FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

Advanced Topics in Resource Analysis: Certification, Incrementality, Concurrency and Array-Sensitivity

Tesis Doctoral

Guillermo Román Díez

Diciembre de 2012

Tesis Doctoral

Advanced Topics in Resource Analysis: Certification, Incrementality, Concurrency and Array-Sensitivity

presentada en la Facultad de Informática
de la Universidad Politécnica de Madrid
para la obtención del título de
Doctor en Software y Sistemas

Candidato: Guillermo Román Díez

Ingeniero en Informática
Máster en Tecnologías de la Información
Universidad Politécnica de Madrid, España

Director: Elvira Albert

Profesora Titular de Universidad
Universidad Complutense de Madrid

Director: Germán Puebla

Profesor Titular de Universidad
Universidad Politécnica de Madrid

Madrid, Diciembre de 2012

A Gloria, mi compañera de viaje.

A mis padres, Martiniano y Eloísa.

Agradecimientos

Me gustaría expresar mi agradecimiento a todas las personas que, de un modo un otro, han ayudado a que este difícil objetivo haya sido posible.

Primero quiero mostrar mi más sincero agradecimiento a Germán Puebla y a Elvira Albert, mis directores de la tesis doctoral, las dos personas sin las cuales este trabajo no hubiera sido posible. Gracias a la confianza y apoyo de Germán Puebla pude dar el salto definitivo desde el trabajo en la empresa privada a la investigación, dándome la privilegiada oportunidad de trabajar en un grupo de investigación como COSTA. Y, por supuesto, gracias a Elvira Albert, cuya inagotable capacidad de trabajo, conocimientos y paciencia están al alcance de muy pocos, y que ha sido absolutamente clave en el desarrollo de esta tesis doctoral. Será muy difícil estarles suficientemente agradecido.

Así mismo, quiero agradecer a otros investigadores del grupo COSTA la ayuda que me han prestado a lo largo de estos años. A Samir Genaim, cuyos enormes conocimientos están al mismo nivel que su capacidad y paciencia para explicarlos. A Puri Arenas, cuya capacidad de trabajo y colaboración siempre va acompañada de su gran sentido del humor. Quiero hacer un agradecimiento especial a Jesús Correas, quien, no sólo me ayudó en mis primeros trabajos de investigación, sino que siempre ha tenido un consejo cuando han aparecido dificultades. También quiero agradecer a María García de la Banda su revisión de la tesis y sus constructivos comentarios, que, sin ninguna duda, me han ayudado a mejorar el resultado final. No quiero olvidarme de otros compañeros de COSTA, Abu Naser Masud, José Miguel Rojas y Diana Ramírez, con los que he compartido el día a día del laboratorio, con las alegrías y dificultades propias de la vida investigadora.

En lo personal, quiero agradecerle a Gloria, mi compañera de viaje, todo su apoyo, paciencia, comprensión, confianza a lo largo de tantos y tantos años. A su lado he aprendido a disfrutar de los buenos momentos y a afrontar y superar con éxito todas las dificultades. Sin ella no sería quien soy.

Y por último, quiero hacer una mención especial a mis padres, Martiniano y Eloísa, quienes, no sólo me han inculcado los valores personales y la educación necesaria para afrontar los retos que se plantean a lo largo de una vida, sino que siempre han estado ahí, tanto en los buenos como en los malos momentos.

A todos ellos, Gracias.

Sinopsis

El *Análisis de Consumo de Recursos* o *Análisis de Coste* trata de aproximar el coste de ejecutar un programa como una función dependiente de sus datos de entrada. A pesar de que existen trabajos previos a esta tesis doctoral que desarrollan potentes marcos para el análisis de coste de programas orientados a objetos, algunos aspectos avanzados, como la eficiencia, la precisión y la fiabilidad de los resultados, todavía deben ser estudiados en profundidad. Esta tesis aborda estos aspectos desde cuatro perspectivas diferentes:

(1) Las estructuras de datos compartidas en la memoria del programa son una pesadilla para el análisis estático de programas. Trabajos recientes proponen una serie de condiciones de *localidad* para poder mantener de forma consistente información sobre los atributos de los objetos almacenados en memoria compartida, reemplazando éstos por variables *locales* no almacenadas en la memoria compartida. En esta tesis presentamos dos extensiones a estos trabajos: la primera es considerar, no sólo los accesos a los atributos, sino también los accesos a los elementos almacenados en arrays; la segunda se centra en los casos en los que las condiciones de localidad no se cumplen de forma incondicional, para lo cual, proponemos una técnica para encontrar las precondiciones necesarias para garantizar la consistencia de la información acerca de los datos almacenados en memoria.

(2) El objetivo del análisis *incremental* es, dado un programa, los resultados de su análisis y una serie de cambios sobre el programa, obtener los nuevos resultados del análisis de la forma más eficiente posible, evitando reanalizar aquellos fragmentos de código que no se hayan visto afectados por los cambios. Los analizadores actuales todavía leen y analizan el programa completo de forma no incremental. Esta tesis presenta un *análisis de coste incremental*, que, dado un cambio en el programa, reconstruye la información sobre el coste del programa de todos los métodos afectados por el cambio de forma incremental. Para esto, proponemos (i) un algoritmo multi-dominio y de punto fijo que puede ser utilizado en todos los análisis globales necesarios para inferir el coste, y (ii) una novedosa forma de almacenar las expresiones de coste que nos permite reconstruir de forma

incremental únicamente las funciones de coste de aquellos componentes afectados por el cambio.

(3) Las *garantías de coste* obtenidas de forma automática por herramientas de análisis estático no son consideradas totalmente fiables salvo que la implementación de la herramienta o los resultados obtenidos sean verificados formalmente. Llevar a cabo el análisis de estas herramientas es una tarea titánica, ya que se trata de herramientas de gran tamaño y complejidad. En esta tesis nos centramos en el desarrollo de un marco formal para la verificación de las garantías de coste obtenidas por los analizadores en lugar de analizar las herramientas. Hemos implementado esta idea mediante la herramienta COSTA, un analizador de coste para programas Java y KeY, una herramienta de verificación de programas Java. De esta forma, COSTA genera las garantías de coste, mientras que KeY prueba la validez formal de los resultados obtenidos, generando de esta forma *garantías de coste verificadas*.

(4) Hoy en día la concurrencia y los programas distribuidos son clave en el desarrollo de software. Los *objetos concurrentes* son un modelo de concurrencia asentado para el desarrollo de sistemas concurrentes. En este modelo, los objetos son las unidades de concurrencia y se comunican entre ellos mediante llamadas asíncronas a sus métodos. La distribución de las tareas sugiere que el análisis de coste debe inferir el coste de los diferentes componentes distribuidos por separado. En esta tesis proponemos un análisis de coste *sensible a objetos* que, utilizando los resultados obtenidos mediante un análisis de *apunta-a*, mantiene el coste de los diferentes componentes de forma independiente.

FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

Advanced Topics in Resource Analysis: Certification, Incrementality, Concurrency and Array-Sensitivity

PhD Thesis

Guillermo Román Díez

December 2012

PhD Thesis

Advanced Topics in Resource Analysis: Certification, Incrementality, Concurrency and Array-Sensitivity

presented at the Computer Science School
of the Technical University of Madrid
in partial fulfillment of the degree of
PhD in Software and Systems

PhD Candidate: **Guillermo Román Díez**
Computer Science Engineer
Master in Information Technologies
Technical University of Madrid, España

Advisor: **Elvira Albert**
Associate Professor
Complutense University of Madrid

Advisor: **Germán Puebla**
Associate Professor
Technical University of Madrid

Madrid, December 2012

To Gloria, my travel partner.

To my parents, Martiniano and Eloísa.

Acknowledgements

I would like to express my gratitude to all people who, in one way or another, have helped me to achieve this challenging goal.

First of all, I want to express my most sincere gratitude to my PhD thesis advisors, Germán Puebla and Elvira Albert, who have made it possible to achieve this objective. Thanks to the confidence and support of Germán Puebla I have been able to finally jump from working in private companies to the research world. He gave me the privileged opportunity to work in the exciting research group COSTA. And, of course, thanks to Elvira Albert, whose endless working capacity, knowledge and patience are exceptional. Her support has been crucial in the development of this thesis. It will be difficult to be sufficiently grateful to both of them.

Likewise, I want to thank other researchers of the COSTA team for the help granted during the last years. Samir Genaim whose knowledge is at the same great level than his capability and patience to transmit it. Puri Arenas, whose work and collaboration come always accompanied by her great sense of humor. I want to specially thank Jesús Correas who, not only helped me in my first research works, but also had a wise advice to give when difficulties appeared. I am also grateful to Maria Garcia de la Banda for detailed comments on the form and contents of the thesis. I do not want to forget other COSTA colleages, Abu Naser Masud, José Miguel Rojas and Diana Ramírez, with whom I share the laboratory days, with the joys and difficulties inherent to research life.

At a personal level, I want to thank Gloria, my travel partner, for all her support, patience, comprehension and confidence during so, so many years. With her, I have learned to enjoy the good moments and to face and overcome the difficulties. Without her, I would not be who I am today.

Finally, I want to make a special mention to my parents, Martiniano and Eloísa who, not only have instilled me personal values and the required education to face all life challenges, but also they always have been there, in good and bad times.

To all of them, Thanks.

Abstract

Resource Analysis (a.k.a. *Cost Analysis*) tries to approximate the cost of executing programs as functions on their input data sizes and without actually having to execute the programs. While a powerful resource analysis framework on object-oriented programs existed before this thesis, advanced aspects to improve the efficiency, the accuracy and the reliability of the results of the analysis still need to be further investigated. This thesis tackles this need from the following four different perspectives.

(1) Shared mutable data structures are the bane of formal reasoning and static analysis. Analyses which keep track of heap-allocated data are referred to as *heap-sensitive*. Recent work proposes locality conditions for soundly tracking field accesses by means of ghost non-heap allocated variables. In this thesis we present two extensions to this approach: the first extension is to consider arrays accesses (in addition to object fields), while the second extension focuses on handling cases for which the locality conditions cannot be proven unconditionally by finding aliasing preconditions under which tracking such heap locations is feasible.

(2) The aim of *incremental* analysis is, given a program, its analysis results and a series of changes to the program, to obtain the new analysis results as efficiently as possible and, ideally, without having to (re-)analyze fragments of code that are not affected by the changes. During software development, programs are permanently modified but most analyzers still read and analyze the entire program at once in a non-incremental way. This thesis presents an *incremental resource usage analysis* which, after a change in the program is made, is able to reconstruct the upper-bounds of all affected methods in an incremental way. To this purpose, we propose (i) a *multi-domain* incremental fixed-point algorithm which can be used by all global analyses required to infer the cost, and (ii) a novel form of *cost summaries* that allows us to incrementally reconstruct only those components of cost functions affected by the change.

(3) Resource guarantees that are automatically inferred by static analysis tools are generally not considered completely trustworthy, unless the tool implementation or the results are formally verified. Performing full-blown verification

of such tools is a daunting task, since they are large and complex. In this thesis we focus on the development of a formal framework for the verification of the resource guarantees obtained by the analyzers, instead of verifying the tools. We have implemented this idea using COSTA, a state-of-the-art cost analyzer for Java programs and KeY, a state-of-the-art verification tool for Java source code. COSTA is able to derive *upper-bounds* of Java programs while KeY proves the validity of these bounds and provides a certificate. The main contribution of our work is to show that the proposed tools cooperation can be used for automatically producing *verified resource guarantees*.

(4) Distribution and concurrency are today mainstream. Concurrent objects form a well established model for distributed concurrent systems. In this model, objects are the concurrency units that communicate via asynchronous method calls. Distribution suggests that analysis must infer the cost of the diverse distributed components separately. In this thesis we propose a novel *object-sensitive* cost analysis which, by using the results gathered by a *points-to* analysis, can keep the cost of the diverse distributed components separate.

Contents

1 Introduction **1**

1.1 Thesis objectives 4

1.2 Structure of the Work 5

1.3 Contributions 7

2 Preliminaries **11**

2.1 Introduction 11

2.2 From Bytecode to a Rule-Based Representation 16

2.2.1 Generation of the Control Flow Graph guided by Class Analysis 16

2.2.2 Rule-Based Representation 18

2.2.3 RBR Semantics 19

2.3 From the RBR to a Cost Relation System 23

2.3.1 Context Sensitive (Pre-)Analyses 23

2.3.2 Heap Analysis 24

2.3.3 Cost Models 24

2.3.4 Size Analysis 25

2.3.5 Generation of Cost Relation System 28

2.4 From the CRS to a Closed-Form Upper Bound 29

2.4.1 Cost Relations compositionality 30

2.4.2 Stand-Alone Relations 30

2.4.3 Bottom-Up Computation 32

3 Conditional Termination of Loops over Heap-Allocated Data 35
3.1 Introduction . . . 35
3.1.1 Organization of the Chapter . . . 37
3.2 Reference Constancy Analysis . . . 37
3.2.1 The Set of Access Paths . . . 38
3.2.2 The Analysis . . . 39
3.2.3 Modular Analysis . . . 45
3.3 Heap-Sensitive Analysis . . . 54
3.3.1 Basic Locality . . . 54
3.3.2 Locality Partition . . . 56
3.3.3 Automatic Transformation . . . 58
3.3.4 Heuristics for References . . . 60
3.4 Inference of Termination Preconditions . . . 62
3.4.1 Inference of Local Termination Preconditions . . . 62
3.4.2 Inference of Global Termination Preconditions . . . 64
3.5 Experimental Evaluation . . . 66
3.6 Related Work . . . 69

4 Incremental Resource Usage Analysis 73
4.1 Introduction . . . 73
4.1.1 Organization of the Chapter . . . 75
4.2 A Fixed-Point Analysis Engine . . . 76
4.2.1 A Global Fixed-Point Analysis Engine . . . 77
4.3 Incremental Inference of Cost Relations . . . 80
4.3.1 Method Summary for Global Properties . . . 80
4.3.2 A Multi-Domain Incremental Fixed-Point Analyzer . . . 82
4.4 Generation of Cost Relations . . . 95
4.5 Incremental Inference of Upper Bounds . . . 97
4.5.1 The Notion of Cost Summary . . . 97
4.5.2 Incremental Inference of Summaries . . . 100
4.6 Experiments . . . 107
4.7 Related Work . . . 113

5 Verified Resource Guarantees **115**
5.1 Introduction . . . 115
5.1.1 Organization of the Chapter . . . 116
5.2 Upper-Bounds for Integer Manipulating Programs . . . 116
5.2.1 Main Components of an Upper Bound . . . 117
5.2.2 UBs Claim as JML Annotations . . . 120
5.3 Verification of Upper Bounds using KeY . . . 121
5.3.1 Verification by Symbolic Execution . . . 122
5.4 Upper Bounds for Heap Manipulating Programs . . . 125
5.4.1 Path-Length Analysis . . . 127
5.4.2 Cyclicity analysis . . . 128
5.4.3 Sharing analysis . . . 129
5.5 Verification of Path-Length Assertions . . . 130
5.5.1 Heap Representation . . . 130
5.5.2 Predicates for Structural Heap Properties . . . 131
5.5.3 Field Update Independence . . . 132
5.5.4 Path-Length Axiomatization . . . 133
5.6 Experimental Evaluation . . . 133
5.7 Related work . . . 136

6 Concurrency: Object-Sensitive Cost Analysis for Concurrent Objects **139**
6.1 Introduction . . . 139
6.1.1 Organization of the Chapter . . . 141
6.2 A Language with Concurrent Objects . . . 141
6.2.1 The Concurrency Model . . . 141
6.2.2 A Rule-based Representation for Concurrent Objects . . . 144
6.2.3 Operational Semantics . . . 146
6.3 Cost and Cost Models for Concurrent Programs . . . 149
6.4 Field-Sensitive Size Analysis for Concurrent OO Programs . . . 150
6.4.1 The Basic Size Analysis . . . 150
6.4.2 Class Invariants in Cost Analysis . . . 154
6.5 Points-to Analysis for Concurrent Programs . . . 155

6.5.1 The Abstract Domain 156
6.5.2 The Transfer Function 157
6.6 Object-Sensitive Resource Analysis 159
6.6.1 Object-Insensitive Analysis 160
6.6.2 Adding Cost Centers to the Equations 162
6.7 Experimental Evaluation 171
6.8 Related Work 173

7 Conclusions and Future Work **175**
7.1 Conclusions 175
7.2 Future Work 178

List of Figures

2.1 Architecture of COSTA 12
2.2 Preliminaries Running Example 14
2.3 Java bytecode for the Runnig Example 15
2.4 CFG of method decList 17
2.5 Compiling bytecode instructions (as they appear in the CFG) to rule-based instructions (t stands for the height of the stack before the instruction). 19
2.6 Operational semantics of bytecode programs in rule-based form . 21
2.7 RBR of the example 22
2.8 CRS of the example 29

3.1 Small examples to illustrate the notion of constant access path . . 38
3.2 Transfer function for reference constancy analysis 41
3.3 Running Example. Method m contains nested loops with iterator and arrays 44
3.4 RBR of the Running Example (left). Program point constancy information (right) 45
3.5 Resulting RBR after applying the transformations 60

4.1 Incremental Algorithm Running Example 75
4.2 Domains dependencies 81
4.3 Adding experiment scheme 111
4.4 Top-down development experiment scheme 112

5.1 Integer manipulating running example 118

5.2 Heap manipulating running example, with (partial) JML annotations 126

6.1 Running Example 143
6.2 The RBR and CFG for method readBlock 145
6.3 Abstract compilation. $\texttt{ABST}(b_{k:i}, \rho)=\langle\alpha_\rho(b_{k:i}), \rho'\rangle$ 152
6.4 Transfer Function (where $l \equiv o_{i\ldots p}$, and $l \oplus q \equiv o_{i\ldots p\oplus q}$). 158
6.5 Points-to analysis results for the running example. 159
6.6 Abstract (cost) operational semantics 166

List of Tables

3.1 Some examples of constraints obtained by using the heap-sensitive extension . . . 67
3.2 Statistics about the Conditional Heap-Sensitive Analysis (times in ms) . . . 68

4.1 Benchmarks information . . . 108
4.2 Touch experiment results . . . 110
4.3 Adding experiment results . . . 111
4.4 Top-down development experiments results . . . 113

5.1 Statistics about integer manipulating programs . . . 134
5.2 Statistics about heap manipulating programs . . . 135

6.1 Statistics about the Object-Sensitive Resource Analysis (times in ms.) . . . 172

Chapter 1

Introduction

Resource Analysis (a.k.a. *Cost Analysis*) tries to approximate the cost of executing programs by means of cost functions on their input data sizes without actually having to execute the programs. In the context of resource analysis, *termination analysis* can be presented as a simple type of resource analysis. Proving termination implies that the amount of resources consumed by the program is finite but we do not have a bound for it. During last two decades, a wide variety of cost analysis frameworks have been proposed, most of them for functional languages [Ben01, Ros89] and for logic programs [NMLGH07, DL93]. Partly due to the complexity of their semantics, cost analysis of imperative and object-oriented languages (OO for short), have received much less attention. However, in the last years, sophisticated resource and termination analysis frameworks for imperative languages have been developed [AAGP11, SMP10, OBvEG09, GMC09, HH10]. These frameworks have shown that it is feasible to apply a resource analysis on object oriented programs, but that some advanced aspects still need to be investigated in order to improve the efficiency, accuracy and reliability of the results. This thesis proposes novel techniques along this direction of research.

Handling shared mutable data structures, such as those stored in the heap, is one of the main challenges for static analyzers of imperative languages [Min06]. In OO languages, this problem is even worse because most of the data resides in fields or in arrays stored in the heap. The crucial aspect is to find a balance between the two extremes, a heap-insensitive analysis, which is too imprecise, and

a fully heap-sensitive analysis, which is sometimes computationally intractable. In the context of cost analysis, a field-sensitive analysis for OO languages has already been proposed [AAGP09, AAG+10, RD11]. This approach is based on the observation that by analyzing program fragments rather than the application as a whole, under certain *locality* conditions that must hold unconditionally, it is possible to keep track of heap-allocated data by means of local (non heap-allocated) variables. However, this work is focused on fields and cannot handle other data structures that are also heap-allocated, like arrays. Besides, since heap references can alias and access the same memory location, the termination of the programs might depend on aliasing conditions. Our work in Chapter 3 presents a novel conditional termination framework for programs with loops over heap-allocated data.

Resource usage analyzers are powerful tools which may be very useful during software development. Analyzers can help not only to detect bugs, like spotting non-terminating loops, but also to improve the quality of the software, helping to reduce the amount of resources consumed by the program. During software development, programs are permanently modified, e.g. because a new implementation of an existing method is provided or because a method is extended with new functionality. In such cases, existing analysis information previously computed may no longer be correct and/or accurate. Resource analysis is a costly task and its execution may require a large amount of resources. Despite the great progress made in static analysis in general, and in resource analysis in particular, most analyzers still read and analyze the entire program at once. Thus, traditional analyzers do not reuse previous analysis information and instead they reanalyze the whole program from scratch after each modification, resulting, in most cases, in an inefficient and non-practical analysis. In the context of termination analysis, a modular approach, based on the composition of the analysis results, has been proposed [RCP12]. However, modularity per se does not handle the recomputation of analysis results after a program modification. While some *incremental* analysis algorithms have been proposed in different contexts [KMSS97, Ryd88, WG97], current resource analyse for OO programs lack an incremental approach for handling modifications in the source code that resuses already computed information and recomputes only those parts of the program affected by a change. This thesis

proposes an incremental analysis framework in Chapter 4.

Another aspect that we consider in this thesis is the reliability of the analysis results. Formally proving the correctness of software and its properties can be crucial for many applications, e.g., in safety critical systems or real time systems. Program properties that are automatically inferred by static analysis tools are generally not considered to be completely trustworthy unless the tool implementation or the results are formally verified. Nowadays, a number of cost analyzers exist [AAGP11, GMC09, HH10] that can generate *resource guarantees* in a fully automatic way. Resource guarantees allow being certain to some extent that programs will run within the indicated amount of resources. Unfortunately, verifying the correctness of modern static analysis tools is rather challenging, because of the sophisticated algorithms used in them and their evolution over time. A simpler alternative is to construct a validating tool [PSS98] which, after every analysis, formally verifies the correctness of the resource guarantees by cost analyzers, as we will pursue in the Chapter 5 of this thesis.

The last issue we tackle in the thesis is the extension of the cost analysis framework to the context of distributed and concurrent programs. Distribution and concurrency are today mainstream. The Internet and the broad availability of multi-processors radically influence software. However, while cost analysis for sequential programming languages has received considerable attention, because of its complexity, concurrency and distribution have been notably less studied. Modern programming languages, like Java or C#, present a concurrency model that needs to consider too many interleaving possibilities. That ends up in a cost analysis limited to very small programs in practice. *Concurrent objects* is a concurrency model based on the notion of concurrently running (groups of) objects, similar to the actor-based and active-objects approaches [SPH10, SM08]. These models take the advantage of the concurrency implicit in the notion of object in order to provide programmers with high-level constructs that help in producing concurrent applications more modularly and in a less error-prone way. We consider such concurency model in Chapter 6 of this thesis for studying the distribution of the cost among objects.

1.1 Thesis objectives

The main objective of this thesis is to extend and improve the existing resource analysis techniques in order to make them capable of handling a larger number of programs and scenarios and doing it in a more accurate way. The starting point for this thesis is the state-of-the-art resource analisys framework proposed in [AAG+12b, AAGP11].

The first concrete objective of this thesis is to generalize the idea of field-sensitive analysis [AAGP09, AAG+10, RD11] to **heap-sensitive** analysis, by handling not only fields accesses, but also array accesses and their indexes. Besides, while in previous work locality conditions must hold unconditionally, in this thesis we seek to generate **aliasing preconditions** which, when satified in the initial state, can guarantee the termination of the program.

The second objective of this thesis is to define **incremental** algorithms to improve the performance of resource analysis. This allows handling modifications in the source code by reusing previously computed information and recomputing only those components of the cost functions affected by the changes.

Our third objective is the design and development of a formal framework for **certification** of the resource guarantees. The aim is to construct a validating tool which, after every cost analysis, formally confirms the correctness of the results. The final step of this objective is to generate correctness proofs that can be translated into *verified resource guarantees*.

The last objective of this thesis is to extend the resource analysis from sequential programs to **concurrent** programs for the *concurrent objects* model. Cost analysis for this concurrency model has been initially studied in [AAG+11]. In this thesis we aim at extending this work by adding to it *object-sensitivity*, in order to automatically separate the cost among different objects (cost centers) by using the results inferred by a *points-to* analysis.

1.2 Structure of the Work

This thesis contains the following chapters:

Chapter 2. Background: Resource Analysis in COSTA

This part of the thesis overviews the techniques used in resource analyses and sets up the terminology used in the thesis.

Chapter 3. Conditional Termination of loops over Heap-Allocated Data

We extend a previously developed semantic *constancy analysis* to consider also arrays. In an ideal scenario where fields or array accesses are unconditionally local, we can transform each heap access by its equivalent access using local variables. However, there are cases when the locality conditions cannot be proven unconditionally, e.g. two input arguments that might alias and access the same field. In such cases, it is often possible to provide *aliasing preconditions* under which tracking such heap locations is feasible. This part of the thesis seeks to generate aliasing preconditions that, when hold in the initial state, guarantee the termination of the program.

Our experimental results, performed on examples that combine the use of arrays with numeric and reference fields, show that this approach is able to automatically infer interesting preconditions and introduces a reasonable overhead on the resource analysis.

Chapter 4. Incremental Resource Usage Analysis

This part of the thesis presents an *incremental* resource usage analysis for a sequential Java-like language which, after a change in the program is made, is able to reconstruct the *upper-bounds* of all affected methods in an incremental way. The main contributions are (1) a *multi-domain* incremental fixed-point algorithm that can be used by all global pre-analyses

required to infer the cost, and that takes care of propagating dependencies among such domains, and (2) a novel form of *cost summaries* that allows us to incrementally reconstruct only those components of cost functions affected by the change.

The incremental analysis has been implemented in COSTA [AAG+09]. An experimental evaluation has been performed in selected benchmarks from the standarized *JOlden* benchmark suite [Sui] and from the *Apache Commons* Project [Pro] by simulating different development scenarios. The experimental results show that the proposed incremental analysis performs very efficiently in practice in comparison with the non-incremental approach, resulting in significant speedups.

Chapter 5. Verified Resource Guarantees

This part of the thesis investigates how to formally prove the correctness of the resource guarantees generated by cost analyzers, instead of performing a full-blown verification of the tools. This objective is implemented using COSTA for producing the resource guarantees, and KeY [BHS06], a state-of-the-art verification tool, for formally verifying the correctness of such resource guarantees.

We start the chapter by describing and formalizing the verification framework for resource guarantees that depend only on integer data. In the first part of this chapter we identify the *Java Modelling Language* annotations needed by KeY to verify the results obatined by the cost analysis of integer manipulating programs.

However, in realistic programs, the resource consumption is often bounded by the size of *heap-allocated* data structures. Bounding their size requires to perform a number of structural heap analyses. The second part of this chapter (i) identifies what exactly needs to be verified to guarantee sound analysis of heap manipulating programs, (ii) provides a suitable extension of the program logic used for verification in order to handle structural heap properties in the context of resource guarantees, and (iii) improves the

underlying theorem prover so that proof obligations can be automatically discharged.

The experimental evaluation shows that the proposed tool cooperation can be used for automatically producing *Verified Resource Guarantees* for both integer manipulating and heap manipulating Java programs.

Chapter 6. Object-Sensitive Cost Analysis for Concurrent Objects

In the concurrent objects model, objects are the concurrency *units* that communicate via *asynchronous* method calls. Distribution suggests that analyses must infer the cost of the diverse distributed components separately. We capture this distribution by means of a novel form of *object-sensitive recurrence equations* that use *cost centers* in order to keep the resource usage assigned to the different components separate.

Object-sensitive cost analysis has been implemented and evaluated on several small applications that are classical examples of concurrent programming showing that splitting the cost of concurrent programs in different cost centers is feasible in practice.

1.3 Contributions

The main contributions of this thesis are the following:

- A generalization of the field-sensitive analysis to a heap-sensitive analysis that handles both fields and array elements in an uniform way. This requires an extension of the reference constancy analysis and of the notion of locality described at [AAGP09] to determine if heap-allocated data behave as local variables.

  Sometimes, heap-allocated data behave locally only under certain aliasing conditions. This thesis introduces the notion of *locality partition* which, by assuming such aliasing conditions, guarantees the locality of the considered heap-allocated data. Based on the locality partitions, we introduce a novel

transformation and composition algorithms for automatically inferring the aliasing preconditions for the initial state that guarantee the termination of the program. An initial version of these contributions has been presented at *Bytecode'12* [AGRD12]:

> E. Albert, S. Genaim, and G. Román-Díez. **Conditional Termination of Loops over Arrays.** In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'12), Tallin, Estonia*, March 2012.

And an extended version of this work is under revision for a special issue of *Bytecode'12* in the journal *Science in Computer Programming*.

- A *multi-domain* incremental analysis engine that can be used by all global pre-analyses required to infer the resource usage of a program. Such pre-analyses include class analysis, sharing, cyclicity, constancy and size analysis. The algorithm is multi-domain in the sense that it interleaves the computation for the different domains and takes into account dependencies among them, in such a way that it is possible to invalidate only partial pre-computed information.

  Even a small change within a method (e.g., adding an instruction) can change the overall cost of the program. A fundamental idea to minimize the amount of information that needs to be recomputed is to be able to distinguish within a *cost summary* the cost subcomponent associated to each method, so that the final cost functions can be recomputed by replacing only the affected subcomponents. This work has been presented as a poster in [ACPRD11] and, as a regular paper, in [ACPRD12]:

  > E. Albert, J. Correas, G. Puebla, and G. Román-Díez. **Towards Incremental Resource Usage Analysis.** In *The Ninth Asian Symposium on Programming Languages and Systems (APLAS'11), Kenting, Taiwan.* Poster Presentaion. Springer, December 2011.

E. Albert, J. Correas, G. Puebla, and G. Román-Díez. **Incremental Resource Usage Analysis.** In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), Philadelphia, Pennsylvania, USA*, pages 25–34. ACM Press, January 2012.

- The main contribution of the third part of the thesis is to show that it is possible to formally and automatically *certificate* the correctness of the resource guarantees created by COSTA. This work is based on the idea that the static analysis tool COSTA and the formal verification tool KeY have complementary strengths. COSTA is able to derive the UB, annotate it within a Java file by using the *Java Modelling Language* (JML). Then KeY is able to prove the validity of the bounds and generate a proof. This work, published as a tool demo at *PEPM'11* [ABG+11a] focuses on integer manipulating programs only:

  E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. **Verified Resource Guarantees using COSTA and KeY**. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'11), Austin, TX, USA*, pages 73–76. ACM Press, January 2011.

  The second phase is the extension of the previous phase to handle heap manipulating programs. In particular, this work identifies the structural properties inferred by COSTA that need to be verified, and extends JML by means of suitable new constructs. A new extension of the program logic used during verification by additional theories for structural heap properties including acyclicity or disjointness of heap regions has been added to KeY. Realizing the cooperation between COSTA and KeY has required non-trivial extensions in both systems. This work has been informally presented in the "KeY Symposium 2011" [ABG+11b] and finally published in [ABG+12]:

  E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-

Díez. **Verified Resource Guarantees for Heap Manipulating Programs.** *10th KeY Symposium*, August 2011.

E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. **Verified Resource Guarantees for Heap Manipulating Programs**. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12), Tallinn, Estonia*. volume 7212 of LNCS, pages 130-145. Springer, March 2012.

- The last part of the thesis contains two relevant contributions (1) a flow-sensitive object-sensitive points-to analysis for the concurrent objects concurrency model; and (2) a novel form of *object-sensitive* cost relation system which relies on information gathered by the points-to analysis in order to generate the cost equations. Interestingly, the resulting cost relations can still be solved to closed-form upper/lower bounds using standard solvers for cost analysis for sequential programs.

  Non-distributed cost analysis for concurrent object model has been initially studied in [AAG+11] and in this thesis we have extended it by adding *object-sensitivity*. This work is under revision for the special issue of *QAPL'12* in the journal *Theoretical Computer Science*.

Chapter 2

Preliminaries

In this chapter, we provide some introductory notions and fix some notation used in resource analysis in general, and in the COSTA system in particular.

2.1 Introduction

Resource Usage Analysis aims at approximating the cost of programs, i.e. the amount of resources required to run a given program in terms of its input values. COSTA is an abstract interpretation based COSt and Termination Analyzer for object oriented programs. It was originally developed for the analysis of Java bytecode programs, and has been recently extended to analyze ABS programs [JHS+12, AAG+12a], an ABStract behavioral language for modeling systems. COSTA receives as input a Java bytecode program and a resource of interest and tries to infer an *upper-bound* (UB) of the resource consumption of the program. The techniques followed by COSTA for inferring the resource consumption of executing a program follow the classical two-fold approach to cost analysis due by Wegbreit [Weg75], (1) a program is first transformed into a set of cost relations [AAG+12b] which (2) can then be solved into a closed-form upper/lower bound [AAGP11]. This section details the techniques used in COSTA for inferring UBs from bytecode programs.

Figure 2.1 shows the architecture of the COSTA analyzer. The input and out-

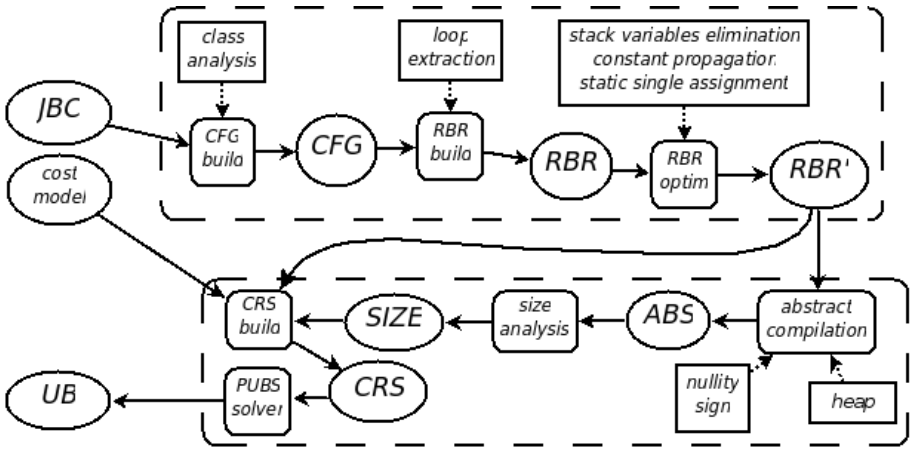


Figure 2.1: Architecture of COSTA

put to the system are depicted on the left: COSTA takes a Java bytecode program (JBC) and a *cost model* and yields as output an *UB* of its resource consumption, together with information about the termination of the program. Ellipses represent *what* the system produces, rounded boxes indicate the *steps* performed by the analyzer and auxiliary analyses are represented by dashed arrows. These phases are described in the current section to make the thesis self-contained and they can be summarized in three main steps:

(1) A *control flow graph* (CFG) is obtained from the bytecode program. The generation of the CFG is guided by a *class* analysis [SJ03] for determining the set of reachable classes at any program point and for solving the dynamic dispatching. The CFG helps to transform the unstructured control flow of bytecode into a *Rule-Based Representation* (RBR), which is a convenient intermediate representation form which later facilitates the formalization of the analysis. Several optimizations can be performed on the RBR to enable more efficient and accurate results, namely; loop extraction, which makes possible to deal with nested loops by extracting loop-like constructs from the CFG; single static assignment, that allows denotational program analyses; stack variables elimination, that enables the removal of a large number of stack variables that correspond to intermediate states; and, constant prop-

agation, where variables are unified to constants when it is feasible. These transformations yield an improved RBR, named RBR' in the figure.

(2) The next step consists in setting up a set of *cost recurrence equations*, or *cost relations* for short, from the rule-based representation of the program [AAG+12b]. Cost relations consist of two parts, (1) an expression that captures the cost of executing the program in terms of its input data, and (2) a set of constraints for defining the applicability conditions and the relations between variable values. Soundness and precision of the cost relations are guaranteed by applying a set of pre-analyses, namely, nullness and sign that allow COSTA to eliminate useless branches from the rules; and heap analyses that are required for the soundness of the size analysis. COSTA applies an *abstract compilation*, and a *size analysis* on the RBR', which, together with the cost model, allow the generation of the *Cost Relation System* (CRS) of the program.

(3) Finally, COSTA tries to solve the CRS and express the cost relations as *cost functions* [AAGP11], that are not in recursive form and hence can be directly evaluated. Since a precise solution for the cost relations seldom exists, COSTA infers an *upper bound* for the cost relations. A *lower-bound* can also be inferred similarly, but in this thesis we focus on the UB. Inferring an UB is a global process which starts by solving the cost relations which do not depend on any other and continue by replacing the computed cost functions on the equations that call such relations until all cost relations are solved.

The rest of the chapter describes all these components in detail.

