve the expected robustness and soundness requirement while still being effective. This work was done in collaboration with researchers from Microsoft Research and New York University under patronage of Microsoft SCOPE product group, and it was published [56] and presented at the international conference on Foundations of Software Engineering (FSE) in 2017.

9.2 Background

Before presenting the analyses we provide more information about the SCOPE language, UDOs and its ecosystem in general.

9.2.1 Cosmos and SCOPE

Cosmos is a distributed computing platform developed at Microsoft for storing and analyzing massive datasets [117]. Designed to run on large clusters consisting of thousands of commodity servers, Cosmos main platform objectives are availability, reliability, scalability, performance and reduced cost. The main components of Cosmos are storage, execution environment and SCOPE, a high-level programming language for Big Data analytics.

More specifically, SCOPE is the programming language used to write *scripts* to be executed in Cosmos. It is primarily a version of SQL with several extensions. The computation model of SCOPE is defined in terms of a directed acyclic graph. Data exchanged between nodes in the graph are in the form of strongly-typed *tables*. A table comprises a set of *columns*, each column containing values of some particular type. The data in a table is organized as a set of *rows*: each row has a field for every column.

The code that executes within a node is either generated by the system or is user code, typically written in C#, called a User-Defined Operator (UDO). A

UDO can be any combination of table filters, projections and joins that are either impossible (or difficult) to express in the SQL-ish subset of the language.

SCOPE API for UDOs. A SCOPE UDO is implemented as a C# class which subclasses one of three base classes [117].

Processor. Implements a method that takes a row from the input table as a parameter and returns zero or more rows.

Reducer. Implements a method that takes a *rowset* (a set of rows from the input table that all have the same values in a specified set of columns) and returns zero or more rows.

Combiner. Like a Reducer, but receives two rowsets and returns zero or more rows.

All three must override a method in their respective base classes that returns the *schema* of their output table. This method is executed during query compilation. Optionally, the method may also indicate that column pruning is allowed and to also attach information to each (output) column indicating which input columns that column depends upon. Without this information, the optimizer must make the conservative assumption that all input columns are read and that no information is available about which columns the output table might be partitioned on. Not only do many UDOs fail to add this optional information, but there is no check to make sure that any declarations are in fact correct.

Figure 9.1 shows an illustrative example of a UDO Reducer. The operator returns an output table that is essentially a copy of the input table where value of the output column indexed by 2 is created using the value of the input column indexed by 0. Columns can either be accessed by integer indices, or more commonly, using string indices. The above example exhibits a structure common to almost all real-world UDOs: it is written as an *iterator*, a C# idiom for coroutines defining a lazy, cooperative state machine that must be polled for each element in the sequence it returns. There is a `foreach` loop iterating over the collection of input rows, code that creates an output row and a `yield` instruction that returns that row.

```
1  IEnumerable<Row> Process(RowSet input_rowset,
2                           Row output_row,
3                           string[] args) {
4    foreach (Row input_row in input_rowset.Rows) {
5      input_row.CopyTo(output_row);
6      string market = input_row[0].String;
7      output_row[2].Set("FOO" + market);
8      yield return output_row;
9    }
10 }
```

Figure 9.1: SCOPE UDO Reducer example.

9.2.2 UDO Representation

While the source code in Figure 9.1 is simple, it is compiled into a much more complicated representation in the resulting bytecode. In order to support UDOs written in any programming language available for .NET, our analysis operates on the bytecode instead of the source code. As depicted in Figure 9.2, the `foreach-yield` loop is implemented using a closure class `<Process>_d__d<>3` whose method `GetEnumerator` essentially populates the compiler generated fields that (1) model the parameters of the original `Process` method and fields that (2) represent the state of the loop iteration.

The `MoveNext` method is a state machine that, depending on the state, invokes the actual enumerator of the input row set and performs one iteration, possibly computing an output row. The analysis on UDOs must be aware of this internal organization, look for these particular methods, associate the internal fields with the original method parameters, and simulate the potentially multiple invocations of `MoveNext`. Additionally, it must also understand how SCOPE operations are represented at the CIL bytecode level.

Both of the analyses we develop in this work take as input a SCOPE *job*, *i.e.,* a compilation of a SCOPE script. Each job has a set of processors (UDOs) that, as mentioned earlier, implement Processor, Reducer or Combiner APIs. For each

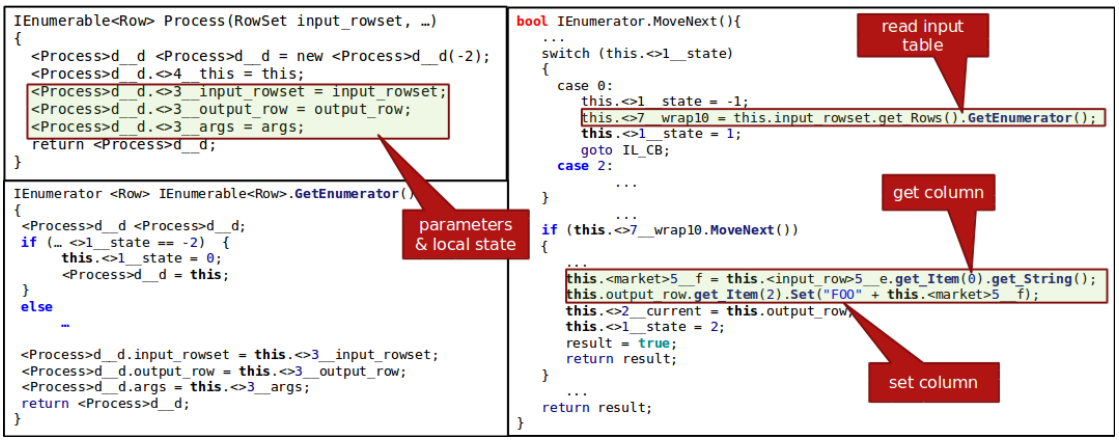


Figure 9.2: Low-level closure representation of the UDO from Figure 9.1. Top-left: `Process` method. Bottom-left: the enumerator generated for `yield return`. Right: `MoveNext` method of the enumerator.

UDO in the job, we find the corresponding closure class and run our analyses on the `MoveNext` method assuming the above closure representation.

9.3 Accessed Columns Analysis

The first analysis statically analyzes the code of a UDO to overapproximate the input columns that are being used by the operator. Using this information, the SCOPE distributed runtime environment can ship over the network only the values of inferred columns instead of values of all of the table columns while executing the UDO, without compromising the correctness of the results.

9.3.1 Approach

We decided to design the analysis to be as simple as possible. First, the analysis is intra-procedural, analyzing method bodies in isolation. Second, the analysis does not distinguish input from output columns, which would otherwise require potentially detailed aliasing information. Lastly, the analysis (soundly) answers that all input columns are read if the UDO has exceptional control-flow, which

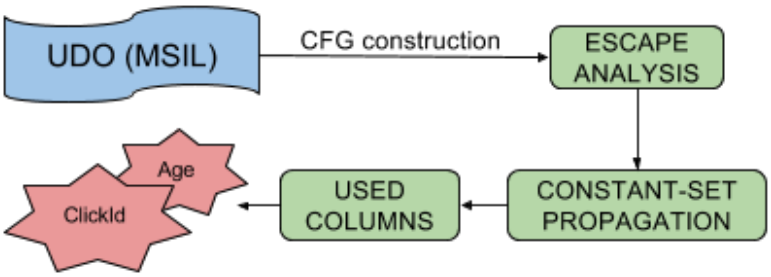


Figure 9.3: Accessed columns analysis pipeline.

would otherwise also require more complicated treatment. These design decisions helped us validate that our analysis is sound with no exceptions, in contrast to *soundy* analyses [77].

Clearly, the above mentioned simplifications may make our analysis not effective. The reason these decisions make sense for computing accessed input columns is based on an empirical observation: while UDOs can become quite complex in terms of the functionality they embody, the way the columns are accessed is typically straightforward. That is, columns are typically accessed directly by string or integer indices, as in Figure 9.1, or by variables that can be resolved as constants at compile-time.

Figure 9.3 is a high-level illustration of our analysis. The analysis takes an UDO in previously described CIL format, performs classical control-flow graph transformations and optimizations described in Section 9.5, and then proceeds to three core subanalyses: escape analysis, constant-set propagation and used columns analysis.