Example 2.1.1. *Figure 2.2 shows a Java program which consists of four classes:* List*, a classical implementation of a linked list;* IncClass *and* IncClass2 *that implement a method to move forward to the next position of the list; and* C *which implements some methods that operate on lists. Method* C.getInc *can return two different object instances, an object of type* IncClass *or an object of type* IncClass2*, depending on the guard of an if-then-else statement. The difference between both instances is in the implementation of* nextElem*: objects of type* IncClass *increment the list one by one, while* IncClass2.nextElem *moves forward two elements at*

```
class C {
  static void modify (List l) {
    IncClass o = getInc(l);
    decList(o, o.nextElem(l));
  }
  static IncClass getInc (List l) {
    if (l.data == 1)
      return new IncClass();
    else
      return new IncClass2();
  }
  static void decList (IncClass o, List l) {
    while (l ≠ null) {
      l.data = l.data - 1;
      l = o.nextElem(l);
    }
  }
}
```

```
class IncClass {
  List nextElem (List l) {
    return l.next;
  }
}
class IncClass2 extends IncClass {
  List nextElem (List l) {
    return l.next.next;
  }
}
class List {
  List next;
  int data;
}
```

Figure 2.2: Preliminaries Running Example

a time. Method C.decList *traverses the list* l *by using* nextElem *of object* o*. Thus, depending on the instance returned by* getInc*,* decList *will decrement by one all elements of the list or only half of them.*

Figure 2.3 depicts the bytecode for each method of the Java source. Let us focus on the instructions of method C.decList*. The indexes* 0*,* 1 *in the bytecode instructions correspond respectively to parameters* o*,* l*. For clarity,variable* this *is ignored because it is not relevant for the cost analysis. Method* C.decList *contains the typical structure for executing a loop in bytecode. The instruction* 0: load_1 *pushes the reference to* l *on the stack. The next instruction* ifnull 13 *evaluates the condition of the loop by comparing the top of the stack with* null *and, if it is* null*, the program counter jumps to the instruction* 13*, exiting from the loop. Otherwise, the execution continues to instruction* 1*, and executes the loop body, whose last instruction,* 12: goto 0*, jumps again to the loop guard. At the end of the conditional instructions the top of the stack is automatically popped.*

Fields are accessed using two instructions, the load *instruction which pushes on the stack the reference variable and* getfield/putfield f *that are used to read/write*

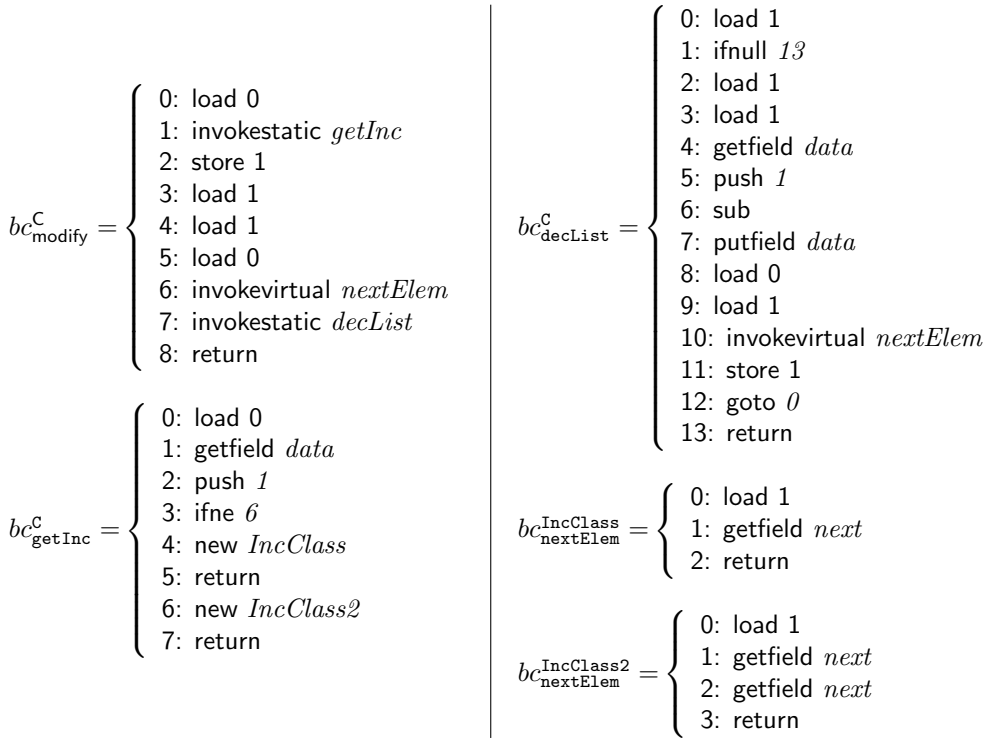


Figure 2.3: Java bytecode for the Runnig Example

the field f of the pushed variable, e.g. 4: load_1 *pushes* l *on the stack and* 5: getfield data *accesses the value of the field* data *of reference* l. *The value obtained by reading a field is pushed on the stack after popping the reference used to access the field. After writing a field, the reference is popped from the stack.*

Non-static methods are executed by means of the bytecode instruction invokevirtual. *Method parameters are previously pushed onto the stack. Dynamic dispatching is handled similarly by evaluating the runtime class of the instance of the first parameter pushed on the stack, which corresponds to the* this *reference. For instance, the instructions* 8: load_0 *and* 9: load_1 *push the parameters* o *and* l *on the stack respectively. The method which will be finally executed depends on the class of the object stored in* o *and it is evaluated by the Java Virtual Machine at runtime.*

2.2 From Bytecode to a Rule-Based Representation

During this phase, Java bytecode is transformed into a *rule-based representation*. The purpose of this transformation is twofold: (1) to represent the unstructured control flow of bytecode into a procedural form, and (2) to have an uniform treatment of stack and local variables.

2.2.1 Generation of the Control Flow Graph guided by Class Analysis

The control flow of Java bytecode is unstructured, and it allows conditional and unconditional jumps. The notion of *Control Flow Graph* facilitates the reasoning on a Java bytecode program. CFGs are created by adopting standard techniques from compiler theory: instructions sequences are splitted into its maximal subsequences of *non-branching* instructions, conforming the *basic blocks* of the initial graph. Blocks are connected by *guarded edges* that describe all possible transitions. Guarded edges are introduced by considering the last bytecode instruction of each block which represents the condition for the control going from one block to another one. The branching instructions considered in this thesis include conditional jumps and dynamic dispatching. Exceptions can be handled similarly by means of guarded edges, but they will be ignored for clarity of the presentation. For further details about the CFG generation we refer to [AAG+12b].

As customary in the analysis of object oriented languages, a *class analysis* [SJ03] is required to precisely approximate *virtual invocations*. Computing a precise approximation of the reachable methods at each program point is not trivial. COSTA constructs the CFG of the entry method and the class analysis infers all possible runtime classes at each program point. This information is used to resolve virtual invocations. The CFGs of the methods that can be called at runtime are constructed iteratively until no new CFG is created.

A subsequent *loop extraction* transformation is applied to the initial CFG to separate the sub-graphs that correspond to loops. This step has been well studied

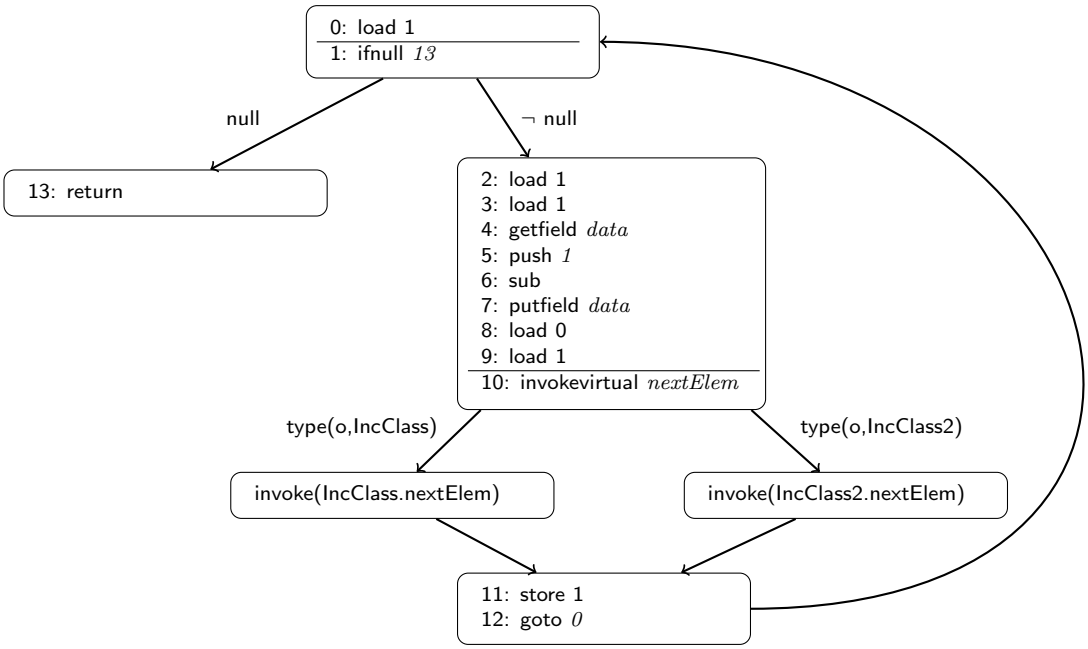


Figure 2.4: CFG of method decList

in the context of program decompilation [All70], termination analysis [AAC+08] and cost analysis [AAG+12b]. Loop isolation is crucial when the program contains nested loops, since it allows reasoning on one loop at a time.

Example 2.2.1. *Figure 2.4 depicts the CFG of method* decList*. Observe that branching instructions, like* ifnull*, generate two edges and their corresponding mutually exclusive conditions. Dynamic dispatching is handled similarly by a branching instruction (see* invokevirtual*) but using the information gathered by the class analysis, which indicates the existence of two possible implementations for* nextElem *(one in class* IncClass *and another in* IncClass2*). If-then-else statements will result in a similar structure that is guarded by the if condition. Note that unconditional branching instructions, like* return *or* goto *are represented by edges without any condition.*

2.2.2 Rule-Based Representation

A *rule-based program* P consists of a set of *procedures* and a set of classes. The set of class names C defined in P is denoted by $Class(P)$. A procedure p with k input arguments $\bar{x} = \langle x_1, \ldots, x_k \rangle$ and m output arguments $\bar{y} = \langle y_1, \ldots, y_m \rangle$ is defined by one or more *guarded rules* which adhere to this grammar:

$$
\begin{array}{rcl}
\mathit{rule} & ::= & p(\bar{x}, \bar{y}) \ \leftarrow \ g, \mathit{body}. \\
g & ::= & \mathit{true} \mid \mathit{exp}_1 \ \mathit{op} \ \mathit{exp}_2 \mid \mathsf{type}(x, C) \\
\mathit{body} & ::= & \epsilon \mid b, \mathit{body} \\
b & ::= & x := \mathit{exp} \mid x := \mathsf{new} \ C \mid x := y.f \mid x.f := y \mid x := \mathsf{newarray}(D, y) \mid \\
 & & x[y] := z \mid z := x[y] \mid x := \mathsf{arraylength}(y) \mid q(\bar{x}, \bar{y}) \\
\mathit{exp} & ::= & x \mid \mathsf{null} \mid n \mid x \ \mathit{aop} \ y \\
\mathit{aop} & ::= & + \mid - \mid / \mid * \\
\mathit{op} & ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{array}
$$

where $p(\bar{x}, \bar{y})$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; *body* the body of the rule; n an integer; x, y and z variables; f a field name; and $q(\bar{x}, \bar{y})$ a procedure call by value.

The RBR supports class definition and includes instructions for object and array creation and manipulation. A class contains a finite set of typed field names, where a type can be (1) an integer; (2) a class $C \in Class(P)$; or (3) an array whose elements are of type integer or $Class(P)$. For the sake of simplicity, the same field name, if used in different classes, has the same type. This can be done by automatically encoding the type into the field name. The set of all field names defined in P is denoted by $\mathit{fields}(P)$. The instruction $\mathsf{new}\ C$ creates an object of type C and returns a reference to it, $\mathsf{newarray}(D, y)$ creates an array of y elements of type $D \in \{\mathit{int}\} \cup Class(P)$ and $\mathsf{arraylength}(y)$ returns the length of the array y. For simplicity, we support only unidimensional arrays.

Classes, in the RBR, are in fact closer to records in C than to classes in Java, as they encapsulate only fields and not methods, and do not use inheritance or interfaces as in Java. We do not give an explicit syntax for defining classes; when needed, we simply use Java's syntax.

The translation from Java bytecode to the rule-based form is performed in two

b_j	dec(b_j)
load i	$s_{t+1} := l_i$
store i	$l_i := s_t$
push n	$s_{t+1} := n$
pop	nop(pop)
dup	$s_{t+1} := s_t$
add	$s_{t-1} := s_{t-1} + s_t$
sub	$s_{t-1} := s_{t-1} - s_t$
lt	$s_{t-1} < s_t$
gt	$s_{t-1} > s_t$
eq	$s_{t-1} = s_t$
null	$s_t = \mathsf{null}$
¬ lt	$s_{t-1} \geq s_t$
¬ gt	$s_{t-1} \leq s_t$

b_j	dec(b_j)
¬ eq	$s_{t-1} \neq s_t$
¬ null	$s_t \neq \mathsf{null}$
type(n, c)	type(s_{t-n}, c)
new c	$s_{t+1} := \mathsf{new}\ c$
getfield f	$s_t := s_t.f$
putfield f	$s_{t-1}.f := s_t$
newarray$c$	$s_t := \mathsf{newarray}(c, s_t)$
aload	$s_{t-1} := s_{t-1}[s_t]$
astore	$s_{t-2}[s_{t-1}] := s_t$
arraylength	$s_t := \mathsf{arraylength}(s_t)$
invoke m	$m(s_{t-n}, \ldots, s_t, s_{t-n})$
return	$out := s_t$
nop(b)	nop(b)

Figure 2.5: Compiling bytecode instructions (as they appear in the CFG) to rule-based instructions (t stands for the height of the stack before the instruction).

steps that are described in detail in [AAG+12b]. First, the CFG is built for each method (as described in Section 2.2.1) and each bytecode instruction is compiled into the RBR representation according to the mapping depicted in Figure 2.5. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. In this thesis, the language does not include features of Java, such as exceptions, static fields, access control and primitive types besides integers, arrays and references. However, COSTA deals with full *sequential* Java bytecode.

2.2.3 RBR Semantics

First, we assume that programs have been verified for well-typeness. By well-typeness we mean that any program variable at a given program point can hold (in any possible execution) either a *reference* or an *integer*, but not both. We refer to such types as *static types*. Given a variable x, we let `stype`(x) denote its static type, which can be, respectively `ref` or `int`. Due to well-typeness, for the case of $x[y] := z$ and $z := x[y]$, the variable x can be (in any execution) either a

reference to an array of integers or to an array of references (because z has always the same static type). For such case, we assume that $\texttt{stype}(x)$ returns `aref` or `aint` respectively. Note that, for simplicity, we assume that variable x implicitly contains information on the program point at which it appears so that its `stype` can uniquely identify this variable.

The execution of rule-based programs mimics standard bytecode [LY96]. The rules in Figure 2.6 define an *operational semantics* for the language (see [AAG+12b] for more details). An *activation record* (ar) has the form $\langle p, bc, tv\rangle$, where p is a procedure name, bc is a sequence of instructions, and tv is a variable mapping. Given a variable x, $tv(x)$ refers to the value of x, and $tv[x{\mapsto}v]$ updates tv by making $tv(x) = v$ while tv remains the same for all other variables. A *heap* h is a partial map from an infinite set of *memory locations* (or reference) to *objects*. We use $h(r)$ to denote the object referred to by r in h. We use $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$ while h stays the same for all locations different from r. For any location r and heap h, $r \in dom(h)$ iff there is an object associated to r in h. Given an object o, $o.f$ refers to the value of the field f in o, and $o[f{\mapsto}v]$ sets the value of $o.f$ to v. We use $h[o.f{\mapsto}v]$ as a shortcut for $h(r)[f{\mapsto}v]$, with $o = h(r)$.

In rule (1), $eval(exp, tv)$ returns the evaluation of the arithmetic or boolean expression exp for the values of the corresponding variables from tv in the standard way; for reference variables, it returns the reference. We assume that well-typing forbids pointer arithmetics. Rules (2), (3) and (4) deal with objects as expected. Procedure $newobject(C)$ creates a new object of class C by initializing its fields to either 0 or null, depending on their types. Rules (5), (6), (7) and (8) account for arrays. For simplicity, an array of length v is modeled as an object o with a special (read-only) field *length* initialized to v, and fields $1, \ldots, v$ corresponding to the array elements. The call $newarray(D, v)$ creates an array of v elements initialized to 0 or null. Note that Rule (7) prevents "simulating" multi-dimensional arrays. Rule (9) (resp., (10)) corresponds to calling (resp., returning from) a procedure. The notation $p[\bar{y}', \bar{y}]$ records the association between the formal and

(1) $$\frac{b \equiv x := exp, \quad v := eval(exp, tv)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[x \mapsto v]\rangle{\cdot}ar; h}$$

(2) $$\frac{b \equiv x := \mathsf{new}\ C, \quad o := newobject(C), \quad r \text{ is a new location not in } dom(h)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[x \mapsto r]\rangle{\cdot}ar; h[r \mapsto o]}$$

(3) $$\frac{b \equiv x := y.f, \quad tv(y) \neq \mathsf{null}, \quad o := h(tv(y))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[x \mapsto o.f]\rangle{\cdot}ar; h}$$

(4) $$\frac{b \equiv x.f := y, \quad tv(x) \neq \mathsf{null}, \quad o := h(tv(x))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv\rangle{\cdot}ar; h[o.f \mapsto tv(y)]}$$

(5) $$\frac{\begin{array}{c} b \equiv x := \mathsf{newarray}(D, y), \quad v =: tv(y), \quad v \geq 0, \\ o = newarray(D, v), \quad r \text{ is a new location not in } dom(h) \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[x \mapsto r]\rangle{\cdot}ar; h[r \mapsto o]}$$

(6) $$\frac{b \equiv x := \mathsf{arraylength}(y), \quad tv(y) \neq \mathsf{null}, \quad o = h(tv(y))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[x \mapsto o.length]\rangle{\cdot}ar; h}$$

(7) $$\frac{\begin{array}{c} b \equiv x[y] := z, \quad tv(x) \neq \mathsf{null}, \quad o := h(tv(x)), \quad v := tv(y), \\ 1 \leq v \leq o.length, \quad tv(z) \text{ is not an array} \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv\rangle{\cdot}ar; h[o.v \mapsto tv(z)]}$$

(8) $$\frac{\begin{array}{c} b \equiv x := y[z], \quad tv(y) \neq \mathsf{null}, \quad o := h(tv(y)), \quad v := tv(z), \\ 1 \leq v \leq o.length \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[x \mapsto o.v]\rangle{\cdot}ar; h}$$

(9) $$\frac{\begin{array}{c} b \equiv q(\bar{x}, \bar{y}), \quad \text{there is a rule } q(\bar{x}', \bar{y}') {:=} g, b_1, \cdots, b_k \in P, \\ tv' := newenv(q), \ tv'[x'_i \mapsto tv(x_i)], \ eval(g, tv') = true \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle q, b_1 \cdots b_k, tv'\rangle{\cdot}\langle p[\bar{y}', \bar{y}], bc, tv\rangle{\cdot}ar; h}$$

(10) $$\frac{}{\langle q, \epsilon, tv'\rangle{\cdot}\langle p[\bar{y}', \bar{y}], bc, tv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, tv[\bar{y} \mapsto tv'(\bar{y}')]\rangle{\cdot}ar; h}$$

Figure 2.6: Operational semantics of bytecode programs in rule-based form

actual return variables. *newenv* creates a new mapping of local variables for the method, where each variable is initialized to either 0 or null.

An execution for a program P starts from an *initial configuration* of the form $\langle start, p(\bar{x}, \bar{y}), tv\rangle; h$, and ends in a *final configuration* $\langle start, \epsilon, tv'\rangle; h'$, where:

1. *start* is an auxiliary name to indicate an initial activation record;
2. $p(\bar{x}, \bar{y})$ is a call to the procedure from which the execution starts;

$$
\begin{array}{rcl}
modify(\langle l\rangle,\langle\rangle) & \leftarrow & s_1 := l, getInc(\langle l\rangle,\langle s_1'\rangle),\\
& & o := s_1', s_1'' := o, s_2 := o, s_3 := l,\\
& & call\_nextElem(\langle s_2, s_3\rangle,\langle s_2'\rangle),\\
& & decList(\langle s_1'', s_2'\rangle,\langle\rangle).\\
getInc(\langle l\rangle,\langle out\rangle) & \leftarrow & s_1 := l, s_1' := s_1.data, call\_if(\langle s_1'\rangle,\langle out\rangle).\\
call\_if(\langle v\rangle,\langle out\rangle) & \leftarrow & v = 0,\\
& & s_1 := new\ \mathsf{IncClass}, out := s_1.\\
call\_if(\langle v\rangle,\langle out\rangle) & \leftarrow & v \neq 0,\\
& & s_1 := new\ \mathsf{IncClass2}, out := s_1.\\
call\_nextElem(\langle o, l\rangle,\langle l'\rangle) & \leftarrow & \mathsf{type(o,IncClass)},\\
& & IncClass.nextElem(\langle l\rangle,\langle l'\rangle).\\
call\_nextElem(\langle o, l\rangle,\langle l'\rangle) & \leftarrow & \mathsf{type(o,IncClass2)},\\
& & IncClass2.nextElem(\langle l\rangle,\langle l'\rangle).\\
decList(\langle o, l\rangle,\langle\rangle\rangle & \leftarrow & while(\langle o, l\rangle,\langle\rangle).\\
while(\langle o, l\rangle,\langle\rangle) & \leftarrow & while^c(\langle o, l\rangle,\langle l'\rangle).\\
while^c(\langle o, l\rangle,\langle l\rangle) & \leftarrow & l = \mathsf{null}.\\
while^c(\langle o, l\rangle,\langle l'\rangle) & \leftarrow & l \neq \mathsf{null},\\
& & s_1 := l, s_2 := l, s_2' := s_2.data,\\
& & s_3 := 1, s_2'' := s_2' - s_3, s_1.data := s_2'',\\
& & s_1' := o, s_2''' := l, call\_nextElem(\langle s_1', s_2'''\rangle,\langle s_1''\rangle),\\
& & l' := s_1'', while(\langle o, l\rangle,\langle l'\rangle).\\
IncClass.nextElem(\langle l\rangle,\langle out\rangle) & \leftarrow & s_1 := l, s_1' := s_1.next, out := s_1'.\\
IncClass2.nextElem(\langle l\rangle,\langle out\rangle) & \leftarrow & s_1 := l, s_1' := s_1.next, s_1'' := s_1'.next, out := s_1''.
\end{array}
$$

Figure 2.7: RBR of the example

3. h is an initial heap; and
4. tv is a variable mapping such that $dom(tv) = \{\bar{x}\} \cup \{\bar{y}\}$, and all variables are initialized to an integer value, null or a reference to an object in h.

Executions can be regarded as *traces* of the form $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_f$ (abbreviated $S_0 \rightsquigarrow^* S_f$), where S_f is a final configuration. Non-terminating executions have infinite traces.

Example 2.2.2. *Figure 2.7 shows the RBR of the program shown in Figure 2.2. By looking at method* decList*, we can observe that each rule in the RBR corre-*

sponds to one block in the CFG. The conditional statement if-then-else of method getInc *is handled by the rule* call_if*. In turn, the guards of the rule* call_if*, decide which block will be executed, the "if" block, when* $v = 0$*, or the "else" one, when* $v \neq 0$ *(note that the guards of the rules are mutually exclusive). Dynamic dispatching is handled similarly: the rule* call_nextElem *includes an instruction* $type(o, Class)$*, which checks if* o *is an instance of type* IncClass *or* IncClass2*, and depending on the type of* o*, calls the corresponding method,* IncClass.nextElem *or* IncClass2.nextElem*, respectively. The rule* while *contains a continuation rule,* while^c*, to evaluate the loop condition and either perform the recursive call (when the loop condition holds) or, otherwise, exit from the loop.*

2.3 From the RBR to a Cost Relation System

Once the RBR is generated, the next step consists in setting up *cost recurrence equations* (a.k.a. *cost relation system*). The global analysis underlying this step is the inference of size relations which determine how the sizes of data changes along the execution of the program. Before applying the size analysis, some previous (pre-)analyses are applied in order to optimize the RBR and to guarantee the soundness of the size analysis.

2.3.1 Context Sensitive (Pre-)Analyses

Nullness Analysis

Nullness analysis aims at keeping track of the reference variables which point to *non-null* objects at concrete program points. The nullness analysis allows us to guarantee that no *NullPointerException* can be thrown when accessing fields or invoking methods using reference variables. When the analysis can ensure that the reference variable is definitely non-null, those branches created by the compiler for handling the *NullPointerException* can be removed. This optimization significantly reduces the number of branches in the program.

Sign Analysis

Sign analysis keeps track of the sign of the program variables at concrete program points. The information about the sign of the data allows us to remove redundant rules that correspond to useless branches created for handling arithmetic exceptions. This results in an improvement in the precision of the resource analysis and the efficiency as the amount or rules analyzed is smaller.

2.3.2 Heap Analysis

Programming languages with dynamic memory allocation, such as Java, allow creating and manipulating heap-allocated data structures. Handling data structures allocated in the program memory (the heap) is a challenging issue in the context of termination analysis, resource usage analysis, garbage collection, etc. The information about the structures stored in the heap is needed for guaranteeing the soundness of the size analysis detailed in Section 2.3.4. COSTA implements three analyses for obtaining information related to local variables, fields and arguments located in the heap:

- **Sharing** [SS05] aims at detecting the pairs of reference variables that may point to common heap locations (see Section 5.4).
- **Cyclicity** [RS06] aims at detecting those variables that points to data structures that might be cyclic (see Section 5.4).
- **Constancy** [GS08] aims at detecting those heap structures that are not modified during the execution of the parts of the program.

2.3.3 Cost Models

Resource analysis is a generic technique that can be used for inferring different types of resources. Traditional resources include the number of executed instructions, the memory consumption, and the number of times a method is invoked. Each resource is defined by a corresponding cost model.

We consider platform independent cost models (e.g., worst-case execution time or energy consumption are excluded). The following is a cost model for approximating the *number of executed instructions*:

$$\mathcal{M}_i(b)=\begin{cases} 0 & b \equiv q(\_,\_) \wedge \text{q is not a method} \\ \mathcal{M}_i(g_1) + \mathcal{M}_i(g_2) & b \equiv g_1 \wedge g_2 \\ 1 & \textit{otherwise} \end{cases}$$

$\mathcal{M}_i$ assigns cost 1 to all instructions except calls to blocks, as they do not appear in the original program. The total memory allocated by the program can be obtained similarly by the cost model $\mathcal{M}_m(b) = size(C)$, if $b \equiv \mathsf{new}\ C$ where *size* returns the number of bytes required to create an object of type C; and $\mathcal{M}_m(b) = 0$, otherwise. A cost model that simply counts the *total number of objects created* along the execution can be defined as $\mathcal{M}_o(b) = 1$ if $b \equiv \mathsf{new}\ C$ and $\mathcal{M}_o(b) = 0$ otherwise We can also count the number of calls to an specific method m , e.g., by adding 1 only when the instruction $m(\_,\_)$ is find and m is not a block.

2.3.4 Size Analysis

The notion of *Size Measure*

From the RBR, the objective of the size analysis is to determine how the *size* of the data is modified along the program execution. To this end the notion of *size measure* is crucial [AAG+12b]. The size of a piece of data at a given program point is an abstraction of the information it contains. In the scenario of cost analysis, data structures are usually abstracted to their *size*. Depending on the type of the variable we are interested in, we can find different size measures:

- **Integer** variables are mapped to their value. The size of integer variables is typically used in loops with a counter and it is required for approximating the maximum number of iterations such loops can execute.
- **Arrays** are mapped to their length, allowing us to bound the number of loop iterations used to traverse an array by using as bound int length.

- **Data Structures** are abstracted by using the *path-length* [SMP10] of the structure, that is, an object is mapped to the length of the maximum path reachable from it by dereferencing. Provided x, a non-cyclic data structure, the size of x is greater than the size of any reference field of x. Thus, to guarantee the soundness of this abstraction, a cyclicity analysis is first performed. Path-length abstraction is used to bound the number of iterations of loops that traverse data structures.

Abstract Compilation

The aim of our abstract compilation is to transform the RBR program into an abstraction of the program with respect to the size measures. For example, when analyzing a loop with an integer counter i that goes from 0 to a threshold, size analysis with respect to *integer value* should see that the size of i in the n-th iteration of the loop is greater by 1 than its size in the $n-1$-th iteration. In this step of the cost analysis, the RBR instructions are *abstracted* by *linear constraints* on the size of its variables. For instance, from x=x+1 we obtain the RBR instruction $s_0 := s_0 + 1$. From this instruction, we are interested in representing that *the value of s_0 after the execution of the instruction is equal to the value of s_0 before the instruction plus 1.* This information is obtained via a *Single Static Assignment* transformation, which produces the instruction $s'_0 := s_0 + 1$, that, represent the value x before/after the instruction, that is, x':=x+1. Field instructions, like $s_0 := s_0.f$, are abstracted by $s'_0 \leq s_0 - 1$, which means that the output size is less than the input size, due to the field access. This step results in an *abstract compilation* which approximates the cost and termination behaviour of the original program.

Example 2.3.1. *Observing the rules **IncClass.nextElem** in the RBR of Figure 2.7 we can determine how the size of the the list* l *changes during the execution of **nextElem**. As it is described in Section 2.3.4, the size of list* l *is abstracted to its path-length, that is "l". The first instruction, $s_1 := l$, just assigns the abstract value of "l" to s_1. Besides, instruction $s'_1 := s_1.next$ will generate the constraint $s'_1 \leq s_1 - 1$, which means that the abstract value stored in s'_1 is less than the size s_1, i.e., less than the size of the list "l". Rule guards, like $l = null$ or $v = 0$, will*

also be abstracted to linear constraints. In the latter case, this abstraction can be done directly because "v" is an integer variable, so its size is represented by its content. But, for the reference variable "l" the constraint $l = null$ is abstracted by the constraint $l = 0$.

Input/Output Size Relations

A RBR rule can contain calls to other rules and the information propagated by the calls by means of the output variables may be needed to obtain the cost of executing a program. A call to a rule in the RBR may have output variables, but the CRS cannot have output variables because a CRS is a set of *mathematical relations*. The objective of this step is to obtain an abstract program where the output variables do not appear. The basic idea relies on computing input/output size relations relating the input and the output variables and use these relations to propagate the effect of calling a rule. COSTA infers input/output size relations of the form $p(\overline{x}, \overline{y}) \rightarrow \varphi$, where φ is a set of linear constraints describing the relation between the size of the input variables ($\overline{x}$) and the output variables ($\overline{y}$).

In order to combine the information obtained for each method, linear constraints are propagated via bottom-up computation. Sound input-output size relations can be obtained by taking the abstract rules generated by abstract compilation, and combining them via a fixpoint computation, using abstract interpretation techniques in order to avoid infinite computations.

Example 2.3.2. *According to the abstract compilation described in the Example 2.3.1, the abstract value returned by the rule* ***IncClass.nextElem*** *in variable* ***out*** *is the abstract value assigned to s'_1, that is, $s'_1 \leq s_1 - 1$. We can express this constraint in terms of the input/output variables (l/out) as follows:*

$$IncClass.nextElem(\langle l \rangle, \langle out \rangle) \leftarrow out \leq l - 1.$$

This constraint states that after executing the method ***IncClass.nextElem****, the size of the list returned by the method is equal to the size of the input list minus 1. Next, by using the obtained input/output size relations, the abstract compilation of the rule* ***modify*** *can determine how the size of s_3 is modified after calling* ***call_nextElem***$(\langle s_2, s_3 \rangle, \langle s'_2 \rangle)$*, resulting in the constraint $s'_2 \leq s_3 - 1$.*

2.3.5 Generation of Cost Relation System

Once the bytecode program has been transformed into its RBR and the size relations have been inferred, the next step is to set up the *Cost Relations System* (CRS) of the program. The CRS defines the cost of executing the program in terms of the size of its input parameters. In particular, each rule in the RBR program defines one equation in the CRS, which is composed of two parts, a *cost expression* and the set of constraints as we will see now.

Given a sequence of instructions B, we use the following functions: $calls(B)$ to obtain the set of elements that are calls to methods or to blocks of the form $q(\overline{w}, \overline{y})$ in B, and $instr(B)$ to refer to the remaining set of elements of B (the instructions). Given a cost model $\mathcal{M}$, a rule $p(\bar{x}, \bar{y}) \leftarrow B$ in the program and its size abstraction φ, we generate the cost equation:

$$p(\bar{x}) = \sum_{b \in instr(B)} \mathcal{M}(b) + \sum_{b(\overline{w},\overline{z}) \in calls(B)} b(\overline{w}), \quad \varphi$$

Given a program P, its cost relation system (CRS) is obtained by applying the above transformation to all rules.

The above cost relations have the following characteristics: (i) they do not have output arguments, as the cost is a function of the input; (ii) given a rule being analyzed, its cost equation is obtained by applying the cost model $\mathcal{M}$ to each of the basic instructions in the body (first summation in the equation); (iii) a call in the program is substituted by a call to its corresponding cost equation (second summation in the equation); (iv) the size abstractions φ are attached to the rules to define the applicability constraints for the equations and the size relations among the variables in the equations.

Example 2.3.3. *Figure 2.8 shows the CRS for counting the number of instructions executed by the program (by using the cost model $\mathcal{M}_i$ defined in Section 2.3.3) represented by the RBR of Figure 2.7. Observe that each rule in the RBR corresponds to one equation in the CRS. The resulting constraints to the right define the applicability conditions of the equations and the size relations between variables. Rule* ***modify*** *defines the cost expression "8 + getInc + call_nextElem + decList", where* 5 *out of the* 8 *corresponds to the cost of each*

$$
\begin{aligned}
modify(l) &= 8 + getInc(l) + call\_nextElem(l,l') + decList(l') && \{l' \leq l-1\}\\
getInc(l) &= 2 + call\_if() && \{\}\\
call\_if() &= 3 && \{\}\\
call\_if() &= 3 && \{\}\\
call\_nextElem(l,l') &= IncClass.nextElem(l,l') && \{l \geq 1, l' \geq 0, l' \leq l-1\}\\
call\_nextElem(l,l') &= IncClass2.nextElem(l,l') && \{l \geq 2, l' \geq 0, l' \leq l-2\}\\
decList(l) &= while(l) && \{\}\\
while(l) &= while^c(l) && \{\}\\
while^c(l) &= 1 && \{l=0\}\\
while^c(l) &= 11 + call\_nextElem(l,l') + while(l') && \{l \geq 1, l' \geq 0, l' \leq l-1\}\\
IncClass.nextElem(l,l') &= 3 && \{l \geq 1, l' \leq l-1\}\\
IncClass2.nextElem(l,l') &= 4 && \{l \geq 2, l' \leq l-2\}
\end{aligned}
$$

Figure 2.8: CRS of the example

basic instruction, and the 3 *corresponds to the cost of making the calls, plus the cost of executing the corresponding methods. Calls to internal blocks, like the calls to the* ***while*** *rule will not add any cost. Variables that are not involved in the equation guards are omitted because they are useless for solving the equations. Note also that the CRS entries do not have output variables, which have been replaced by constraints using the input/output size relations between the variables involved in the equation.*

2.4 From the CRS to a Closed-Form Upper Bound

The third phase in resource analysis consists in transforming the CRS obtained in Section 2.3 into *cost functions*, that is, cost expressions without recurrences. As an exact solution often does not exist, the objective of the resource analysis is to infer *Upper/Lower Bounds*, which are an over/under approximation of the worst/best-case cost. While this thesis will focus on the computation of the UB, the problem of LB is dual. A cost function is in closed-form when the function is a *basic cost expression* [AAGP11], i.e., a cost expression with all dependencies

(calls to other relations) solved. The final objective of the resource analysis is to obtain a *Closed-Form Upper Bound* of the cost of the program for a given resource. For brevity, UB will refer to the "closed-form upper bound".

2.4.1 Cost Relations compositionality

An important feature of the CRS is its *compositionality*. This feature allows us computing the UB of the CRS by concentrating on one relation at a time. Compositionality results in an effective mechanism when all recursions are direct, so that, the first step consists in transforming all relations into direct recursion (see [AAGP11] for details), resulting in a CRS with only one relation per *Strongly Connected Component* (SCC). This transformation allows us focusing on one relation at a time, starting by *stand-alone cost relations*, i.e. relations that do not depend on any other relation.

2.4.2 Stand-Alone Relations

Intuitively, to infer an UB for a recursive relation (a loop or a recursive method), the method proposed in [AAGP11] infers an UB in the number of iterations ($\#iter$) and an UB on the cost of a single execution of its body (C_{iter}), resulting in the UB $\#iter * C_{iter}$.

Bounding the number of iterations

The problem of bounding the number of recursive iterations has been widely studied in the context of termination analysis [AAC+08, PR04]. Termination analyzers usually prove that the number of iterations of the loop is bounded by proving that there exists a *ranking function* [Flo67] for that loop. A ranking function is a function f in terms of the loop arguments such that f decreases in any two consecutive calls and it is bounded from below. This function guarantees the absence of infinite traces and can be safely used to bound the number of iterations ($\#iter$).

Example 2.4.1. *Consider the CRS of Figure 2.8. Only recursive cost relations will iterate, so let us focus on the cost relations of the* ***while*** *rule. Observe*

that, depending on the ***nextElem*** *implementation, the length of the list can be decremented by 1 or 2. Thus, the size of the list is decremented by at least one at each iteration, and* $\mathsf{nat}(l)$ *can be used as a ranking function of the loop. Function* $\mathsf{nat}(x) = max(\{0, x\})$ *is used by the UB solver to avoid negative evaluations. The ranking function determines the UB for the number of iterations of the loop, that is* $\#iter = \mathsf{nat}(l)$.

Bounding the cost of a single iteration of the loop

To compute the worst-case cost of a ginven call iteration of the loop, it is necessary to determine how the values of the variables are modified within the loop. For that purpose, the method of [AAGP11] uses (1) the *loop invariant* (ψ), which is a set of linear constraints that relates the values of the variables inside the loop with the variable values when entering into the loop; and (2) the *size relations* φ between the program variables at different program points (see Section 2.3.4). Intuitively, by using the *loop invariant* and the *size relation*, it is possible to maximize the cost expression and obtain the worst possible cost for one iteration of the loop.

Example 2.4.2. *Let us infer the worst case of executing the while loop for the CRS of Figure 2.8. The following invariant is obtained for this cost relation* $\psi = \{l_0 \geq l + 1\}$*, which simply states that the length of the list in the recursive call is strictly smaller that the length of the initial list (*l_o*). In the while loop, the invariant is not relevant for obtaining the worst cost of the cost relation because it is constant. Thus, we just have to add the cost of the body of the while loop, that is* 11*, plus the cost of the method called. The call to* ***call_nextElem*** *can add* 3 *or* 4 *depending on the* ***nextElem*** *selected. To obtain the UB we have to select the worst case, i.e.,* 4*. Adding both costs we obtain that the worst case for a single iteration of the body is* $C_{iter} = 15$.