9.3.2 Escape Analysis

A valid platform assumption is that upon entering the `MoveNext` method no other method or object has an access to the input row objects. Our analysis first checks whether this invariant also holds upon exiting the method via *escape* subanalysis. To see why this check is important, consider the following code fragment taken from a real-world SCOPE script.

```
1  IEnumerable<Row> Process(RowSet input, ...) {
2    ...
3    foreach (Row current in input.Rows) {
4      outputRow[0].Set(CreateBondEntity(current));
5      ...
6      outputRow[1].Set(RandomCurrentBondEntity());
7      yield return outputRow;
8    }
9  }
```

The method call on line 4 can potentially save a reference to the input row `current`. Since our analysis is intra-procedural, there is no way of knowing whether some other object has an access to an input row after line 4. Hence, the method call on line 6 can potentially read (i.e., access) some column of `current`. Our analysis would hence miss such accesses which would result in unsoundness.

Escape subanalysis checks if an object of type `Row` potentially *escapes* the method body. That is, escape analysis checks if some command in the UDO body passes an object of type related to `Row` as a parameter to some method call or saves it to some field. If so, the analysis claims that a row may *escape* and conservatively answers that all table input columns are read. Otherwise, no other object or method has access to the input rows (i.e., all column accesses are contained within `MoveNext` body). Our analysis then proceeds to the next subanalysis.

9.3.3 Constant-set Propagation

The next subanalysis closely resembles widely known constant propagation. In fact, the major point where our subanalysis and constant propagation differ is in the way they handle join points. Consider the following code fragment.

```
1  if (...) { column = "Age" }
2  else     { column = "age" }
```

Such code fragments are frequent in UDO programs as programmers often consult table schemas to make sure they got column name capitalizations right. After the `if-then-else` statement, at the join point, a typical constant propagation implementation would not consider `column` variable to be constant. Since our goal is to actually infer the columns being accessed, we can soundly save the information that column can take values in the set {`"Age"`, `"age"`} instead of saying that `column` can take on any value. This is why we called this subanalysis *constant-set* propagation, since for each non-reference variable we save the set of constant values the variable can take. Speaking in terms of abstract interpretation, we simply perform disjunctive completion of the abstract domain for constant propagation [48].

9.3.4 Used Columns Analysis

The last subanalysis we undertake is named *used-columns* analysis. In the similar spirit as previous subanalyses, its main characteristic is simplicity. For each method call on an object of type `Row`, we check whether the method being called is known at compile time and is `get_Item`, `get_Schema` or `Reset`. Only the mentioned methods can truly be trusted, in the sense they definitely do not access any columns of the calling `Row` object. Then, we check if for each `Row.get_Item(var)`, our constant-set propagation inferred a set of constants for `var`. If both checks pass, we take the union of these sets of constants as our overapproximation of accessed input columns. If either of the checks fail, the analysis answers that all input columns are being accessed.

The returned set of columns overapproximates both input and output columns being accessed by a UDO, due to the lack of aliasing information. Thus, all input columns that are not in this set are not being used by the UDO. We note that column information for an input table is available at compile time since SCOPE needs table schemas for compilation.

9.4 Computing Input/Output Dependencies

The second analysis we introduce is more sophisticated. It computes column input/output relationships induced by the UDO and pass-through columns. Precise dependency information allows for more aggressive, but still conservative, optimization of query plans. Identifying pass-through columns enables significant savings in network/computation bandwidth.

Example 12. Consider again the UDO presented in Figure 9.1 and the input table schema {`JobGuid(0)`, `SubmitTime(1)`} where integer index for the column name is given in parentheses. The outcome of our second analysis looks like:

- Inputs = {`JobGuid(0)`, `SubmitTime(1)`}
- Outputs = {`JobGuid(0)`, `SubmitTime(1)`, `NewColumn(2)`}
- Pass-through = {`JobGuid(0)`, `SubmitTime(1)`}
- OtherDependencies = {`NewColumn(2)` ← literal + `JobGuid(0)`}

Inputs (resp. Outputs) is the set of input (resp output) columns observed by the analysis. The indices represent the column index associated with the column name. Pass-through is the set of output columns that were computed using one single input column without modifications. OtherDependencies refers to other dependencies observed by the analysis that are not pass-through. In this case, `NewColumn(2)` refers to a new column that depends on a literal (`"FOO"`) and the input column `JobGuid(0)`. ■

9.4.1 Approach

Our solution is inspired by the work of Xia *et al.* [113] that proposes a data dependency analysis for SCOPE programs using an abstract interpretation engine Clousot [78]. The analysis computes dependencies over *traceables*: tables, columns, and row counters. For each output column of a UDO, the analysis reports traceables upon which that column depends on.

The hard constraint of having a sound and robust implementation prevented us from using their work. Clousot makes the optimistic assumption that any references that are not *must aliases* are distinct. As pointed out in [113], there is no aliasing for input tables and fields in the UDO closure classes.

However, considering objects reachable in method calls forces us to be more conservative in three ways:

- Use a conservative points-to analysis to support a may-alias and escape analysis on top of it.
- Use range analysis (intervals) and constant propagation for tracking column indices.
- Add support for exceptional control-flow.

9.4.2 Analysis Sketch

Figure 9.4 shows the analysis sketch. Given a UDO, the analysis first makes sure it has a proper representation of the heap effects. Therefore, we first compute a points-to graph (PTG) to get an alias heap abstraction of the closure constructor. Then, we compute the PTG for the `GetEnumerator` method (taking as input the constructor's PTG) and finally, the PTG of `MoveNext` method. This ensures we reach this method with all closure fields properly initialized and updated. With the proper PTG we can now run the column analysis to discover columns indices and map column names to indices and, finally, the dependency analysis.

Points-to analysis. We based our points-to analysis on a well-known flow-sensitive analysis by Salcianu *et al.* [99]. It is essentially a forward data-flow analysis that builds points-to graphs, where each node (identified by a program location) represents the set of all objects that might be allocated at that location and edges stand for potential references between those objects. Given a PTG, we can determine whether two access paths (in the form of v, $v.f$, $v.f.g$, etc.) may alias by simply traversing the PTG and checking if they both can reach the same node.

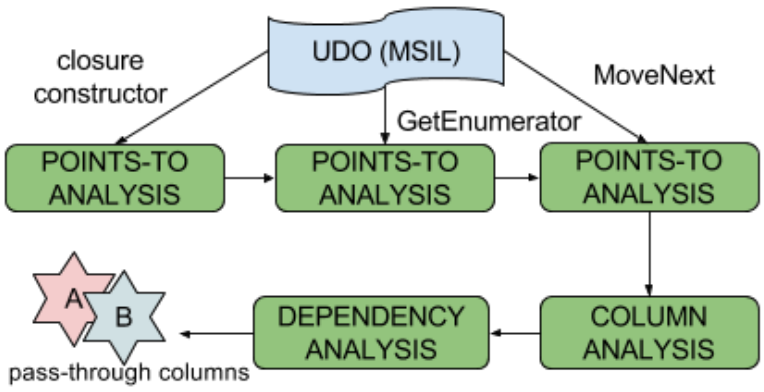


Figure 9.4: Columns dependency analysis pipeline.

Our points-to analysis can handle complex .NET programs constructs such as delegates, lambdas and predicates (appearing frequently in LINQ queries). It can run both intra- and inter-procedurally. For the sake of performance, we decided to run it intra-procedurally with some exceptions. We analyze invocations of closure auxiliary methods and lambda expressions appearing in parameters. To handle collections, we use conservative summaries in the spirit of [35].

Column Analysis. The goal of this analysis is to determine the set of potential indices (columns) used to access a row. We compute an interval analysis to determine the possible integer values that a column index may have, which is more sophisticated than constant-set propagation. In addition, we perform a basic string analysis to discover columns accessed by name and map them to their corresponding indices.

9.4.3 Dependency Analysis

As mentioned earlier, we based the analysis on [113]. Given a UDO, the analysis identifies a set of *traceables* and computes the following mappings:

- $DepVar(v)$ tracks traceables flowing to variables.
- $DepHeap(o.f)$ tracks traceables flowing to fields.

- $Esc(m)$ tracks traceables escaping through non-pure methods (in the case of inter-procedural analysis).

The analysis is essentially a forward data-flow analysis that propagates traceables through variables. Here we show some of the most interesting rules ($\rightarrow$ means propagate):

Row rules. Detect and propagate rows.

- `a = Rows(b)`: $DepVar(b) \rightarrow DepVar(a)$ propagate all traceable rows from table `b` to `a`.
- `a = Current(b)`: $DepVar(b) \rightarrow DepVar(a)$ propagate current traceable row from table `b` to `a`.

Column rules. Detect and propagate columns.

- `a = Read(r,i)`: $\{t_i \mid t \in DepVar(r)\} \rightarrow DepVar(a)$ reads the i^{th} column of row `r`.
- `Write(r,i,a)`: $DepVar(a) \rightarrow t_i$, $\forall t \in DepVar(r)$ writes the value of `a` into the i^{th} column of row `r`.
- `Copy(a,b)`: $DepVar(a) \rightarrow DepVar(b)$ propagates all traceable columns from row `a` into row `b`, similar to multiple applications of the previous two rules.