UB for Stand-Alone Relations

When the number of iterations is bounded ($\#iter$) and the worst-case cost for each iteration (C_{iter}) is obtained, the cost of the loop can be bounded by $\#iter * C_{iter}$.

The cost of non-recursive relations is obtained by replacing the calls with the UBs obtained for them and adding the resulting expressions.

Example 2.4.3. *Consider the CRS of Figure 2.8. The cost of non-recursive relations (e.g.* ***getInc(l)****) is obtained by adding* 2 *(see the equation of* ***getInc****) and the cost of* ***call_if****, which is* 3*. This results in* $UB_{getInc(l)} = 5$*. Recursive relations, like* ***while****, are obtanied by multiplying ranking function calculated in the Example 2.4.1 (i.e.* $\mathsf{nat}(l)$*) and the cost per iteration obtained in Example 2.4.2 (i.e.* $C_{iter} = 15$*), resulting in the cost expression* $UB_{while(l)} = 15 * \mathsf{nat}(l)$*.*

2.4.3 Bottom-Up Computation

The UB of the entry method, is computed by a bottom-up computation, which firstly solves the stand-alone relations and then composes the computed UBs on the equations that call such relations. The composition of the cost in the different cost relations requires a maximization operation [AAGP11] in order to obtain the worst-case cost for such call and express it in terms of the input arguments in the calls. This maximization operation uses the loop invariants. For the recursive cost relations, and the size relations to relate the values of the variables before calling the loop (or method call) and the entry of its parent scope. Cost relations with constant cost (which do not depend on any input argument), can be directly replaced into the callers without requiring maximization, beacuse its cost is constant.

Example 2.4.4. *Let us now compute the UB for the relation* ***modify(l)****. We first have to compute the UB for all relations called by* ***modify****, given in Figure 2.8. .* UB_{getInc} *has been computed in the previous example and* $UB_{call\_nextElem}$ *can be obtained similarly as* $UB_{call\_nextElem} = 4$*. The UB for* ***decList*** *uses* UB_{while} *and results in* $UB_{decList(l)} = 15 * \mathsf{nat}(l)$*. In order to compose* $UB_{modify(l)}$*, we have to maximize the expressions that come from the calls with respect to the input arguments of* ***modify****. By using the size relations of* ***modify****,* $\varphi = \{l' \leq l - 1\}$*, we obtain that the expression that bounds the cost of* ***decList*** *in the context of* ***modify*** *is "*$15 * \mathsf{nat}(l-1)$*". Expressions from* ***getInc*** *and* ***call_nextElem*** *can be directly propagated to* ***modify*** *because they are constant. Finally,* $UB_{modify(l)}$ *is*

obtained by adding the cost of all calls and the cost of its cost expression, resulting in the ***Closed-Form Upper Bound*** *for the running example:*

$$UB_{modify(l)} = 8 + 5 + 4 + (15 * \mathsf{nat}(l - 1)).$$

Chapter 3

Conditional Termination of Loops over Heap-Allocated Data

This chapter presents our main results on proving conditional termination of loops over heap-allocated data. These results were publised in *Bytecode'12* and an extended and revised version is currently under revision for its special issue to be published in the *Science in Computer Programming* journal.

3.1 Introduction

It is well known that shared mutable data structures, such as those stored in the heap, are the bane of formal reasoning and static analysis (see e.g. [Min06, BNR08]). This problem is exacerbated in object-oriented programs, since most data reside in objects and arrays stored in the heap. Analyses that keep track (resp. do not keep track) of heap-allocated data are referred to as *heap-sensitive* (resp. *heap-insensitive*). In most cases, neither of the two extremes (fully heap-insensitive analysis or a fully heap-sensitive analysis) is acceptable: the former produces too imprecise results while the latter is often computationally intractable. There has been significant interest in developing techniques that result in a good balance between the accuracy of analysis and its associated computational cost. A number of heuristics exist that differ in how the value of

heap-allocated data is modeled. A well-known heuristics is *field-based* analysis, in which only one variable is used to model all instances of a field, regardless of the number of objects for the same class that may exist in the heap. This approach is efficient, but loses precision quickly.

The approach we propose in this thesis is based on the observation that, by analyzing *program fragments* (or *scopes*), rather than the application as a whole, it is often possible to keep track of the values of heap-allocated data in a similar way as for non heap-allocated variables. Such fragments can be built starting from methods, loops, or even blocks of contiguous sentences. Our final goal is to be able to instrument programs in which accesses to heap-allocated data are *replaced* with (or replicated by) equivalent accesses to, non-heap allocated, *ghost* variables whose values represent the values of the corresponding heap-allocated data. The instrumented program can then be input to any heap-insensitive static analysis, which can now obtain heap-sensitive information, since the ghost variables expose the heap-allocated values.

The kind of properties that can benefit from our approach are those that require a *local* or *compositional* reasoning, i.e., they require the inference of the property for certain fragments, rather than a global inference for the whole program execution. Termination, the target application of our analysis, is a property that requires such kind of local reasoning, where the scopes of interest are the loops. Basically, in order to prove termination, the analysis has to keep track of how the size of the data involved in loop guards changes when the loop goes through its iterations. This information is used for determining a *ranking function* for the loop as it is described in Section 2.4.

Obviously, not all heap-allocated data are *transformable*, i.e., their behaviour reproducible using ghost variables. In the most general characterization, the replacement is possible when two sufficient conditions hold within the scope: (a) the memory location where the heap-allocated data is stored does not change, i.e., the reference to such data remains *constant*, and (b) all accesses (if any) to such memory location are done through the same reference (and not through aliases). This characterization captures the situations in which heap-allocated data behave *locally* (i.e., like non heap-allocated variables) in the given scopes.

3.1.1 Organization of the Chapter

This chapter is organized as follows: Section 3.2 is devoted to present the reference constancy analysis for programs written in the RBR language described in Chapter 2. Section 3.3 introduces the heap-sensitive analysis in three main steps: (1) we define a simple locality condition which relies on the information inferred by a reference constancy analysis; (2) we discuss that such condition might only hold under some *aliasing* (or not *aliasing*) conditions among the heap accesses; and, (3) e finally present a transformation that replaces the heap accesses which meet the locality condition by local variables for the given locality partition.

In Section 3.4 we propose an approach to infer the aliasing preconditions based on two notions of termination: *local termination*, which guarantees that the loops defined in a given scope S are terminating; and, *global termination*, which guarantees by composition that the loops of S as well as those of scopes that are transitively called from S are terminating.

Section 3.5 summarizes our experimental results performed on some micro-benchmarks which are interesting because of their use of arrays and fields, and the fact that they can be only proven terminating under certain (non-trivial) aliasing preconditions. Finally, Section 3.6 relates our approach to previous work.

3.2 Reference Constancy Analysis

In this section, we develop a *reference constancy analysis* which allows us to obtain the *access paths* to the fields, arrays and array elements that are *constant* in the considered scopes. The idea behind this analysis is similar in spirit to that of the classical numeric constant propagation analysis [CC77]. However, in addition to numerical constants, the values computed by our analysis can include symbolic expressions that refer to locations in (the initial) heap. Such expressions encode as well the way that the corresponding memory locations are reached (e.g., the dereferenced fields).

Example 3.2.1. *Consider the examples in Figure 3.1 . S_1 and S_2 are used to*

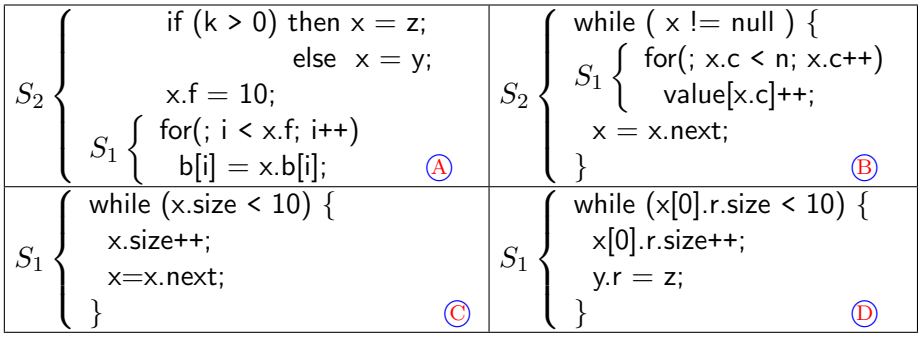


Figure 3.1: Small examples to illustrate the notion of constant access path

delimit scopes. In A*, the reference* x *remains constant within the scope of loop* S_1 *since its value does not change. However, if we consider the whole code fragment* S_2*,* x *is no longer constant, since* x *can take two different values before the loop. In* B*, all occurrences of* x *are constant within the scope* S_1 *of the inner loop. However,* x *takes different values in different iterations of the outer loop, and thus* x *is not constant in the whole scope* S_2*. In* C*,* x *is not constant because it is updated at each iteration of the loop. In* D*, it cannot be ensured that* $x[0].r$ *is constant, since if* $x[0]$ *and* y *are aliases, updating* $y.r$ *changes* $x[0].r$*.*

3.2.1 The Set of Access Paths

We start by defining the set of (symbolic) abstract values that our analysis assigns to each variable, at each program point. We refer to these values as *access paths*. They are defined in terms of the (symbolic) input parameters of the initial call. We denote these parameters by $\mathcal{L} = \{l_1, \ldots, l_n\}$, where l_i represents the value of the i-th parameter. An access path will be denoted by ℓ (possibly subscribed or primed), and it can be one of the following values:

1. $n \in \mathbb{Z}$, which represents the corresponding integer;

2. ℓ_{null}, which represents the value null;

3. $l_i \cdot A_1 \cdots A_n$ or $l_i[\ell'] \cdot A_1 \cdots A_n$, where each A_i is of the form f_i or $f_i[\ell'']$ and $f_i \in \mathit{fields}(P)$. Note that n can also be 0, in which case we do not access

any field. Moreover, ℓ' and ℓ'' are different from ℓ_{null}.

An access path ℓ might refer to an integer or a value or a reference to an object or to an array. Observe also that each l_i inherits the static type of its corresponding parameter. In addition to the access paths defined by the above rules, we use a special one, denoted by $\ell_{\mathtt{any}}$, that represents any value. Note that $\ell_{\mathtt{any}}$ cannot appear as part of any other access path.

Example 3.2.2. *Intuitively, our analysis will assign to each variable (at each program point) an access path which describes its possible values* whenever the execution reaches that point. *Suppose that a variable x, at some program point, is assigned the access path ℓ. Let us intuitively explain the meanings for some possible values of ℓ: (1) if $\ell = 5$, then the value of x is 5; (2) if $\ell = l_2$, then the value of x is the value of its second initial parameter; (3) if $\ell = l_1{\cdot}f{\cdot}g$, assuming that the first parameter points to an object o, then x has the value of $o{\cdot}f{\cdot}g$ when evaluated in the initial state; (4) if $\ell = l_2[l_4]$, assuming that the second initial parameter points to an array a and that the fourth parameter is an integer n, then the value of x is like that of $a[n]$ (again, when evaluated in the initial state); and (5) if $\ell = \ell_{\mathtt{any}}$, then the value of x can be any reference or integer value, depending on the type of x.*

The set of all access paths, w.r.t. a given set $\mathcal{L}$ of initial parameters, is denoted by $AP(\mathcal{L})$. Given an access path ℓ, we denote by $\ell[l_1/\ell_1, \ldots, l_n/\ell_n]$ the access path that results from simultaneously replacing each occurrence of l_i by ℓ_i. If the result includes $\ell_{\mathtt{any}}$, i.e., it is an non constant access path, then we assume that $\ell[l_1/\ell_1, \ldots, l_n/\ell_n]$ is actually $\ell_{\mathtt{any}}$. This replacement operation trivially extends to any entity that involves access paths. The set of access paths $AP(\mathcal{L})$ is partially ordered by $\sqsubseteq_a$ such that for any $\ell \in AP(\mathcal{L})$ we have $\ell \sqsubseteq_a \ell$ and $\ell \sqsubseteq_a \ell_{\mathtt{any}}$. We let $\ell_1 \sqcup_a \ell_2$ be ℓ_1 if $\ell_1 = \ell_2$; otherwise $\ell_{\mathtt{any}}$.

3.2.2 The Analysis

In order to assign access paths to variables at program point level, we need first to define such program points. For this, we assume that the program's rules are

uniquely numbered starting from 1. The k-th program rule $p(\bar{x}, \bar{y}) \leftarrow g, b_1^k, \ldots, b_t^k$ has $t+1$ program points. The first one, k:1, after the execution of the guard g and before the execution of b_1, then k:2 between the execution of b_1 and b_2, until k:t+1 after the execution of b_t. For an initial configuration $C_0 = \langle start, p(\bar{x}, \bar{y}), tv\rangle; h$, we assume that the call $p(\bar{x}, \bar{y})$ corresponds to program point 0:0. The set of all program points of a program P, including 0:0, is denoted by $pps(P)$.

The analysis receives as input a program P and a procedure name p which we refer to as the *entry*. We assume that p has n arguments, and their symbolic values, as above, are denoted by $\mathcal{L} = \{l_1, \ldots, l_n\}$. The analysis assigns to each program point k:$j \in pps(P)$ an abstract state, from which it is possible to obtain the access path of each variable, at any program point. Given a set of (typed) variables $\mathcal{V}$, defined at a given program point, an abstract state over $\mathcal{V}$ and $\mathcal{L}$ has the form $\langle \phi, \theta\rangle$, where $\phi : \mathcal{V} \mapsto AP(\mathcal{L})$ maps variables to access paths; and $\theta \subseteq \mathit{fields}(P) \cup \{\texttt{aint}, \texttt{aref}\}$ is a set of field and array-types which are guaranteed to be constant, i.e., are not modified in any execution that reaches the corresponding program point. As it described in Section 2.2.2, `aint` and `aref` are the values returned by the function $\texttt{stype}(x)$ when x points to an integer and when it points to a reference variable, respectively. Our main interest is in inferring ϕ, the set θ is auxiliary to soundly construct ϕ during the analysis.

We let $\mathcal{S}(\mathcal{V}, \mathcal{L})$ be the set of all abstract states, w.r.t. some $\mathcal{V}$ and $\mathcal{L}$. We say $\langle \phi_1, \theta_1\rangle \sqsubseteq_s \langle \phi_2, \theta_2\rangle$ if $\theta_2 \subseteq \theta_1$, and $\phi_1(x) \sqsubseteq_a \phi_2(x)$ for any $x \in \mathcal{V}$. We let $\langle \phi_1, \theta_1\rangle \sqcup_s \langle \phi_2, \theta_2\rangle = \langle \phi, \theta_1 \cap \theta_2\rangle$ where $\phi(x) = \phi_1(x) \sqcup_a \phi_2(x)$ for any $x \in \mathcal{V}$. We denote by $\mathcal{A}(\mathcal{V}, \mathcal{L})$ the complete lattice $\langle \mathcal{S}(\mathcal{V}, \mathcal{L}), \top_s, \bot_s, \sqcup_s, \sqsubseteq_s\rangle$, where (1) the top element $\top_s$ corresponds to $\langle \phi, \emptyset\rangle$ in which $\phi(x) = \ell_{\texttt{any}}$ for any $x \in \mathcal{V}$; and (2) the bottom element $\bot_s$ is a symbolic value that represents an empty abstract state. Next we lift $\mathcal{A}(\mathcal{V}, \mathcal{L})$ in order to represent a set of abstract states, one for each k:$j \in pps(P)$. We represent such states as sets of elements of the form k:$j \mapsto \langle \phi, \theta\rangle$ where $\langle \phi, \theta\rangle \in \mathcal{S}(\mathcal{V}_{k:j}, \mathcal{L})$. Here $\mathcal{V}_{k:j}$ is the set of (typed) variables that are available at program point k:j. Such set must include an abstract state for each k:$j \in pps(P)$. The set of all such states is denoted by $\bar{\mathcal{S}}_P$. We use $\bar{\mathcal{A}}_P$ to denote the complete lattice $\langle \bar{\mathcal{S}}_P, \top_p, \bot_p, \sqcup_p, \sqsubseteq_p\rangle$ where $\top_p$, $\bot_p$, $\sqcup_p$, and $\sqsubseteq_p$ are defined by lifting the corresponding ones of $\mathcal{A}(\mathcal{V}_{k:j}, \mathcal{L})$.

The analysis is based on a transfer function τ, depicted in Figure 3.2, that

Instruction b		*Transfer function* $\tau(b, \langle\phi, \theta\rangle)$
(1)	$x := y.f$	$\langle\phi[x \mapsto \ell, \theta\rangle$
(2)	$x.f := y$	$\langle\phi, \theta \backslash \{f\}]\rangle$
(3)	$x := n$	$\langle\phi[x \mapsto n], \theta\rangle$
(4)	$x :=$ null	$\langle\phi[x \mapsto \ell_{\mathsf{null}}], \theta\rangle$
(5)	$x := y$	$\langle\phi[x \mapsto \phi(y)], \theta\rangle$
(6)	$x := y\ aop\ z$	$\langle\phi[x \mapsto \ell], \theta\rangle$
(7)	$x :=$ newarray(D, y)	$\langle\phi[x \mapsto \ell_{\mathtt{any}}], \theta\rangle$
(8)	$x := y[z]$	$\langle\phi[x \mapsto \ell], \theta\rangle$
(9)	$x[y] := z$	$\langle\phi, \theta \backslash \{\mathtt{stype}(x)\}\rangle$
(10)	$x :=$ new C	$\langle\phi[x \mapsto \ell_{\mathtt{any}}], \theta\rangle$
(11)	$x :=$ arraylength(y)	$\langle\phi[x \mapsto \ell_{\mathtt{any}}], \theta\rangle$
(12)	*otherwise*	$\langle\phi, \theta\rangle$

where we have the following conditions in rules:

(1) If $f \in \theta \wedge \phi(y) \neq \ell_{\mathtt{any}}$ then $\ell = \phi(y) \cdot f$; otherwise $\ell = \ell_{\mathtt{any}}$.

(6) If $\phi(y)$ and $\phi(z)$ are numbers, then $\ell = \phi(y)\ aop\ \phi(z)$; otherwise $\ell = \ell_{\mathtt{any}}$.

(8) If $\mathtt{stype}(y) \in \theta \wedge \phi(y) \neq \ell_{\mathtt{any}} \wedge \phi(z) \neq \ell_{\mathtt{any}}$ then $\ell = \phi(y)[\phi(z)]$; otherwise $\ell = \ell_{\mathtt{any}}$.

Figure 3.2: Transfer function for reference constancy analysis

defines the effect of executing each (simple) instruction on a given abstract state $\langle\phi, \theta\rangle$. Let us explain the different cases of the transfer function:

(1) When a variable x is assigned the value of $y \cdot f$, the transfer function updates the access path of x accordingly: if y is not $\ell_{\mathtt{any}}$, and field f has not been updated so far, then the resulting access path is the concatenation of that of y with the symbol f; otherwise $\ell_{\mathtt{any}}$.

(2) When a field f is assigned a value, f is eliminated from θ, i.e., it is marked as a field that has been updated and it is not constant. Note that in any subsequent execution step, an access $y \cdot f$ (of case (1)) will result in $\ell_{\mathtt{any}}$.

(3) This case simply updates the access path of x to be number n.

(4) Similarly to the above case, it updates the access path of x to be ℓ_{null}.

(5) This case updates the access path of x with that of y.

(6) If the access path of y and z are numbers, then the access path of x is updated to be the result of applying the corresponding arithmetic operator; otherwise it is updated to $\ell_{\texttt{any}}$ to indicate that it can be any number.

(7) In this case, when creating a new array, the access path of x is updated to $\ell_{\texttt{any}}$ to indicate that it can be any reference value.

(8) If the access paths of y and z are not $\ell_{\texttt{any}}$, and it is guaranteed that the accessed array has not been modified (its static type is still in θ), then the access path of x is computed accordingly; otherwise it is $\ell_{\texttt{any}}$.

(9) It eliminates the static type of array x from θ. It is analogue to case (2).

(10) This case is similar to case (7). The access path of x is updated to $\ell_{\texttt{any}}$, to indicate that its value might be any reference value.

(11) Simply maps x to $\ell_{\texttt{any}}$ since the length of the array can be any number.

The remaining instructions do not alter constancy information. The analysis starts from an abstract state $I_0^\# \in \bar{\mathcal{A}}_P$ that assigns $\bot_p$ to each program point $k{:}j \in \mathit{pps}(P)$, except for 0:0 which is assigned the abstract state $\langle \phi, \mathit{fields}(P) \cup \{\texttt{aint}, \texttt{aref}\}\rangle$ where ϕ maps each x_i (resp. y_i) of the initial call to l_i (resp. ℓ_{null} or 0 depending on its type). Then, it iteratively computes $I_{i+1}^\# = I_i^\# \sqcup_p F_P^\#(I_i^\#)$ until it reaches a state in which $I_{i+1}^\# = I_i^\#$. The operator $F_P^\# : \bar{\mathcal{S}}_P \mapsto \bar{\mathcal{S}}_P$ is defined as $F_P^\#(X) = F_1^\#(X) \cup F_2^\#(X) \cup F_3^\#(X)$ where each $F_i^\#$ is as follows:

$$F_1^\#(X) = \left\{ k{:}j{+}1 \mapsto \langle \phi', \theta' \rangle \;\middle|\; \begin{array}{l} b_j^k \in P \text{ which is not a call} \\ k{:}j \mapsto \langle \phi, \theta \rangle \in X \\ \langle \phi', \theta' \rangle = \tau(b_j^k, \langle \phi, \theta \rangle) \end{array} \right\}$$

$$F_2^\#(X) = \left\{ k'{:}1 \mapsto \langle \phi', \theta' \rangle \;\middle|\; \begin{array}{l} b_j^k \equiv q(\bar{w}, \bar{z}) \in P,\ q \text{ is defined by rule } k' \\ k{:}j \mapsto \langle \phi, \theta \rangle \in X \\ \phi'' = \mathit{init}(k'), \phi' = \phi''[x_i \mapsto \phi(w_i)], \theta' = \theta \end{array} \right\}$$

$$F_3^{\#}(X) = \left\{ k'{:}j{+}1 \mapsto \langle \phi', \theta' \rangle \;\middle|\; \begin{array}{l} p(\bar{x}, \bar{y}) \leftarrow g, b_1^k, \ldots, b_t^k \in P,\ b_j^{k'} \equiv p(\bar{w}, \bar{z}) \in P \\ k{:}t{+}1 \mapsto \langle \phi, \theta \rangle \in X,\ k'{:}j \mapsto \langle \phi'', \theta'' \rangle \in X \\ \phi' = \phi[y_i \mapsto \phi''(y_i)],\ \theta' = \theta'' \end{array} \right\}$$

Let us explain each $F_i^{\#}$:

- $F_1^{\#}$ handles cases in which the instruction b_j^k is a simple instruction (not a call). It uses the current abstract state for program point $k{:}j$ and the transfer function τ in order propagate the information to program point $k{:}j{+}1$.

- $F_2^{\#}$ handles cases in which $k{:}j$ is a call $q(\bar{w}, \bar{z})$ to another procedure. In this case we propagate the abstract state of program point $k{:}j$ to the entry program point $k'{:}1$ for each rule k' that defines q. This is done by first creating an initial mapping ϕ'' that maps each variable of rule k' to either 0 or ℓ_{null}, depending on its type at that program point, and then modifying the access path of each x_i to be as that of the i-th actual parameter.

- In a similar way, $F_3^{\#}$ handles cases in which we propagate the return values to the calling context.

Example 3.2.3. *Our running example is shown in Figure 3.3. Class* **ListIter** *implements the* **Iterator** *interface. In OO programming, the* iterator pattern *(also* enumerator*) is a design pattern in which the elements of a* collection *are traversed systematically using a* cursor*. The cursor points to the current element to be visited and there is a method, called* next*, which returns the current element and advances the cursor to the next element, if any. In order to simplify the example, the method* next *in Figure 3.3 returns (the new value of) the cursor itself and not the element stored in the node. The important point, though, is that the* state *field is updated at each call to* next*. Class* **List** *implements a linked list in the standard way. Finally, class* **UseIterator** *contains the method of interest* m*. The interesting features of this method are: the combined use of fields and arrays, that it contains two nested loops which must be analyzed compositionally and that its termination can be only proven conditionally.*

```
class ListIter implements
          Iterator⟨List⟩ {
 List state;
 public ListIter(List l) {
    state = l;
 }
 public List next() {
    List obj = this.state
    this.state = obj.next;
    return obj;
 }
 public boolean hasNext() {
    return (this.state != null);
 }
 public void remove() {
    throw new Unsupported-
          OperationException();
 }
}
```

```
class List {
 int data;
 List next;
 List(int x, List y) {
      data = x; next = y;
 }
}
class UseIterator {
 public static void m(int[ ] a,int[ ] b, ListIter y) {
   while (y.hasNext()){
      List o = y.next();
      int i= o.data; int j = i;
      while(a[i] > 0) {
        a[i]--;
        b[j]++;
      }
   }
 }
}
```

Figure 3.3: Running Example. Method m contains nested loops with iterator and arrays

Figure 3.4 shows to the left the RBR of the inner while *loop of method* m *of the running example and to the right the abstract states, computed by the analysis, for some selected program points. Each abstract state corresponds to the result* after *analyzing the instructions in the corresponding line in the left side. We use* l_1, l_2, l_3, l_4 *to refer to, respectively, the initial values of* a, b, i *and* j. *Observe that at program point* ①, *i.e., before executing* $s_0 := s_0[s_1]$, *the accesses paths assigned to* s_0 *and* s_1 *are respectively* l_1 *and* l_3. *This means that the corresponding array access will always refer to the memory location* $l_1[l_3]$. *We will see that this piece of information is crucial to determine wether the corresponding heap access is transformable into a local variable or not. Similarly, we can conclude that the array accesses at* ② *and* ③ *will always refer to* $l_1[l_3]$ *and* $l_2[l_4]$ *respectively.*

In what follows, we let $I_P \in \bar{\mathcal{A}}_P$ be the result of the analysis, which is computed iteratively as described above. It is actually the least fixpoint of $\lambda X. I_0^\# \sqcup_p F_P^\#(X)$.

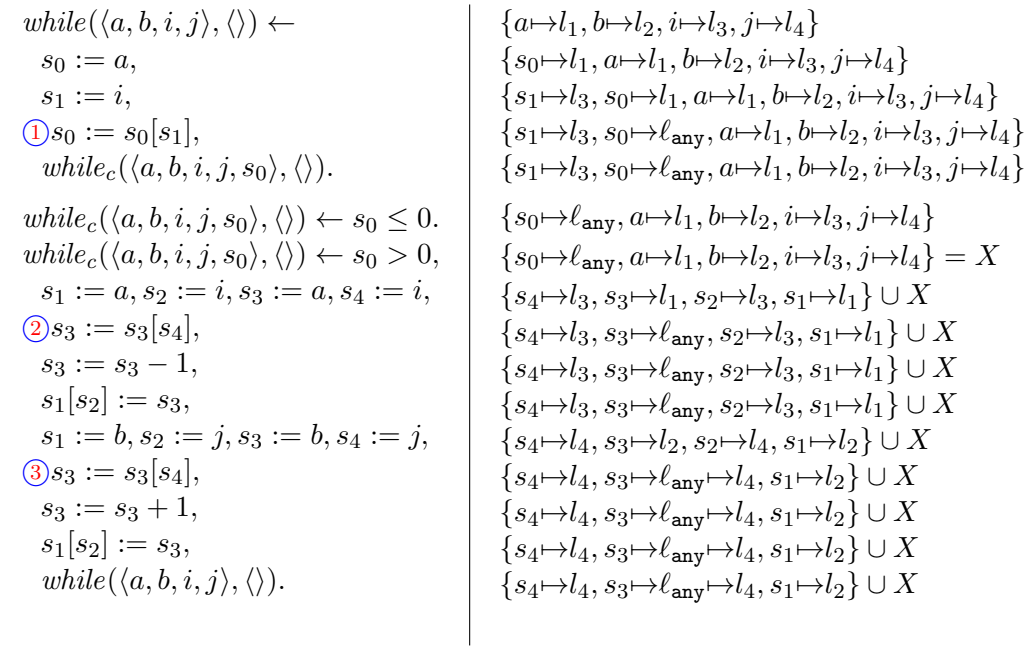


Figure 3.4: RBR of the Running Example (left). Program point constancy information (right)

We use $\langle \phi_{k:j}, \theta_{k:j} \rangle$ to refer to the abstract state assigned to program point $k{:}j$ in $I_P^\#$. In addition, for a given procedure p, we define:

$$\langle \phi_p, \theta_p \rangle = \sqcup_s \{ \langle \phi, \theta \rangle \mid p(\bar{x}, \bar{y}) \leftarrow g, b_1^k, \ldots, b_t^k \in P, k{:}t{+}1 \mapsto \langle \phi, \theta \rangle \}$$

We refer to this abstract state as the *summary* of procedure p, which describes the access paths of its parameter upon exit from p. We assume that $\mathrm{dom}(\phi_p)$ contains always variables with names $\bar{x}$ and $\bar{y}$ (just to avoid renamings).

3.2.3 Modular Analysis

References are often not globally constant, but they can be constant when we look at smaller fragments. Fortunately, the analysis can be applied modularly by partitioning the procedures (and therefore rules) of P into fragments which we refer to as *scopes*, provided that there are no mutual calls between scopes. The

smaller the scopes are, the more precise the analysis result will be. Therefore, the strongly connected components (SCCs) of the program are the smallest scopes we can consider. We assume that each scope has a single entry, this is not a restriction since otherwise the analysis can be repeated for each entry.

Given a program P, we let $S_1, \ldots, S_n$ be the partitioning of its procedures into scopes, where the entry procedure of each S_i is p_i. Since scopes are not mutually recursive, we can assume that if there is a call from a procedure in S_i to a procedure in S_j, then $i \geq j$. We refer to an inter-scope (resp. intra-scope) call as an *external* (resp. *internal*) call. Our aim is to apply the analysis of Section 3.2.2 in a modular way, by analyzing each scope separately, starting from S_1, then S_2, etc. Each scope S_k is analyzed by assuming an initial state, as in the non-modular case, but with a call to the entry procedure $p_k(\bar{x}, \bar{y})$.

In order to achieve this modularity, we extend the transfer function τ for the case of external procedure calls, such that it uses the corresponding summaries. Let $p(\bar{w}, \bar{s})$ be an external call, the result of $\tau(p(\bar{w}, \bar{s}), \langle \phi, \theta \rangle)$ is $\langle \phi', \theta' \rangle$ where:

1. $\theta' = \theta \cap \theta_p$
2. $\forall z \in \mathrm{dom}(\phi) \setminus \bar{s}$, we have $\phi'(z) = \phi(z)$; otherwise
3. $\forall s_i \in \bar{s}$, then $\phi'(s_i) = \mathtt{ren}(\phi_p(y_i), \phi, \theta)$ where $\mathtt{ren}$ is:

   $\mathtt{ren}(\ell, \phi, \theta)$:
   if ℓ includes a field or array-type $f \notin \theta$ **then return** $\ell_{\mathtt{any}}$
   else return $\ell[l_1/\phi(w_1), \ldots, l_n/\phi(w_n)]$

Intuitively, in (1) fields and array-types that might be updated during the execution of p are eliminated in the calling context; in (2) variables in the calling context which are not output variables of p keep their current access paths; and, in (3) the access paths of the output variables $\bar{s}$ are incorporated into the calling context.

Example 3.2.4. *We demonstrate the modular analysis on the example of Figure 3.3. We focus on method* next*, on the inner while loop of Example 3.2.3, and on the (outer) while loop (while$_m$) inside method* m*, which calls method* next *and*

procedure while (and hence reuses their summaries). The translation of next *and the outer loop into the rule based representation is as follows:*

$next(\langle this\rangle, \langle r\rangle) \leftarrow$
$\quad obj := this.state,$
$\quad s_0 := obj.next,$
$\quad this.state := s_0,$
$\quad r := obj.$

$while_m(\langle a, b, y, o, i, j\rangle, \langle o, i, j\rangle) \leftarrow$
$\quad s_0 := y,$
$\quad hasNext(\langle s_0\rangle, \langle s_0\rangle),$
$\quad while^c_m(\langle a, b, y, o, i, j, s_0\rangle, \langle o, i, j\rangle).$

$while^c_m(\langle a, b, y, o, i, j, s_0\rangle, \langle o, i, j\rangle) \leftarrow s_0 = 0.$
$while^c_m(\langle a, b, y, o, i, j, s_0\rangle, \langle o, i, j\rangle) \leftarrow s_0 \neq 0,$
$\quad s_0 := y,$
$\quad$ ④$next(\langle s_0\rangle, \langle s_0\rangle),$
$\quad o := s_0,$
$\quad s_1 := o.data,$
$\quad i := s_1,$
$\quad j := s_1,$
$\quad while(\langle a, b, i, j\rangle, \langle\rangle),$
$\quad while_m(\langle a, b, y, o, i, j\rangle, \langle o, i, j\rangle).$

Let us consider the scopes $S_1 = \{next\}$, $S_2 = \{while, while_c\}$ *and* S_3=$\{while_m,$ $while^c_m\}$. *We first analyze* S_1 *which, in addition to the abstract states for each program point, computes this summary for next:*

$$\langle\phi_{next}, \theta_{next}\rangle{=}\langle\{this{\mapsto}l_1, r{\mapsto}l_1{\cdot}state\}, \{next, \texttt{aint}\}\rangle$$

The set θ_{next} *does not include the field state, which indicates that its memory location might be modified during the execution of next. The meaning of the access path* $r \mapsto l_1{\cdot}state$ *is that the returned value of the method next is equal to the value of dereferencing (upon entering the procedure) the first input argument using the field state. Let us explain how this summary is reused when analyzing the scope* S_3. *When reaching program point* ④ *for the first time, we will have the abstract state:*

$$\langle\phi_0, \theta_0\rangle{=}\langle\{a{\mapsto}l_1, b{\mapsto}l_2, y{\mapsto}l_3, s_0{\mapsto}l_3, o \mapsto l_4, i \mapsto l_5, j \mapsto l_6\}, \{state, next, \texttt{aint}\}\rangle$$

Note that θ_0 *includes all fields and array-types that appear in* S_1, S_2 *and* S_3, *since no one has been updated so far. In order to incorporate the effect of executing the method next into the calling context, we apply the transfer function*

$\tau(next(\langle s_0\rangle, \langle s_0\rangle), \langle \phi_0, \theta_0\rangle)$, *which results in:*

$$\langle \phi_1, \theta_1\rangle = \langle \{a \mapsto l_1, b \mapsto l_2, y \mapsto l_3, s_0 \mapsto l_3 \cdot state, o \mapsto l_4, i \mapsto l_5, j \mapsto l_6\}, \{next, \texttt{aint}\}\rangle$$

Now, field state is not in θ_1*. The access path* $s_0 \mapsto l_3 \cdot state$ *is obtained by taking that of* r*, i.e.,* $l_1 \cdot state$ *and renaming* l_1 *to* $\phi_0(s_0) = l_3$*. We take* $\phi_0(s_0)$ *since* l_1 *refers to the value of the first argument when calling method next, which is* s_0*. In the next iteration of the analysis, we reach* ④ *with the abstract state:*

$$\langle \phi_2, \theta_2\rangle = \langle \{a \mapsto l_1, b \mapsto l_2, y \mapsto l_3, s_0 \mapsto l_3 \cdot state, o \mapsto l_4, i \mapsto l_5, j \mapsto l_6\}, \{next\}\rangle$$

Observe that `aint` *is not included in* θ_2*, since it is removed when incorporating the summary of the inner while loop (computed from the analysis results shown in Example 3.2.3), which modifies an array of integers. Next, applying* $\tau(next(\langle s_0\rangle, \langle s_0\rangle), \langle \phi_2, \theta_2\rangle)$ *results in:*

$$\langle \phi_3, \theta_3\rangle = \langle \{a \mapsto l_1, b \mapsto l_2, y \mapsto l_3, s_0 \mapsto \ell_{any}, o \mapsto l_4, i \mapsto l_5, j \mapsto l_6\}, \{next\}\rangle$$

The access path $s_0 \mapsto \ell_{any}$ *is also obtained from* $r \mapsto l_1 \cdot state$ *as before. However, in this case* `ren` *returns* ℓ_{any} *since* $l_1 \cdot state$ *includes the field state and state* $\notin \theta_3$*.*

Given an access path $\ell \neq \ell_{\texttt{any}}$ and an initial state $C_0 = \langle start, p_i(\bar{x}, \bar{y}), tv\rangle; h$, we let $[\![\ell]\!](C_0)$ be the value that corresponds to ℓ in C_0. For instance, for local variables the value is obtained from the variable mapping $[\![l_2]\!](C_0) = tv(x_2)$, for fields the heap must be accessed as well $[\![l_2 \cdot f]\!](C_0) = h(tv(x_2)) \cdot f$, etc. (See the proof of Theorem 3.2.5 for the exact definition). Now we state the soundness theorem, assuming that the analysis results for each program point $k{:}j$ are obtained in a modular way as explained above.