Heap rules. Propagate traceables from heap locations to variables, and vice versa. We use points-to analysis to determine heap locations.

- `a = b.f`: $DepHeap(b.f) \rightarrow DepVar(a)$ propagate all traceables from the heap locations obtained by following the field access `b.f` to `a`.
- `a.f = b`: $DepVar(b) \rightarrow DepHeap(a.f)$ propagate all traceables from `b` to the heap locations obtained by following the field access `a.f`.

If the method under analysis invokes another method, we check if the traceables of interest (i.e., input-output rows) can be reachable from the parameters. If so, in the intra-procedural case we give up (mark them as escaping) as we cannot tell what the non-analyzed callee is going to do. In the inter-procedural case, we apply the analysis on the callee.

Aliasing. We use the points-to graphs for detecting aliasing pairs (variables and fields) and to resolve method invocations. Also, every time we obtain traceables for a variable v we make sure we also include the traceables from its aliases. Similarly, for $DepHeap(v.f)$ we use PTGs nodes for referring to objects. For instance, if $PT(v) = \{A, B\}$ the analysis produce two locations $\{A.f, B.f\}$ for $v.f$.

9.4.4 Computing Pass-through Columns

Pass-through analysis is a byproduct of the dependency analysis. A pass-through column is an output column whose value is taken directly from one input column. In order to determine if a column is pass-through during the dependency analysis we check that *only* one unchanged input column is used for its computation and no other value.

9.5 Implementation

We have implemented our analyses[1] on top of the .NET static analysis framework presented in Chapter 8. Our implementation works over the SSA intermediate representation and utilize existing framework facilities to model implicit CIL stack operations with explicit top-of-the-stack variable operations. We also make use of many provided well-known analyses [27], like control-flow, data-flow, copy-propagation, live-variables, points-to and escape analyses, among others. Additionally, we have extended the framework by implementing the constant-set propagation, range and dependency analyses directly on top of it.

[1]Source code available at: `https://github.com/Microsoft/rudder`.

9.6 Evaluation

We aim to answer the following research questions.

RQ1: What is the ratio between used and total number of input columns?

RQ2: How many pass-through columns are discovered?

RQ3: Are analysis running times within an acceptable bound?

We run the analyses on about 4000 real-world SCOPE projects extracted from Cosmos' top jobs (in terms of resource usage) executed on a single day of April, 2016. Table 9.1 shows the results for both analyses. There were 1151 UDOs in total[2]. The analysis times ranged between 100 ms to a couple of seconds. Median analysis time for the accessed columns analysis is 65 ms and 463 ms for the dependency analysis. We note our analyses as well as the underlying framework are not yet fully optimized for performance. We believe a mature implementation would experience running times measured in few hundred milliseconds. The total number of columns involved in the UDOs, according to the declared schemas, were 25014 for input and 24941 for output columns.

The first analysis is about 6-8 times faster than the second analysis but more imprecise. In many cases it cannot conclude a precise answer and, for the sake of soundness, abruptly returns that all columns are potentially used. This imprecision is mainly due to exceptional control-flow, escaping rows, untrusted row methods, and non-constant variables used for column accesses. Nevertheless, for about 25% of the UDOs the analysis obtained results, discovering that at least 37% of the columns were not accessed. We feel these are very good results that justify the simplicity of the analysis.

[2]We ignore jobs not using UDOs or using only compiled generated UDOs.

The second analysis is slower but more precise, as expected, and handles more UDOs. It discovers that about 50% of the columns are unused. In addition, it discovers that about 74% of used columns are in fact pass-through. The imprecision mainly comes from dealing with complex index computation for column accesses, traceable escaping through method invocations (needs inter-procedural analysis), and complex data structures like SCOPE maps (see future work 9.8).

We could not measure actual query times since we did not have access to the Cosmos databases, but according to the SCOPE team experts, a reduction in the number of columns passed to a UDO and the savings in marshaling induced by pass-through columns can be drastic. It is important to emphasize that even a small percentage of resource savings yields huge savings in total as the analyzed scripts run daily at Microsoft and operate on hundreds of GB of data.

RQ1: What is the ratio between used and total number of input columns?

The first analysis discovers that at least 37% of the columns were not accessed while the second analysis discovers that about 50% of the columns are unused.

RQ2: How many pass-through columns are discovered?

Only the second analysis computes them and discovers that about 74% of used columns are in fact pass-through.

RQ3: Are analysis running times within an acceptable bound?

Median analysis time is 65 ms and 463 ms for the first and second analysis respectively.

Analysis	UDOs with results	Unused Cols		Pass-through Outputs
		Inputs	Outputs	
Accessed Columns	25%	37%		N/A
Columns Dependency	76%	54%	50%	74%

Table 9.1: Statistics for 1151 real-world UDOs from about 4000 SCOPE jobs.

9.7 Related Work

There are two previous efforts directly related to our work. PeriSCOPE is a static analysis tool that optimizes SCOPE execution plans by analyzing UDOs [61]. The authors present three analyses that compute UDO information useful for optimizing the query execution. One of the analysis is in fact used columns analysis. Unfortunately, it is not clear how general soundness of their algorithm can be argued. For instance, the authors do not explain how they deal with the cases where a `Row` object can escape a method body. In our work, on the other hand, soundness is an imperative.

As we discussed in Section 9.4, Xia et al. present a static analysis that infers column dependencies in SCOPE UDOs [113]. Their approach relies on an optimistic must-aliasing assumption which violates our soundness principle. Such assumption could prevent building different sound analyses on top of their infrastructure.

We mention few other related works. Our dependency analysis closely resembles data and control dependence in compiler optimizations [27, 85]. Also, points-to and alias analysis are a classical topic in static analysis community [39, 102, 35]. However, these efforts are orthogonal to the work presented here.

9.8 Conclusions

We implemented two static analyses aimed at obtaining unused column information and input/output dependencies in SCOPE UDOs. We put a special

focus on being sound and robust while designing and implementing the analyses. Our implementation successfully analyzed thousands of SCOPE scripts and found many input columns that are never used and a significant amount of pass-through columns in real-world UDOs. The inferred information can be used to drastically optimize execution of SCOPE scripts. This works shows how static analysis techniques can be used to improve performance of industrial-strength applications.

Lessons learned. One of the most important lessons we learned is that in an industrial setting it is much more important to be robust and sound than to be precise. Even the most sophisticated analysis will not make its way into production if its soundness cannot be guaranteed and clearly argued. Our recommendation is to start with the simplest analysis possible. Then if needed, more sophisticated analysis can be built on top, keeping the same principles in mind.

Another important lesson learned is that static analyses techniques can be successfully applied for real-world programs at large. The trick behind our success was to incorporate domain specific knowledge into the analysis to get sensible results. SCOPE scripts, while potentially arbitrarily complex, typically follow a simple structure and manipulate certain data structures in a predictable way. We believe there are other problem domains where such tailored static analyses can prove themselves extremely valuable.

Future work. Our next steps are the evaluation of the actual impact of the analyses on execution performance of real-world SCOPE scripts and the implementation of an IDE plugin that automatically generates data-dependency annotations for the UDOs, thus hinting the developer where optimizations can be gained and validating her assumptions. At the same time, we plan to enhance our analyses. For instance, the analysis for computing input columns accessed by a UDO can be improved by making it more inter-procedural by using function inlining (since we have observed that UDO scripts in general tend to be non-recursive). This way, we can gain more precision during escape analysis. Likewise, we would like to support other features of SCOPE such as SCOPE maps, JSON, and other structured column types that are used in scripts to encode sparse columns.

Resumen

Optimización de Consultas Big Data

En este capítulo presentamos los detalles de dos análisis estáticos desarrollados utilizando nuestro *framework* de análisis de programas presentado en el capítulo anterior.

La programación de aplicaciones Big Data se realiza a menudo utilizando lenguajes de procesamiento de datos que combinan construcciones de estilo relacional con operadores imperativos definidos por el usuario. Ejemplos de sistemas basados en este paradigma son, por ejemplo, Spark [116], SCOPE [44, 117] y U-SQL [21]. Un importante componente de dichos sistemas es el *optimizador de consultas* que funciona solo en el esqueleto relacional de un programa. Los operadores definidos por el usuario (UDO) son opacos y no analizados durante la optimización. Por lo tanto, los optimizadores de consultas a menudo pierden oportunidades para mejorar significativamente el ahorro de recursos para aplicaciones Big Data.

El objetivo de este proyecto es inferir automáticamente información útil sobre UDOs durante el tiempo de compilación que se puede utilizar para optimizar el procesamiento de consultas. En particular, nos enfocamos en los programas Big Data escritos en SCOPE, un lenguaje de procesamiento de consultas desarrollado por Microsoft. Los SCOPE *scripts* reciben, analizan y devuelven tablas de datos, similares a SQL. En la práctica, los operadores definidos por el usuario puede

operar en tablas que pueden tener varios cientos de columnas y tener cientos de GB de tamaño. De acuerdo con los expertos del equipo SCOPE, la mayoría de los recursos computacionales y de red consumidos durante la ejecución de una consulta SCOPE es, de hecho, irrelevante para el resultado de la consulta. Por ejemplo, SCOPE pasará todas las columnas de la tabla a un UDO aunque el operador podría usar sólo una pequeña fracción de las columnas de entrada para producir la tabla de salida. Esto pasa ya que el sistema no tiene un conocimiento detallado de los trabajos internos de un UDO y por lo tanto debe suponer de forma conservadora que se utilizan todas las columnas de entrada. Las técnicas que desarrollamos en este trabajo. analizan automáticamente los UDO y proporcionan al optimizador de consultas esa información tan valiosa.

Evaluamos nuestros análisis en miles de *scripts* de consultas SCOPE que se ejecutan internamente en Microsoft diariamente. Corroboramos que la implementación de nuestros análisis logran la robustez esperada y la corrección requerida, mientras que al mismo tiempo son efectivos. Este trabajo se realizó en colaboración con investigadores de Microsoft Research y la Universidad de Nueva York bajo patrocinio del grupo de producto encargado de SCOPE en Microsoft, y se publicó [56] y se presentó en la conferencia internacional sobre Fundamentos de Ingeniería de Software (FSE) en el año 2017.

Chapter 10

Other Clients of the Framework

In this chapter we provide a high-level overview of a few additional client applications of our static program analysis framework presented in Chapter 8 in which we have also participated.

10.1 Memory Consumption Analysis

Consume.NET[1] is an open-source tool that implements the memory consumption static analysis presented in [41] for .NET programs and it is entirely built on top of our program analysis framework.

It consists in a symbolic quantitative static and flow-insensitive analysis for computing parametric upper bounds of the number of simultaneously live objects. The analysis builds summaries quantifying the peak amount of live objects and escaping objects. Since it is a summary-based compositional analysis, method summaries are built by resorting to the summaries of their callees.

In contrast to other related work that infers consumption based on recurrence equation solving [29] or that verifies user-provided consumption annotations [43], our analysis infers memory consumption based on symbolic calculation over iteration spaces. More specifically, the technique for building summaries is essentially based on quantifying the number of visits to statements (such as allocations and

[1]Source code available at: `https://github.com/edgardozoppi/consume-net`.

method calls) over an iteration space given as an invariant predicate that involves the formal parameters of the method under analysis. For this reason, our implementation relies on the symbolic calculator Barvinok [107] to operate and solve complex non-linear symbolic arithmetical expressions.

In order to obtain all the required information, the analysis makes use of the results computed by a couple of additional static analyses, some of which are already provided by the framework, such as the points-to analysis to construct not only a points-to graph for each method, but also a call graph for the entire program, as well as the escape analysis to determine which objects escape their allocating method.

10.2 Boogie Bytecode Translator

Boogie [73] is an intermediate verification language, intended as a layer on which to build program verifiers for other languages. Several program verifiers have been built in this way, including VCC and HAVOC verifiers for C, as well as the verifiers for Dafny, Chalice and Spec#.

Boogie is also the name of a tool [34], that accepts the Boogie language as input, optionally infers some invariants in the given Boogie program, and then generates verification conditions that are passed to a satisfiability modulo theories (SMT) solver such as Z3 [50].

BCT [36] is a translator from .NET CIL bytecode into the Boogie verification language. It provides a vehicle for converting any program checker for Boogie into a checker for a language that compiles to the .NET platform. It was built using CCI [15] to load the CIL bytecode of the .NET program to be translated.

However, given the several issues and bugs found and reported for BCT, and the lack of support of this tool in the last couple of years, a new open-source project named TinyBCT was recently created to overcome those issues.

While still under development, TinyBCT[2] is heavily inspired by the original BCT tool and it is entirely built on top of our static program analysis framework. Among other features of the framework, it makes use of the TAC intermediate rep-

[2]Source code available at: `https://github.com/m7nu3l/TinyBCT`.

resentation and both copy propagation code transformations. It has full support for correctly translating lambdas and delegates, and utilizes the class hierarchy call graph analysis (CHA) to simulate, using nested conditional checks, the dynamic method dispatch of virtual method invocations and interface member accesses, as described in [36]. In order to support non-analyzable external libraries referenced by the program to be translated into Boogie, TinyBCT synthesizes TAC instructions for dummy method bodies to generate a simplified version of the real behavior of the methods defined in those libraries.

10.3 Thrown Exceptions Analysis

Contractor.NET[3] [119, 118] is an automated open-source tool that takes a piece of software and generates a finite abstract representation of it, named EPA. By analyzing the resulting finite state machine, the user can discover potential anomalies in the input contract such as invariants, preconditions or postconditions that might be too weak, therefore allowing unwanted behavior. It also allows the user to augment the original contract specification for the input class with the inferred typestate information, therefore enabling the verification of client code. Contractor.NET support two different backends to solve reachability queries needed to construct an EPA for a class: the Code Contracts static checker Clousot [78], based on abstract interpretation, and Corral [68], a bounded software model checker.

Contractor.NET mostly works at the AST level and was primarily built using CCI [15] to load the CIL bytecode of the .NET program under analysis. However, it makes use of our program analysis framework to implement a simple yet useful static analysis to collect all the exception types that might be thrown by a particular method during run-time. Note that the exceptions can be directly thrown by a given method or indirectly thrown by some other method transitively called by it. For this reason, we designed the analysis to build summaries for each analyzed method. They consist in a set of exception types that the method associated with the summary may throw at run-time during its execution. Summaries are used

[3]Source code available at: `http://lafhis.dc.uba.ar/contractor/contractor.net-web/`.

to compute other summaries. At each method call site the summary corresponding to the callee is applied in the context of the caller to build its own summary. Therefore, it is a summary-based inter-procedural static analysis.

To resolve invocation targets the analysis makes use of the class hierarchy call graph analysis (CHA) provided by our framework. In the presence of non-analyzable external libraries referenced by the program, the analysis utilizes synthetic summaries to inform the set of possible thrown exceptions of the methods defined in those libraries. A more sophisticated analysis could filter exception types that are known to be catched by an exception handler. In this scenario, the provided class hierarchy (CH) analysis would be useful to determine all the exception subtypes that can be catched by a particular exception handler.

Resumen

Otros Clientes del Framework

En este capítulo proporcionamos una descripción general de alto nivel de algunas aplicaciones cliente adicionales de nuestro *framework* de análisis estático de programas presentado en el Capítulo 8 en las cuales también hemos participado.

Análisis de Consumo de Memoria

Consume.NET[4] Es una herramienta de código abierto que implementa el análisis estático de consumo de memoria presentado en [41] para programas .NET y está completamente construido sobre nuestro *framework* de análisis de programas.

Consiste en un análisis simbólico, cuantitativo, estático e insensible a flujo para calcular cotas superiores paramétricas del número de objetos simultáneamente vivos. El análisis crea resúmenes que cuantifican la cantidad máxima de objetos vivos y los objetos que escapan. Como se trata de un análisis composicional basado en resúmenes de métodos, los mismos se construyen utilizando los resúmenes de sus métodos invocados.

En contraste con otros trabajos relacionados que infieren consumo basado en la resolución de ecuaciones de recurrencia [29] o que verifican anotaciones de consumo proporcionadas por el usuario [43], nuestro análisis infiere el consumo de memoria basado en cálculos simbólicos sobre espacios de iteración. Por esta razón, nuestra implementación utiliza la calculadora simbólica Barvinok [107] para operar y resolver complejas expresiones aritméticas simbólicas no lineales.

[4]Código fuente disponible en: `https://github.com/edgardozoppi/consume-net`.

Traductor de Bytecode a Boogie

Boogie [73] es un lenguaje de verificación intermedio, pensado como una capa sobre la cual se pueden construir verificadores de programas para otros lenguajes. Se han construido varios verificadores de programas de esta manera, incluidos los verificadores VCC y HAVOC para C, así como también los verificadores para Dafny, Chalice y Spec#.

Boogie también es el nombre de una herramienta [34], que acepta el lenguaje Boogie como entrada, opcionalmente infiere algunos invariantes en el programa Boogie dado, y luego genera condiciones de verificación que se pasan a un solucionador de satisfacibilidad módulo teorías (SMT) como Z3 [50].