Theorem 3.2.5 (soundness)**.** *Given a scope* S_i*, a program point* $k{:}j$ *in* S_i*, and a variable* $x \in \mathcal{V}_{k:j}$ *such that* $\phi_{k:j}(x) = \ell \neq \ell_{any}$*. Then, for any trace* $C_0 = \langle start, p_i(\bar{x}, \bar{y}), tv\rangle; h \rightsquigarrow^* \langle q, bc, tv'\rangle \cdot ar; h'$ *where* bc *corresponds to program point* $k{:}j$*, it holds that* $tv(x) = [\![\ell]\!](C_0)$*.*

Proof. Let us formally define $[\![\ell]\!](C)$, i.e., how to interpret the access path $\ell \neq \ell_{\texttt{any}}$

in a configuration $C = \langle q, bc, tv\rangle{\cdot}ar; h$:

$$
\begin{aligned}
[\![\ell_{\mathsf{null}}]\!](C) &= \mathsf{null} \\
[\![n]\!](C) &= n \\
[\![l_i{\cdot}f_1{\cdots}f_n]\!](C) &= [\![f_1{\cdots}f_n]\!](C, tv(x_i)) \\
[\![l_i[\ell]{\cdot}f_1{\cdots}f_n]\!](C) &= [\![f_1{\cdots}f_n]\!](C, h(tv(x_i)).v)) \text{ where } v = [\![\ell]\!](C)
\end{aligned}
$$

$$
\begin{aligned}
[\![f{\cdot}A_1{\cdots}A_n]\!](C, x) &= [\![A_1{\cdots}A_n]\!](C, h(x).f) \\
[\![f[\ell]{\cdot}A_1{\cdots}A_n]\!](C, x) &= [\![A_1{\cdots}A_n]\!](C, h(h(x).f).v) \text{ where } v = [\![\ell]\!](C) \\
[\![\epsilon]\!](C, x) &= x
\end{aligned}
$$

The first two cases are straightforward. The third (resp. sixth) case evaluates l_i (resp. $l_i[\ell]$), and then makes a recursive call to dereference (when $n > 0$) the corresponding object with $f_1 \dots f_n$, which is done iteratively by the last three cases.

Let us consider first the analysis of a standalone scope S, i.e., a scope that does not call procedures from other scopes. Afterwards, we consider the modular case. Assume a given standalone scope S with an entry procedure p, the analysis is applied iteratively using $I^{\#}_{i+1} = I^{\#}_i \sqcup_p F^{\#}_{i+1}(I^{\#}_i)$, starting from an abstract state $I^{\#}_0$ that maps every program point $k{:}j$ of S to $\bot_p$, except for program point 0:0 which is mapped to the initial abstract state $\langle \phi, \mathit{fields}(P) \cup \{\texttt{aint}, \texttt{aref}\}\rangle$ in which $\phi(x_i) = l_i$; and $\phi(y_i)$ is 0 or ℓ_{null} depending on the type of y_i.

Similarly to $F^{\#}$, we define $F(X) = \{C' \mid C \in X,\ C \rightsquigarrow C'\}$, $I_{i+1} = I_i \cup F(I_i)$, and $I_0 = \{C_0\}$ where $C_0 = \langle \mathit{start}, p(\bar{x}, \bar{y}), tv_0\rangle; h_0$. Each I_i represents a set of reachable states in at most i derivation steps, when starting from C_0. For simplicity, we assume that each y_i in the initial state is either 0 or null, this does not restrict the correctness statements since these values are never used (they are overwritten upon exit).

In order to show that Theorem 3.2.5 is correct for the standalone case, it is enough to show that the following holds:

> For any $C_0 = \langle \mathit{start}, p(\bar{x}, \bar{y}), tv_0\rangle; h_0$, if $C = \langle q, bc, tv\rangle{\cdot}ar; h \in I_i$, and bc corresponds to program point $k{:}j$ (different from 0:0), then $k{:}j \mapsto \langle \phi, \theta\rangle \in I^{\#}_i$ such that

(1) $\phi(x) = \ell \neq \ell_{\texttt{any}} \Rightarrow tv(x) = [\![\ell]\!](C_0)$;

(2) if $f \in \theta$, then:

(2.1) if $f \in \mathit{fields}(P)$, $r \in \mathrm{dom}(h_0)$, and $h(r)$ is an object that has a field with name f, then $h_0(r).f = h(r).f$;

(2.2) if $f \in \{\texttt{aint}, \texttt{aref}\}$, $r \in \mathrm{dom}(h_0)$, and $h_0(r)$ is an array of static type f, then the elements of $h_0(r)$ and $h(r)$ have the same values;

We say that $\langle \phi, \theta \rangle$ *correctly approximates* C. Note that our main interest is in showing that (1) holds, (2) is an auxiliary statement required for doing this.

We prove the above claim, for an arbitrary C_0, by induction on the number of iterations when computing I_i and $I_i^{\#}$ (i.e., on i). The claim trivially holds for $i = 0$. We assume it holds for I_i and $I_i^{\#}$, and now we show that it holds for I_{i+1} and $I_{i+1}^{\#}$.

Pick an arbitrary $C' = \langle q', bc', tv' \rangle \cdot ar'; h' \in I_{i+1}$, and let bc' correspond to program point $k{:}j$. Assume $C' \notin I_i$, otherwise the claim holds by the induction hypothesis. Thus, $C' \in F(I_i)$, i.e, we have $C = \langle q, bc, tv \rangle \cdot ar; h \in I_i$ such that $C \rightsquigarrow C'$. We prove that the claim holds for such case by considering all possible ways to move from C to C'. In particular we prove that there is $\langle \phi, \theta \rangle$ in $F(I_{i+1}^{\#})$, associated to the program point of bc', that correctly approximates C'.

First note that, by the induction hypothesis, there is $k{:}j \mapsto \langle \phi, \theta \rangle \in I_i^{\#}$ that correctly approximates C, i.e., satisfies conditions (1) and (2) for C. In what follows, when we refer to $\langle \phi, \theta \rangle$, we mean exactly this one.

Let us start by considering the simple instructions, i.e., those that do not correspond to calling (or returning from) a procedure. For such cases, using $\langle \phi, \theta \rangle$ and $F_1^{\#}$, we compute $k{:}j{+}1 \mapsto \langle \phi', \theta' \rangle$. We claim that $\langle \phi', \theta' \rangle$ correctly approximates C'.

Case 1: $x := y.f$ We have $\theta' = \theta$, and ϕ' is different from ϕ only by the value of x. Trivially, Condition (2) holds since the heap is not modified, and Condition (1) holds for any variable different from x. We show that Condition (1) holds also for x. If $\phi'(x) = \ell_{\texttt{any}}$ then Condition (1) trivially holds for x. The other possibility

is that $\phi'(x) = \ell{\cdot}f$, in which case $tv(y) = tv'(y) = \ell \neq \ell_{\texttt{any}}$ and $f \in \theta$, then we have $tv'(x) = tv(y).f = [\![\ell]\!](C_0).f = [\![\ell{\cdot}f]\!](C_0)$.

Case 2: $x.f := y$ Condition (1) holds since no variable is updated. Also Condition (2) holds since the only field that has been modified is removed from the set θ.

Case 3: $x := n$ Condition (1) holds for any variable different from x, and it trivially holds for x since its access path is the number n. Condition (2) also holds since these instructions does not modify the heap.

Case 4: $x := \mathsf{null}$ Straightforward using the same reasoning as in Case 3.

Case 5: $x := y$ Condition (1) holds for any variable different from x. It holds also for x because $tv'(x) = tv(y) = [\![\phi(y)]\!](C_0) = [\![\phi(x)]\!](C_0)$.

Case 6: $x := y \ aop \ z$ Condition (1) holds for any variable different from x. If $\phi'(x) = \ell_{\texttt{any}}$ then it also trivially holds for x. If $\phi'(x) \neq \ell_{\texttt{any}}$, then $\phi(y)$ and $\phi(z)$ must be numbers, and thus $tv'(x) = tv(y) + tv(z) = [\![\phi(x)]\!](C_0) + [\![\phi(z)]\!](C_0) = [\![\phi'(x)]\!](C_0)$.

Case 7: $x := \mathsf{newarray}(D, y)$ Clearly Condition (1) holds for any variable different from x. It holds also for x since $\phi'(x) = \ell_{\texttt{any}}$. Condition (2) holds since no field or array-type is modified.

Case 8: $x := y[z]$ Analogue to Case 1.

Case 9: $x[y] := z$ Analogue to Case 2.

Case 10: $x := \mathsf{new}\ C$ Analogue to Case 7.

Case 11: $x := \mathbf{arraylength}(y)$ Condition (1) holds for any variable different from x. It holds also for x since $\phi'(x) = \ell_{\texttt{any}}$. Condition (2) holds since no field or array-type is modified.

Now we consider the case of a procedure call. For this, we assume that C is of the form $\langle q, q'(\bar{w}, \bar{z}) \cdot bc'', tv\rangle{\cdot}ar; h$, and we make an execution step using a corresponding rule $q'(\bar{x}, \bar{y}) \leftarrow g, b_1^{k'}, \cdots, b_n^{k'} \in P$. Thus, according to the language's semantics we get $C' = \langle q', b_1^{k'} \cdots b_n^{k'}, tv'\rangle{\cdot}\langle q, q[\bar{y}/\bar{z}] \cdot bc'', tv\rangle{\cdot}ar; h$ where tv' maps every variable to either 0 or null, except for the formal parameters $\bar{x}$ whose

values are obtained from the actual ones $\bar{w}$. Note that C' corresponds to program point k':1. In the abstract setting, using $\langle \phi, \theta \rangle$, rule k' and $F_2^\#$ we compute $k'{:}1 \mapsto \langle \phi', \theta' \rangle$. We claim that $\langle \phi', \theta' \rangle$ correctly approximates C'. This is because Condition (2) holds since $\theta' = \theta$ and the derivation step does not modify any field or array; and Condition (1) also holds since all we do is to copy the values of the access paths of actual parameters from those of the formal parameters, the rest of local variables are initialized to 0 or ℓ_{null} for which Condition (1) trivially holds.

Now we consider the case of a return from a procedure. For this we assume that C is of the form $\langle q, \epsilon, tv \rangle \cdot \langle q', q[\bar{y}/\bar{z}] \cdot bc', tv'' \rangle \cdot ar; h$, which according to the language's semantics can, in a single step, only lead to $C' = \langle q', bc', tv' \rangle \cdot ar; h$ where tv' maps all variables as in tv'', except for the output variables $\bar{z}$ for which we have $tv'(z_i) = tv(y_i)$. Now, note that there must be an abstract state C'' of the form $\langle q', q(\bar{w}, \bar{z}) \cdot bc', tv'' \rangle \cdot ar; h'$ in I_i from which we have (transitively) obtained C. Assume that C'' corresponds to program point $k'{:}j$. Thus, C' corresponds to program point $k'{:}j{+}1$. By the induction hypothesis, $I_i^\#$ must include $k'{:}j \mapsto \langle \phi'', \theta'' \rangle$ that correctly approximates C''. It is easy to see that, using $F_3^\#$, together with $\langle \phi, \theta \rangle$ and $\langle \phi'', \theta'' \rangle$, we get $k'{:}j{+}1 \mapsto \langle \phi', \theta' \rangle$ that correctly approximates C'.

We have proved that whenever $F(I_i)$ introduces a state C', then $F^\#(I_i^\#)$ introduces an abstract state $\langle \phi', \theta' \rangle$ that correctly approximates C'. Merging $F^\#(I_i^\#)$ with $I_i^\#$ using $\sqcup_p$ keeps this approximation correct since the only changes that can happen are: the access path of a variable x is upgraded to $\ell_{\mathtt{any}}$, or a field or array-type f is removed from θ. In both cases, respectively, Conditions (1) and (2) still hold.

Q.E.D.

Consider now the modular case. Let us first note the following immediate consequence of Theorem 3.2.5. Assume that we have analyzed a scope S (that does not call any other scope), and that we have obtained the following summary for procedure p:

$$\langle \phi_p, \theta_p \rangle = \sqcup_s \{ \langle \phi, \theta \rangle \mid p(\bar{x}, \bar{y}) \leftarrow g, b_1^k, \ldots, b_t^k \in P, k{:}t{+}1 \mapsto \langle \phi, \theta \rangle \}$$

Let $C \rightsquigarrow^* C'$ where $C = \langle q, p(\bar{w}, \bar{z}) \cdot bc, tv \rangle \cdot ar; h$, $C' = \langle q, bc, tv' \rangle \cdot ar; h'$, and q (the procedure currently executed in C) is not in the same scope of p, i.e., p is an external call. Then, clearly, if $\phi_p(y_i) = \ell \neq \ell_{\texttt{any}}$, we have $tv'(z_i) = [\![\ell]\!](C)$.

Now let us change the definition of $F(X)$ in order to collect the reachable states that correspond to program points in a given scope only. This can be done by changing F such that when $C = \langle q, p(\bar{w}, \bar{z}) \cdot bc, tv \rangle \cdot ar; h$ and p is external, then instead of making a single derivation step we use $C \rightsquigarrow^* C'$ where $C' = \langle q, bc, tv' \rangle \cdot ar; h'$, i.e., we execute p completely.

We claim that Theorem 3.2.5 still holds when applied to a given scope S. The proof is the same as for the standalone scope, except that, we must extend it to handle calls to external procedures.

Assuming the C corresponds to program point $k{:}j$, and that $k{:}j \mapsto \langle \phi, \theta \rangle \in I_i^{\#}$ correctly approximates C, we show that $\langle \phi', \theta' \rangle = \tau(p(\bar{w}, \bar{z}), \langle \phi, \theta \rangle)$ correctly approximates C'. Condition (2) holds since $\theta' = \theta \cap \theta_p$. Condition (1) clearly holds for any variable not in $\bar{z}$. Next, we prove that it holds also for those variables in $\bar{z}$.

Applying the definition of τ for external calls, we get that each $\phi'(z_i)$ equals to $\texttt{ren}(\phi_p(y_i), \phi, \theta)$. According to the definition of `ren`, if $\phi_p(y_i)$ includes a field or array-array type $f \notin \theta$ we get $\phi'(z_i) = \ell_{\texttt{any}}$, so for such case Condition (1) trivially holds. Assume that there is no such f, i.e., we are in the else branch of `ren`. In such case, if $\phi_p(y_i) = \ell_{\texttt{any}}$, or it includes l_j such that $\phi(w_j)$ is $\ell_{\texttt{any}}$, then $\phi'(z_i) = \ell_{\texttt{any}}$, so for this case too Condition (1) trivially holds. Now, assume this is not the case, then, $\phi'(z_i) = \ell'$ where ℓ' is obtained from $\phi_p(y_i)$ by replacing each l_j by $\phi(w_n)$. We prove that $tv(z_i) = [\![\ell']\!](C_0)$: As we have commented above (at the beginning of the proof of the modular analysis) we have $tv(z_i) = [\![\phi_p(y_i)]\!](C)$. Now when evaluating $[\![\phi_p(y_i)]\!](C)$, we reach base-cases in which we need to evaluate $[\![l_j]\!](C)$, which is equal to $tv(x_i)$, and by the induction hypothesis it is equal to $[\![\phi(x_j)]\!](C_0)$. Thus, $[\![\phi_p(y_i)]\!](C)$ is equal to $[\![\ell]\!](C_0)$. □

3.3 Heap-Sensitive Analysis

This section presents the core of our method in three steps: we provide in Section 3.3.1 some auxiliary definitions and the basic notion of (unconditional) locality; then, Section 3.3.2 introduces the notion of *conditional partition* that will let us transform a heap access by a local variable under certain conditions; finally, we present in Section 3.3.3 an automatic transformation that actually carries out the conversion.

3.3.1 Basic Locality

Let us first introduce two auxiliary notions which define the set of *read and write access paths* to fields and to arrays within a scope. Intuitively, such sets provide information on how a field or array-type is accessed in a scope (and in its reachable scopes). This information is needed in Section 3.3.3 for soundly tracking the values that are stored in such heap location.

Definition 3.3.1. *Given a scope S and a field f, the set of* read access paths *for f in S, denoted by $R(S,f)$, is defined as $R(S,f) = R^+(S,f) \cup R^*(S,f)$ where*

$$R^+(S,f) = \{\ell \mid b_j^k \equiv x := y.f \in S, \ell = \phi_{k:j}(y), \ell \neq \ell_{null}\}$$
$$R^*(S,f) = \{\ell' \mid b_j^k \equiv q(\bar{x},\bar{y}) \in S, q \in S' \neq S, \ell \in R(S',f), \ell' = \texttt{ren}(\ell, \phi_{k:j}, \theta_{k:j})\}$$

The set of write access paths for f in S, denoted $W(S,f)$, is computed analogously, by considering instructions of the form $y.f := x$.

Let us explain the above definition. In $R^+(S,f)$, for each access $x := y.f$ we add the access path that the analysis has computed for y. Computing the read access paths for a scope S requires computing the read access paths for all other scopes transitively called from S. This is done in $R^*(S,f)$. For each call such that q is the entry of scope S' we take $R(S',f)$ and rename it according to the calling context using `ren` as defined in Section 3.2.3. Note that all access paths in $R(S,f)$ and $W(S,f)$ refer to memory locations, they do not include ℓ such that $\ell \in \mathbb{Z}$ or $\ell = \ell_{\mathsf{null}}$. The above definition extends for arrays in a natural way.

Definition 3.3.2. *Given a scope S and an array-type $f \in \{\texttt{aref}, \texttt{aint}\}$, the set of* read access paths *for f in S, denoted by $R(S, f)$, is defined as $R(S, f) = R^+(S, f) \cup R^*(S, f)$ where*

$$R^+(S, f) = \{\ell \mid b_j^k \equiv x := y[z] \in S, \texttt{stype}(y){=}f, \phi_{k:j}(y){\neq}\ell_{null}, \ell{=}\phi_{k:j}(y)[\phi_{k:j}(z)]\}$$

and $R^*(S, f)$ is as in Definition 3.3.1. The set of write access paths for f in S, denoted $W(S, f)$, is computed analogously, by considering instructions of the form $y[z] := x$.

Example 3.3.3. *Using the results of the constancy analysis in Examples 3.2.3 and 3.2.4, we have that the read/write access sets are: $R(S_1, next) = \{l_1.state\}$ and $W(S_1, next) = \{\}$ and $R(S_1, state) = W(S_1, state) = \{l_1\}$. In the scope of the inner loop, we have that $R(S_2, \texttt{aint}) = W(S_2, \texttt{aint}) = \{l_1[l_3], l_2[l_4]\}$, since the array content is read and modified using the references a[i] and b[j]. In S_3, we have that $R(S_3, next) = \{l_3{\cdot}state\}$, $W(S_3, next) = \{\}$, $R(S_3, state) = W(S_3, state) = \{l_3\}$ and $R(S_3, \texttt{aint}) = W(S_3, \texttt{aint}) = \{l_1[l_5], l_2[l_6]\}$.*

Intuitively, in order to ensure a sound transformation, a field can be considered *local in a scope S* if all read and write accesses to it in *all* reachable scopes are performed through the same access path. This makes it safe to replace such heap access by a corresponding local variable.

Definition 3.3.4 (locality)**.** *Given a field or an array-type f and a scope S, we say that f is local in S if $R(S, f) \cup W(S, f) = \{l\}$.*

Example 3.3.5. *From the results computed in Example 3.3.3, we have that the array type* `aint` *is not local in the scope S_2 corresponding to the inner loop, because $R(S_2, \texttt{aint}) = W(S_2, \texttt{aint}) = \{l_1[l_3], l_2[l_4]\}$, i.e., the union contains more than one element. But we have that next and state are local in both S_1 and S_3. Consider again the small examples in Figure 3.1: we have that in Ⓐ, field f is local in S_1 because $R(S_1, f) = \{l_1\}$ and $W(S_1, f) = \emptyset$. However, it is not local in S_2 because $R(S_2, f) \cup W(S_2, f) = \{\ell_{any}\}$. In Ⓑ, we have that c is local in S_1 because $R(S_1, c) \cup W(S_1, c) = \{l_1\}$, while as before it is not local in S_2 because $R(S_2, c) \cup W(S_2, c) = \{\ell_{any}\}$. In Ⓒ, size is not local because $R(S_1, size) \cup W(S_1, size) = \{\ell_{any}\}$. Also, in Ⓓ, we have that $R(S_1, r) \cup W(S_1, r) = \{\ell_{any}\}$.*

3.3.2 Locality Partition

In ideal scenarios in which fields are unconditionally local, we can transform each local field access in the considered scope into an equivalent access using a local variable which exposes the value of the field. However, there are cases in which the read and write sets do not provide enough information for tracking the values stored in the corresponding locations. In such cases, it is often possible to provide preconditions under which tracking such locations is possible. When such conditions are used, any property for the locations that we infer (e.g., using static analysis) is sound only for inputs that satisfy the precondition.

Definition 3.3.6 (aliasing preconditions)**.** *An aliasing precondition φ is a Boolean formula $\bigvee_i(\bigwedge_j)c_{ij}$ where each c_{ij} is an* atomic aliasing proposition *of the form $\ell_1 \approx \ell_2$ or $\ell_1 \not\approx \ell_2$.*

The meaning of $\ell_1 \approx \ell_2$ (resp. $\ell_1 \not\approx \ell_2$) is, as expected, that ℓ_1 and ℓ_2 alias (resp. do not alias). For simplicity in the notation, we will use the term *aliasing* for access paths that represent memory locations as well as integer values. Note that, by definition, some propositions are *true*, e.g., $l_1 \approx l_1$, and some are *false*, e.g., $l_1 \not\approx l_1$. Moreover, for any access path ℓ we let $\ell \not\approx \ell_{\texttt{any}}$ and $\ell \approx \ell_{\texttt{any}}$ be both *false*. This is because such accesses are not constant. When an aliasing precondition φ implies another precondition φ' we write $\varphi \models \varphi'$. We will be mainly interested in implied atomic aliasing propositions, e.g., $\varphi \models \ell_1 \approx \ell_2$ and $\varphi \models \ell_1 \not\approx \ell_2$.

Example 3.3.7. *In our running example, we cannot precisely track the write accesses to the arrays (*a *or* b*) in S_2 because the memory location accessed depends on an aliasing condition: if* a *and* b *point to the same array, the content of such array may be modified using both accesses,* a[i] *or* b[j]*. Furthermore, if* i ≈ j*, both accesses are modifying exactly the same element of the array. Thus, the trackability of array accesses and, as a consequence, the number of ghost variables needed to track them depends on some preconditions that are given in terms of the initial parameters. E.g. assuming the precondition $l_1[l_3] \not\approx l_2[l_4]$ over the read/write sets in Example 3.3.3, we will need two different ghost variables to safely represent these array references, because they are pointing to different memory locations. However, if we assume that $l_1[l_3] \approx l_2[l_4]$, all accesses point*

to the same array element, so we just need one ghost variable to track both array accesses.

Note that every $\ell \in R(S,f) \cup W(S,f)$ is associated to a set of program points in which the corresponding field/array access appear. The set of all such program points is denoted by $\texttt{rwpps}(\ell)$.

Definition 3.3.8 (locality partition)**.** *Given a field or an array-type f, a scope S, and an aliasing precondition φ. We say that a partition $G_1, \ldots, G_n$ of $R(S,f) \cup W(S,f)$ is a locality partition for f w.r.t. φ if:*

1. $\forall 1 \leq i \leq n.\ \forall \ell_1, \ell_2 \in G_i.\ \varphi \models \ell_1 \approx \ell_2$; *and*
2. $\forall 1 \leq i < j \leq n.\ \forall \ell_1 \in G_i.\ \forall \ell_2 \in G_j.\ \varphi \models \ell_1 \not\approx \ell_2$.
3. $\forall 1 \leq i < j \leq n.\ \forall \ell_1 \in G_i.\ \forall \ell_2 \in G_j.\ \texttt{rwpps}(\ell_1) \cap \texttt{rwpps}(\ell_2) = \emptyset$.

We denote the locality partition by $\mathcal{P}_f^{\varphi}$.

Let us explain the above definition: Condition (1) requires that the access paths in each G_i alias, i.e., they all refer to the same memory location to do all heap accesses in G_i trackable. Condition (2) requires that access paths from different partitions do not alias, i.e., they refer to different memory locations. The main idea is that now each component G_i can be used to track the value stored in a corresponding memory location by means of a different ghost variable. Condition (3) requires that every (trackable) access in the program always refers to the same memory location. This condition is added for simplifying the presentation and it could be omitted if we use a polyvariant transformation [AAG+10, RD11] which clones the code for each calling pattern. We say that the memory location induced by G_i is trackable.

If $\ell_{\texttt{any}} \in R(S,f) \cup W(S,f)$ then there is no partition that satisfies the above definition. This is because in (1) we can take $\ell_1 = \ell_2 = \ell_{\texttt{any}}$ for which $\ell_1 \approx \ell_2$ does not hold. Observe that Definition 3.3.4 induces a locality partition w.r.t. the aliasing precondition *true*, i.e., the heap access is unconditionally local.

Example 3.3.9. *Partitions can be built by considering all possible equalities and disequalities of the elements in $R(S,f) \cup W(S,f)$. Consider the read/write access*

sets in S_2 in Example 3.3.3, the following two locality partitions can be generated: (1) $G_1 = \{l_1[l_3], l_2[l_4]\}$ which gives us the precondition $\psi_1 = \{l_1[l_3] \approx l_2[l_4]\}$, or if we refer to the source code variables, then $\psi_1 = \{\mathsf{a[i]} \approx \mathsf{b[j]}\}$; (2) G_1=$\{l_1[l_3]\}$, G_2=$\{l_2[l_4]\}$ which gives us the precondition $\psi_2 = \{l_1[l_3] \not\approx l_2[l_4]\}$ and equivalently, using the source code variables, $\psi_2 = \{\mathsf{a[i]} \not\approx \mathsf{b[j]}\}$.

3.3.3 Automatic Transformation

In addition to identifying when memory locations are trackable w.r.t. a given precondition φ, we need to find a way to actually track them. As mention before, our approach is based on instrumenting the program with extra local (ghost) variables that expose the values of those locations to a heap-insensitive analysis. This is done as follows: (1) for each trackable location induced by G_i, we introduce a ghost variable g; (2) when the content of the memory location is modified, we modify g accordingly; and (3) when the memory location is read, we read the value from g. This approach has one clear advantage: there is no need to change existing static analysis tools to make them heap-sensitive, we simply apply them on the transformed program, and then the properties inferred for the ghost variables hold also for the corresponding memory locations.

Definition 3.3.10 (locatity transformation)**.** *Given a scope S, and a corresponding locality partition $\mathcal{P}_f^{\varphi} = \langle G_1, \ldots, G_n \rangle$. The instrumented program $\mathcal{T}(\mathcal{P}_f^{\varphi})$ is obtained by transforming all rules of S^* as follows:*

1. *Generate n different ghost variables names, $\bar{g} = \langle g_1, \ldots, g_n \rangle$;*
2. *Every procedure call or rule head $p(\bar{x}, \bar{y})$ is replaced by $p(\bar{x}{\cdot}\bar{g}, \bar{y}{\cdot}\bar{g})$; and*
3. *For every $\ell \in G_i$, and $k{:}j \in \mathtt{rwpps}(\ell)$, the field or array access at program point $k{:}j$ is replaced by g_i.*

Given k different fields $f_1, \ldots, f_k$ with locality partitions $\mathcal{P}_{f_1}^{\varphi_1}, \ldots, \mathcal{P}_{f_k}^{\varphi_k}$, we let $\mathcal{T}(\mathcal{P}_{f_1}^{\varphi_1}, \ldots, \mathcal{P}_{f_k}^{\varphi_k})$ be the program obtained by applying the above steps on each $\mathcal{P}_{f_i}^{\varphi_i}$, iteratively.

Let us explain the above transformation: in step (1) we create the ghost variables $\langle g_1, \ldots, g_n \rangle$, where g_i will be used to track the content of the memory location induced by G_i; in (2) we add the ghost variables as input and output arguments to all rules in S^*; and in (3) we simply replace accesses to the memory locations by accesses to the corresponding ghost variables. When several fields or arrays are going to be transformed, the instrumented program is obtained by applying the transformation on each corresponding locality partition iteratively. This is safe since $f_i \neq f_k$, guaranteeing that the different partitions refer to different memory locations.

Example 3.3.11. *Using the preconditions and partitions of Example 3.3.9, we apply Definition 3.3.10 twice, once for each precondition and obtain the two versions depicted in the first two columns of Figure 3.5, where* $\overline{arg}$ *stands for* a, b, i, j*. The one in the first column corresponds to the transformation for* ψ_1*, and has one ghost variable* g_1*, while the one in the second column corresponds the one for* ψ_2 *and has two ghost variables* g_1 *and* g_2*. Observe that the second version always terminates, while the first one might not because both array accesses,* `a[i]` *and* `b[j]`*, modify the same array location. Therefore, using the transformed program, a heap-insensitive termination analyzer would infer that the while loop at hand terminates for the precondition* ψ_2*, i.e.* $\{\mathsf{a} \not\approx \mathsf{b} \vee \mathsf{i} \not\approx \mathsf{j}\}$.

We say that a configuration C satisfies an aliasing precondition φ, denoted by $C \models \varphi$, iff for any ℓ_1 and ℓ_2 such that $\varphi \models \ell_1 \approx \ell_2$ (resp. $\varphi \models \ell_1 \not\approx \ell_2$), it holds that $[\![\ell_1]\!](C) = [\![\ell_2]\!](C)$ (resp. $[\![\ell_1]\!](C) \neq [\![\ell_2]\!](C)$). The following soundness theorem states that any reachable state in the original program, has a corresponding "equivalent" one in the transformed program. Given a ghost variable g_i, we let ℓ_{g_i} be the memory location that g_i tracks.

Theorem 3.3.12. *Let* S *be a scope with an entry* p*,* $\mathcal{P}^{\varphi_1}_{f_1}, \ldots, \mathcal{P}^{\varphi_k}_{f_k}$ *be locality partitions such that* $f_i \neq f_j$*,* $C_0 = \langle start, p(\bar{x}, \bar{y}), tv_0 \rangle; h_0$ *such that* $C_0 \models \varphi_1 \wedge \ldots \wedge \varphi_n$*, and* $C'_0 = \langle start, p(\bar{x} \cdot \bar{g}, \bar{y} \cdot \bar{g}), tv_0 \rangle; h_0$ *such that* tv' *extends* tv *with* $tv'(g_i) = [\![\ell_{g_i}]\!]C_0$*: If* $C_0 \rightsquigarrow^n \langle q, bc_n, tv_n \rangle \cdot ar; h_n$ *is a possible execution using* S*, then* $C'_0 \rightsquigarrow^n \langle q, bc_n, tv'_n \rangle \cdot ar; h'_n$ *is a possible execution using the corresponding* $\mathcal{T}(\mathcal{P}^{\varphi_1}_{f_1}, \ldots, \mathcal{P}^{\varphi_k}_{f_k})$ *and* $tv_n(x) = tv'_n(x)$ *for any* $x \in \mathrm{dom}(tv_n)$.

(pre-cond. ψ_1)	(pre-cond. ψ_2)
$while(\langle \overline{arg}, g_1 \rangle, \langle g_1 \rangle) \leftarrow$	$while(\langle \overline{arg}, g_1, g_2 \rangle, \langle g_1, g_2 \rangle) \leftarrow$
$s_0 := a, s_1 := i, s_0 := g_1,$	$s_0 := a, s_1 := i, s_0 := g_1,$
$while_c(\langle \overline{arg}, g_1, s_0 \rangle, \langle g_1 \rangle).$	$while_c(\langle \overline{arg}, g_1, g_2, s_0 \rangle, \langle g_1, g_2 \rangle).$
$while_c(\langle \overline{arg}, g_1, s_0 \rangle, \langle g_1 \rangle) \leftarrow s_0 \leq 0.$	$while_c(\langle \overline{arg}, g_1, g_2, s_0 \rangle, \langle g_1, g_2 \rangle) \leftarrow s_0 \leq 0.$
$while_c(\langle \overline{arg}, g_1, s_0 \rangle, \langle g_1 \rangle) \leftarrow s_0 > 0,$	$while_c(\langle \overline{arg}, g_1, g_2, s_0 \rangle, \langle g_1, g_2 \rangle) \leftarrow s_0 > 0,$
$s_1 := a, s_2 := i,$	$s_1 := a, s_2 := i,$
$s_3 := a, s_4 := i,$	$s_3 := a, s_4 := i,$
$s_3 := g_1, s_3 := s_3 - 1,$	$s_3 := g_1, s_3 := s_3 - 1,$
$g_1 := s_3, s_1 := b,$	$g_1 := s_3, s_1 := b,$
$s_2 := j, s_3 := b,$	$s_2 := j, s_3 := b,$
$s_4 := j, s_3 := g_1,$	$s_4 := j, s_3 := g_2,$
$s_3 := s_3 + 1, g_1 := s_3,$	$s_3 := s_3 + 1, g_2 := s_3,$
$while(\langle \overline{arg}, g_1 \rangle, \langle g_1 \rangle).$	$while(\langle \overline{arg}, g_1, g_2 \rangle, \langle g_1, g_2 \rangle).$

Figure 3.5: Resulting RBR after applying the transformations

Proof. The correctness of this Theorem is straightforward given the correctness of the access path analysis and the definition of locality partitions, as we explain below.

First, by the correctness of the access path analysis, and the definition of locality partition, it is clear that each ghost variable correctly tracks a corresponding memory location. This is because in the read and write sets we have all possible accesses, and the partition condition guarantees by definition the each G_i refers to different memory locations. Moreover, each instruction can refer to only one location. Second, the initial value of each g_i is the value stored in the memory location that it tracks. □

3.3.4 Heuristics for References

In the above transformation, we track all possible heap accesses. However, it is also safe not to track all of them. This means that we can select some of the G_i to be transformed. This is especially interesting when the heap accesses are reference fields or arrays of references. In theses cases, it is usually a good heuristics to track the locations that are used for traversing the data structures,

but do not track reference fields that are part of the data structure itself. This distinction can often be discovered because cursors are both read and updated, hence $G_i \cap R(S, f) \neq \emptyset$ and $G_i \cap W(S, f) \neq \emptyset$, while references that are part of the data structure are typically only read, i.e., $G_i \cap W(S, f) = \emptyset$.

Example 3.3.13. *In order to prove termination of methods that use method* next *to traverse the list (e.g., method* m*), we need to infer that the "size of* state*" decreases every time we execute method* next*. We use the path-length abstraction [SMP10] (i.e., the longest reachable path from the considered reference) as a size measure to check how the size of non-cyclic data structures is modified. By applying Definition 3.3.8 (or simply Definition 3.3.4), both reference fields* state *and rest are local in the scope of the method unconditionally. Thus, it is possible to convert them into respective ghost variables v_s (for* state*) and v_n (for rest). The rule defining* next *of Example 3.2.4 would be transformed into:*

$$next(\langle this, v_s, v_n\rangle, \langle v_s, v_n, r\rangle) \quad \leftarrow \quad obj := v_s, s_o := v_n, v_s := s_0, r := obj.$$

for which we cannot infer that the path-length of v_s in the output is smaller than that of v_s in the input. In particular, the path-length abstraction approximates the effect of the instructions by the constraints $\{obj = v_s, s_0 = v_n, v'_s = s_0, r = obj\}$. Primed variables are due to a single static assignment. The problem is that the transformation replaces the assignment $s_0 := obj.next$ with $s_0 := v_n$. Such assignment is crucial for proving that the path-length of v_s decreases at each call to next. If, instead, we transform this rule w.r.t. the field state only:

$$next(\langle this, v_s\rangle, \langle v_s, r\rangle) \quad \leftarrow \quad obj := v_s, s_0 := obj.next, v_s := s_0, r := obj.$$

the path-length abstraction approximates now the effect of the instructions by $\{obj = v_s, s_0 < obj, v'_s = s_0, r = obj\}$ which implies $v'_s < v_s$. The important point is that, in the second constraint, when accessing a field of an acyclic data structure, the corresponding path-length decreases. This enables proving termination of loops that use next *(like in* m*) by relying only on the field-insensitive version of path-length (note that [SMP10] is not field-sensitive).*

In summary, our approach can be used to prove termination of programs that use common patterns in OO languages such as *iterators* and *enumerators* by using a heuristics that tries to transform only those reference heap accesses which are

Input: A scope S
Output: Local termination preconditions
1: **proc** infer_local_term_preconditions
2: Choose the set of fields of interest $\mathcal{F}_S = \{f_1, \ldots, f_k\}$
3: Generate all possible locality partitions $\mathcal{P}^*$ for $\mathcal{F}_S$
4: $\varphi = \mathit{false}$
5: **for each** $\langle \mathcal{P}^{\varphi_1}_{f_1}, \ldots, \mathcal{P}^{\varphi_k}_{f_k} \rangle \in \mathcal{P}^*$ **do**
6: **if** `Termin` *returns* true *on* $\mathcal{T}(\mathcal{P}^{\varphi_1}_{f_1}, \ldots, \mathcal{P}^{\varphi_k}_{f_k})$ **then**
7: $\varphi = \varphi \vee (\varphi_1 \wedge \cdots \wedge \varphi_k)$
8: **return** φ

Algorithm 1: Inference of local termination preconditions for a scope S

used as cursors to the data structures. This is achieved by requiring that the field (or array) is both read and written in the scope.

3.4 Inference of Termination Preconditions

In this section, we describe our approach for inferring aliasing preconditions that, when hold in the initial state, guarantee the termination of the program under consideration. Our approach is defined in two stages: (i) we first find preconditions that guarantee *local termination*, i.e., they guarantee that the loops defined in a given scope S are terminating, ignoring the termination behavior of loops defined in scopes that are called from S; and (ii) in a second step, we use the local preconditions in order to obtain conditions on *global termination* which guarantee that the loops of S as well as those of scopes that are transitively called from S are terminating. Note that if S does not call any other scope, then local and global termination are equivalent for S.

3.4.1 Inference of Local Termination Preconditions

Given a scope S, the purpose of this section is to describe how to infer an aliasing precondition φ which guarantees the local termination of S. Algorithm 1 outlines the steps to generate φ. At line 2, we choose the set of fields of interest $\mathcal{F}_S$ from all fields that are accessed in S^*. Note that any subset chosen leads to

a safe transformed program since the lack of some fields only implies computing less precise information. For instance, for the sake of efficiency, one can select only the fields that might affect the termination behaviour of the program (see, e.g., the approximation of [AAG+08]).