BCT [36] es un traductor de .NET CIL *bytecode* al lenguaje de verificación Boogie. Proporciona un vehículo para convertir cualquier verificador de programas para Boogie en un verificador para un lenguaje que compila para la plataforma .NET. Fue construido utilizando CCI [15] para cargar el CIL *bytecode* del programa .NET que quiere traducir.

Sin embargo, dados los diversos problemas y errores encontrados e informados en BCT, y la falta de soporte de esta herramienta en los últimos años, un nuevo proyecto de código abierto llamado TinyBCT[5] se creó recientemente para solucionar esos problemas.

Análisis de Excepciones Lanzadas

Contractor.NET[6] [119, 118] es una herramienta de código abierto que toma una componente de software y genera una representación abstracta finita del mismo, llamada EPA. Al analizar la máquina de estados finitos resultante, el usuario puede descubrir posibles anomalías en el contrato de entrada, como invariantes, condiciones previas o condiciones posteriores que pueden ser demasiado débiles, permitiendo comportamiento no deseado. También le permite al usuario aumentar

[5]Código fuente disponible en: `https://github.com/m7nu3l/TinyBCT`.

[6]Código fuente disponible en: `http://lafhis.dc.uba.ar/contractor/contractor.net-web/`.

la especificación de contratos original para la clase de entrada con la información de *typestate* inferida, permitiendo la verificación de código cliente. Contractor.NET admite dos *backends* diferentes para resolver las consultas de alcanzabilidad necesarias para construir una EPA para una clase: el verificador estático de Code Contracts Clousot [78], basado en interpretación abstracta, y Corral [68], un verificador de modelos de software acotado.

Contractor.NET trabaja principalmente a nivel de AST y se creó principalmente utilizando CCI [15] para cargar el CIL *bytecode* del programa .NET bajo análisis. Sin embargo, hace uso de nuestro *framework* de análisis de programas para implementar un análisis estático simple pero a su vez útil para recopilar todos los tipos de excepciones que un determinado método podría lanzar durante el tiempo de ejecución.

Chapter 11

Related Work

Many closely related program analysis resources exist, but only a few support applications targeting the .NET platform. For this reason, Ferrara *et al.* [55] presents a tool that converts .NET CIL to Java bytecode for static analysis leveraging. A formal translation is introduced and proved sound with respect to the language semantics. The main result of this work is to take advantage of already existing, mature and sound analyzers for Java bytecode by applying them to the (translated) CIL bytecode.

However, it does not support some specific features of .NET that are not available in Java, such as unsafe code and low-level pointer manipulation. Additionally, delegates and more complex language constructs such as `async` and `await` statements are translated using reflection which is often not very well supported by many Java static analyzers, leading to precision loss due to the translation process. Instead, we consider that both platforms are different enough to justify the development of a specific static analysis framework for .NET programs.

Analysis of .NET Programs

Most of the following tools and libraries are related but orthogonal to our work. Moreover, our framework can be extended by implementing a code provider for each of them. That way, it could be easily adopted by applications that need to perform static analyses but are already built on top of those libraries. In addition,

non of these tools implement or provide a sequential intermediate representation like three-address code (TAC). They all work at the abstract syntax-tree (AST) level or directly at the bytecode level, and as discussed in Section 8.2, these are not adequate code representations for implementing static analyses.

Roslyn. The .NET Compiler Platform (Roslyn) [84] is the official open-source C# and Visual Basic compiler developed by Microsoft. It provides rich code analysis features that enable building tools with the same APIs that are used by the Visual Studio IDE. It counts with full Microsoft support and its widely used by a large community.

The core mission of the Roslyn APIs is opening up the black boxes and allowing tools and end users to share in the wealth of information compilers have about code. Instead of being opaque source-code-in and object-code-out translators, through Roslyn, compilers become platforms: APIs that can be used for code-related tasks in client tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and interactive use of programming languages, among others. Roslyn enable developers to build analyzers and code fixes that find and correct coding mistakes. Analyzers understand the syntax and structure of code and detect practices that should be corrected. Code fixes provide one or more suggested fixes for addressing coding mistakes found by analyzers. They can point out practices that often lead to bugs (code smells), unmaintainable code or standard guideline validation (coding styles).

It defines distinct concrete syntax-tree data structures for each language, as well as a more general and unique AST for both of them, called `IOperation`. Several compiler phases including parsing, semantic checking and code generation are also exposed to the end user. Semantic analysis is restricted to recognizing the role of token identifiers (similar to querying the symbol table of the compiler) and the type inference/checking of expressions.

It is a great tool to build syntax-based code analysis or to compile on the fly arbitrary source code. However, it does not provide a useful intermediate

representation, such as a TAC, suitable to implement a static analysis on top of it. Only complex tree-based code representations are available. Also, because of being a compiler, it takes source code as input, so is not possible to analyze third-party libraries referenced by the program under analysis for which its source code is not available. Neither to analyze .NET programs written in a different programming language other than C# or Visual Basic, such as F#. Classical control-flow and data-flow analyses are not provided either[1]. However, it allows users to efficiently modify and transform source code and to generate executable .NET programs.

CCI. The Common Compiler Infrastructure (CCI) [15] is a set of open-source libraries developed by Microsoft Research for creating, reading and manipulating .NET metadata and CIL bytecode. It comes in two flavors: CCI Metadata API and CCI Code & AST API. Most applications that use the latter also make use of the former, so typically both APIs are used together.

The CCI Metadata API allows applications to efficiently read or modify .NET assemblies, modules and debugging files (PDBs). It supports the functionality of the .NET `System.Reflection` and `System.Reflection.Emit` APIs, but with much better performance. It also provides additional functionality that is not available in either .NET API.

The CCI Code & AST API is essentially an extension of CCI Metadata and provides similar functionality. However, instead of representing method bodies as a flat list of CIL instructions, the CCI Code & AST API represents method bodies with a language-independent hierarchical object model that is similar to source code. It subsumes the functionality provided by `System.CodeDom` API.

These libraries are complementary to our work, since they provide access to raw CIL bytecode in the case of Metadata API, or an ad hoc decompiled AST version of C# in the case of Code & AST API, although it does not support many of the latest features added to the language in the last couple of years. Non of these libraries provide a TAC representation suitable for implementing static

[1]Roslyn provides related but very restricted APIs to compute statement reachability and lexical scoping of local variables.

analyses, nor offer classical control-flow and data-flow analyses. However, it allows the transformation and generation of .NET programs using the object model or by direct bytecode manipulation.

Initially, our framework made use and built on top of the CCI Metadata API to read .NET assemblies and CIL bytecode. Later on, we decided to decouple our implementation from directly using CCI because of the many issues and bugs we found during development and the lack of support given to these currently deprecated libraries. Instead, we have opted to use the new official and cross-platform `System.Reflection.Metadata` API fully supported by Microsoft (and used by Roslyn internally), that provides the same features more efficiently and up to date. However, it is still possible to use CCI within our framework through the CCI code provider we have implemented, to load metadata and construct instances of our more general object model and simplified bytecode.

Cecil. Mono.Cecil [8] is an open-source library to generate and inspect programs and libraries in the ECMA [14] CIL format. Cecil allows reading .NET binaries using a simple and powerful object model, without having to load assemblies to use Reflection. It also enables the modification of .NET binaries, by adding new metadata structures or altering CIL bytecode. Cecil has been around since 2004 and is widely used in the .NET community because of its up to date features and active support.

This library provides almost the same functionality of CCI Metadata API, thus it has the same benefits but also lacks the same features required to easily implement static analyses, such us an adequate intermediate code representation. Only raw CIL bytecode is provided with no built-in static analyses included. In fact, Cecil could be used to implement an additional code provider for our framework.

ILSpy. ILSpy [16] is an open-source .NET metadata browser and decompiler tool. Many front-ends are available, including a standalone GUI application comparable to our Analysis Explorer tool and a Visual Studio extension plug-in.

This tool provides similar functionality than CCI Code & AST API, but is currently supported and up to date with the latest language features. For instance, it

correctly decompiles `async` code and generates C# source code that makes use of most syntactic sugar language constructs. Internally, it takes advantage of Cecil to load .NET metadata and CIL bytecode. The reverse engineer process of decompilation is done by several code transformation phases that convert the original CIL bytecode to a tree-based code representation similar to an AST. At each phase, specific low-level code patterns are used to recognize higher-level constructions.

Even that ILSpy internally implements a few static analyses required by the decompilation process, its goal is not to provide a general .NET program analysis framework. Instead, it completely focus on generating human readable C# source code from .NET assemblies.

Program Analysis Frameworks

There are several frameworks that provide the same kind of features, but to the best of our knowledge, all of them target different platforms. We are not aware of any other publicly available framework supporting the analysis of .NET programs.