Line 3 computes all possible partitions of the read and write sets for the elements $f \in \mathcal{F}_S$, denoted $\mathcal{P}^*$. For each field f, each corresponding partition $\langle G_1, \ldots, G_n \rangle$ is created by adding aliasing propositions that state the following: (1) Any $\ell_1, \ell_2 \in G_i$ are equal; and (2) Any $\ell_1 \in G_i$ and $\ell_2 \in G_j$ are different when $i \neq j$. Stating that two access paths ℓ_1 and ℓ_2 are equal (resp. different) is done by writing $\ell_1 \approx \ell_2$ (resp. $\ell_1 \not\approx \ell_2$). However, when ℓ_1 and ℓ_2 have a similar structure, this can be done by comparing their corresponding components. For example, if $\ell_1 = l_1[l_2]$ and $\ell_2 = l_1[l_3]$, we can use $l_2 \approx l_3$ (resp. $l_2 \not\approx l_3$), and if $\ell_1 = l_1[l_2]$ and $\ell_3 = l_3[l_4]$, we could use $l_1 \approx l_3 \wedge l_2 \approx l_4$ (resp. $l_1 \not\approx l_3 \vee l_2 \not\approx l_4$).

In line 6, we assume the existence of a heap-insensitive termination analysis procedure `Termin` that is able to answer the question: *does S locally terminate for any input?*. Note that local termination does not prove termination of loops in scopes invoked from S; however, it needs to transform them in order to track the modifications that invoked scopes might perform on the sizes of data. The answer of `Termin`, as expected, is not definitive, it might be *yes* or *don't-know*. For each locality partition of the involved fields (line 5), we run the local termination analyzer on the program resulting from applying the locality transformation in Definition 3.3.10 w.r.t. such locality partition. If `Termin` returns *true*, according to Theorem 3.3.12, S locally terminates when the precondition holds in the input state. In line 7, we perform the disjunction of the current result with the preconditions obtained from the previous partitions such that the final result is the disjunction of all preconditions for which the program terminates.

Example 3.4.1. *Let us apply Algorithm 1 on the scope S_2 (inner loop) of our running example. Step at line 2 gives us the type* `aint`*. At line 3, the two partitions of Example 3.3.9 are generated. Thus, the* foreach *loop performs two iterations. When considering the locality partition $\mathcal{P}^{\psi_2}_{\texttt{aint}}$ where the precondition is $\psi_2 = \{l_1 \not\approx l_2 \vee l_3 \not\approx l_4\}$, the transformed program of Figure 3.5 (right) is constructed and* `Termin` *returns* true *in line 6. Hence, φ (initialized to* false*) takes now the*

value ψ_2. In the next iteration, the locality partition $\mathcal{P}^{\psi_1}_{\texttt{aint}}$ is considered and the transformed program of Figure 3.5 (left) is constructed. In this case, `Termin` *returns* don't-know *and hence φ remains with the value ψ_2 assigned in the previous iteration, which is returned as result in line 8 of Algorithm 1.*

3.4.2 Inference of Global Termination Preconditions

Let us explain intuitively how Algorithm 2 infers global termination preconditions. Consider a scope S that includes a call $b^k_j = p(\bar{x}, \bar{y}) \in S$ to a procedure p that is defined in a different scope S'. Moreover, assume that S' does not call procedures that are defined in other scopes. In a first step, we have inferred local termination preconditions φ_1 and φ_2 for S and S', respectively, by means of Algorithm 1. In this step, we will combine φ_1 and φ_2 into global termination preconditions ψ_1 and ψ_2, respectively. For S', clearly we can take $\psi_2 = \varphi_2$, since it does not call any other scope. For S, we seek a precondition ψ_1 such that $\psi_1 \models \varphi_1$; and ψ_2 holds whenever the execution reaches the call to p. We take $\psi_1 = \varphi_1 \wedge \psi'_2$, where ψ'_2 is obtained from ψ_2 by replacing each l_i by $\phi_{k:j}(x_i)$. Intuitively, ψ'_2 is the global termination precondition of p, but expressed in terms of the input to the entry of S. This process is applied for all scopes in reverse topological order, i.e., starting from the one which does not call any other scope, until the one that includes the main entry procedure.

Input: Scopes $S_1, \ldots, S_n$
Output: Local and Global termination preconditions
1: **proc** infer_global_termination_preconditions
2: **for each** $i \leftarrow 1 \ldots n$ **do**
3: $\texttt{LC}[i] = \texttt{LocalCondTermin}(S_i)$
4: $\varphi \leftarrow \texttt{LC}[i]$
5: **for each** external call $b^k_j \equiv p(\bar{x}, \bar{y}) \in S_i$ where $p \in S_h$ **do**
6: $\varphi \leftarrow \varphi \wedge \psi[l_1/\phi_{k:j}(x_1), \ldots, l_m/\phi_{k:j}(x_m)]$ where $\psi = \texttt{GC}[h]$
7: $\texttt{GC}[i] \leftarrow \varphi$

Algorithm 2: Computing termination preconditions

Algorithm 2 outlines the main steps to generate global termination preconditions for all scopes. The outer loop iterates over the scopes in reverse topological

order and computes a global termination precondition for each scope S_i. At line 3, we compute a local termination condition $\texttt{LocalCondTermin}(S_i)$ for S_i using Algorithm 1. This condition is also used at line 4 as its initial global termination precondition. Then, the inner loop traverses all calls to external scopes, for each such call, at line 6, it translates the global termination condition of the corresponding S_h to be in terms of the input of S_i, and adds it to the global precondition of S_i. When the algorithm terminates, $\texttt{LC}[i]$ and $\texttt{GC}[i]$ will be, respectively, the local and global termination precondition for the scope S_i.

Example 3.4.2. *Consider the analysis of the scope S corresponding to method* m *in Figure 3.3. Inferring global termination preconditions for* m *requires the analysis of all scopes invoked in the method. Let us focus on S_1 (method next), S_2 (the inner loop) and S_3 (the outer loop). The application of Algorithm 1 on S_1 computes the precondition true (see Example 3.3.13). Algorithm 1 on S_2 computes the precondition $\varphi = \{l_1 \not\approx l_2 \vee l_3 \not\approx l_4\}$ (see Example 3.4.1). For scope S_3, Algorithm 2 is able to prove termination (ignoring the inner loop) unconditionally by using a unique ghost variable for the field* state *and considering the size relations inferred in Example 3.3.13. Once all local conditions for S_1, S_2 and S_3 have been computed, Algorithm 2 proceeds as follows: $\texttt{LC}[1]$ and $\texttt{LC}[2]$ are initialized to values true and φ, respectively. As there are no external calls in S_1 nor in S_2, the* foreach *loop at line 5 is not executed for any of the scopes, and $\texttt{GC}[1] = true$ and $\texttt{GC}[2] = \varphi$ (line 7). In the iteration on S_3, $\texttt{GC}[3]$ is initialized to $\varphi' = true$. The interesting point is in the call to the inner loop, for which it is required a renaming according to the access path information ϕ stored at the program point just before the inner loop. According to Examples 3.2.4 and 3.3.13, we have $\{i \mapsto l_3{\cdot}state{\cdot}data, j \mapsto l_3{\cdot}state{\cdot}data\} \in \phi$. Line 6 of Algorithm 2 computes $\varphi' = \varphi[l_1/l_1, l_2/l_2, l_3/l_3{\cdot}state{\cdot}data, l_4/l_3{\cdot}state{\cdot}data]$, which results in $\varphi' = \{l_1 \approx l_2 \vee l_3{\cdot}state{\cdot}data \approx l_3{\cdot}state{\cdot}data\} = \{l_1 \approx l_2\}$ as global precondition for S_3, and thus for method* m.

3.5 Experimental Evaluation

In this section, we present an experimental evaluation of our implementation on the set of micro-benchmarks shown in Table 3.1. The main focus of the examples is on the combined use of arrays with numeric and reference fields, and on requiring interesting aliasing conditions for proving their termination.

Table 3.1 aims at showing different types of preconditions that can be obtained by our system for the programs shown on the leftmost column. Column **Preconditions** displays the aliasing preconditions found by COSTA to guarantee termination of each method. Variables are named using this convention: integers are represented by variables i,j,k,l, arrays by variables a,b,c, and x,y,z,w represent reference variables. Identifiers that start by f are declared as fields. Observe that the preconditions obtained show different types of aliasing conditions that can be inferred for guaranteeing the termination of the methods. Benchmarks m2, m7, m8 and m9 show preconditions that involve aliasing between objects, while m3, m4, m5 and m6 require aliasing conditions for arrays and their indexes. Method m10 needs preconditions that combine arrays and object references aliasing. Note also that the experiments show that we can obtain both disjunctive and conjunctive conditions. For instance, benchmark m3 contains a disjunction of different conjunctive components, that is obtained from a condition $fa[i] \neq fb[j]$ encoded as $fa \neq fb$ or $i \neq j$. One interesting method is m8 because it receives four objects as inputs parameters and all of them modify the same integer field fi. The loop will terminate in four different cases (marked in the table): ① if $x \approx y$, the loop condition never holds; ② if $y \approx w$ and the rest of the objects do not alias, in that case y.fi decreases while x.fi does not change; ③ if $x \approx z$ and the rest of the objects do not alias, it is like the previous one because x.fi is incremented by z.fi; and, ④ where $x \approx z$ and $y \approx w$ but $x \not\approx y$, because x.fi is incremented by z.fi and y.fi is decremented by w.fi.

None of these examples can be handled by the heap-sensitive analyses in [AAGP09, AAG+10, RD11] for two reasons: (1) such analyses were designed only for fields and could not track array contents and (2) they could only handle unconditional locality, while all our examples rely on aliasing preconditions.

Table 3.2 aims at showing the overhead introduced by the heap sensitive

Program	**Preconditions**

void m2 (A x) {
  while(x.fi > 0) fi−−;
}
	$this \approx x$

void m3(int i, int j, int k) {
  while (fa[i] > fb[j]) fc[k]++;
}
	$((fb \approx fc \wedge j \approx k) \wedge (fa \not\approx fb \vee i \not\approx j) \wedge (fa \not\approx fc \vee i \not\approx k)) \vee$
	$(fa \approx fb \approx fc \wedge i \approx j \approx k) \vee$
	$((fa \approx fb \wedge i \approx j) \wedge (fa \not\approx fc \vee i \not\approx k) \wedge (fb \not\approx fc \vee j \not\approx k))$

void m4(int i, int j)
  while (fa[i] > 0) {
  fb[i]−−; fa[j]++;
  }
}
	$i \not\approx j \wedge fa \approx fb$

void m5(int[] a,int i,int j,int k) {
  while(a[i] > 0) {
  a[j]−−; a[k]−−;
  }
}
	$(i \approx j \wedge i \not\approx k) \vee (i \approx j \approx k) \vee (i \approx k \wedge i \not\approx j)$

void m6(int[] a,int i,int j,int k,int l){
  while(a[l] > 0) {
  while(a[i] < 0) {
  a[j]++; a[k]++;
  }
  while(a[i] < 0) a[j]++;
  a[l]−−;
  }
}
	$(i \approx j \wedge i \not\approx k \wedge i \not\approx l \wedge k \not\approx l) \vee$
	$(i \approx j \wedge j \approx k \wedge i \not\approx l)$

void m7(A x, A y, A z){
  while(x.fi < y.fi) z.fi++;
}
	$(x \approx y \wedge x \not\approx z) \vee$
	$(x \approx y \approx z) \vee$
	$(x \approx z \wedge x \not\approx y)$

void m8(A x, A y, A z, A w) {
  while (x.fi < y.fi) {
  z.fi++; w.fi−−;
  }
}
	①$(x \approx y \wedge x \not\approx z \wedge x \not\approx w \wedge z \not\approx w) \vee$
	①$(x \approx y \wedge z \approx w \wedge x \not\approx z \wedge x \not\approx w) \vee$
	①$(x \approx y \approx z \wedge x \not\approx w) \vee$
	①$(x \approx y \approx z \approx w) \vee$
	①$(x \approx y \approx w \wedge x \not\approx z) \vee$
	②$(y \approx w \wedge x \not\approx y \wedge x \not\approx z \wedge x \not\approx w \wedge y \not\approx z) \vee$
	③$(x \approx z \wedge x \not\approx y \wedge x \not\approx w \wedge y \not\approx w) \vee$
	④$(x \approx z \wedge y \approx w \wedge x \not\approx y \wedge x \not\approx w)$

void m9(A x, A y, A z) { m8(x,y,x,z);}	$(y \approx z \wedge x \not\approx y) \vee (x \approx y \wedge x \not\approx z) \vee (x \approx y \approx z) \vee$ $(x \not\approx y \wedge y \not\approx z \wedge x \not\approx z) \vee (y \approx z \wedge x \not\approx y)$

void m10(A x, A y, int i) {
  while(x.fa[fi] > 0) y.fb[i]−−;
}
	$(x.fa \approx y.fb \wedge this.fi \approx i)$

Table 3.1: Some examples of constraints obtained by using the heap-sensitive extension

	Unconditional			*Conditional Analysis*					
Method	T_N^U	T_H^U	O_H	$\#_P$	$\#_T$	T_G	T_E	T_T	$\mathsf{T}_{T/P}$
Running	121	210	1.73	4	3	10	670	720	180
m2	7	10	1.42	2	1	0	60	70	35
m3	20	40	2.00	5	3	0	340	360	72
m4	40	30	0.75	5	1	10	240	260	52
m5	40	40	1,99	5	3	0	240	270	54
m6	70	90	1.28	22	11	60	3000	3100	140
m7	20	30	1.50	5	3	0	150	170	34
m8	30	60	2.00	15	8	140	590	650	43
m9	70	80	1.40	15	8	60	640	720	48
m10	30	30	1.00	2	1	0	160	180	90
Average	44.8	62	1.40	8	4.2	28	609	650	74.8

Table 3.2: Statistics about the Conditional Heap-Sensitive Analysis (times in ms)

analysis w.r.t. the heap insensitive one, and w.r.t. the unconditional version of our heap-sensitive analysis, that can only handles unconditional locality. The experiments have been performed on an Intel(R) Core(TM)2 Duo CPU 2.53GHz with 4GB of RAM running Linux 3.2.0. Columns under *Unconditional* show the total time taken by the two unconditional analyses: T_N^U shows the time taken by the heap-insensitive analysis and column T_H^U the time (in milliseconds) taken by the unconditional heap-sensitive analysis (i.e., using our analysis by relying on Definition 3.3.4 instead of Definition 3.3.8 and applying the transformation using the precondition *true*). The average overhead introduced by the heap sensitive analysis is 1.40, which is reasonably small.

Columns under *Conditional Analysis* show information gathered from applying the conditional heap-sensitive analysis. Column $\#_P$ shows the number of partitions for the recursive scopes generated for all heap accesses in this scope (and transitive ones). This number corresponds also to the total number of versions for which COSTA tries to prove termination after having applied the transformation in Section 3.3.3. Besides, column $\#_T$ shows how many of such versions are proven to be terminating. In all cases that could not be proven terminating, we have checked that there might be actually non-terminating derivations. Thus, our analysis is as precise as possible for these examples. As regards times, column T_G shows the time taken by the reference constancy analysis, the generation of

the locality partitions and the transformed programs. This phase is quite efficient and, only in one case (m8), it takes more than 10% of the total analysis time. Column T_E shows the time taken for proving termination of all generated versions, and T_T shows the time taken by the whole analysis of the program (which includes some pre-processing steps).

Our experiments clearly show that overhead introduced by the conditional analysis is directly related to the number of partitions in each case. Thus, additionally, in column $\mathsf{T}_{T/P}$, we show the total time of the program divided by the number of partitions. The most relevant conclusion is that the analysis taken by one partition is similar to the unconditional heap-sensitive analysis, i.e., it does not increase significantly and it is less than twice the time of the heap-insensitive analysis, on average from 62 to 74.8.

3.6 Related Work

Traditionally, existing approaches to reason on shared mutable data structures either track all possible updates of heap-allocated data (endangering efficiency) or abstract all field updates into a single element (sacrificing accuracy). Our work does not fall into either category, as it does not track all heap-allocated updates but rather only those which behave like non heap-allocated variables. As our experiments show, our approach is sufficiently precise for scope-based reasoning while introducing a reasonable overhead, as required in important applications, such as termination and resource analysis.

Miné's [Min06] value analysis for C takes a different approach by enriching the abstract domain to make the analysis field-sensitive. The motivation here is different from ours, his analysis is developed to improve points-to analysis in the presence of pointer arithmetics. Similarly, [CL05] enriches a numeric abstract domain with alien expressions (field accesses). Without additional information, such as reference constancy analysis, this domain would be rather limited (imprecise) for bytecode.

We have applied our approach to the context of termination analysis. In this sense, our work continues and improves over the stream of work on termination

analysis of object-oriented bytecode programs [AAGP09, OBvEG09, AAC+08, SHP06, RS06]. For numeric data, termination analyzers rely on a *value* analysis which approximates the value of numeric variables (e.g. [CH78]). Some field-sensitive value analyses have been developed over the last years (see the work of Miné mentioned above [Min06]). For heap-allocated data structures, *path-length* [SHP06] is an abstract domain which provides a safe approximation of the length of the longest reference chain reachable from the variables of interest. This allows proving termination of loops that traverse acyclic data structures such as linked lists, trees, etc. However, the path-length abstract domain, and its corresponding abstract semantics, as defined in [SHP06] is field-insensitive in the sense that the elements of such domain describe path-length relations among local variables only and not among reference fields. Thus, analysis results do not provide explicit information about the path-length of reference heap-allocated data.

As regards to the reference constancy analysis, equivalent notions have been defined for other languages (see [AFKT03] and its references) and for different purposes. Our work adapts and extends such analyses to consider arrays and fields in a uniform way. Also, the analysis in this chapter generalizes our previous reference constancy analyses [AAGP09] which infers information only on class fields, to consider arrays, integer variables and arithmetic expressions. There is also a lot of work devoted to infer the shape of the heap (see, e.g., [SRW99], [Rey02]). In general, more accurate aliasing analysis [SRW99] can be used to improve the precision of our analysis when computing the read and write sets, but at a higher performance cost.

Techniques which rely on separation logic in order to track the depth (i.e., the path-length) of data-structures [BCDO06] would have the same limitation as path-length based techniques, if they are applied in a field-insensitive manner. For example, in the iterator example shown in Section 3.3.4, we have seen that given an object x of type ListIter, it is necessary an analysis which is able to model the path-length of field x.state and not that of x. Namely, we need a heap-sensitive analysis based on path-length, which is one of our contributions in this thesis. However, by applying these techniques which track the depth (i.e., the path-length) on our transformed programs, we expect them to infer the required

information without any modification to their analyses.

Finally, conditional termination has been considered before in [BIK12, CGLA+08]. In these works, the focus is in inferring termination conditions on the *numerical variables*, and not on the reference variables as we do.

Chapter 4

Incremental Resource Usage Analysis

This chapter presents our approach to *incremental resource usage analysis* that has been presented at *PEPM'12* [ACPRD12] and also as a poster paper in *APLAS'11* [ACPRD11].

4.1 Introduction

The starting point for the *Incremental Analysis* is the global cost analysis described in Chapter 2. Cost is inferred by a sequence of *global* analyses (or whole-program analyses) which must analyze the whole program in order to obtain sound and precise results. Despite the great progress made in static analysis, most global analyzers still read and analyze the entire program at once in a non-incremental way. In particular, all resource analyses to date are non-incremental (COSTA and other analyzers [GMC09, JH11]). During software development, programs are often modified, e.g., because a new implementation of an existing method is provided (which improves its efficiency or fixes its correctness) or because an existing code is extended with new functionality (typically by extending a class with further methods). In such cases, the existing analysis information for the program may no longer be correct and/or accurate. However, resource

analysis is a costly task and starting analysis from scratch is inefficient in most cases. A key challenge for static analysis techniques is achieving a satisfactory combination of precision and scalability. Making precise (and hence expensive) static analysis incremental is a step forward in this direction.

In this chapter, we present an incremental approach of resource usage analysis of an imperative and object-oriented programming language. The difficulty when devising an incremental analysis framework is to recompute the least possible information and do it in the most efficient way. In our setting, we achieve it by means of the following two steps wich are our main contributions:

- A *multi-domain* incremental analysis engine which can be used by all global pre-analyses required to infer the resource usage of a program (including the class analysis, sharing, cyclicity, constancy and size analysis as mentioned above). The engine is multi-domain in the sense that it interleaves the computation for the different domains and takes into account dependencies among them, in such a way that it is possible to invalidate only partial pre-computed information.

- Even a small change within a method (e.g., adding an instruction) can change the overall cost of the program. A fundamental idea to minimize the amount of information that needs to be recomputed is to be able to distinguish, within a *cost summary*, the cost subcomponent associated to each method, so that the final cost functions can be recomputed by replacing only the affected subcomponents.

Our incremental analysis has been implemented in the COSTA system. Experimental results are performed on selected benchmarks from the standardized *JOlden* benchmark suite [Sui] and from the *Apache Commons* Project [Pro]. Our results show that the proposed incremental analysis achieves a significant speedup with respect to the non-incremental approach. To the best of our knowledge, this is the first incremental resource usage analysis framework.

```
class C
 void main (List l) {
    int s = len(l);
    if (s % 2 ≠ 0 || s<2)
       return;
    mod(l);
 }
 void mod (List l) {
    Inc o = get(l);
    dup(o, o.incr(l));
 }
 Inc get (List l) {
    return new Inc();
 }
 void dup (Inc o, List l) {
    while (l != null) {
       l.data = l.data * 2;
       l = o.incr(l);
    }
 }
```

```
 int len(List l) {
    int i = 0;
    for(; l != null; l = l.next) {
       i++;
    }
    return i;
 }
}  //end of class C

class Inc {
 List incr (List l) {
    return l.next.next;
 }
}
class Inc2 extends Inc {
 List incr (List l) {
    return l.next;
 }
}
class List {
 List next;
 int data;
}
```

Figure 4.1: Incremental Algorithm Running Example

4.1.1 Organization of the Chapter

The chapter is organized as follows: Section 4.2 presents a typical event-based global analysis algorithm. Section 4.3 introduces our multi-domain incremental analysis algorithm which, given a change on a method, reconstructs the information which needs to be recomputed for a given set of domains.

Section 4.4 explains how incremental resource analysis handles the CRS generation. Section 4.5 presents the incremental UBs inference that recomputes the cost functions only of those subcomponents affected by the change. Section 4.6 presents the experimental results obtained by applying our approach over a set of realistic programs, and Section 4.7 concludes relating our apprach to previous work.

4.2 A Fixed-Point Analysis Engine

The analysis algorithms are developed on programs written in the *RBR* defined in Section 2.2.2. In this section, we present a global analysis fixed-point engine for the RBR which is parametric w.r.t. the analysis domain and that will later be extended to support incremental analysis.

Example 4.2.1 (RBR)**.** *The running example of this section, shown in Figure 4.1, is a simple example to show the interaction among the different domains and the reanalysis required by the incremental extensions. Method* main *receives a list of integers, checks the length of the list and modifies some of its elements by invoking* mod*. Some methods of the example are transformed into the following RBR:*

$$
\begin{array}{rcl}
mod(\langle l\rangle,\langle\rangle) & \leftarrow & get(\langle l\rangle,\langle o\rangle),\\
 & & call\_incr(\langle o,l\rangle,\langle l'\rangle),\\
 & & dup(\langle o,l'\rangle,\langle\rangle)\\
get(\langle l\rangle,\langle r\rangle) & \leftarrow & r := new\ \mathsf{Inc}\\
dup(\langle o,l\rangle,\langle\rangle) & \leftarrow & while(\langle o,l\rangle,\langle l'\rangle)\\
while(\langle o,l\rangle,\langle l\rangle) & \leftarrow & l = \mathit{null}\\
while(\langle o,l\rangle,\langle l'''\rangle) & \leftarrow & l \neq \mathit{null},\\
 & & l'.data := l.data * 2,\\
 & & call\_incr(\langle o,l'\rangle,\langle l''\rangle),\\
 & & while(\langle o,l''\rangle,\langle l'''\rangle)\\
(*)call\_incr(\langle o,l\rangle,\langle l'\rangle) & \leftarrow & \mathbf{type}(o,\mathsf{Inc}),\\
 & & Inc.incr(\langle l\rangle,\langle l'\rangle)\\
Inc.incr(\langle l\rangle,\langle l''\rangle) & \leftarrow & l' := l.next,\\
 & & l'' := l'.next
\end{array}
$$

The RBR of the program is built during the execution of class analysis. Virtual invocations are statically resolved and simulated by means of dispatch rules (e.g., call_incr). Class analysis for this program determines that the object o *returned by method* get *is an instance of* Inc*. Thus, the dispatch rule defined by procedure call_incr (*) does not include a call to* Inc2.incr*.*

4.2.1 A Global Fixed-Point Analysis Engine

Algorithm 3 presents an event-based global fixed-point analysis engine for the RBR, similar to other worklist algorithms [NNH99]. The algorithm is parametric w.r.t. the abstract domain D that describes some property of interest. As usual, the domain D is defined for all possible descriptions, which form a complete lattice for which all ascending chains are finite. We assume that the operations of *least upper bound* (denoted $\sqcup$), *greatest lower bound* (denoted $\sqcap$) and $\sqsubseteq$ are defined on the particular abstract domain. Function alpha returns the abstraction of an instruction in D. Function $\mathsf{restrict}(ST, V, D)$ projects an abstraction ST onto the variables in the set V w.r.t. domain D, and $\mathsf{extend}(ST, V, D)$ extends the abstraction ST to the variables in V w.r.t. domain D. The analysis results for a procedure are computed with respect to a specific calling pattern CP, which is a description in the abstract domain of the values that the input arguments can take. The analysis is *monovariant*, i.e., the goal of the analysis is to compute for each procedure p in the program at most one *answer* of the form $CP \mapsto AP$, where AP is the *answer pattern*, which is also a description in the abstract domain of the values that the output arguments can take, and the *call pattern* CP is general enough to cover all possible patterns for p that appear during the analysis of the program.

The algorithm uses two global data structures: (1) the *local answer table* $\mathcal{L}^D$ for domain D, where the answers for all procedures are stored, and (2) the *queue of events* $\mathcal{Q}$, which initially contains as single element the pair (m, CP) with the entry procedure m and a corresponding call pattern CP. The analysis of a method is carried out in *process_analysis*, where we analyze all rules defining a procedure in Line 27 (L27 for short) and traverse the instructions in its body from left to right (L29). When the instruction is not a procedure call, we abstract it according to the abstract domain (L35). As usual, the abstract description obtained from one instruction is conjoined with the previously computed one (L36). The analysis results obtained from the different rules which define a procedure are joined together (L37). When the instruction is a procedure call (L30), *get_proc_answer* first checks if a previously computed answer exists in $\mathcal{L}$ (L11). We assume that *get* returns the answer for a given call properly renamed w.r.t. the arguments in

```
1: proc analysis(m,CP,D)
2:    Q = ∅
3:    L^D=∅
4:    Q.add(m,CP)
5:    while (¬Q.empty()) do
6:       (p,CP)=Q.extract_first()
7:       process_analysis(p,CP,D)

8: function get_proc_answer(p,CP,D)
9:    CP' = CP
10:   AP' = ⊥
11:   if (L^D.exists(p)) then
12:      (CP^L ↦ AP^L)= L^D.get(p)
13:      if (CP ⊑ CP^L) then
14:         return AP^L
15:      CP' = CP ⊔ CP^L
16:      AP' = AP^L
17:   Q.add(p,CP')
18:   L^D.update(p,CP' ↦ AP')
19:   return AP'

20: proc invalidate_callers(p)
21:   for all (p' in callers(p)) do
22:      if (L^D.exists(p')) then
23:         (CP' ↦ AP')=L^D.get(p')
24:         Q.add(p',CP')

25: proc process_analysis(p,CP,D)
26:   AP=⊥
27:   for all (R_i : p ← b_i1, ..., b_in) do
28:      ST=extend(CP,vars(R_i),D)
29:      for each (b_ij, 1 ≤ j ≤ n) do
30:         if (b_ij = q(_,_)) then
31:            CP'=restrict(ST,vars(b_ij),D)
32:            ST'=get_proc_answer(b_ij,CP',D)
33:            ST'=extend(ST',vars(R_i),D)
34:         else
35:            ST'=extend(alpha(b_ij,D),vars(R_i),D)
36:         ST= ST ⊓ ST'
37:      AP = AP ⊔ ST
38:   (CP^L ↦ AP^L)=L^D.get(p)
39:   if (AP ⋢ AP^L) then
40:      invalidate_callers(p)
41:      L^D.update(p,CP ↦ AP)
```

Algorithm 3: Fixed-point algorithm (operators $\sqcup$, $\sqsubseteq$, $\sqcap$ are parametric w.r.t. the analysis domain, D)

the call (L12). If the existing calling pattern is general enough (L13), we just use the previous answer (L14). Otherwise, since the algorithm is monovariant, we join the calling patterns (L15), reanalyze the corresponding method (L17). At the end of *process_analysis*, if the answer for p has changed (L39), we need to invalidate the information for all rules that invoke p (calling *invalidate_callers* in L40), and update the local answer table $\mathcal{L}$ (L41). *invalidate_callers* adds to $\mathcal{Q}$ those methods that invoke p, denoted *callers(p)* (L21), and have an entry in $\mathcal{L}$ (L22).

Observe that the *analysis* engine adds entries to $\mathcal{Q}$ during its execution when (i) a rule must be analyzed for a given calling pattern in L17, either because there

was not an answer for it or because it had been analyzed for a less general calling pattern and (ii) when the answer of a rule invoked from it changes (L24). The execution finishes when there are not more events to process in $\mathcal{Q}$ (L5).

Example 4.2.2 (alg. 3)**.** *The following table shows some relevant states of execution Algorithm 3 when analyzing method* main *of the running example w.r.t. the calling pattern* $\rho \equiv \{l : \{\mathsf{List}\}\}$ *for the domain "*class*". In the* class *domain, an abstract value represents the set of classes that the corresponding variable can be typed to. Thus, the CP indicates that the type of the input list* l *is* List.

(1)	$\mathcal{Q}$: *(*main,{*l*:{List}}*)* $\mathcal{L}$: *(*main,{*l*:{List}} $\mapsto \bot$*)*
(2)	$\mathcal{Q}$: *(*len,{*l*:{List}}*),(*mod,{*l*:{List}}*)* $\mathcal{L}$: *(*main,{*l*:{List}} $\mapsto \bot$*), (*len,{*l*:{List}} $\mapsto \bot$*), (*mod,{*l*:{List}} $\mapsto \bot$*)*
(3)	$\mathcal{Q}$: *(*mod,{*l*:{List}}*)* $\mathcal{L}$: *(*main,{*l*:{List}} $\mapsto \bot$*), (*len,{*l*:{List}} $\mapsto \bot$*), (*mod,{*l*:{List}} $\mapsto \bot$*),* *(*get,{*l*:{List}} $\mapsto \bot$*), (*dup,{*o*:$\bot$, *l*:{List}} $\mapsto \bot$*), (*)*
(4)	$\mathcal{Q}$: *(*get,{*l*:{List}}*), (*dup,{*o*:$\bot$,*l*:{List}}*),* $\mathcal{L}$: *(*main,{*l*:{List}} $\mapsto \bot$*), (*len,{*l*:{List}} $\mapsto \bot$*), (*mod,{*l*:{List}} $\mapsto \bot$*),* *(*get,{*l*:{List}} $\mapsto$ {*r*:{Inc}}*), (*dup,{*o*:$\bot$, *l*:{List}} $\mapsto \bot$*)*
(5)	$\mathcal{Q}$: *(*dup,{*o*:$\bot$,*l*:{List}}*), (*mod,{*l*:{List}}*)* $\mathcal{L}$: *(*main,{*l*:{List}} $\mapsto \bot$*), (*len,{*l*:{List}} $\mapsto \bot$*), (*mod,{*l*:{List}} $\mapsto \bot$*),* *(*get,{*l*:{List}} $\mapsto$ {*r*:{Inc}}*), (*dup,{*o*:$\bot$, *l*:{List}} $\mapsto \bot$*)*
(6)	$\mathcal{Q}$: *(*mod,{*l*:{List}}*), (*Inc.incr,{*l*:{List}}*),* dup,{*o*:{Inc.incr}, *l*:{List}} $\mathcal{L}$: *(*main,{*l*:{List}} $\mapsto \bot$*), (*len,{*l*:{List}} $\mapsto \bot$*), (*mod,{*l*:{List}} $\mapsto \bot$*),* *(*get,{*l*:{List}} $\mapsto$ {*r*:{Inc}}*), (*dup,{*o*:$\bot$, *l*:{List}} $\mapsto \bot$*),* *(*Inc.incr,{*l*:{List}} $\mapsto \bot$*)*
⋮	⋮

In this example, internal rules are ignored because they do not add any relevant information to the class *analysis. Relevant iterations proceed as follows: At iteration (1), method* main *is analyzed using the default CP,* {*l*:{List}}*. Method* main *calls* len *and* mod *which are added to* $\mathcal{Q}$ *and to* $\mathcal{L}$*, see L17 and L18 of Alg. 3, with the default AP,* $\bot$*. At (3), method* mod *is analyzed, adding to* $\mathcal{Q}$ *and* $\mathcal{L}$ *methods* get *and* dup*. The call to* incr *is not resolved at (4) and (5) because the callgraph is being built during class analysis and the type of o is not known yet.*

Iteration (4) corresponds to the analysis of method get *where greater AP for* get *is found. Thus, according to* invalidate_callers, mod *is added again to* $\mathcal{Q}$. *Iteration (6) analyzes* mod *and the dynamic dispatching of* incr *can be solved, thus the calls to* Inc.incr *and to* dup *are added to* $\mathcal{Q}$. *The algorithm iterates until a fixpoint is reached (which is shown later in Example 4.3.2).*

4.3 Incremental Inference of Cost Relations

The goal of this section is to support incremental analysis in all global pre-analyses required to infer the CRS which was explained in Section 2.3. First, Section 4.3.1 describes the notion of *method summary* which comprises the analysis information that has been computed globally in a non-incremental way in order to set up the CRS. Section 4.3.2 introduces a multi-domain incremental analysis which, given a change and the method summaries, is able to reconstruct the summaries for all domains.

4.3.1 Method Summary for Global Properties

All analysis information included within a method summary can be computed by using the generic fixed-point engine in Algorithm 3 for each domain. As it is described in Section 2.3.1, several analyses are executed consecutively and the information inferred for one domain is used for analyzing subsequent domains. Figure 4.2 shows the *dependency graph* between domains, where the order of execution and the dependencies between domains are shown. Given a domain D, we will refer to the set of domains reachable from D in the dependency graph (including D) as *dep(D)*.

Definition 4.3.1 (method summary)**.** *Given a method* $m(\bar{x}, \bar{y})$, *a* method summary *for* m *is a tuple of five answers* $CP_D \mapsto AP_D$ *for the following domains:*

(1) D=class, *where* x:$\{C_1, \ldots, C_n\}$$\in$$AP_{cl}$ *(respectively* CP_{cl}*) represents the set of classes that variable* x *may be typed to after (respectively before) executing* m *[SJ03] (see Section 2.2.1);*

(2) D=sharing, *where* $(x, y) \in AP_{sh}$ *(respectively* CP_{sh}*) means that* x *and* y *might share after (respectively before) executing* m *[SS05] (see Section 5.4.3);*

(3) D=acyclicity, *where* $x \in AP_{ac}$ *(respectively* CP_{ac}*) means that* x *may point to a cyclic data structure after (respectively before) executing* m *[RS06] (see Section 5.4.2);*

(4) D=constancy, *where* $x \in AP_{cn}$ *if the shape of* x *may have changed during the execution of* m *[GS08] (this analysis is context-insensitive, i.e. the AP does not depend on the CP);*

(5) D=size, *where* AP_{sz} *(respectively* CP_{sz}*) are a set of linear constraints describing the relation between the size of the variables* $\bar{x}$ *and* $\bar{y}$ *after (respectively before) executing* m *[AAGZ11] (see Section 2.3.4).*

Let us mention the most relevant issues related to the five points in the above definition and explain the domain dependencies in Figure 4.2. The class analysis information determines the overall code that must be analyzed in the next steps. This domain is finite since it only contains the classes available in the program at run-time. In size analysis, we assume that the size of a heap-allocated data structure is its path-length (i.e., the length of the longest path reachable from it), if the data structure is not *cyclic*. Hence, acyclicity is a soundness requirement for size analysis. Besides, if two variables x and y share and there is a reference field assignment $x.f = y$, then no safe information can be provided regarding the acyclicity of x nor y since cycles might be introduced. Hence, sharing is a soundness requirement for acyclicity, constancy and, hence, for size. It is essential to know the arguments of m whose shape remains *constant* upon return because in such case their path-length is preserved on exit from m [GS08]. (5) Once all previous analyses have been performed, size relations can be inferred in order to determine how the size of data is modified along the program's execution.

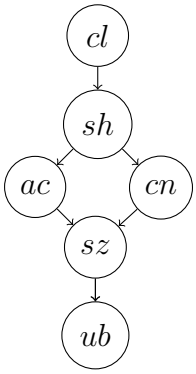


Figure 4.2: Domains dependencies

Example 4.3.2 (summaries)**.** *The following method summaries are obtained for the RBR in Example 4.2.1 by sequentially performing each of the global analyses above in a top down order (the results of sharing, acyclicity and constancy are not shown as they do not detect any sharing between data structures, cycles, nor changes in the data structures shapes, respectively). Note that* r *stands for the return value:*

Method	Summaries	
void mod(l)	*class*	$\{l{:}\{\mathsf{List}\}\} \mapsto \bot$
	size	$\{l \geq 2\} \mapsto \{l \geq 2\}$
Inc get(l)	*class*	$\{l{:}\{\mathsf{List}\}\} \mapsto \{r{:}\{\mathsf{Inc}\}\}$
	size	$\{l \geq 2\} \mapsto \{l \geq 2, r = 1\}$
void dup(o,l)	*class*	$\{o{:}\{\mathsf{Inc}\}, l{:}\{\mathsf{List}\}\} \mapsto \bot$
	size	$\{o = 1, l \geq 0\} \mapsto \{o = 1, l \geq 0\}$
List incr(l)	*class*	$\{l{:}\{\mathsf{List}\}\} \mapsto \{r{:}\{\mathsf{List}\}\}$
	size	$\{l \geq 2\} \mapsto \{l \geq r + 2, r \geq 0\}$

Observe that after creating the object in get*, the type of the object is instantiated to* Inc *and its size is 1 (this means that it is not null). Also, notice that, after analyzing* len*, the* ***size*** *analysis of* main *learns from the conditional statement that the size of the list is greater than or equal to two. This size constraint is used as precondition of the next methods. It is important to understand the size relation obtained for method* incr *which allows us to know that the size of the output list is the size of the input list decreased by two. This piece of information is essential to bound the cost.*

4.3.2 A Multi-Domain Incremental Fixed-Point Analyzer

When performing incremental analysis, after a modification in a program, a change in the analysis results associated to one domain must *invalidate* the results previously inferred by subsequent dependent domains. Algorithm 4 presents the extensions required to make Alg. 3 incremental by relying on the summaries in Definition 4.3.1 and the dependencies in Figure 4.2. We use the notation

noincr:m to refer a procedure m defined in Alg. 3 (see L10 and L30 in Alg. 4). The incremental algorithm uses method *noincr:analysis* of Alg. 3 (and its data structures) and two implementations of *get_proc_answer*, *noincr:get_proc_answer* and *incr:get_proc_answer*. In L32 of *noincr:process_analysis* of Alg. 3, we invoke *incr:get_proc_answer* instead of *noincr:get_proc_answer*.

In contrast to other approaches [HPMS00], the granularity of our analysis is set at the level of methods, i.e., we establish the method as the smallest piece of code whose analysis information will be stored and reanalyzed in case of changes. Procedure *incremental_fixpoint* receives the signature of the method m which has been changed. If there are multiple methods changed, changes will be handled one after another. Note that class analysis determines the CFG of the program being analyzed. Hence, after a change, the callers information used in L15 by Alg. 4 is recomputed. Observe that the objective of Alg. 4 is to recompute invalidated information, but not to improve the precision of previously computed results. Trying to improve the precision would require further recomputation in some cases.

Algorithm 4 uses three global data structures: (1) the *global answer table* $\mathcal{G}$, which contains the set of summaries for all previously analyzed methods; (2) the *queue of pending events* $\mathcal{P}$, which is formed by pairs of the form $(m, \mathcal{D})$ where $\mathcal{D}$ is a list of domains for which m must be reanalyzed in the order stated in Figure 4.2; and (3) a list, $\mathcal{R}$, to store all methods that have been reanalyzed in the current iteration of the while loop. The main goal of the incremental algorithm is, starting from the modified method, to propagate the new information obtained from the modification. When a method changes, its code must be reanalyzed with respect to all domains. This is done in the algorithm in L2-3 where we add to $\mathcal{P}$ the modified method for all domains and invalidate its entries in $\mathcal{G}$. The three main aspects of the algorithm are described below:

Multi-domain Our aim is to handle multiple domains in the context of incremental analysis such that the minimal amount of reanalysis is performed. Our approach consists in interleaving the computation of the incremental fixed point for all domains by means of validity flags. The idea is that a change in a summary for a specific domain invalidates only the entries for those dependent domains (in

```
1: proc incremental_fixpoint(m)
2:   P =[(m,D)]; C = ∅;
3:   G.invalidate(m,D)
4:   while (¬P.empty()) do
5:     R=∅
6:     (m', D_m') =P.extract_first()
7:     for each D in D_m' do
8:       R.add(m',D)
9:       (CP_m' ↦ AP_m')=G.get(m',D)
10:      noincr:analysis(m',CP_m',D)
11:    for all ((n,D) in R) do
12:      (CP_n^G ↦ AP_n^G)=G.get(n,D)
13:      (CP_n^L ↦ AP_n^L)=L^D.get(n,D)
14:      if (AP_n^L ⋤ AP_n^G)) then
15:        P.add_dom(callers(n),dep(D))
16:      G.update(n,CP_n^L↦AP_n^L⊔AP_n^G)
17:    P.remove(R)
18: function get_proc_answer(p,CP,D)
19:   CP'=CP
20:   if (is_method(p) ∧ G.exists(p,D))
      then
21:     (CP^G ↦ AP^G)=G.get(p,D)
22:     if (G.valid(p,D)) then
23:       if (CP ⊑ CP^G) then
24:         return AP^G
25:       else
26:         G.invalidate(p,dep(D))
27:         P.add_dom({p},dep(D))
28:   CP'=CP ⊔ CP^G
29:   R.add((p,D))
30:   AP=noincr:get_proc_answer(p,CP',D)
31:   return AP
```

Algorithm 4: Generic incremental fixed-point algorithm.

our case the dependencies in Fig. 4.2). This is handled in the algorithm by means of $\mathcal{G}$ such that each entry stored in $\mathcal{G}$ has a flag to indicate whether the entry for this particular domain is valid. Initially all entries are valid. We use function *valid(p,D)* to check if the summary for p and domain D is valid. The call *invalidate(p,$\mathcal{D}$)* sets up to invalid the flags for the set of domains $\mathcal{D}$ for method p. Invalidated entries must be reanalyzed. This occurs in the algorithm when the entries to be reanalyzed are added to $\mathcal{P}$ in L15 and L27. To that end, *add_dom* receives a set of methods $\mathcal{M}$ and a set of domains $\mathcal{D}$ and, for each $m \in \mathcal{M}$, $D \in \mathcal{D}$, if there exists an entry $(m, \mathcal{D}_m)$, it is updated to $(m, add(\mathcal{D}_m, D))$, where the addition of D is ordered as established in Fig. 4.2. If an entry does not exist, it adds $(m, \{D\})$.

Descendants L7-10 take care of reanalyzing those methods that are pending to be reanalyzed. We use the CP stored in $\mathcal{G}$ to initiate the analysis (L9-L10). Besides, all reanalyzed methods are added to $\mathcal{R}$ in L8 in order to later (L11) decide the recomputation that must be done (see ancestors paragraph). The call

noincr:analysis (L10) reanalyzes m for a particular domain by calling Alg. 3. During the execution of *noincr:analysis* (Alg. 3) those methods reachable from m (descendants) whose answer for such domain must be recomputed will also be reanalyzed. Observe that the execution of *noincr:analysis* uses *incr:get_proc_answer* instead of *noincr:get_proc_answer*. The new function *get_proc_answer* in Alg. 4 differs from the one in Alg. 3 in that, for method calls, it tries to reuse an existing answer from the method summary (stored in $\mathcal{G}$) (L20) if its calling pattern is general enough and the entry has not been invalidated (L22-23). Otherwise, both calling patterns are joined (L28) and the pair of method signature and domain is added in L29 to the list $\mathcal{R}$. Entries of the method summary for dependent domains are invalidated (L26) and added to $\mathcal{P}$ (L27). If there is no summary for the method, or if it is an intermediate procedure of a method (L20), the function *noincr:get_proc_answer* of Alg. 3 is invoked, and it analyzes m in the non-incremental way (L30). As we have seen in the non-incremental analysis, the analysis of one method may produce a new answer that must be propagated to its callers, by means of *noincr:invalidate_callers* (L40 of Alg. 3). Observe that, in the incremental case, $\mathcal{L}$ only contains information regarding descendants of m which have been reanalyzed. Therefore, *noincr:invalidate_callers* only invalidates methods which have been reanalyzed in the current call to *noincr:analysis* (L22 of Alg. 3). The execution of *noincr:analysis* finishes when no new information is propagated and a fixpoint for the reanalyzed methods is reached. This process is repetated for all domains for method the considered method (L7).

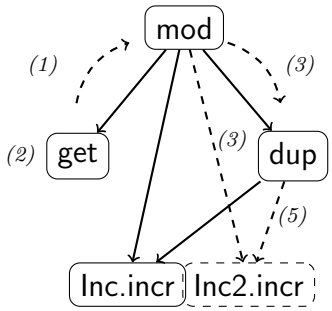
Ancestors When the call *noincr:analysis* finishes (L10), reanalyzed methods have been added to the list $\mathcal{R}$. Now, we need to take care of reanalyzing all those methods that relied on answers for methods in $\mathcal{R}$. The list $\mathcal{P}$ is used for this purpose. Initially, $\mathcal{P}$ contains an entry for the changed method and all domains (L2). It is later updated in L15 as follows. For each element in $\mathcal{R}$, we know that it is reanalyzed for such domain (L11). We compare the new answer $AP_n^{\mathcal{L}}$ with the one in the summary $AP_n^{\mathcal{G}}$ (L12-14). If the new one is not contained in the previous one (L14), we need to reanalyze all methods that invoke m (L15) for such domain and its dependent domains. However, some methods that invoke m may have been already reanalyzed in this iteration (during L7-10) and, therefore,

they do not need to be reanalyzed again. All methods in $\mathcal{R}$ are removed from $\mathcal{P}$ in L17. Finally, the fixed point is reached when there are no more methods to analyze in $\mathcal{P}$.

Example 4.3.3 (algorithm 4)**.** *Let us consider the following new implementation of method* get*:*

```
Inc get(List l) {
  if ((l.data % 2)==0)
    return new Inc();
  else
    return new Inc2();
}
```

As a result we have a new implementation of method get *and thus execute procedure* incremental_fixpoint*(*get*). The following iterations of Algorithm 4 are performed due to this change:*



Iter	$\mathcal{P}$
(1)	$\{(\mathsf{get}, cl), (\mathsf{get}, sz)\}$
(2)	$\{(\mathsf{get}, sz), (\mathsf{mod}, cl), (\mathsf{mod}, sz)\}$
(3)	$\{(\mathsf{mod}, cl), (\mathsf{mod}, sz)\}$
(4)	$\{(\mathsf{mod}, sz), (\mathsf{dup}, sz)\}$
(5)	$\{(\mathsf{dup}, sz)\}$

The above left figure graphically represents the iterations of the incremental analysis algorithm. Dashed lines (arrows and boxes) represent the recomputed information due to the application of the delta and the arrow are labeled with the step that computes it. The table to the right shows the contents of $\mathcal{P}$ at each iteration. Intuitively, the algorithm proceeds as follows:

(1) Procedure analysis(get,_,*class*) *of Alg. 3 is executed. The new answer pattern for* get *is* $\{r{:}\{\mathsf{Inc}, \mathsf{Inc2}\}\}$*. The class analysis info for* get *stored in $\mathcal{G}$ does not contain the newly generated answer pattern. Therefore, all direct ancestors of* get *must be reanalyzed, namely* mod *is added to $\mathcal{P}$ for*

dep(class). *This is shown graphically by a dashed arrow from* get *to* mod *labeled with (1).*

(2) The size *analysis for* get *does not propagate new information its callers.*

(3) analysis(mod,_,*class*) *is launched. During class analysis, calls to* dup *and* incr *are found. According to the new results obtained for* get*, variable* o *can now be of types* {Inc, Inc2}*. Due to this polymorphism, an invocation to* Inc2.incr* *is found in addition to the previous* Inc.incr*. Since there are no entries for* Inc2.incr *in* $\mathcal{G}$*, it is analyzed during the execution of* analysis *and added to* $\mathcal{G}$ *for the domain* class*. Besides, the entry for* dup *in* $\mathcal{G}$ *has* $CP_1 \equiv \{o{:}\{\mathsf{Inc}\}, l{:}\{\mathsf{List}\}\}$ *and thus it does not cover the new* $CP_2 \equiv \{o{:}\{\mathsf{Inc}, \mathsf{Inc2}\}, l{:}\{\mathsf{List}\}\}$*. Both* ρ*s are joined (resulting in* CP_2*) and* dup *is reanalyzed for the class domain w.r.t.* CP_2*. During the class analysis of* dup *a new call to* Inc2.incr *is found, but now* $\mathcal{G}$ *contains valid information for* Inc2.incr*, and thus the AP stored in* $\mathcal{G}$ *can be directly used. As* dup *has been reanalyzed for domain* class*, entries for* dup *for* dep(class) *in* $\mathcal{G}$ *are invalidated and added to* $\mathcal{P}$*. Note that (*dup,cl*) is not added to* $\mathcal{P}$ *because it has been handled as a descendant of* mod*.*

(4) Similarly to (3), during the analysis of mod *for* size *domain, the entry for (*Inc2.incr,*size) is added to* $\mathcal{G}$*.*

(5) Polymorphism of the call to incr *forces that the size relations for procedure* while *change. Now, we combine the size results of* Inc.incr *and* Inc2.incr*, since any of the two methods can be executed within the loop.*

At the end of (5) a fixed-point is reached since there are no further changes in the answer pattern for any domain. Importantly, only affected methods have been reanalyzed and their information in $\mathcal{G}$ *is up-to-date. Methods* main*,* len *and* Inc.incr *have not required reanalysis for any domain. The following table shows these summaries that have changed w.r.t. the ones in Ex. 4.3.2:*

*Method references include the class they belong to when disambiguation is needed.

Method	Summaries	
void mod(l)	cl	$\{l$:{List}$\} \mapsto \bot$
	sz	$\{l \geq 2\} \mapsto \{l \geq 2\}$
Inc get(l)	cl	$\{l$:{List}$\} \mapsto \{r$:{Inc, ***Inc2***}$\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq 2, r = 1\}$
void dup(o,l)	cl	$\{o$:{Inc, ***Inc2***}$, l$:{List}$\} \mapsto \bot$
	sz	$\{o = 1, l \geq 0\} \mapsto \{o = 1, l \geq 0\}$
List Inc.incr(l)	cl	$\{l$:{List}$\} \mapsto \{r$:{List}$\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq r + 2, r \geq 0\}$
List Inc2.incr(l)	***cl***	$\{l$:{List}$\} \mapsto \{r$:{List}$\}$
	sz	$\{l \geq 2\} \mapsto \{l \geq r + 1, r \geq 0\}$

Theorem 4.3.4 (termination of Alg 4)**.** *Given a program P, its analysis results stored in $\mathcal{G}$, and a method m which has been modified,* incremental_fixpoint(m) *terminates.*

Proof. For proving termination of the incremental algorithm (Alg. 4), we define the pair (Λ, L) where:

- Λ is a measure of the distance to $\top$ of the elements in $\mathcal{L}$, defined as

$$\Lambda = \sum_{CP_i \mapsto AP_i \in \mathcal{G}} (\delta(CP_i) + \delta(AP_i)) \tag{4.1}$$

  where $\delta(ST)$ is the number of abstract states from the abstract substitution ST to $\top$. The ascending chain condition (ACC) of the abstract domains guarantees that this distance is finite from any state (using widening when needed).

- L is the length of $\mathcal{P}$.

Let $>^{\Lambda,L}$ be the lexicographical ordering induced by $>_{\mathbb{N}}$ over $\mathbb{N}^2$:

$$(\Lambda_1, L_1) >^{\Lambda,L} (\Lambda_2, L_2) \iff (\Lambda_1 > \Lambda_2) \vee (\Lambda_1 = \Lambda_2 \wedge L_1 > L_2)$$

Since $>^{\Lambda,L}$ is a well-founded ordering, termination of Algorithm 4 can be concluded if we show that for each iteration i of Algorithm 4, $(\Lambda_{i-1}, L_{i-1}) >^{\Lambda,L} (\Lambda_i, L_i)$.

We will assume that if in $\mathcal{G}$ there is no entry for method p, it contains $p : \bot \mapsto \bot$ for every rule p that is required by the analysis. At each iteration i of the *while* loop in L4-17 of Alg. 4, Λ and L may change in the following cases:

- $\mathcal{G}$ is updated in *incremental_fixpoint* in L16. Regarding the calling pattern information, $CP_n^{\mathcal{L}}$ stored in $\mathcal{G}$ in L16 comes from the information contained in $\mathcal{L}$ (L13). We will show that $\delta(CP_n^{\mathcal{L}}) \leq \delta(CP_n^{\mathcal{G}})$ for any $(n, D) \in \mathcal{R}$.

  The analyses of pending methods in $\mathcal{P}$ are launched in the loop at L7-10. These analyses use as calling patterns the information already existing in $\mathcal{G}$ (L9). Therefore, *noincr:process_analysis* will analyze the pending methods using the calling pattern in $\mathcal{G}$.

  During the execution of *noincr:process_analysis* there may be calls to other methods which are handled by *incr:get_proc_answer* (L32 of Alg. 3). In *incr:get_proc_answer* there are three possibilities:

  - If there is a valid entry in $\mathcal{G}$ for an invoked method m which is applicable (L23), the answer pattern in $\mathcal{G}$ is used (L24), without updating $\mathcal{L}$.
  - If it is not applicable, the calling pattern for m is lubbed with the existing calling pattern in $\mathcal{G}$ (L28), and then *noincr:get_proc_answer* updates $\mathcal{L}$ with this lubbed calling pattern (L18, Alg. 3).
  - If there is no entry in $\mathcal{G}$, the calling pattern for m is stored in $\mathcal{L}$, (L18, Alg. 3).

  In all cases, $\delta(CP_n^{\mathcal{L}}) \leq \delta(CP_n^{\mathcal{G}})$, since $CP_n^{\mathcal{L}} \sqsupseteq CP_n^{\mathcal{G}}$ in any case.

- Regarding answer patterns, $AP_n^{\mathcal{L}} \sqcup AP_n^{\mathcal{G}}$ stored in $\mathcal{G}$ in L16 trivially verifies $AP_n^{\mathcal{L}} \sqcup AP_n^{\mathcal{G}} \sqsupseteq AP_n^{\mathcal{G}}$, and thus $\delta(AP_n^{\mathcal{L}} \sqcup AP_n^{\mathcal{G}}) \leq \delta(AP_n^{\mathcal{G}})$.

In all cases, $\Lambda_{i-1} \geq \Lambda_i$. Let $\mathcal{G}_i$ be the state of $\mathcal{G}$ after iteration i. There are two cases:

- If $\Lambda_{i-1} > \Lambda_i$, then $(\Lambda_{i-1}, L_{i-1}) >^{\Lambda,L} (\Lambda_i, L_i)$ holds.
- If $\Lambda_{i-1} = \Lambda_i$, then all entries $CP \mapsto AP \in \mathcal{G}_{i-1}$ remain unchanged in $\mathcal{G}_i$. We now prove that no new element is added to $\mathcal{P}$. Elements are added to $\mathcal{P}$ in L27 and L15.
  - L27 is not executed because $CP \sqsubseteq CP^{\mathcal{G}}$ is true in L23.
  - L15 is not executed because $AP_n^{\mathcal{L}} \not\sqsubseteq AP_n^{\mathcal{G}}$ is false in L14.

  Since no elements are added to $\mathcal{P}$, its size decreases in L6. Thus, if $\Lambda_{i-1} = \Lambda_i$ and the size of $\mathcal{P}$ decreases, then $(\Lambda_{i-1}, L_{i-1}) >^{\Lambda,L} (\Lambda_i, L_i)$ holds.

In every case the lexicographical order strictly decreases, thus the termination of Alg. 4 is proved. □

In order to prove the correctness of the results stored in $\mathcal{G}$ at the end of the execution of the incremental algorithm (Alg. 4), we study the correctness of the incremental analysis for only one domain. Let us introduce some notation and definitions:

Definition 4.3.5 (summaries graph)**.** *Given a program P, a domain D, and a global answer table $\mathcal{G}$ with entries of the form $(m, D) : CP \mapsto AP$, a* summaries graph *G_P is a directed graph represented by the pair $\langle N, E \rangle$ where N is the set of nodes and $E \subseteq N \times N$ is the set edges defined as follows:*

- *N is the set of methods m for which there is an entry $(m, D) : CP \mapsto AP$ stored in $\mathcal{G}$*
- *E is the set of edges, generated from $\mathcal{G}$ according to the following rules. For every entry $(m, D) : CP \mapsto AP \in \mathcal{G}$ and every node $n \in callers(m)$,*
  - *there is an edge from n to m labeled with CP, and*
  - *there is an edge from m to n labeled with AP*

Let P_0 be the initial program, and P_1 the program after the modification of method m. We will use G_{P_0} and G_{P_1} to represent the summaries graphs of P_0

and P_1 and $\mathcal{G}_{P_0}$ and $\mathcal{G}_{P_1}$ to refer to the global answer tables generated by the non-incremental algorithm (Alg. 3) for P_0 and P_1, respectively. We will use $\mathcal{G}|_M$ to refer to the subset of $\mathcal{G}$ that contains the entries related to methods in the set M.

Given two answer tables $\mathcal{G}_1$ and $\mathcal{G}_2$, we say that $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ when $\forall(m, D) : CP_1 \mapsto AP_1 \in \mathcal{G}_1$, there exists $(m, D) : CP_2 \mapsto AP_2 \in \mathcal{G}_2$ such that $CP_1 \sqsubseteq CP_2 \wedge AP_1 \sqsubseteq AP_2$.

Definition 4.3.6 (affected edge)**.** *Let G_{P_0} and G_{P_1} be the summaries graphs for programs P_0 and P_1, respectively. We say that an edge ST_{P_1} of G_{P_1} that goes to node m is an* affected edge *when:*

- *Node m exists in G_{P_1} but not in G_{P_0}, or*
- *The corresponding edge from P_0, ST_{P_0}, satisfies the following condition: $ST_{P_0} \not\sqsupseteq ST_{P_1}$*

An affected path *in a summaries graph is a path formed by affected edges only.*

Definition 4.3.7 (affected methods)**.** *Let G_{P_0} and G_{P_1} be the summaries graphs for programs P_0 and P_1, respectively. We say that a method m is* affected *when there is an affected edge that goes to m.*

In what follows we exploit the fact that there is only one method m which has been modified. It can be extended to several methods applying it one at a time.

Proposition 4.3.8. *All affected nodes of G_{P_1} are reachable from the modified method m by an affected path.*

Proof. Let us prove it by contradiction. Let us assume the following:

> *(*) "There exists an affected node n not reachable from the node corresponding to the modified method m by an affected path."*

We use the correctness result of Alg. 3, which is similar to other worklist algorithms ([NNH99], Sec. 6.1). Observe that Alg. 3 returns the least fixed point of the analysis regardless of the order in which events are processed.

If n is an affected node, then, by definition, there is at least one affected edge from another method n' to n, and n' is neither connected from m by an affected path, since otherwise there would be an affected path from m to n. Let N be the set of affected nodes connected to n by affected paths. This set does not contain m. All edges ST_{P_1} that connect nodes not in N to nodes in N are such that $ST_{P_1} \sqsubseteq ST_{P_0}$, where ST_{P_0} is the corresponding edge in G_{P_0}. Let M be the set of such nodes not in N.

It is straightforward to prove that if $ST_{P_1} \sqsubseteq ST_{P_0}$ for an edge that connects node a to node b, where $a, b \in N$ and ST_{P_i} corresponds to a call pattern, then the corresponding entries $(b, D) : ST_{P_0} \mapsto AP_{P_0} \in \mathcal{G}_{P_0}$ and $(b, D) : ST_{P_1} \mapsto AP_{P_1} \in \mathcal{G}_{P_1}$ satisfy $AP_{P_1} \sqsubseteq AP_{P_0}$, and vice versa for ST_{P_i} corresponding to an answer pattern. Therefore, $\mathcal{G}_{P_1}|_M \sqsubseteq \mathcal{G}_{P_0}|_M$. Since Alg. 3 obtains the least fixed point, it should have obtained results for methods in N such that $\mathcal{G}_{P_1}|_N \sqsubseteq \mathcal{G}_{P_0}|_N$, since the information used during the analysis of N, i.e., the one related to methods in M, is not affected by the modification. But edges in N are affected edges, and therefore the non-incremental algorithm in Alg. 3 would have not obtained the least fixed point when handling events related to methods in N for P_1, which is a contradiction. □

In order to handle changes in several methods, this proposition can be extended for global answer tables which are obtained from the incremental algorithm.

Theorem 4.3.9 (correctness of Alg 4)**.** *Given a program P, its analysis results stored in $\mathcal{G}$, and a method m which has been modified,* incremental_fixpoint(m) *returns a correct $\mathcal{G}$ for all methods in P.*

Proof. For proving the correctness, let start by proving that one iteration of the loop in L4-17 of the incremental analysis (Alg. 4) correctly propagates its new information to other nodes. We consider a global answer table, $\mathcal{G}_0$, the results from the analysis of method m, extracted from $\mathcal{P}$, and with entry $m : CP_m \mapsto AP_m \in \mathcal{G}_0$. The analysis of m starts by calling *noincr:analysis*, which consecutively calls *process_analysis*. During the analysis of m, we distinguish the following two cases:

- **Descendants**: the reanalysis of a method may find calls to another method m' with calling pattern CP' (L32 of Alg. 3). Let CP'_0 represent the call pattern of the corresponding entry for m' in $\mathcal{G}_0$, if it exists. One of the following possibilities may occur:

  (a) m calls m', which has no entry in $\mathcal{G}_0$. As before, we will consider w.l.o.g. that $m' : \bot \mapsto \bot$ exists in $\mathcal{G}_0$. Method m' must be reanalyzed for the calling pattern CP'. Alg. 4 covers this case by evaluating in L20 if the method exists in $\mathcal{G}_0$ and forcing its reanalysis by calling *noincr:get_proc_answer* with call pattern CP' in L30.

  (b) m calls m' with a call pattern CP' that is not contained in the call pattern CP'_0 stored in $\mathcal{G}_0$, that is, $(m', D) : CP'_0 \mapsto AP'_0 \in \mathcal{G}_0$ and $CP' \not\sqsubseteq CP'_0$. In this case, m' must be reanalyzed taking into account the new information. This condition is evaluated in *incr:get_proc_answer* (L23) which compares the new call pattern with the one stored in $\mathcal{G}_0$. Since the condition in L23 does not hold, m' is scheduled for reanalysis by calling *noincr:get_proc_answer* (L30) with $CP'_0 \sqcup CP'$ as calling pattern. m' will be reanalized because it is added to $\mathcal{Q}$ in L17 of Alg. 3 and this reanalysis will transitively handle all methods that have a call pattern not contained in $\mathcal{G}_0$.

  (c) m calls m' with a call pattern CP' that is contained in the one CP'_0 stored in $\mathcal{G}_0$, that is, $CP' \sqsubseteq CP'_0$. The information stored in $\mathcal{G}_0$ is valid for m' in this iteration, so we can reuse the information stored in $\mathcal{G}_0$. This is evaluated by Alg. 4 in L23 and the information stored in $\mathcal{G}_0$ is returned in L24.

- **Ancestors**: the reanalysis of a method produces a resulting AP' for the reanalyzed method m. AP_0 represents the answer pattern of the corresponding entry for m in $\mathcal{G}_0$. Let m' represent a caller of method m. One of the following possibilities may occur:

  (a) Method m' has not been reanalyzed during the execution of *process_analysis* for method m. In this case we can have two possibilities.

* If $AP' \sqsubseteq AP$, no new information is generated and, thus, no new analysis is required. This case is handled by Algorithm 4 in L14, checking if the new AP, stored in $\mathcal{L}$, is greater than the one stored in $\mathcal{G}_0$, that is, if $AP_n^{\mathcal{L}} \sqsubseteq AP_n^{\mathcal{G}}$. In that case, the algorithm does not generate any new event.
* If $AP' \not\sqsubseteq AP$, new information must be propagated to m'. This case is handled by Algorithm 4 in L14, checking if the new AP, stored in $\mathcal{L}$, is greater than the one stored in $\mathcal{G}_0$, that is, if $AP_n^{\mathcal{L}} \not\sqsubseteq AP_n^{\mathcal{G}}$. In that case, the algorithm adds m' to $\mathcal{P}$, forcing its reanalysis in subsequent iterations.

(b) Method m' has been reanalyzed during the execution of *process_analysis* for method m. This is because m' has a new calling pattern found during the non-incremental analysis. In this case we can have two possibilities.

* If $AP' \sqsubseteq AP$, no new information is generated and, thus, no new analysis is required. This case is handled by the incremental algorithm (Alg. 4) in L14 and by the non-incremental algorithm (Alg. 3) in L39, not generating any new event if $AP' \sqsubseteq AP$.
* If $AP' \not\sqsubseteq AP$, we have to propagate new information to m'. This case is handled by Alg. 3 in L39-40 by calling *invalidate_callers*, if the new answer pattern is greater than the one stored in $\mathcal{L}$ (L39: $AP \not\sqsubseteq AP^{\mathcal{L}}$). Procedure *invalidate_callers* of Alg. 3 checks if $\mathcal{L}$ has an entry for m', if true, L24 schedules a new event to be processed for m', forcing its reanalysis.

Once the execution of the reanalysis is completed, $\mathcal{G}_0$ must be updated with the new information to continue the analysis of the rest of remaining methods that are pending to be analyzed with the up-to-date information. This is done by the incremental algorithm in the loop in L11-16, that takes the information from $\mathcal{L}$ and copies it to $\mathcal{G}$.

Up to now, we have proved: (i) all affected methods are reachable from m, the modified method, by propagating the new information along the affected edges

(Prop. 4.3.8), and (ii) one iteration of the loop in L4-17 of the incremental algorithm correctly propagates the new information to the ancestors and descendants of the method taken from $\mathcal{P}$. The incremental algorithm starts by executing *incremental_fixpoint* from the modified method m using the calling pattern information taken from $\mathcal{G}$ (L9) and propagates its new information to its neighbours, generating a sequence of global answer tables, $\mathcal{G}_1 \rightsquigarrow \mathcal{G}_2 \rightsquigarrow \mathcal{G}_3 \ldots \mathcal{G}_n$, one $\mathcal{G}$ for each iteration of the loop in L4-19. As it is proved in (i), all affected methods are reachable starting from m, and the incremental algorithm starts the propagation of new information from m. Note that m is the only method that has been modified, the rest of the methods only propagate new information by receiving new information from affected edges. Thus, when a fixpoint is reached, that is, when no new information is propagated, the global answer table for the final iteration, $\mathcal{G}_n$, contains all the information propagated from the modified method through the whole program and consequently, $\mathcal{G}_n$ will contain correct information for all methods.

Proving the correctness of the algorithm handling multiple domains is straightforward. All methods that are reanalyzed must be invalidated and reanalyzed for all dependant domains. The invalidation of one $\mathcal{G}$ entry just forces its reanalysis but the behaviour of the algorithm is the same than handling only one domain. $\mathcal{G}$ entries are invalidated by the incremental algorithm in L26 and L15, that add, not only for the current domain, but also for all dependant domains.

□

4.4 Generation of Cost Relations

Given the size relations, the second phase of cost analysis (see Section 2.3) is the generation of CRs. This step is performed locally to each rule and hence it is already "incremental" (or local). Nevertheless, in order to link with the next phase of the incremental cost analysis (in Section 4.5), we revise what CRs are and how the incremental analysis has to treat them. Intuitively, the generation of an equation for a rule consists of the next steps: (1) apply the selected cost model to each instruction in the rule (a cost model maps an instruction into its

corresponding cost), (2) abstract each basic instruction by a size constraint and, (3) when we find a call to a method, the size constraint is the size relation in the summary above and the cost of the call is defined by a corresponding equation.

Example 4.4.1. *Consider the original program before the change in Example 4.2.1. By using the cost model that counts number of instructions, the first rule of Example 4.2.1 is transformed into the equation:* $mod(l) = 3 + get(l) + call\_incr(l) + dup(l')$ $\{l \geq l' + 2, l' \geq 0\}$ *where the* 3 *stands for the three calls in the instruction. The cost of the calls to the methods will be defined by corresponding equations. For the running example, we get:*

$$
\begin{array}{ll}
mod(l) = 3 + get(l) + call\_incr(l') + dup(l') & \{l \geq l' + 2\} \\
get(l) = 1 & \{\} \\
while(l) = 3 + call\_incr(l) + while(l') & \{l' \geq 0, l \geq l' + 2\} \\
dup(l) = 1 + while(l) & \{l \geq 0\} \\
call\_incr(l) = 0 + Inc.incr(l) & \{\} \\
while(l) = 0 & \{l = 0\} \\
Inc.incr(l) = 2 & \{l \geq 2\}
\end{array}
$$

The resulting constraints to the right define the applicability conditions of the equations and the size relations between the variables. We omit in the equations the variables that are not involved in the equation guards and because they are useless for solving the equations.

After a modification in a program, when Algorithm 4 finishes, we need to generate new CRs for all methods that have been reanalyzed (i.e., those that have belonged to $\mathcal{R}$). This is because their size relations may have changed; and, besides, for the changed method its accumulated cost can change as well.

Example 4.4.2. *Consider now the modification in Example 4.3.3. When Alg 4 finishes* $\mathcal{R}$ *has contained* {get, mod, dup, Incr2.inc} *for at least one domain. Thus, the following CRs must be generated by using the standard CR generation as explained above:*

$$
\begin{array}{ll}
mod(l) = 3 + get(l) + call\_incr(l') + dup(l') & \{l \geq l' + 1\} \\
get(l) = 3 & \{\} \\
dup(l) = 1 + while(l) & \{l \geq 0\} \\
while(l) = 3 + call\_incr(l) + while(l') & \{l' \geq 0, l \geq l' + 1\} \\
while(l) = 0 & \{l = 0\} \\
call\_incr(l) = 0 + Inc.incr(l) & \{\} \\
call\_incr(l) = 0 + Inc2.incr(l) & \{\} \\
Inc2.incr(l) = 1 & \{l \geq 1\}
\end{array}
$$

When comparing the equations with the ones in Example 4.4.1, we observe that after merging the size information gathered for the two implementations of incr*, in the new CR, we lose information and have to assume that the length of the list decreases (in the worst-case) by one. Cost relations* main*,* len *and* Inc.incr *do not need to be updated.*

4.5 Incremental Inference of Upper Bounds

As it is described in Chapter 2, the third phase in cost analysis consists in transforming the CRs obtained in Section 4.4 into *cost functions*, i.e., cost expressions without recurrences. Since a precise solution often does not exist, cost analyzers infer upper bounds/lower bounds (UBs/LBs) from them which are, resp., over/under-approximations of the worst/best-case cost. For the sake of concreteness, we focus on UBs (the problem of LBs is dual). In Section 4.5.1, we first introduce the notion of UB summary which specifies the information that needs to be stored in order to recompute UBs after a change in a program (and hence in its CR). Section 4.5.2 presents an algorithm to support incremental inference in this step.

4.5.1 The Notion of Cost Summary

Our starting point is the technique [AAGP11] (described in Chapter 2, Section 2.4) which proposes an automatic approach to obtaining UBs from CRs by (1) first, transforming all relations into direct recursion (this process leaves one relation per SCC) and (2) then, obtaining an UB for the standalone CRs (which

do not call any other relation) and consecutively replacing such UBs in the equations which call such relations until all CRs are solved. W.l.o.g., we consider polynomial CRs defined by a set of equations, each of them containing at most one recursive call:

$$\langle C(\overline{x})=\texttt{exp}+\sum_{i=1}^{k} D_i(\overline{y}_i) + C(\overline{y}), \varphi\rangle \tag{4.2}$$

In [AAGP11], automatic techniques to solve C (as well exponential and logarithmic relations) are proposed. The following definition summarizes the solving process described intuitively in Section 2.4:

Definition 4.5.1 (upper bound [AAGP11])**.** *An upper bound for* $C(\overline{x})$ *is* $\mathrm{UB}_C(\overline{x}) = \#iter * \texttt{mexp}$ *(base cases are ignored for simplicity) where:*

1. *#iter is an upper bound on the number of recursive calls of* C*,*
2. *the* size relations *(*φ*) are linear constraints on variables* $\overline{x}$ *and* $\overline{y}_i \cup \overline{y}$*,*
3. *the* invariant *(*ψ*) relates variables* $\overline{y}_i \cup \overline{y}$ *to their initial values* $\overline{x}$ *(we denote the initial value of a variable* x *as* x_0*),*
4. $\texttt{ub}_i$ *are upper bounds for each call* $D_i(\overline{y}_i)$*,*
5. *for each recursive equation, we have that:*
   $\texttt{mexp}'$=*maximize(*$\texttt{exp}$,$\overline{x}$,ψ,φ*)+* $\sum_{i=1}^{k}$ *maximize*$(\texttt{ub}_i, \overline{x}, \psi, \varphi)$
   where function maximize$(e, \overline{x}, \psi, \varphi)$ *returns the maximization of* e *for* ψ *and* φ *w.r.t. the equation entry variables* $\overline{x}$*,*
6. *if* C *has* k *recursive equations,* $\texttt{mexp}$=$max(\texttt{mexp}'_1, ..., \texttt{mexp}'_k)$*, where* $\texttt{mexp}'_j$ *is the maximized cost of equation* j *obtained in point 5, with* $j = 1, \ldots, k$*.*

Observe that the process of obtaining $\texttt{mexp}$ requires an invariant generation phase and a maximization of expressions. We cannot make any incrementalization of these two parts because they are already locally obtained from the CRs (see the details in [AAGP11]).

Example 4.5.2 (ub)**.** *Let us apply the above definition to solve the CR* mod *in Example 4.4.1 which has obtained before applying the modification in Example 4.3.3. The standalone CRs* Inc.incr *and* get *are already solved since they are not recursive. Next, we solve* while(l') *which is required to solve* dup*. The following invariant holds for this CR* $\psi = \{l'_0 \geq l' + 2\}$*, which simply states that the length of the list in the recursive call is strictly smaller than the length of the initial list. An UB of #iter for while is* $\mathsf{nat}(l'/2)$*. Function* $\mathsf{nat}(v)$=$\max(\{v, 0\})$ *is used by the UB solver to avoid negative evaluations. By applying Definition 4.5.1,* $UB_{\mathsf{dup}}(l') = 1+(3+2)*\mathsf{nat}(l'/2)$*. Finally, when computing an UB for* $\mathsf{mod}(l)$*, the UB for* dup *has to be maximized w.r.t. the entry variable* l*,* $\varphi = \{l \geq l'+2, l' \geq 0\}$ *and* $\psi = \{l_0 = l, l_0 \geq 2\}$*. This results in* $UB_{\mathsf{mod}}(l) = 6 + (1 + 5 * \mathsf{nat}(l/2 - 1))$*, where we can observe that the maximum cost of* dup *within* mod *occurs when* $l' = l - 2$*.*

In the above example, it can be seen that an UB is a global expression which includes the UBs of the relations it calls. If the CR associated to one method m changes, since it is not possible to distinguish within an UB which part of the cost is associated to m, the whole expression must be recomputed. This affects the UBs of all methods from which m is reachable and often forces recomputation of all cost functions upwards in the program call graph until reaching the main method. A fundamental idea to support incremental inference of UBs is to *annotate* each cost subexpression with the name of the relation it comes from. If, additionally, we keep the invariants and the size relations, given an annotated UB for a method m, it is possible to replace the cost subexpressions associated to those methods invoked from m whose UB has changed by the new (maximized) UBs, without having to recompute the whole UB for m. Thus, instead of using the UBs in Definition 4.5.1, we use the notion of *UB summary*.