Soot. As mentioned before, our work is inspired in the Java optimization framework Soot [105, 70]. Since it was first released in 2000, it has been widely used for educational purposes and research in general, and it is currently supported by a large program analysis community around the world. Soot allows the manipulation, analysis and transformation of Java programs. Its key features include a simplified TAC intermediate representation of Java bytecode, pointer analysis and call graph construction algorithms and the ability to produce executable code as output. While Jimple is the most popular and fundamental IR provided, a SSA form is also available.

The design of our framework was greatly influenced by Soot. For this reason, both provide very similar features, although Soot is a much more mature project, with many users and client applications, and has a wealth of analyses already implemented on top of it. However, our framework design tries to be more general to support not only .NET related programming languages, but also allows the possibility to analyze other similar platforms in the future.

WALA. The T. J. Watson Libraries for Analysis (WALA) [18] provide static analysis capabilities for Java bytecode and related languages. It also supports the analysis of JavaScript applications. The initial WALA infrastructure was developed as part of a IBM research project. In 2006, IBM donated the software to the community under an open-source license and since then, many research projects and publications made use of it.

WALA consists in a set of Java libraries for static and dynamic program analysis. Their key design goals are robustness, efficiency and extensibility. Instead, our framework design prioritizes simplicity and usability over performance. Similar to our general language-agnostic approach, it supports multiple language front-ends, enabling the analysis of both Java bytecode and JavaScript source code. They also claim to have an internal port for .NET CIL, but there is no further information available about it. WALA mainly focuses on computing analysis data, with limited code transformation support. Its main SSA-based TAC intermediate representation is immutable, with no provided code generator. Some of the main features supported by WALA are context-sensitive pointer analysis and call graph construction, inter-procedural data-flow analysis solver and other general analysis utilities and data structures.

LLVM. The LLVM Compiler Infrastructure [17] is a collection of modular and reusable compiler and toolchain technologies, often used to develop compiler front-ends and back-ends. Despite its name, LLVM has little to do with traditional virtual machines. Since it first started in 2000, it has grown to be an umbrella project consisting of a number of subprojects, many of which are used in production by a wide variety of commercial and open-source applications, as well as being widely used in academic research. A major strength of LLVM is its versatility, flexibility and reusability.

The LLVM Core libraries are written in C++ and provide a modern source and target-independent optimizer, along with code generation support for many popular CPUs. These libraries are built around a well specified code representation known as the LLVM intermediate representation (LLVM IR), which is a low-level,

SSA-based and strongly typed assembly language. Related tools built on top of the core libraries are also available, including compilers, debuggers and linkers.

It provides a wealth of state-of-the-art static analyses and code transformations already implemented and ready to be used in client applications. However, so far there is no support for translating CIL bytecode to the LLVM IR or a front-end for any of the .NET framework programming languages.

Resumen

Trabajo Relacionado

Existen muchos recursos de análisis de programas estrechamente relacionados, pero solamente unos pocos soportan aplicaciones basadas en la plataforma .NET. Por este motivo, Ferrara *et al.* [55] presenta una herramienta que convierte .NET CIL al bytecode de Java para beneficio de los análisis estáticos. Se introduce una traducción formal y se prueba con respecto a la semántica del lenguaje. El principal resultado de este trabajo es aprovechar los analizadores maduros y correctos ya existentes para el bytecode de Java y aplicarlos al bytecode CIL traducido.

Sin embargo, no soporta algunas características específicas de .NET que no están disponibles en Java, como el uso de código inseguro y la manipulación de punteros a bajo nivel. Además, los *delegates* y otras construcciones del lenguaje más complejas como las instrucciones `async` y `await` se traducen utilizando *reflection* que comúnmente no es soportado por muchos analizadores estáticos de Java, lo que ocasiona una pérdida de precisión debido al proceso de traducción. Por lo tanto, consideramos que ambas plataformas son lo suficientemente diferentes como para justificar el desarrollo de un *framework* de análisis estático específico para programas .NET.

Análisis de Programas .NET

La mayoría de las herramientas y bibliotecas están relacionadas pero son ortogonales a nuestro trabajo. Además, nuestro *framework* puede extenderse imple-

mentando un proveedor de código para cada una de ellas. De esa manera, podría ser fácilmente adoptado por aplicaciones que necesitan realizar análisis estáticos pero que ya están construidas sobre otras bibliotecas. Además, ninguna de estas herramientas implementa o proporciona una representación intermedia secuencial como es el código de tres direcciones (TAC). Todos ellos trabajan al nivel del árbol de sintaxis abstracta (AST) o directamente al nivel del bytecode, y como se explica en la Sección 8.2, estas no son representaciones de código adecuadas para implementar análisis estáticos de programas.

Herramientas para el Análisis de Programas

Existen varios *frameworks* que proporcionan el mismo tipo de características, pero según sabemos, todos ellos se enfocan en plataformas diferentes. No tenemos conocimiento de ningún otro *framework* disponible públicamente que soporte el análisis de programas .NET.

Part IV

Epilogue

Chapter 12

Conclusions

In this thesis we present the design and implementation of a wide range of static analyses for the .NET platform. We mainly focus in scalability, since this is the most important concern when applying static analysis techniques to real-world, industrial-size programs. We target the .NET platform given its popularity in the industry and the rich set of features it provides, ranging from object-oriented to functional paradigms, including concurrent programming and low-level pointer manipulation. The combination of all these features make static analysis very challenging.

As modern development is increasingly moving to large online cloud-backed repositories such as GitHub, BitBucket or Visual Studio Team Services, is natural to wonder what kind of analysis can be performed on large bodies of code. In the first part of this thesis, we explore an analysis architecture in which static analysis is executed on a *distributed cluster* composed of legacy VMs available from a commercial cloud provider. In this context, we present a static analysis approach based on the actor model and designed for *elasticity* (i.e., to scale gracefully with the size of the input).

To demonstrate the potential of our approach, we show how a typical call graph analysis can be implemented and deployed in Microsoft Azure. Additionally, we extend our distributed call graph analysis to support incremental source code updates submitted to centralized source control repositories like Git, and

show how the previously computed results can be updated without having to recompute them from scratch. Only reanalyzing the modified parts of the program is required, giving a significant performance improvement. We experimentally validate our analyses using a combination of both synthetic and real benchmarks. Our call graph analysis implementation is able to handle inputs that are almost 10 million LOC in size. Our results show that our analysis scales well in terms of memory pressure independent of the input size, as we add more VMs. Despite using stock hardware and incurring a non-trivial communication overhead, our processing time for some of the benchmarks of close to 1 million LOC can be about 5 minutes, excluding compilation time. As the number of analysis VMs increases, we show that the analysis time does not suffer. We also demonstrate that querying the results can be performed with a median latency of 15 ms. Lastly, our incremental call graph analysis shows significant time savings of about 250X average speedup in comparison to the full exhaustive analysis.

At the same time, the poor offer of program analysis related tools targeting the .NET platform, and the difficulty of analyzing .NET programs with the few existing available ones, puts in evidence an arising necessity of providing better tools. In the second part of this thesis, we present a static program analysis framework specifically designed for analyzing .NET programs. We also describe all of its features in detail, including a few intermediate code representations such as a three-address code suitable for implementing a static analysis on top of it, and a rich set of analyses and transformations such as type inference, control-flow and data-flow analyses, and call graph and points-to graph construction, among many others. We don't know of any other static analysis framework publicly available to the .NET community providing these kind of features that developers can use to build their own analyses for the .NET platform.

To demonstrate the capabilities of our framework, we also present a few client applications that take advantage of its features to implement their own custom analyses targeting the .NET platform. Including a Big Data query optimization analysis to automatically detect unused columns and dependencies between input and output tables of user-defined operators written in a .NET-based programming language like C#.

12.1 Future Work

As future work we plan to investigate the performance of other instances of our distributed framework and understand the impact of changing the granularity of actors (e.g., from basic blocks to modules). We want to combine distributed processing with incremental analysis: we are ultimately interested in deploying an Azure-based distributed incremental analysis that can respond quickly to frequent updates in the code repository. We would also like to incorporate the analysis into an IDE and perform user studies.

Additionally, we plan to further evaluate our incremental call graph analysis by performing experiments to help us understand in which kind of scenarios the incremental analysis version is more efficient in comparison to just running the full exhaustive analysis. We would like to find a heuristic to determine which analysis version is more convenient to use considering the percentage of source code changed (i.e., by measuring modified documents or methods).

Regarding our .NET static program analysis framework, we plan to complete the support for Java bytecode to also enable the analysis of Java related programs. We are also very interested in supporting the generation of executable .NET assemblies from our simplified bytecode or even directly from our three-address code. An alternative but related approach would be the implementation of an interpreter for those intermediate representations. Additionally, we want to provide an implementation of the IFDS/IDE framework for performing inter-procedural data-flow analysis efficiently.

Finally, we want to migrate the code base to target .NET Core instead of the .NET Framework to support cross-platform development. We also would like to integrate our framework with the Visual Studio Code text editor by developing an extension with similar features to the one we already have for the full Visual Studio IDE. In addition, we plan to provide new analyses and code transformations, as well as actively support the framework by fixing bugs, implementing more tests and writing better documentation for the community.

Resumen

Conclusiones

En esta tesis presentamos el diseño y la implementación de una amplia gama de análisis estáticos para la plataforma .NET. Nos centramos principalmente en la escalabilidad, ya que esta es la problemática más importante a la hora de aplicar técnicas de análisis estático a programas industriales de tamaño real. Nos enfocamos en la plataforma .NET dada su popularidad en la industria y el amplio conjunto de características que ofrece, que van desde paradigmas orientados a objetos hasta funcionales, incluyendo programación concurrente y manipulación de punteros a bajo nivel. La combinación de todas estas características hace del análisis estático un desafío.

Dado que el desarrollo moderno se está moviendo cada vez más a grandes repositorios en línea respaldados en la nube, como GitHub, BitBucket o Visual Studio Team Services, es natural preguntarse qué tipo de análisis se puede realizar en grandes cantidades de código. En la primera parte de esta tesis, exploramos una arquitectura de análisis en la que el análisis estático se ejecuta en un *cluster distribuido* compuesto por máquinas virtuales disponibles por medio de un proveedor comercial de la nube. En este contexto, presentamos un enfoque de análisis estático basado en el modelo de actores y diseñado con foco en *elasticidad* (es decir, para escalar con gracia con el tamaño de la entrada).

Al mismo tiempo, la escasa oferta de herramientas relacionadas con el análisis de programas para la plataforma .NET, y la dificultad de analizar los programas

.NET con las pocas existentes, ponen en evidencia una necesidad creciente de proporcionar mejores herramientas. En la segunda parte de esta tesis, presentamos un marco de análisis estático de programas diseñado específicamente para analizar programas .NET. También describimos todas sus características en detalle, incluyendo algunas representaciones intermedias de código, como un código de tres direcciones adecuado para implementar un análisis estático, y un amplio conjunto de análisis y transformaciones como la inferencia de tipos, análisis de flujo de control y flujo de datos, y construcción de *call graphs* y *points-to graphs*, entre muchos otros. No conocemos ningún otro *framework* de análisis estático disponible públicamente para la comunidad .NET que proporcione este tipo de funcionalidades, que los desarrolladores pueden usar para crear sus propios análisis para la plataforma .NET.

Appendix

The following tables and figures show some additional analysis statistics, including analysis and compilation times, both in terms of absolute values and compared to each other, as well as memory consumption as a function of the number of worker VMs used.

	X1,000							**X10,000**							**X100,000**					**X1,000,000**		
Machines	1	2	4	8	16	32	64	1	2	4	8	16	32	64	4	8	16	32	64	16	32	64
Compilation	6	11	9	20	28	32	49	48	57	68	58	67	72	88	188	205	204	207	259	281	215	352
Analysis	25	25	26	22	22	21	28	130	115	77	52	52	47	41	609	371	298	237	284	3,704	2,666	2,514

	Azure-PW							**ILSPY**							**ShareX**						
Machines	1	2	4	8	16	32	64	1	2	4	8	16	32	64	1	2	4	8	16	32	64
Compilation	305	266	269	272	276	285	308	121	115	177	216	129	189	165	85	94	88	105	108	121	162
Analysis	670	373	298	238	210	183	190	1,281	1,214	1,063	931	677	576	568	280	246	190	152	122	105	111

Table A.1: Analysis and compilation times for synthetic and real benchmarks (in *seconds*).

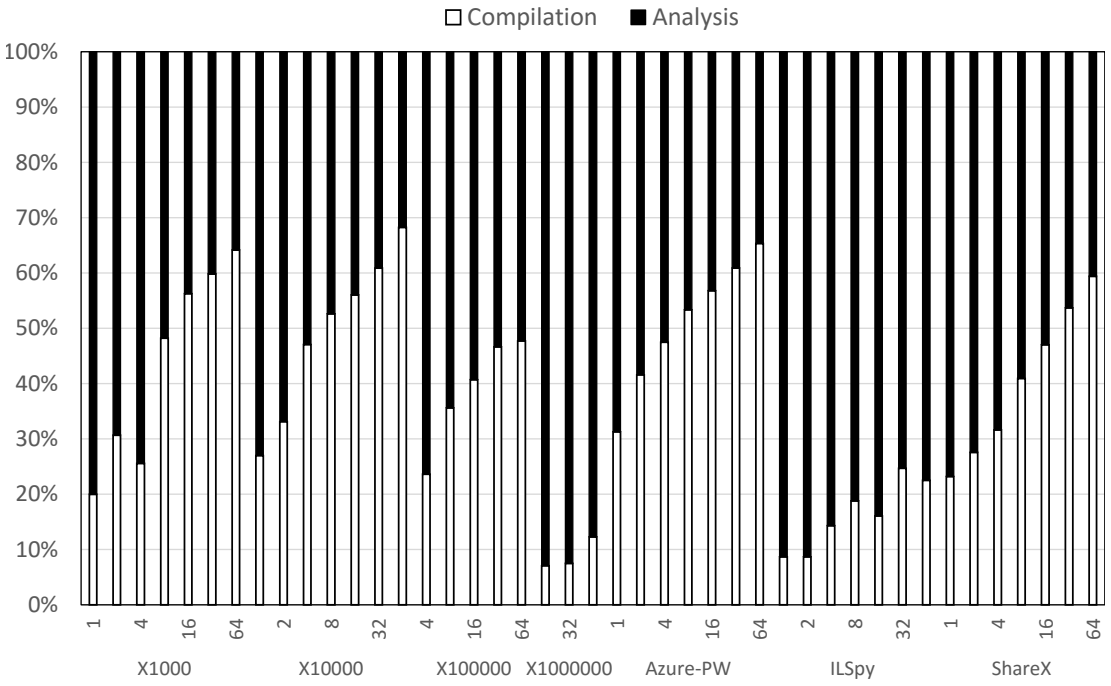


Figure A.1: Compilation time compared to analysis time for the benchmarks across a number of VM configurations, in *ms*. Analysis time can be *relatively* large when fewer VMs are involved. For 16 or 64 VMs, the analysis time can often be half of compilation time.

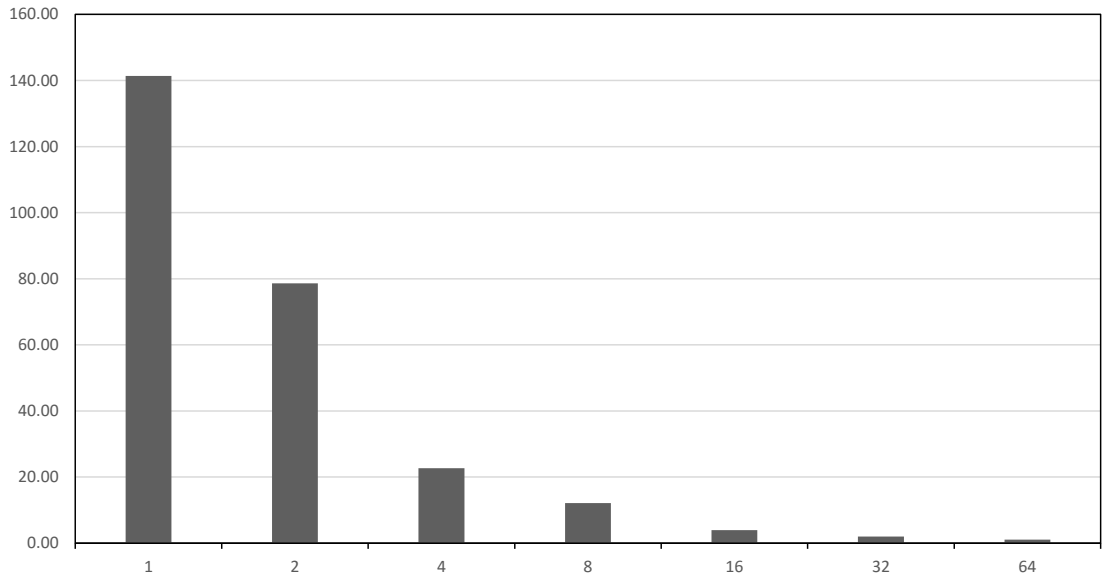


Figure A.2: Average memory consumption in KB/method, as a function of the number of worker VMs used, averaged over all the synthetic benchmarks, normalized by the number of reachable methods. We similarly observe a steady decrease as the number of worker VMs goes up. Fitting an exponential trend line to this data gives us the following formula: $M = 169.09/e^{0.728 \cdot m}$ with $R^2 = 0.99545$.