Definition 4.5.3 (upper bound summary)**.** *In the same conditions of Definition 4.5.1, an* UB summary *for* $C(\overline{x})$ *is a tuple* $\mathrm{UB}^{\mathsf{S}}_{\mathsf{C}}(\overline{x}) = \langle \#iter \cdot \mathtt{aexp}, \psi, \varphi \rangle$*, where*

$\mathtt{aexp} = maximize(\mathtt{exp}, \psi, \varphi, \overline{x}) + \sum_{i=1}^{k} [maximize(\mathsf{remove\_annot}(\mathtt{aub}_i), \psi, \varphi, \overline{x})]_{D_i}$*: such that:*

- $\mathtt{aub}_i$ *is the annotated cost expression in the upper bound summary of* D_i*,*

- *function* remove_annot *removes the annotations of an expression, and*
- $[e]_D$ *is an annotation of* e *with the name of the relation* D *it originates.*

The notation aexp.*set_expr(*$[e]_m$*,exp) is used to rewrite in* aexp *the annotated subexpression* e *with* exp *keeping the same annotations.*

An important observation in the above definition is that the annotations refer only to direct calls from the relations.

Example 4.5.4 (UB summary)**.** *UB summaries for some selected CRs are:*

$$
\begin{aligned}
UB_{\mathsf{while}}(l) = \;& \langle (3 + [2]_{\mathsf{Inc.incr}(l')}) * \mathsf{nat}(l/2), \\
& \{l_0 \geq l + 2, l \geq 0\}, \\
& \{l = l'\}\rangle \\
UB^{\mathsf{S}}_{\mathsf{dup}}(l) = \;& \langle 1 + [(3 + 2) * \mathsf{nat}(l/2)]_{\mathsf{while}(l')}, \\
& \{l_0 = l, l_0 \geq 0\}, \\
& \{l = l', l \geq 0\}\rangle \\
UB^{\mathsf{S}}_{\mathsf{mod}}(l) = \;& \langle 3 + [1]_{\mathsf{get}(l')} + [2]_{\mathsf{incr}(l'')} + [(1 + 5 * \mathsf{nat}(l/2 - 1))]_{\mathsf{dup}(l''')}, \\
& \{l_0 = l, l_0 \geq 2\}, \\
& \{l = l', l = l'', l \geq l''' + 2, l''' \geq 0\}\rangle \\
UB^{\mathsf{S}}_{\mathsf{main}}(l) = \;& \langle 6 + [2 + 3 * \mathsf{nat}(l)]_{\mathsf{len}(l')} + [7 + 5 * \mathsf{nat}(l/2 - 1)]_{\mathsf{mod}(l'')}, \\
& \{l_0 = l\}, \\
& \{l = l', l = l'', l \geq 2\}\rangle
\end{aligned}
$$

Observe that the annotations refer only to direct calls from the relations, i.e., those of transitive calls are removed before maximizing the expression.

Note that the only difference with the UB of Example 4.5.2 is in the annotations.

4.5.2 Incremental Inference of Summaries

The input to the algorithm for reconstructing UB summaries is the table of summaries, named $\mathcal{U}$, which were previously computed, and the list of methods that have been reanalyzed in Algorithm 4, named $\mathcal{C}$ (i.e., those elements that have

belonged to $\mathcal{P}$ along the execution of Algorithm 4.) The first important point to notice is that (1) the summaries for all methods in $\mathcal{C}$ must be recomputed, as well as (2) those fragments of the summaries of the ancestors of methods in $\mathcal{C}$ that correspond to the cost of the reanalyzed methods. However, the actions to perform in each case are different: (1) while the summaries of $\mathcal{C}$ must be fully recomputed, as a change in the size relations might affect all components of an UB (#iter, invariants, size relations and maximized expressions can be different), (2) in the summaries of the ancestors of a method m in $\mathcal{C}$, we just need to replace those subexpressions annotated as m with the new maximized UB for m. As in Algorithm 4, we use a flag in the summaries table $\mathcal{U}$ to indicate wether the content of an entry is valid or not. An important idea is to first process all methods in $\mathcal{C}$ and, since full summaries for them might not be yet produced (as information about relations invoked from them might not be valid), generate only UB *skeletons*.

Definition 4.5.5 (upper bound skeleton)**.** *In the same conditions of Definition 4.5.1, an upper bound skeleton for* $C(\overline{x})$ *is a tuple* $SK_C(\overline{x})=\langle \#iter\cdot\texttt{sexp}, \psi, \varphi\rangle$*, where* `sexp` *is the annotated expression* $\texttt{exp}+\sum_{i=1}^{k}[\_]_{D_i}$ *and* $\_$ *denotes any value. In what follows, function* do_skeleton$(C(\overline{x}))$ *generates the upper bound skeleton for* C*.*

The difference between summaries and skeletons is that the UBs of the invoked relations are not filled (we write $\_$) and maximization is not yet performed. Once the skeletons have been computed for all summaries in $\mathcal{C}$, the algorithm can treat in the same way $\mathcal{C}$ and their ancestors (i.e., actions 1 and 2 above must not be distinguished anymore). In particular, given a relation m, all we need to do is replace the UBs of the relations invoked from m by their new maximized expressions (when needed). This is done by function *do_summary* of Algorithm 5.

Example 4.5.6 (skeleton)**.** *The change of Example 4.3.3 leads to the following skeletons for* mod *and* dup*:*

$$
\begin{aligned}
SK_{\mathsf{while}}(l) = \; & \langle (3 + max([\_]_{\mathsf{Inc.incr}(l')}, [\_]_{\mathsf{Inc2.incr}(l')})) * (l), \\
& \{l_0 \geq l + 1, l \geq 0\}, \\
& \{l = l'\}\rangle \\
SK_{\mathsf{dup}}(l) = \; & \langle 1 + [\_]_{\mathsf{while}(l')}, \\
& \{l = l_0, l_0 \geq 0\}, \\
& \{l = l', l \geq 0\}\rangle \\
SK_{\mathsf{mod}}(l) = \; & \langle 3 + [\_]_{\mathsf{get}(l')} + max([\_]_{\mathsf{Inc.incr}(l'')}, [\_]_{\mathsf{Inc2.incr}(l'')}) + [\_]_{\mathsf{dup}(l''')}, \\
& \{l_0 = l, l_0 \geq 2\}, \\
& \{l = l', l = l'', l \geq l''' + 1, l''' \geq 0\}\rangle
\end{aligned}
$$

Note that the skeleton of dup *differs from the initial one because of the new implementation of* incr. *This leads to a different #iter and introduces the* max *expression which includes the worst-case costs of both implementations of* incr. *As* main *has not been reanalyzed, its skeleton will not be computed again.*

```
 1: proc reconstruct_summaries()
 2:    P = ∅
 3:    for all m(x̄) in C do
 4:       ⟨aexp_m, ψ, φ⟩=do_skeleton(m(x̄))
 5:       U.update(m,⟨aexp_m, ψ, φ⟩)
 6:       U.invalidate(m(x̄) ∪ ancestors(m))
 7:       P.add(m(x̄) ∪ ancestors(m))
 8:    for all m(x̄) in P do
 9:       do_summary(m(x̄))
10:    U.validate_all()
11: function do_summary(m(x̄))
12:    ⟨aexp_m, ψ, φ⟩ = U.get(m)
13:    if m ∉ P then
14:       return remove_annot(aexp_m)
15:    for all [_]_p(ȳ) in aexp_m do
16:       if (m ∈ C) ∨ U.is_invalid(p) then
17:          exp_p = do_summary(p(ȳ))
18:          exp_p = maximize(exp_p,x̄,ψ,φ)
19:          aexp_m.set_expr([_]_p,exp_p)
20:    P.remove(m);
21:    U.update(m,⟨aexp_m, ψ, φ⟩)
22:    return remove_annot(aexp_m)
```

Algorithm 5: Incremental Upper Bounds Algorithm

Intuitively, Algorithm 5 works as follows. Procedure *reconstruct_summaries* updates the entries for all methods in $\mathcal{C}$ with their skeletons and activates the invalid flag for all relations that require reprocessing (i.e., $\mathcal{C}$ and their ancestors). It also builds the list $\mathcal{P}$ made up of such relations. Function *do_summary* takes care of replacing the affected components by the new maximized UBs. The base

case of the recursion (L13) is when the relation is not in $\mathcal{P}$ (either because its recomputation was not needed or because it has already been recomputed). We remove its annotations because $\mathcal{U}$ only keeps the outer level of annotations, as seen in Definition 4.5.3. If it is not a base case (L15-19), we need to obtain new maximized expressions for those subexpressions of $\mathtt{aexp}_m$ when (i) the expression is in $\mathcal{C}$ (this is because its size relations might have changed and we need to maximize again all components) or (ii) because the invoked relation is invalid. Lines 17-19 take care of recursively obtaining the summary for the subexpression, maximizing it and placing it inside the summary. Once all components of $\mathtt{aexp}_m$ have been treated (L20), the relation m is removed from the list of pending relations to process $\mathcal{P}$ and its summary is updated (L21). The result is returned without annotations in order to use it from the calling site.

Example 4.5.7 (algorithm 5)**.** *The change in Example 4.3.3 forces the execution of* do_skeleton *for all methods in* $\mathcal{C} = \{\mathsf{mod}, \mathsf{dup}, \mathsf{get}, \mathsf{Inc2.incr}\}$*. All those methods and their ancestors (*main*) are added to* $\mathcal{P}$*. Let us assume that* mod *is the first summary computed. Since* mod *is in* $\mathcal{C}$*, it needs the summary of* dup*. Hence,* do_summary(dup) *is invoked and a new summary of* dup *is produced by using its skeleton in Example 4.5.6. Then,* dup *is removed from* $\mathcal{P}$ *and we obtain:*

$$\begin{aligned} UB^{\mathsf{S}}_{\mathsf{while}}(l) = \; & \langle (3 + max([2]_{\mathsf{Inc.incr}(l')}, [1]_{\mathsf{Inc2.incr}(l')})) * \mathsf{nat}(l), \\ & \{l_0 \geq l + 1, l \geq 0\}, \\ & \{l = l'\}\rangle \\ UB^{\mathsf{S}}_{\mathsf{dup}}(l) = \; & \langle 1 + [(3 + 2) * \mathsf{nat}(l)]_{\mathsf{while}(l')}, \\ & \{l = l_0, l_0 \geq 0\}, \\ & \{l = l', l \geq 0\}\rangle \end{aligned}$$

In order to obtain UB_{mod}*, we need to maximize* UB_{dup}*, which leads to:*

$$\begin{aligned} UB^{\mathsf{S}}_{\mathsf{mod}}(l) = \; & \langle 3 + [3]_{\mathsf{get}(l')} + max([2]_{\mathsf{Inc.incr}(l'')}, [1]_{\mathsf{Inc2.incr}(l'')})) + [1 + 5 * \mathsf{nat}(l - 1)]_{dup(l''')}, \\ & \{l_0 = l, l_0 \geq 2\}, \\ & \{l = l', l = l'', l \geq l''' + 1, l''' \geq 0\}\rangle \end{aligned}$$

For brevity, the recomputation of the UBs for incr *and* get *is not described. Next, the summary of* main *is recomputed. Since* main *was invalidated but not reanalyzed, its summary can be reused as a skeleton, maximizing again only its invalidated subexpressions. Only* UB_{mod} *must be maximized, and* UB_{len} *can be reused:*

$$\begin{aligned} UB^{\mathsf{S}}_{\mathsf{main}}(l) = \quad & \langle 5 + [2 + 3 * \mathsf{nat}(l)]_{\mathsf{len}(l')} + [9 + 5 * \mathsf{nat}(l-1)]_{\mathsf{mod}(l'')}, \\ & \{l = l_0\}, \\ & \{l = l', l = l'', l \geq 2\}\rangle \end{aligned}$$

All in all, the incremental recomputation of UBs has avoided computing the skeleton (#iter, invariant) and one maximization for main*, and summaries of* len *and* lnc.incr *remain the same.*

Theorem 4.5.8 (termination of Alg. 5)**.** *Given a set of relations* $\mathcal{C}$ *whose CRs have changed and a table of upper bounds summaries* $\mathcal{U}$*, reconstruct_summaries terminates.*

For proving the theorem, let us introduce some notation. Given a table of summaries $\mathcal{U}$, where C and D are two relations defined in $\mathcal{U}$, we say that C depends on D, denoted $C \mapsto D$, iff there is an upper bound summary such that $UB^S_C = \langle \#iter \cdot \mathtt{aexp}, \psi, \varphi\rangle$, where $\mathtt{aexp} = \mathtt{exp}_0 + \sum_{i=1}^{k} [\mathtt{exp_i}]_{D_i}$ and $\mathtt{exp}_i$ denote unannotated expressions. A *directed graph* G is a pair $\langle N, E\rangle$ where N is the set of nodes and $E \subseteq N \times N$ is the set edges. We associate to each table of summaries $\mathcal{U}$ a graph $G_{\mathcal{U}}$, which is the directed graph obtained from $\mathcal{U}$ by taking the set of entries in $\mathcal{U}$ as N and where $(C, D) \in E$ iff $C \mapsto D$. A relation D is *reachable* from a relation C in $\mathcal{U}$ iff there is a path from C to D in $G_{\mathcal{U}}$.

The resulting graph is a directed acyclic graph, since the entries in $\mathcal{U}$ are in closed-form. Given a node n, $ancestors(n)$ is the set of nodes in the graph from which n is reachable (i.e., the set of entries whose closed-form upper bound depends on the expression of n). $descendants(n)$ is the set of nodes in the graph reachable from n (the set of entries in $\mathcal{U}$ which must be computed to obtain the upper bound of n). *ancestors* and *descendants* can also be applied to sets of nodes. The *level* of a node n in a directed acyclic graph is the length of the longest path reachable from n. The *depth* of a directed acyclic graph is the longest path in the graph.

We start the proof by demonstrating that all reanalyzed methods and their ancestors will be added to the list of pending methods.

Proposition 4.5.9. *All reanalyzed methods and their ancestors will be added to* $\mathcal{P}$*.*

Proof. The first step of the algorithm is to invalidate all methods that depend on methods reanalyzed and whose size relations may be affected by the modification in the source code. Let us assume the following statement:

> *(*) "There exists an entry* m *in* $\mathcal{U}$ *whose upper-bound must be computed and it is not included in* $\mathcal{P}$ *after the execution of lines 3-7"*

The proof now is by contradiction. We make a case distinction:

- If $m \in \mathcal{C}$: m is added to $\mathcal{P}$ in L7, which contradicts the assumption (*);
- else, if $m \in ancestors(\mathcal{C})$: m is added to $\mathcal{P}$ in L7, which contradicts the assumption (*);

□

In the second step, we demonstrate that all pending methods will be processed and, at the end of the algorithm, all pending upper-bounds are computed. We assume that L18-19 are correct.

Lemma 4.5.10. *All methods in* $\mathcal{P}$ *are processed and, at the end of the algorithm, all invalidated upper-bounds are computed.*

Proof. Assuming the proposition 4.5.9, $\mathcal{U}$ contains all reanalyzed methods as invalids in L8. Termination of function *do_summary* is ensured by the fact that $G_{\mathcal{U}}$ is a directed acyclic graph. This is because the entries in $\mathcal{U}$ are in closed form. The recursive *do_summary* function performs a depth traversal of this graph.

As regards correctness, the proof is by induction on the depth of $G_{\mathcal{U}}$.

- **Base case**: Let $G_{\mathcal{U}}$ be a graph of depth 0 and m be a node in $G_{\mathcal{U}}$ of level 0. There are no paths starting from m in the graph. In L13 of function *do_summary* there are two cases:

- If $m \notin \mathcal{P}$, then the summary stored in $\mathcal{U}$ is correct.
- Otherwise, since there is no edge from m to any other node, loop in L15 does not iterate, and m is removed from $\mathcal{P}$. The skeleton generated and stored in $\mathcal{U}$ (L4-5) is correct.

- **Inductive case**: Let us suppose that the summaries stored in $\mathcal{U}$ are correct for a graph $G^i_{\mathcal{U}}$ of depth smaller or equal than i.

  Let $G^{i+1}_{\mathcal{U}}$ a graph of depth $i+1$ and m be a node of level $i+1$ in $G^{i+1}_{\mathcal{U}}$. According to L13, there are two cases:

  - If m is not in $\mathcal{P}$, we have two situations:
    * m was not added to $\mathcal{P}$ in L7. This is because $m \notin \mathcal{C}$ and $m \notin ancestors(\mathcal{C})$. The summary for m stored in $\mathcal{U}$ is not affected by the change, and therefore correct.
    * m was added to $\mathcal{P}$ in L7, its entry in $\mathcal{U}$ recomputed (L15-19), updated in $\mathcal{U}$ (L21) and removed from $\mathcal{P}$ (L20). Therefore, in $\mathcal{U}$ the entry has already been correctly recomputed.
  - Otherwise, all relations from which m directly depends on are traversed in L15. They correspond in $G^{i+1}_{\mathcal{U}}$ to the nodes for which there is an edge from m to them. Let n_j, $1 \leq j \leq k$ be those nodes.

    All relations in $\mathcal{P}$ are marked as *invalid* in $\mathcal{U}$ (L6), and are not validated again until the algorithm finishes (L10).

    L16 leads to several cases:
    * If $m \in \mathcal{C}$, then its skeleton was generated in L4 and its summary is reconstructed (L17-19), updated in $\mathcal{U}$ (L21) and removed from $\mathcal{P}$ (L20) avoiding its recomputation. The recursive call to *do_summary* in L17 is applied to n_j, which in $G^{i+1}_{\mathcal{U}}$ have levels $l_j < i+1$. The induction hypothesis guarantees that *do_summary* produces correct results for a graph of depth less or equal than i. The subgraph formed by n_j and all nodes reachable from n_j is a graph of depth less or equal than i, and thus the hypothesis holds.

Consequently, the summary for m is reconstructed using correct information from n_j.

* If $m \notin \mathcal{C}$, those nodes n_j which are invalid in $\mathcal{U}$ are correctly reconstructed in L17-19 as in the previous case, and the summary for m is updated in $\mathcal{U}$ with the correct maximized subexpressions for n_j (L21).

□

Proof. [**Theorem 4.5.8**] Proof of theorem is straightforward by combining Proposition 4.5.9 and Lemma 4.5.10. □

4.6 Experiments

COSTA implements all global analyses in the method summary definition (see Definition 4.3.1) and, in addition, it performs nullness and sign analyses (see Section 2.3.1). The incremental multi-domain algorithm has been implemented and applied to all domains.

Our experimental evaluation has been performed on slightly modified versions of programs `Voronoi`, `Health`, `TSP`, and `MST`, from the JOlden benchmark suite [Sui], available at `http://costa.ls.fi.upm.es`. The modifications (described in [AGGZ10] in detail) are performed in order to overcome some limitations inherent to the size analysis and the UB solver of COSTA and are not related to the incremental extensions presented in this chapter. Furthermore, we have used as benchmarks the following programs borrowed from the Apache-Commons Project [Pro]: StringEncrypt and ParseTarHeader from Apache-Commons-Math, and TestOrthogonal and TestDistance from Apache-Commons-Compress. The source code of all of them is available at the Apache-Commons web site. The cost model used in our experiments is the number of bytecode instructions required for executing the corresponding programs.

Table 4.1 shows some information about the size and complexity of the benchmark programs. For each program, the column $\#_{\mathsf{BY}}$ shows the number of bytecode instructions, $\#_{\mathsf{RU}}$ indicates the number of RBR rules, and $\#_{\mathsf{EQ}}$ the number

	Program Info.			Analysis Times		
Experiment	$\#_{\mathsf{BY}}$	$\#_{\mathsf{RU}}$	$\#_{\mathsf{EQ}}$	$\mathbf{T}_{\mathsf{CRs}}$	$\mathbf{T}_{\mathsf{UB}}$	$\mathbf{T}_{\mathsf{T}}$
MST	250	120	82	9870	570	10440
TSP	189	78	55	760	10940	11700
Health	209	73	47	4020	240	4260
Voronoi	202	66	40	460	140	600
StringEncrypt	204	136	101	419	750	1169
ParseTarHeader	341	164	115	1200	2590	3790
TestOrthogonal	221	88	56	500	180	680
TestDistance	150	93	62	550	90	640

Table 4.1: Benchmarks information

of relations in the CR. The table also contains information about the analysis times (in ms) taken by the non-incremental analysis. The total analysis time (T_{T}) is split into the time taken to build the CRs ($\mathbf{T}_{\mathsf{CRs}}$) and the time to obtain a closed-form UB from the CRs ($\mathbf{T}_{\mathsf{UB}}$).

Our experiments are based on making a series of systematic modifications to the benchmark programs and comparing the time taken by the incremental approach with the time taken by the non-incremental one. We use the notation $P_{i-1} \rightarrow P_i$ to represent a program change, where P_{i-1} and P_i correspond to the versions of the program before and after the change, respectively. We refer to the non-incremental analysis of a program P starting from method m as $\mathcal{A}(P, m)$, while $\mathcal{A}^{\Delta}(P_i, m_i)$ is used to represent the incremental analysis of P_i starting from m_i with respect to the information computed and stored in $\mathcal{G}_{i-1}$ and $\mathcal{U}_{i-1}$ in the previous analysis. In what follows, as abbreviation, we use $\mathcal{T}_i$ to denote both $\mathcal{G}_i$ and its associated $\mathcal{U}_i$. Given a sequence of changes in a program, $P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_i$, the successive incremental analyses can be denoted as:

$$\mathcal{A}(P_0, m_0) \xrightarrow{\mathcal{T}_0} \mathcal{A}^{\Delta}(P_1, m_1) \xrightarrow{\mathcal{T}_1} \ldots \xrightarrow{\mathcal{T}_{i-1}} \mathcal{A}^{\Delta}(P_i, m_i)$$

We have performed a series of experiments which aim at evaluating the benefits of using our proposed incremental approach. The first two experiments capture the most common development scenario where (most of) the program is available and the programmer is making relatively small changes which affect one

method at a time. In our expected usage scenario, analysis is triggered whenever the user saves the class file he/she is editing. Depending on the case, the change can be minimal (for example, after refactoring the code) or important, where the analysis results for the new version of the method drastically differ from the previous ones and they have to be propagated to calling methods. This first extreme case is captured by our first experiment (*Touch experiment*), where we simply replace the code of a method with a new version which in reality is identical to the previous one. In this case, analysis will reanalyze the updated method, but no change has to be propagated. The second extreme case is captured by our second experiment (*Adding experiment*), where we replace a missing implementation of a method, which simply returns the default value of the return type[†], with the final implementation.

We use the term *unweighted speedup* (S) to denote the ratio between the time required to perform the non-incremental analysis of a program, and the time required by the incremental analysis. In addition to S, experiments tables also contain the *weighted speedup* (W), weighting formula S with respect to the number of bytecode instructions of the modified method. Larger methods are more likely to be changed, thus W provides a more realistic estimate.

(1) Touch experiment: In this experiment we just have one version of the program, P_0, on which the incremental and non-incremental analyses are performed. The incremental analysis is systematically performed by starting from each method m_i in P_0 and using the previously computed $\mathcal{T}_0$:

$$\mathcal{A}(P_0, m_0) \xrightarrow{\mathcal{T}_0} \mathcal{A}^{\Delta}(P_0, m_i)$$

The speedup (S) is computed as the ratio between n times the time taken by the non-incremental analysis of P_0, and the addition of the times of the incremental analysis starting the analysis from different methods in P (m_i):

$$\mathsf{S} = \frac{n \times time(\mathcal{A}(P_0, m_0))}{\sum_{i=1}^{n} time(\mathcal{A}^{\Delta}(P_0, m_i))}$$

[†] Methods that return a non-void type keep a default `return` statement: `return null` for methods that return an object type, `return 0` for methods that return an integer, etc.

Incremental Vs. Non incremental Speedup						
	Unweighted Speedup			Weighted Speedup		
Benchmark	S_{CRs}	S_{UB}	S_T	W_{CRs}	W_{UB}	W_T
MST	29.05	6.97	24.77	18.89	4.10	15.78
TSP	2.72	3.97	3.85	1.77	2.09	2.07
Health	7.39	2.35	6.59	5.35	1.61	4.73
Voronoi	4.18	6.22	4.53	2.45	2.49	2.46
StringEncrypt	10.90	7.09	8.11	7.56	3.92	4.74
ParseTarHeader	5.52	2.03	2.54	8.09	3.31	4.07
TestOrthogonal	3.25	10.00	3.96	2.64	4.27	2.94
TestDistance	3.09	4.95	3.26	4.05	6.09	4.25
Arith. Mean	**8.26**	**5.45**	**7.20**	**6.35**	**3.49**	**5.13**

Table 4.2: Touch experiment results

The above results (Table 4.2) clearly show that the incremental approach is much more efficient than the non-incremental one when handling small changes in the program, with a weighted speedup of over 5.

(2) Adding experiment: This experiment captures the second extreme case in which we replace a missing implementation of a method with its final code. In this situation, the analysis of the new version will require triggering analysis of possibly multiple (transitively) calling methods.

In this case, the experiment considers different initial versions of the program P_0^i, each one missing the implementation of one method. Each P_0^i is then analyzed using the non-incremental algorithm, $\mathcal{A}(P_0^i, m_0)$. Subsequently, the code of m_i is restored producing the version P_1 (the final version of the program), and incrementally reanalyzed, $\mathcal{A}^\Delta(P_1, m_i)$, using $\mathcal{T}_0^i$:

$$\mathcal{A}(P_0^i, m_0) \xrightarrow{\mathcal{T}_0^i} \mathcal{A}^\Delta(P_1, m_i)$$

This procedure is illustrated in Figure 4.3, using as example a small call graph composed by four methods. Colored nodes represent empty methods, and white nodes represent implemented methods.

This procedure is systematically applied in order to modify all methods of the benchmark program. The speedup of the incremental approach is calculated as follows:

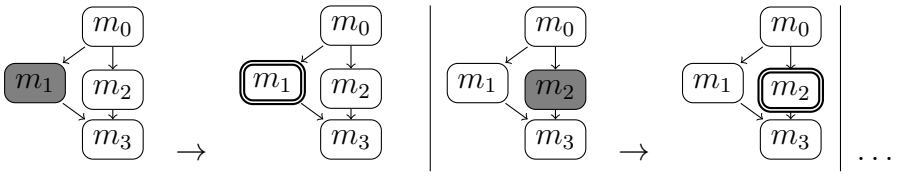


Figure 4.3: Adding experiment scheme

$$\mathsf{S} = \frac{n \times time(\mathcal{A}(P_1, m_0))}{\sum_{i=1}^{n} time(\mathcal{A}^\Delta(P_1, m_i))}$$

Experimental results (Table 4.3) clearly show that the incremental approach efficiently handles method modifications in a program. It is efficient in both parts of the resource usage analysis, in the generation of `CRs` and the `UB` solving. Altogether it achieves a significant improvement over non-incremental analysis, being almost two times faster.

Incremental Vs. Non incremental Speedup						
	Unweighted Speedup			**Weighted Speedup**		
Benchmark	$\mathsf{S}_{\mathsf{CRs}}$	S_{UB}	S_{T}	$\mathsf{W}_{\mathsf{CRs}}$	W_{UB}	W_{T}
MST	2.17	1.14	2.07	2.03	1.31	1.97
TSP	1.08	1.33	1.31	1.14	1.46	1.43
Health	2.57	1.17	2.41	1.75	1.23	1.71
Voronoi	1.55	2.07	1.64	1.31	1.86	1.40
StringEncrypt	1.26	1.30	1.28	2.04	2.40	2.26
ParseTarHeader	1.54	1.30	1.37	2.46	2.35	2.39
TestOrthogonal	1.14	1.79	1.26	1.22	1.55	1.30
TestDistance	1.38	1.80	1.43	1.90	2.44	1.96
Arith. Mean	**1.59**	**1.49**	**1.60**	**1.73**	**1.82**	**1.80**

Table 4.3: Adding experiment results

(3) Top-down development experiment: A question which remains to be answered is whether the incremental approach can be less efficient than analyzing

the whole program. This question is important since there is no formal guarantee that the incremental analysis will be more efficient than the analysis from scratch. In fact, it is possible to find situations where global analysis can be more efficient. To assess this situation, our third experiment tries to perform a stress test of the worst possible situation that can arise. This occurs when we analyze in an incremental fashion a program, by adding a method at a time, following a top-down order in the call graph. In the experiment, we start with empty implementations that lack the content of the method for all methods. We progressively add the implementations one by one starting from the root of the call graph. This scenario will require the largest possible number of reanalysis.

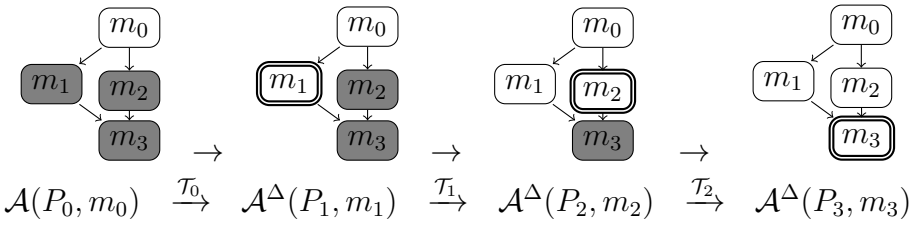


Figure 4.4: Top-down development experiment scheme

The scenario is illustrated in Figure 4.4. The experiment starts from the analysis of an initial version $\mathcal{A}(P_0, m_0)$ where all methods except the main method m_0 are empty. In the second step, the code of a method m_1, directly invoked by m_0, is added generating a new version of the program P_1, and the incremental analysis $\mathcal{A}^\Delta(P_1, m_1)$ is applied by using $\mathcal{T}_0$. In the following steps, the contents of the remaining methods are added one by one (m_i), producing different versions of the program (P_i). The speedup (S) of the incremental analysis is computed as

$$\mathsf{S} = \frac{\sum_{i=1}^{n} time(\mathcal{A}(P_i, m_i))}{\sum_{i=1}^{n} time(\mathcal{A}^\Delta(P_i, m_i))}$$

Table 4.4 shows that, even in the extreme case of having the reanalyze a large number of methods, the use of our incremental analysis is not worse than the global one. Only in one example (TestOrthogonal) there is a small slowdown in

Incremental Vs. Non incremental Speedup						
	Unweighted Speedup			Weighted Speedup		
Benchmark	S_{CRs}	S_{UB}	S_{T}	W_{CRs}	W_{UB}	W_{T}
MST	3.42	1.43	2.96	3.42	1.48	2.94
TSP	1.04	1.34	1.32	1.05	1.40	1.38
Health	1.18	1.09	1.17	1.05	0.96	1.04
Voronoi	1.33	1.74	1.40	1.10	1.52	1.17
StringEncrypt	1.35	1.31	1.32	1.51	1.63	1.58
ParseTarHeader	1.29	1.26	1.27	1.29	1.41	1.37
TestOrthogonal	1.09	1.85	1.23	0.84	1.16	0.91
TestDistance	1.36	2.26	1.44	1.39	2.16	1.46
Arith. Mean	**1.51**	**1.53**	**1.51**	**1.46**	**1.46**	**1.48**

Table 4.4: Top-down development experiments results

the total *weighted speedup*. While, there is an overall gain of 1.48. This, together with the first two experiments, indicates that incremental analysis will provide important gains in the most common and realistic scenarios while not introduce overhead in the less optimal scenarios.

a

4.7 Related Work

The most related approach to ours is [HPMS00], which develops a generic incremental analysis algorithm for constraint logic programs (CLP). In addition to the language differences, their incremental algorithm does not handle domain dependencies like ours, which is fundamental for an application such as resource usage which relies on multiple pre-analyses with dependencies among them. Besides, our work provides novel definitions for cost summaries which enable the incremental reconstruction of cost functions, a problem that has not been considered before. Another difference is that the granularity of the analysis is our case is at the level of methods, while [HPMS00] considers modifications at the level of rules. This is because in a CLP program the notion of method does not exist as such. A method can be seen as a predicate definition and hence the changes in CLP are handled at the level of rules. This finer-grained modularity does not fit

well in the imperative setting.

Other approaches to incremental analysis are developed for other purposes, e.g., [WG97] proposes an efficient incremental parser for general context-free grammars which allows generating incremental tools. The work in [Hed89] develops an approach to incremental static semantic analysis for object-oriented languages using door attribute grammars as a way to maintain incremental information, while our work is mostly focused on the reconstruction of the analysis information and the *cost summaries*. An incremental analysis based on incremental specifications such as those found in formal models is presented in [GL98], while we do not rely on specifications. The notion of summary has been previously used in other contexts [RHS95, DDA08] different from incremental analysis. It is also worth mentioning recent work on incremental analysis [GKP12] which defines an incremental analysis via domain specific solvers, for declarative modeling language based on first-order logic with sets and relations. The latter work is also related to directed incremental symbolic execution (DiSE) [PYRK11], a technique which in principle is more related to testing than to static analysis. However, the novelty of DiSE is to combine the efficiencies of static analysis techniques to compute program difference information with the precision of symbolic execution to explore program execution paths and generate path conditions affected by the differences. We believe that a combined approach like this one could be also adopted for the inference of resource consumption information.

Modular analysis [CC02, Log11] is related to incremental analysis in that it aims at reducing the time and memory required to perform analysis by splitting the program into smaller parts and storing analysis results, either automatically or by using user-provided summaries. Our technique is modular in the sense that it automatically stores summaries, though it does not split the program into smaller parts. On the other hand, modularity per se does not handle the efficient recomputation of analysis results after a program change.

Chapter 5

Verified Resource Guarantees

This chapter presents our main results on the certification of resource guarantees that were presented in the international conferences *PEPM'11* [ABG+11a] and *FASE'12* [ABG+12].

5.1 Introduction

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of systems. The analysis algorithms used in COSTA for inferring the main components of the UB generation were proven correct at a theoretical level. However, there is no guarantee that correctness is preserved in the actual implementation which is rather involved. Verifying the correctness of modern static analyzers like COSTA is rather challenging, among other things, because of the sophisticated algorithms used in them, their evolution over time, and, possibly, proprietary considerations. A simpler alternative is to construct a validating tool [PSS98] which, after every run of the analyzer, formally confirms that the results are correct and, optionally, generates correctness proofs. Such proofs could then be translated to *resource certificates* [CW00, Nec97].

KeY [BHS06] is a state-of-the-art source code verification tool for the Java programming language. Its coverage of Java is comparable to that of COSTA (nearly full sequential Java, plus a simplified concurrency model). KeY imple-

ments a logic-based setting of symbolic execution that allows deep integration with aggressive first-order simplification. While the degree of automation of KeY is very high on loop- and recursion-free programs, the user must in general supply suitable invariants to deal with loops and recursion. In general, invariants that are sufficient to prove complex functional properties cannot be inferred automatically. However, simpler invariants that are sufficient to establish UBs *can* be automatically derived in many cases and this is exactly COSTA's forte. Our work is based on the insight that the static analysis tool COSTA and the formal verification tool KeY have complementary strengths: COSTA is able to derive UBs of Java programs including the invariants needed to obtain them. This information is enough for KeY to *prove* the validity of the bounds and provide a certificate. The main contribution of this work is to show that, using the Java verification tool KeY, it is possible to formally and automatically verify the correctness of the UBs obtained by COSTA.

5.1.1 Organization of the Chapter

This chapter is structured as follows: Section 5.2 reviews the main components of the UBs inferred by COSTA by means of a running example and describes the JML annotations needed to verify the correctness of the UB. Section 5.3 describes how KeY verifies the JML annotations generated by COSTA by using dinamic logic by focusing on integer manipulating programs only. Section 5.4 presents the additional components that need to be verified for carrying out the extension for handling heap manipulating programs. Section 5.5 describes how the KeY logic has been extended to express and verify structural heap properties and path-length assertions. Experimental evaluation is described in Section 5.6. Section 5.7 relates our approach with existing related work.

5.2 Upper-Bounds for Integer Manipulating Programs

In this section, we summarize the techniques used in COSTA for automatically inferring UBs (we refer to Section 2.4 for more details), and we identify the proof

obligations that need to be verified using KeY for integer manipulating programs.

5.2.1 Main Components of an Upper Bound

Figure 5.1 shows a Java source code (JML annotated), that implements the insert sort algorithm. COSTA receives a non-annotated version of the above program and, for the cost model that counts the number of executed bytecode instructions, produces the (asymptotic) UB $\mathsf{insert\_sort}(a)=a^2$, where a refers to $\mathsf{A.length}$. The underlying analysis used in COSTA infers UBs for each iterative and recursive constructs (loops) and then composes the results in order to obtain an UB for the method of interest. As mentioned before, in order to infer an UB for a single loop, it first infers an UB A on the cost of a single execution of its body, an UB I on the number of iterations that it can make, and then $A*I$ is an UB for the loop. In order to infer A and I COSTA relies on several program analysis components that provide essential information (see Section 2.4 for further details):

Ranking functions. For each loop, COSTA infers a linear function from the loop variables to $\mathbb{N}$ which is decreasing at each iteration. For example, for the loop at line 17, it infers function $f(a, j) = \mathsf{nat}(a - j)$. This function can be safely used to bound the number of iterations. In the example, if a_3 and j_3 are the initial values of a and j, then it is guaranteed that $f(a_3, j_3)$ is an UB on the number of iterations of the loop.

Loop invariants. For each loop in the program, COSTA infers an invariant that involves the loop's variables and their initial values (i.e., their values before entering the loop). Let us denote by i_1 the initial value of i when entering the loop at line 9. COSTA infers the invariant $i \leq i_1$, which states that i is always smaller than or equal to its initial value when the program reaches the loop condition. This information, together with the size relations below, is needed to compute the worst-case cost of executing one loop iteration.