List of Algorithms

List of Figures

List of Tables

References

[1] Apache Subversion (SVN). http://subversion.apache.org. Retrieved January, 2019.

[2] BitBucket. https://bitbucket.org. Retrieved January, 2019.

[3] Git. https://git-scm.com. Retrieved January, 2019.

[4] GitHub. https://github.com. Retrieved January, 2019.

[5] GitHut: a small place to discover languages in GitHub. https://githut.info. Retrieved January, 2019.

[6] Linux Cross Referencer (LXR). http://lxr.linux.no/. Retrieved January, 2019.

[7] Mercurial. https://www.mercurial-scm.org. Retrieved January, 2019.

[8] Mono.Cecil. http://cecil.pe. Retrieved January, 2019.

[9] .NET customers. https://dotnet.microsoft.com/platform/customers. Retrieved January, 2019.

[10] .NET is open-source. https://dotnet.microsoft.com/platform/open-source. Retrieved January, 2019.

[11] .NET programming languages. https://dotnet.microsoft.com/languages. Retrieved January, 2019.

[12] PYPL: PopularitY of Programming Language index. http://pypl.github.io/PYPL.html. Retrieved January, 2019.

[13] Representational State Transfer (REST). https://en.wikipedia.org/wiki/Representational_state_transfer. Retrieved January, 2019.

[14] Standard ECMA-335: Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm. Retrieved January, 2019.

[15] The Common Compiler Infrastructure (CCI). https://github.com/Microsoft/cci. Retrieved January, 2019.

[16] The ILSpy .NET Decompiler. http://www.ilspy.net/. Retrieved January, 2019.

[17] The LLVM Compiler Infrastructure. https://llvm.org/. Retrieved January, 2019.

[18] The T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/. Retrieved January, 2019.

[19] TIOBE Programming Community index. https://www.tiobe.com/tiobe-index. Retrieved January, 2019.

[20] Top computer languages. http://statisticstimes.com/tech/top-computer-languages.php. Retrieved January, 2019.

[21] U-SQL data processing language. http://usql.io/. Retrieved January, 2019.

[22] Visual Studio Team Services. https://www.visualstudio.com/team-services. Retrieved January, 2019.

[23] What is .NET? https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet. Retrieved January, 2019.

[24] O. Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, 1996.

[25] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. *ACM SIGPLAN Notices*, 1995.

[26] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[27] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[28] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani. Parallelizing top-down interprocedural analyses. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2012.

[29] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap space analysis for garbage collected languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.

[30] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[31] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[32] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[33] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1996.

[34] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S.

de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387, 2006.

[35] M. Barnett, M. Fähndrich, F. Logozzo, and D. Garbervetsky. Annotations for (more) precise points-to analysis. *In IWACO*, pages 11–18, 2007.

[36] M. Barnett and S. Qadeer. BCT: A translator from MSIL to Boogie. In *Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, 2012.

[37] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2003.

[38] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.

[39] B. Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *In POPL*, pages 25–37. ACM, 1998.

[40] J. Bornholt and E. Torlak. Scaling program synthesis by exploiting existing code. *Machine Learning for Programming Languages*, 2015.

[41] V. Braberman, D. Garbervetsky, S. Hym, and S. Yovine. Summary-based inference of quantitative bounds of live heap objects. *Science of Computer Programming*, 92:56–84, 2014.

[42] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[43] R. Castaño, J. P. Galeotti, D. Garbervetsky, J. Tapicer, and E. Zoppi. On Verifying Resource Contracts using Code Contracts. In *Proceedings of the First Latin American Workshop on Formal Methods, (LAFM)*, 2013.

[44] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

[45] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of concrete type-inference in the presence of exceptions. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998.

[46] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[47] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical report, Laboratoire IMAG, Université scientifique et médicale de Grenoble, 1977.

[48] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[49] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[50] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[51] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[52] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the International Conference on Software Engineering*. IEEE Press, 2013.

[53] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of Java language features. In *Proceedings of the International Conference on Software Engineering*, 2014.

[54] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. In *ACM SIGPLAN Notices*. ACM, 2013.

[55] P. Ferrara, A. Cortesi, and F. Spoto. CIL to Java-bytecode translation for static analysis leveraging. In *2018 IEEE/ACM 6th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 40–49. IEEE, 2018.

[56] D. Garbervetsky, Z. Pavlinovic, M. Barnett, M. Musuvathi, T. Mytkowicz, and E. Zoppi. Static analysis for optimizing Big Data queries. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 932–937. ACM, 2017.

[57] D. Garbervetsky, E. Zoppi, T. Ball, and B. Livshits. Toward full elasticity in distributed static analysis. *Microsoft Research Technical Report*, 2016.

[58] D. Garbervetsky, E. Zoppi, and B. Livshits. Toward full elasticity in distributed static analysis: the case of callgraph analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 442–453. ACM, 2017.

[59] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 2001.

[60] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 1997.

[61] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, pages 121–133, 2012.

[62] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2007.

[63] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2011.

[64] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2001.

[65] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(2):11, 2007.

[66] U. Ismail. Incremental call graph construction for the Eclipse IDE. *University of Waterloo Technical Report*, 2009.

[67] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[68] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification*, pages 427–443. Springer, 2012.

[69] M. S. Lam, J. Whaley, B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Symposium on Principles of Database Systems*, June 2005.

[70] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[71] J.-L. Lassez, V. Nguyen, and E. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 1982.

[72] Y. Y. Lee, S. Harwell, S. Khurshid, and D. Marinov. Temporal code completion and navigation. In *Proceedings of the International Conference on Software Engineering*, 2013.

[73] K. R. M. Leino. This is boogie 2. *manuscript KRML*, 178(131), 2008.

[74] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the Conference on Compiler Construction*, 2003.

[75] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the International Conference on Compiler Construction*, 2006.

[76] B. Liu, J. Huang, and L. Rauchwerger. Rethinking incremental and parallel pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(1):6, 2019.

[77] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 2015.

[78] F. Logozzo. Clousot: Static contract checking with abstract interpretation. *Formal Verification of Object-Oriented Software*, page 5.

[79] N. P. Lopes and A. Rybalchenko. Distributed and predictable software model checking. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.

[80] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2013.

[81] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2010.

[82] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 2013.

[83] M. Mendez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[84] Microsoft Corporation. The .NET Compiler Platform (Roslyn). `https://github.com/dotnet/roslyn`. Retrieved January, 2019.

[85] S. S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[86] K. Murray, J. P. Bigham, et al. Beyond autocomplete: Automatic function definition. In *Proceedings of the Visual Languages and Human-Centric Computing Symposium*, 2011.

[87] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the International Conference on Software Engineering*.

[88] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[89] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the International Conference on Software Engineering*, 2012.

[90] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-oriented Languages, Systems, and Applications*, 1991.

[91] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[92] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 1994.

[93] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[94] J. Rodriguez and O. Lhoták. Actor-based parallel dataflow analysis. In *International Conference on Compiler Construction*, pages 179–197. Springer, 2011.

[95] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[96] B. G. Ryder. Incremental data flow analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 167–176. ACM, 1983.

[97] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, 2015.

[98] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[99] A. Sălcianu and M. Rinard. Purity and side effect analysis for java programs. In *In VMCAI*, pages 199–215. Springer, 2005.

[100] A. L. Souter and L. L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 682–691. IEEE, 2001.

[101] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[102] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 1996.

[103] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2000.

[104] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2000.

[105] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[106] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[107] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

[108] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 2007.

[109] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[110] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[111] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, 2004.

[112] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *ACM Sigplan Notices*, volume 34, pages 187–206. ACM, 1999.

[113] S. Xia, M. Fähndrich, and F. Logozzo. Inferring dataflow properties of user defined table processors. In *SAS*, pages 19–35, 2009.

[114] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[115] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2010.

[116] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[117] J. Zhou, N. Bruno, M. Wu, P. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet mapreduce. *VLDB J.*, 21(5):611–636, 2012.

[118] E. Zoppi. Enriqueciendo Code Contracts con Typestates. Master's thesis, Universidad de Buenos Aires (UBA), 2012.

[119] E. Zoppi, V. Braberman, G. de Caso, D. Garbervetsky, and S. Uchitel. Contractor.NET: Inferring Typestate Properties to Enrich Code Contracts. In *Proceedings of the 1st Workshop on Developing Tools as Plug-ins (TOPI)*, pages 44–47. ACM, 2011.