Size relations. Given a fragment of code or a scope (details below), COSTA infers relations between the values of the program variables at a certain program

```
1   void insert_sort(int A[]) {
2     int i, j, v;
3     //@ ghost int i_0 = i; int j_0 = j; int a_0 = a;
4     i=A.length-2;
5     //@ assert (i = i_0 - 2 ∧ j = j_0 ∧ a = a_0)
6     //@ ghost int i_1 = i; int j_1 = j; int a_1 = a;
7     //@ loop_invariant i ≤ i_1
8     //@ decreases i > 0 ? i : 0
9     while ( i ≥ 0 ) {
10        //@ ghost int i_2 = i; int j_2 = j; int a_2 = a;
11        j=i+1;
12        v=A[i];
13        //@ assert j = i_2 + 1 ∧ i_2 ≥ 0
14        //@ ghost int i_3 = i; int j_3 = j; int a_3 = a;
15        //@ loop_invariant j ≤ a_3
16        //@ decreases a - j>0 ? a - j : 0
17        while ( j < A.length && A[j] < v) {
18          A[j-1]=A[j];
19          j++;
20        }
21        A[j-1]=v;
22        i--;
23      }
24  }
```

Figure 5.1: Integer manipulating running example

point of interest within the scope and their initial values when entering the scope. For example, at program point 13, it infers that $j = i_2 + 1$, where i_2 is the value of i when entering the scope that contains line 13 (i.e., the scope here is the loop body). In this case the relation is a simple consequence of the instruction at line 11. In general, however, it may not be trivial to infer such relations nor to prove that they are correct.

Upper Bounds. Once the above information has been inferred, it is straightforward to compute an UB for the method. Let us show this process on the integer manipulating running example:

Inner loop. The process starts from the innermost loops. Thus, we start with the loop at line 17. Assuming that executing the condition costs (at most) c_1 instructions, and that the cost of each iteration (i.e., the loop body) is c_2 instructions, then it is clear that $\mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1$ is an UB on the cost of this loop (because c_1 and c_2 are constant).

Outer loop. Next, we move to the outer loop at line 9. Let us assume that the cost of the comparison is c_3 instructions, the code at lines 11–12 are c_4 instructions, and the code at lines 21–22 are c_5 instructions. Then, the cost of each iteration of this loop is $c_3 + c_4 + \underline{\mathsf{nat}(a_3 - j_3) * (c_1 + c_2) + c_1} + c_5$, where the underlined subexpression corresponds to the cost of the inner loop computed above. Note that in this case, each iteration might have a different cost, since $a_3 - j_3$ is not the same for all iterations. Simply multiplying the number of iterations $\mathsf{nat}(i_1)$ by such a cost is unsound. The solution is to find an expression U in terms of the initial values of a_1, i_1, j_1 that does not change during the loop and such that $U \geq a_3 - j_3$ in all iterations. Then, $\mathsf{nat}(i_1) * [c_3 + c_4 + \underline{\mathsf{nat}(U) * (c_1 + c_2) + c_1} + c_5] + c_3$ is an UB for the loop. In order to find such U, COSTA uses the loop invariant (line 7) and the size relations (line 13) as follows: it solves the parametric integer programming problem of maximizing the objective function $a_3 - j_3$ w.r.t. the loop invariant and the size relations where i_1, a_1, j_1 are the parameters. This produces an expression in terms of i_1, a_1, j_1 which is greater than or equal to $a_3 - j_3$ in all iterations of the loop. In our example, it is $U = a_1 - 1$.

Method. We finally can compute the cost of the insert_sort method. Assume that the cost of line 4 is c_6, then the cost of the method is

$$c_6 + \underline{\mathsf{nat}(i_1) * [c_3 + c_4 + \mathsf{nat}(a_1 - 1) * (c_1 + c_2) + c_1 + c_5] + c_3}.$$

We need to express this UB in terms of the input parameter a. For this, COSTA maximizes (using parametric integer programming) i_1 and $a_1 - 1$ w.r.t. the size relation at line 5 and, respectively, obtains $a - 2$ and $a - 1$. Therefore, the UB for insert_sort is:

$$c_6 + \mathsf{nat}(a-2) * [c_3 + c_4 + \underline{\mathsf{nat}(a-1) * (c_1 + c_2) + c_1} + c_5] + c_3$$

5.2.2 UBs Claim as JML Annotations

To justify that the UBs obtained by COSTA are correct, we need to provide formal correctness proofs for all the claims above. This includes the ranking functions, invariants, size relations, the cost model that provides all c_i, and the underlying PIP solver.

Correctness of the cost model is trivial as it is a simple mapping from each instruction to a number. Correctness of the underlying PIP solver is also straightforward if we use the maximization procedure defined in [AAGP11], which is based only on the Gaussian elimination algorithm. Therefore, we concentrate on verifying the correctness of the ranking functions, size relations and invariants. They are inferred by large software components whose correctness has not been verified. We now briefly describe the translation of the different pieces of information generated by COSTA into JML annotations on the Java program, which will allow their verification in KeY.

Ranking functions. For a given loop, when COSTA infers a ranking function of the form $\mathsf{nat}(\ell)$, we translate it to the JML annotation

//@ decreasing $\ell > 0 \ ? \ \ell : 0$

since $\mathsf{nat}(\ell)$ can be defined as an if-then-else. COSTA might provide also ranking functions of the form $\log(\mathsf{nat}(\ell) + 1)$, which are handled similarly.

Invariants. COSTA infers an invariant φ for each loop. This invariant involves the loop variables $\bar{v}$ and auxiliary variables $\bar{w}$ such that each w_i represents the initial value of v_i. The JML annotation for this invariant consists of one line defining all $\bar{w}$ as ghost variables

//@ ghost int $w_1 = v_1$*;...; int* $w_n = v_n$

and one line for declaring the loop invariant

//@ loop_invariant φ.

Size relations. Size relations are linear constraints between the values of a set of variables of interest between two program points. As we have seen, this allows composing the cost of the different program fragments. For each loop (or method call), COSTA infers the relation φ between the values before the loop entry (or the call) and the entry of its parent scope. Suppose that the loop (or the call) is at line L_l, its parent scope starts at line L_p, and that $\bar{v}$ are the variables of interest at L_l and $\bar{w}$ represent their values at L_p. Then we add the JML annotation

//@ ghost int $w_1 = v_1$*;. . .; int* $w_n = v_n$

immediately after line L_p to capture the values of $\bar{v}$ at line L_p, and the JML annotation

//@ assert φ

immediately before line L_l to state that the relation φ must hold at the program point.

Additional size relations inferred by COSTA are input-output size relations. These are linear constraints that relate the return value of a given method to its input values. For example, suppose that we replace "*i––*" in line 22 of the insert_sort program by "i=decrement(i)" where decrement is defined by

int decrement(int x) {return x-1;}.

Then COSTA infers the relation "$\varphi \equiv$*\result=x-1*" which is used to bound the number of iterations of that loop. In order to verify this relation in KeY we add the JML annotation "*//@ ensures* φ" to the contract of decrement:

```
/*@ public behavior
  @ requires true;
  @ ensures \result=x-1;
  @ signals_only Exception;
  @ signals (Exception) true; @*/
```

5.3 Verification of Upper Bounds using KeY

We now describe the verification techniques used in KeY to prove program correctness, focusing on those relevant to UB verification.

5.3.1 Verification by Symbolic Execution

The program logic used by KeY is *JavaCard Dynamic Logic* (JavaDL) [BHS06], a first-order dynamic logic with arithmetic. Programs are first-class citizens similar to Hoare logics but, in dynamic logic, correctness assertions can appear arbitrarily nested. JavaDL extends sorted first-order logic by a program modality $\langle\cdot\rangle\cdot$ (read "diamond"). Let p denote a sequence of executable Java statements and ϕ an arbitrary JavaDL formula, then $\langle\mathsf{p}\rangle\phi$ is a JavaDL formula which states that program p terminates and in its final state ϕ holds. A typical formula in JavaDL looks like

$$\mathsf{i} \doteq i0 \wedge \mathsf{j} \doteq j0 \;\texttt{->}\; \langle\overbrace{\mathsf{i} = \mathsf{j} - \mathsf{i}; \mathsf{j} = \mathsf{j} - \mathsf{i}; \mathsf{i} = \mathsf{i} + \mathsf{j};}^{\mathsf{p}}\rangle(\mathsf{i} \doteq j0 \wedge \mathsf{j} \doteq i0)$$

where i, j are program variables represented as *non-rigid* constants. Non-rigid constants and functions are state-dependent: their value can be changed by programs. The *rigid* constants $i0, j0$ are state-independent: their value cannot be changed. The formula above says that if program p is executed in a state where i and j have values $i0$, $j0$, then p terminates *and* in its final state the values of the variables are swapped. To reason about JavaDL formulas, KeY employs a sequent calculus whose rules perform *symbolic execution* of the programs in the modalities. Here is a typical rule:

$$\textsf{ifSplit}\ \frac{\Gamma, b \Longrightarrow \langle\texttt{\{p\}rest}\rangle\phi, \Delta \quad \Gamma, \neg b \Longrightarrow \langle\texttt{\{q\}rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\texttt{if (b) \{p\} else \{q\} rest}\rangle\phi, \Delta}$$

As values are symbolic, it is in general necessary to split the proof whenever an implicit or explicit case distinction is executed. It is also necessary to represent the *symbolic* values of variables throughout execution. This becomes apparent when statements with side effects are executed, notably assignments. The assignment rule in JavaDL looks as follows:

$$\textsf{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{x} := \mathsf{val}\}\langle\mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle\mathsf{x} = \mathsf{val};\ \mathsf{rest}\rangle\phi, \Delta}$$

The expression in curly braces in the premise is called *update* and is used in KeY to represent symbolic state changes. An *elementary* update $loc := val$ is a pair of a location (program variable, field, array) and a value. The meaning of updates is the same as that of an assignment, but they can be composed in different ways to represent complex state changes. Updates u_1, u_2 can be composed into *parallel updates* $u_1 \| u_2$. In case of clashes (updates u_1, u_2 assign different values to the same location) a last-wins semantics resolves the conflict. This reflects left-to-right sequential execution. Apart from that, parallel updates are applied simultaneously, i.e., they do not depend on each other. Update application to a formula/term e is denoted by $\{u\}e$ and forms itself a formula/term. Application of updates is similar to explicit substitutions, but is aware of aliasing.

Loops and recursive method calls give rise to infinitely long symbolic executions. Invariants are used in order to deal with unbounded program structures (an example is given below). Exhaustive application of symbolic execution and invariant rules results in formulas of the form $\{u\}\langle\rangle\phi$ where the program in the modality has been fully executed. At this stage, symbolic updates are applied to the postcondition ϕ resulting in a first-order formula that represents the weakest precondition of the executed program wrt ϕ.

To verify UBs in KeY the annotated source code files provided by COSTA are loaded. For methods where COSTA did not generate a contract, KeY provides the following default contract:

```
/*@ public behavior
  @ requires true;
  @ ensures true;
  @ signals_only Exception;
  @ signals (Exception) true; @*/
```

This contract requires to prove termination for any input and ensures that all possible execution paths are analyzed. Abrupt termination by uncaught exceptions is allowed (signals clauses). To prove that a method m satisfies its contract, a JavaDL formula is constructed which is valid iff m satisfies its contract. Slightly simplified, for insert_sort this formula (using the default contract) is:

$\forall o; \forall a0; \{\mathsf{a} := a0 \parallel \mathsf{self} := o\}(\neg(\mathsf{a} \doteq \mathsf{null}) \land \neg(\mathsf{self} \doteq \mathsf{null}) \rightarrow$

$\langle$ try{ self.insert_sort(a)@NestedLoops; }
catch(Exceptione){exc = e; }$\rangle$(exc $\doteq$ null $\vee\, instance_{\mathsf{Exception}}$(exc))

The above formula states that for any possibly value o of self and any value $a0$ of the argument a which satisfy the implicit JML preconditions (self and a are not null), the method invocation self.insert_sort(a) *terminates* (required by the use of the diamond modality) and in its final state no exception has been thrown or any thrown exception must be of type Exception.

Verification of Proof-Obligations

The proof obligation formulae must be proven valid by executing the method insert_sort symbolically starting with the execution of the variable declarations. Ghost variable declarations and assignments to ghost variables (*//@ set var=val;*) are symbolically executed just like Java assignments.

Verifying Size Relations. If a JML assertion assert φ; is encountered during symbolic execution, the proof is split: the first branch must prove that the assertion formula φ holds in the current symbolic state; the second branch continues symbolic execution. In the insert_sort example, a proof split occurs exactly before entering each loop. This verifies the size relations among variables as derived by COSTA and encoded in terms of JML assertion statements (see Section 5.2.2). Input-output size relations encoded in terms of method contracts are proven correct as outlined in Section 5.3.1.

Verifying Invariants and Ranking Functions. Verification of the loop invariants and ranking functions obtained from COSTA is achieved with a tailored loop invariant rule that has a variant term to ensure termination:

$$\mathsf{loopInv}\ \frac{\begin{array}{ll}(i) & \Gamma \Longrightarrow Inv \wedge dec \geq 0, \Delta \\ (ii) & \Gamma, \{\mathcal{U}_A\}(b \wedge Inv \wedge dec \doteq d0) \ \Rightarrow \\ & \quad \{\mathcal{U}_A\}\langle \mathsf{body}\rangle(Inv \wedge dec < d0 \wedge dec \geq 0), \Delta \\ (iii) & \Gamma, \{\mathcal{U}_A\}(\neg b \wedge Inv) \Longrightarrow \{\mathcal{U}_A\}\langle \mathsf{rest}\rangle\phi, \Delta \end{array}}{\Gamma \Longrightarrow \langle \mathsf{while\ (b)\ \{\ body\ \}\ rest}\rangle\phi, \Delta}$$

Inv and dec are obtained, respectively, from the *loop_invariant* and *decreasing* JML annotations generated by COSTA. Premise (i) ensures that invariant Inv is valid just before entering the loop and that the variant dec is non-negative. Premise (ii) ensures that Inv is preserved by the loop body and that the variant term decreases strictly monotonic while remaining non-negative. Premise (iii) continues symbolical execution upon loop exit. The integer-typed variant term ensures loop termination as it has a lower bound (0) and is decreased by each loop iteration. Using COSTA's derived ranking function as variant term obviously verifies that the ranking function is correct. The update $\mathcal{U}_A$ assigns to all locations whose values are potentially changed by the loop a fixed, but unknown value. This allows using the values of locations that are unchanged in the loop during symbolic execution of the body.

Generated Proofs. A single proof for each method is sufficient to verify the correctness of the derived loop invariants, ranking functions and size relations. The reason is that the contracts capturing the input-output size relations are not more restrictive w.r.t. the precondition than the default contracts are. Hence, with the verification of the input-output size relation contracts, we analyze all feasible execution paths and prove correctness of all loop invariants, ranking functions and JML assertion annotations. We stress that the proofs run fully automatic. Much of the time is needed to derive specific instances of arithmetic properties. As future work, we plan to do proof profiling and to reduce the search time by hashing frequently occuring normalisation steps.

5.4 Upper Bounds for Heap Manipulating Programs

When input arguments of a method are of reference type, its UB is usually not specified in terms of the concrete values within the data structures, but rather in terms of some *structural* properties of the involved data structures. For example, if the input is a list, then the UB would typically depend on the length of the list instead of the concrete values in the list.

```
1  //@ requires \acyclic(x)
2  //@ ensures \acyclic(r)
3  //@ ensures \depth(r) ≤ \depth(x) + 1
4  public static List insert(List x, int v) {
5     //@ ghost List x_0 = x;
6     List p = null;
7     List c = x;
8     List n = new List(v, null);
9     //@ ghost List c_0 = c
10     //@ assert \depth(n) = 1 ∧ \depth(c_0) = \depth(x_0)
11     //@ decreasing \depth(c)
12     //@ loop_invariant \depth(c_0) ≥ \depth(c)
13     //@ loop_invariant \acyclic(n) ∧ \acyclic(p) ∧ \acyclic(x) ∧ \acyclic(c)
14     //@ loop_invariant \disjoint({n, x})∧\disjoint({n, c}) ∧ \disjoint({n, p})
15     //@ loop_invariant !\reachPlus(p, x)∧!\reachPlus(n, x)∧!\reach(n, p)
16     while ( c != null ∧ c.data < v){
17        p = c;
18        c = c.next;
19     }
20     if ( p == null ) {
21        n.next = x;
22        x = n;
23     } else {
24        n.next = c;
25        p.next = n;
26     }
27     return x;
28  }
```

Figure 5.2: Heap manipulating running example, with (partial) JML annotations

Example 5.4.1. *Consider the program in Figure 5.2 where class* List *implements a linked list as usual. For method* insert*, COSTA infers the UB* $c_1 * \mathsf{nat}(x) + c_2$ *where* x *refers to the length of variable* x*, and* c_1/c_2 *are constants representing the cost of the instructions inside/before and after the loop. The UB depends on the length of* x*, because the list is traversed at lines 16–19.*

The example shows that cost analysis of heap manipulating programs requires inferring information on how the size of data structures changes during the execution, similar to the invariants and size-relations that are used to describe how

the values of integer variables change. To do so, we first need to fix the meaning of "size of a data structure". We use the path-length measure which maps data structures to their *depth*, such that the depth of a cyclic data structure is defined to be ∞. Recall that the depth of a data structure is the maximum number of nodes (i.e. objects) on a path from the root to a leaf. Using this size measure, COSTA infers invariants and size relations that involve both integer and reference variables, where the reference variables refer to the depth of the corresponding data structures. Once the invariants are inferred, synthesizing the UBs follows the same pattern as in Section 5.2. In the following, we identify the essential information of the path-length analysis (and related analyses) that must be verified later by KeY.

5.4.1 Path-Length Analysis

Path-length analysis [SMP10] (see also the size analysis in Section 2.3.4) is based on abstracting program states to linear constraints that describe the corresponding path-length relations between the different data structures. For example, the linear constraint $x < y$ represents all program states in which *the depth of the data structure to which* `x` *points is smaller than the depth of the data structure to which* `y` *points*. Starting from an initial abstract state that describes the path-length relations of the initial concrete state, the analysis computes path-length invariants for each program point of interest. In order to verify the path-length information inferred by COSTA using KeY, we have extended JML with the new keyword `\depth` that gives the depth of a data structure to which a reference variable points. In particular, for invariants, size-relations, and contracts, if the corresponding constraints include a variable x, corresponding to a reference variable `x`, we replace all occurrences of x by `\depth(x)`.

Example 5.4.2. *We explain the various path-length relations inferred by COSTA for the method* `insert` *of Figure 5.2, and how they are used to infer an UB. Due to space limitations, we only show the annotations of interest. For the loop at lines 16–19, COSTA infers that the depth of the data structure to which* `c` *points decreases in each iteration. Since the depth is bounded by* 0*, it concludes that* $\mathsf{nat}(c)$ *is a ranking function for that loop. As a part of the loop invariant, COSTA*

infers that $c_0 \geq c$ where c_0 refers to the depth of the data structure to which c *points before entering the loop and c to the depth of the data structure to which* c *points after each iteration. Using this invariant, together with the knowledge that the depth of c_0 equals to the depth of* x*, we have that $c_1 * \mathsf{nat}(x) + c_2$ is an UB for* insert *(since the maximum value of* c *is exactly* x*). Another essential relation inferred by the path-length analysis (captured in the* ensures *clause in line* 3*) is that the depth of the list returned by* insert *is smaller than or equal to the depth of* x *plus one. This is crucial when analyzing a method that uses* insert *since it allows tracking the size of the list after inserting an element.*

Path-length relations are obtained by means of a fixpoint computation which (symbolically) executes the program over abstract states. As a typical example, executing x=y.f adds the constraint $x' < y$ to the abstract state if the variable y *points to an acyclic* data structure, and $x' \leq y$ otherwise. On the other hand, executing x.f=y adds the constraints $\bigwedge\{z' \leq z + y \mid$ z might share with x$\}$ if it is guaranteed that x does not become cyclic after executing this statement. This is because, in the worst case, x might be a leaf of the corresponding data-structure pointed to by z, and thus the length of its new paths can be longer than the old ones at most by y. This constraint means that the depth of any reference variable that might share a common region (in the heap) with x is modified to be smaller than or equal to its previous depth plus the depth of y. Obviously, to perform path-length analysis, we require information on (a) whether a variables certainly points to an acyclic data structure; and (b) which variables might share common regions in the heap.

5.4.2 Cyclicity analysis

The cyclicity analysis of COSTA [GZ10] infers information on which variables *may* point to (a)cyclic data structures. This is essential for the path-length analysis. The analysis abstracts program states to sets of elements of the form: (1) x$\rightsquigarrow$y which indicates that starting from x one *may* reach (with at least one step) the object to which y points; (2) $\circlearrowleft^{\mathsf{x}}$ which indicates that x *might* point to a cyclic data structure; and (3) x$\diamond$y which indicates that x *might* alias with y.

Starting from an abstract state that describes the initial reachability, aliasing and cyclicity information, the analysis computes invariants (on reachability, aliasing and cyclicity) for each program point of interest by means of a fixpoint computation which (symbolically) executes the program instructions over the abstract states. For example, when executing y=x.f, then y inherits the cyclicity and reachability properties of x; and when executing x.f=y, then x becomes cyclic if before the instruction the abstract state included $\circlearrowleft^{y}$, $y \rightsquigarrow x$, or $y \diamond x$.

On the verification side, to make use of the inferred cyclicity relations, we extend JML by the new keyword *acyclic* which *guarantees* acyclicity. In contrast to COSTA, JML and KeY use shape predicates with *must*-semantics. Acyclicity information is then added in JML annotations at entry points of contracts and loops where we specify all variables which are guaranteed to be acyclic. For loop entry points as invariants (as in line 13) and for contracts as pre- and postconditions (as in lines 1, 2). To make use of the reachability relations we extend JML by the new keyword $\backslash\mathit{reachPlus}(x, y)$, which indicates that y *must* be reachable from x in at least one step, and use the standard keyword $\backslash\mathit{reach}(x, y)$ which indicates that y *must* be reachable from x in zero or more steps (i.e., they might alias). The *may*-information of COSTA about reachability and aliasing is then added as *must*-predicates in JML (in loop entries and contracts) as follows: let A be the set of judgments inferred by COSTA for a given program point, then we add $!\backslash\mathit{reachPlus}(x, y)$ whenever $x \rightsquigarrow y \notin A$, and we add $!\backslash\mathit{reach}(x, y)$ whenever $x \rightsquigarrow y \notin A \wedge x \diamond y \notin A$ (for example, in line 15).

5.4.3 Sharing analysis

Knowledge on possible sharing is required by both path-length and cyclicity analyses. The sharing analysis of COSTA is based on [SS05] where abstract states are sets of pairs of the form $x \bullet y$ which indicate that x and y might share a common region in the heap. Similarly to the path-length and cyclicity analysis, the sharing invariants are propagated from an initial state by means of a fixpoint computation to the program points of interest. For example, when executing y=x.f, the variable y will only share with anything that shared with x (including x itself); on the other hand, when executing x.f=y, the variable x keeps its previous sharing

relations, and in addition it might share with y and anything that shared with y before.

Obviously, KeY needs to know about the sharing information inferred by COSTA to verify acyclicity and path-length properties. To this end, we extended JML by the new keyword *disjoint* which states that its argument, a set of variables, *does not share* any common region in the heap (for example, in line 14).

5.5 Verification of Path-Length Assertions

As explained in the previous section, structural heap properties, including acyclicity, reachability and disjointness, are essential both for path-length analysis and for the verification of path-length assertions. However, while the path-length analysis performed by COSTA maintains cyclicity and sharing, the complementary properties are used as primitives on the verification side. The reason is that the symbolic execution machinery of KeY starts with a completely unspecified heap structure that subsequently is refined using the inferred information about acyclicity and disjointness. In the following we explain how structural heap properties are formalized in the dynamic logic (JavaDL) used in this chapter and implemented in KeY [BHS06].

5.5.1 Heap Representation

First we briefly explain the logical modeling of the heap in JavaDL.* The heap of a Java program is represented as an element of type $Heap$. The $Heap$ data type is formalized using the theory of arrays and associates locations to values. A location is a pair (o, f) of an object o and a field f. The *select* function allows to access the value of a location in a heap h by $select(h, o, f)$. The complementary update operation which establishes an association between a location (o, f) and a value val is $store(h, o, f, val)$. To improve readability, when the heap h it is clear from the context, we use the familiar notation $o.f$ and $o.f := val$ instead of select and store expressions. Based on this heap model, we define a rule for

*Note that this is *not* the heap model described in earlier publications on KeY such as [BHS06]. In the present chapter we use an explicit heap model based on [Wei11].

symbolic execution of field assignments (cf. the assign rule in Section 5.3.1). It simply updates the global heap program variable with the updated heap object:

$$\text{assign}\ \frac{\Gamma \Longrightarrow \{\mathsf{heap} := \mathit{store}(\mathsf{heap}, \mathsf{o}, \mathsf{f}, \mathsf{v})\}\langle \mathsf{rest}\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathsf{o.f} = \mathsf{v};\ \mathsf{rest}\rangle\phi, \Delta}$$

5.5.2 Predicates for Structural Heap Properties

For the sake of readability, in Section 5.4, we gave simplified versions of the predicates *depth*, *acyclic*, *reach*, *reachPlus* and *disjoint* as compared to the actual implementation. In reality, these predicates have an extra argument that restricts their domain to a given set of fields. For example, instead of $\backslash\mathit{depth}(x)$ we might actually have $\backslash\mathit{depth}(\{x.\mathsf{next}\}, x)$ which refers to the depth of x considering only those paths that go through the field next. A syntactic analysis infers automatically a safe approximation of these sets of fields by taking the fields explicitly used in the corresponding code fragment.

Ultimately, the various structural heap properties are reduced to reachability between objects which, therefore, must be expressible in the underlying program logic. The counterpart of JML's *reach* predicate in Java DL is

$$\backslash\mathit{reach} : \mathit{Heap} \times \mathit{LocSet} \times \mathit{Object} \times \mathit{Object} \times \mathit{int}$$

and expresses *bounded reachability* (or n-reachability): an object e is n-reachable from an object s with respect to a heap h and a set of locations l (of type LocSet) if and only if there exists a sequence $s = o_1 o_2 \cdots o_n = e$ where $o_{i+1} = o_i.f_i$ and $(o_i, f_i) \in l$ for all $0 < i < n$. The predicate $\backslash\mathit{reach}(h, l, s, e, n)$ is formally defined as $n \geq 0 \wedge s \neq \mathsf{null} \wedge ((n \doteq 0 \wedge s \doteq e) \vee \exists f.(o, f) \in l \wedge \backslash\mathit{reach}(h, l, s.f, e, n-1))$. As a consequence, from null nothing is reachable and also null cannot be reached.

Location sets in Java DL are formalized in the data type LocSet which provides constructors and the usual set operations (see [Wei11] for a full account). Here we need only three location set constructors: the constructor *empty* for the empty set, the constructor $\mathit{singleton}(o, f)$ which takes an object o and a field f and constructs a location set with location (o, f) as its only member, and the constructor $\mathit{allObjects}(f)$ which stands for the location set $\{(o, f) \mid o \in \mathit{Object}\}$.

Example 5.5.1. $\backslash\mathit{reach}(h, \mathit{allObjects}(\mathit{next}), \mathit{head}, \mathit{last}, 5)$ *is evaluated to true iff the object last is reachable from object head in five steps by a chain of* next *fields.*

Based on \reach we could directly axiomatize structural heap predicates such as \acyclic$(h, l, o)$ or \disjoint$(h, l, o, u)$. Instead we prefer to reduce structural heap predicates to \reachPlus$(h, l, o, u)$ which is the counterpart of the JML function of the same name in Section 5.4.2 and expresses reachability in at least one step. This has several advantages over using \reach: (1) the definition of predicates such as \acyclic does not use the step parameter of the \reach predicate and one would use existential quantification to eliminate it which impedes automation; and (2) for \reachPlus$(h, l, o, u)$ to hold one has to perform at least one step using a location in l. This renders the definition of properties such as \acyclic less cumbersome as the zero step case has been excluded.

The predicate \reachPlus can defined with the help of \reach and this definition can be used if necessary, however, in the first place we use a separate axiomatization of \reachPlus. This helps to avoid (or at least to delay as long as possible) the reintroduction of the step parameter and, hence, an additional level of quantification. We describe in the following section one central difficulty that arises when reasoning about structural heap properties and how we solved it to achieve higher automation.

5.5.3 Field Update Independence

When reasoning about structural heap predicates one often ends up in a situation where one has to prove that a heap property is still valid after updating a location on the heap, i.e, after executing one or several field assignments. For instance, we might know that \acyclic$(h, l, u)$ holds and have to prove that after executing the assignment o.f=v; the formula \acyclic$(store(h, o, f, v), l, u)$ holds.

A precise analysis of the effect of a field update is expensive and makes automation significantly harder. As it is common in this kind of situation, it helps to *optimize the common case*. In the present context, this means to decide in most cases efficiently that a field assignment does not effect a heap property at all. This is sufficiently achieved by two simple checks:

1. The expression $singleton(o, f) \subseteq l$ checks whether an updated location $o.f$ is in the location set l of the heap property to be preserved. This turns out

to be inexpensive for most (if not all) practically occurring cases. Whenever this check fails, the resulting *store* can be removed from the argument of the heap property. For instance, an assignment o.data=5 to the data field of a list does not change the list structure which depends solely on the next field. In that case we can rewrite $\backslash\mathit{acyclic}(\mathit{store}(h, \mathsf{o}, \mathsf{data}, 5), l, u)$ to $\backslash\mathit{acyclic}(h, l, u)$.

2. To check whether an object o whose field has been updated is reachable from one of the other mentioned objects, is more expensive than the previous one, but still cheaper than a full analysis. For example, we can check whether the object o is reachable from object u in case of $\backslash\mathit{acyclic}(\mathit{store}(h, o, f, v), l, u)$. If the answer is negative we can again discard the store expression.

5.5.4 Path-Length Axiomatization

In general, the JML assertions generated by COSTA refer to the path-length of a data structure o as $\backslash\mathit{depth}(l, o)$ where l is the location set restricting the depth to certain locations. This JML function is mapped to the Java DL function $\backslash\mathit{depth}(h, l, o)$ which is evaluated to the maximal path-length of o in heap h using only locations from l. Its axiomatization is based on the n-reachability predicate $\backslash\mathit{reach}$ expressing that there exists an object u reachable in $\backslash\mathit{depth}(h, l, o)$ steps and that there is no object z reachable from o in more than $\backslash\mathit{depth}(h, l, o)$ steps. This definition is not used by default by the theorem prover, instead, automated proof search relies mainly on a number of lemmas that state more useful higher-level properties. For instance, given a term like $\backslash\mathit{depth}(\mathit{store}(h, o, f, v), l, u)$ there is a lemma which checks that o is reachable from u and some acyclicity requirements. If that is positive then the lemma allows us to use the same approximation for $\backslash\mathit{depth}$ in case of a heap update as detailed in Section 5.4.1.

5.6 Experimental Evaluation

The implementation of our approach has required the following non-trivial extensions to COSTA and KeY (note that COSTA works on Java bytecode, and

Bench	COSTA					KeY			Total
	$\mathbf{T}_{size}$	$\mathbf{T}_{inv}$	$\mathbf{T}_{rf}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	**Nodes**	**Branches**	$\mathbf{T}_{ver}$	
slm	22	20	26	112	4	3641	36	6700	6816
nlf	30	16	24	106	6	5665	37	2800	2912
bubsort	38	24	144	296	14	14890	230	57800	58110
inssort	30	12	46	142	6	9875	167	29300	29448
selsort	40	20	112	232	8	12564	209	40700	40940
pastri	66	38	138	394	14	29723	337	110100	110508

Table 5.1: Statistics about integer manipulating programs

KeY on Java source): (1) output the proof obligations using the original variable names (at the bytecode level, operand stack variables are often used); (2) place the obligations in the Java source at the precise program points where they must be verified (entry points of loops); (3) finding a suitable JML format for representing proof obligations on UBs has required a considerable number of iterations (defining ghost variables, introducing *assert* constructs, etc.); (4) implement the JML *assert* construct in KeY which was not supported hitherto.

Regarding heap manipulating programs, it requires the following extensions to both COSTA and KeY: (1) generate and output in COSTA the JML annotations *depth*, *acyclic* and *disjoint* so that KeY can parse them; (2) synthesize suitable proof obligations in Java DL that ensure correctness of the resource analysis; (3) axiomatize the JML *depth*, *acyclic* and *disjoint* functions in KeY as described in Section 5.5 and implement heuristics for automation; and (4) implement heuristic checks in KeY that allow fast verification of the common case as described in Section 5.5.4.

Table 5.1 shows some experiments using a set of representative programs, which include sorting algorithms for integer manipulating programs, namely bubble sort (bubsort), insert sort (inssort), and selection sort (selsort); a method to generate a Pascal Triangle (pastri); simple (slm) and nested loops (nlf). All times are measured in ms and were obtained using an Intel Core2 Duo at 2.53GHz with 4Gb of RAM running a Linux 2.6.32. Table 5.2 shows our experiments using a set of representative programs that perform common list operations as well as searching for an element in a binary tree. Columns $\mathbf{T}_{size}$, $\mathbf{T}_{inv}$ and $\mathbf{T}_{rf}$

Bench	COSTA			KeY			Total
	$\mathbf{T}_{heap}$	$\mathbf{T}_{ana}$	$\mathbf{T}_{jml}$	**Nodes**	**Branches**	$\mathbf{T}_{ver}$	
traverse	14	36	2	1208	52	2300	2338
create	54	150	8	1499	47	3100	3258
insert	282	374	16	19252	636	40800	41190
indexOf	26	86	4	2439	67	5900	5990
reverse	72	130	8	14206	673	20900	21038
array2List	62	154	8	1457	37	2600	2762
copy	76	132	10	14147	673	22600	22742
searchtree	142	202	6	2389	97	3700	3908

Table 5.2: Statistics about heap manipulating programs

(in Table 5.1), show, respectively, the times taken by COSTA to obtain the size relations, loop invariants and ranking functions. Column $\mathbf{T}_{heap}$ (in Table 5.2) shows the time taken by COSTA to perform the heap analysis (cyclicity, sharing and path-length). Columns $\mathbf{T}_{ana}$ and $\mathbf{T}_{jml}$ show the time taken to perform the whole analysis (which includes the previous times) and to generate the JML annotations. Column $\mathbf{T}_{ver}$ shows the time taken by KeY in order to verify the JML annotations generated by COSTA. As time measurements for Java are imprecise, we state in addition the number of nodes and branches of the generated proof to provide some insight on the proof complexity. Column **Total** shows the time taken by the whole process.

Our preliminary experiments show already that a proof-carrying code approach to verified resource guarantees can be fully automatic using COSTA and KeY. In our framework the code originating from an untrusted *producer* should be bundled with the proof generated by COSTA + KeY for a given resource consumption. A notable result of our experiments is that KeY was able to spot a bug in COSTA, as it failed to prove correct one invariant which was incorrect. In addition, KeY could provide a concrete counterexample that helped understand, locate and fix the bug, which was related to a recently added feature of COSTA.

5.7 Related work

We have demonstrated that automatic verification of the upper bounds inferred by COSTA using KeY is feasible. Instead of verifying the correctness of the underlying static analysis, we take the alternative approach of verifying the correctness of their results. Interestingly, this approach, though weaker in principle than verification of the analyzer, has advantages in the context of mobile code. Following proof-carrying-code [Nec97] principles, code originating from an untrusted *producer* can be bundled together with the proof generated by KeY for its declared resource consumption. This way, the code *consumer* can check locally and automatically using KeY whether the claimed resource guarantees are verified.

Many software verification tools including KeY [BHS06], Why [FM07], VeriFast [SJPW08], or Dafny [Lei10] rely on automatic theorem proving technology. While most of these systems are expressive enough to model and prove heap properties of programs, such proofs are far from being automatic. The main reason is that functional verification of heap properties requires complex invariants that cannot be found automatically. In addition, automated reasoning over heap-allocated symbolic data is far less developed than reasoning over integers or arrays.

With this work we also show that the automation built into a state-of-the-art verification system is sufficient to reason successfully about resource-related heap properties. The main reasons for this are: (a) the required invariants are inferred automatically in the resource analysis stage; (b) a limited and carefully axiomatized signature for heap properties expressed in logic is used. This confirms the findings of the SLAM project [BBL+10] that existing verification technology can be highly automatic for realistic programs and a restricted class of properties.

There exist several other cost analyzers, like [GMC09, HH10], that automatically infer resource guarantees for different programming languages. However, none of them formally prove the correctness of the UBs they infer. An exception is [CW00], which verifies and certifies resource consumption (for a small programming language and not for heap properties). For the particular case of memory resources, [DP11] formally certifies the correctness of the static analyzer. We have taken the alternative approach of certifying the correctness of the upper bounds

that the tool generates. This is not only much simpler, but has the additional advantage that the generated proofs can act as resource certificates.

Chapter 6

Concurrency: Object-Sensitive Cost Analysis for Concurrent Objects

This chapter presents our work on cost analysis for concurrent objects which adds to the analysis described in [AAG+11] object-sensitivity to separate the cost in different cost centers. This work is under revision for the special issue of *QAPL'12* to be published in the journal *Theoretical Computer Science*.

6.1 Introduction

Distribution and concurrency are currently mainstream. The Internet and the broad availability of multi-processors radically influence software. Many standard desktop programs have to deal with distribution aspects like network transmission delay and failure. Furthermore, many chip manufactures are turning to multicore processor designs as a way to increase performance in desktop, enterprise, and mobile processors. This brings renewed interest in developing both new concurrency models and associated programming languages techniques that help in understanding, analyzing, and verifying the behavior of concurrent and

distributed programs.

One of the most important features of a program is its resource consumption. By resource, we mean not only traditional cost measures (e.g., number of executed instructions, or memory consumption) but also concurrency-related measures (e.g., number of tasks spawned, number of requests to remote servers). Automatically inferring the resource usage of concurrent programs is challenging because of the inherent complexity of concurrent behaviors.

In addition to traditional applications, like optimization [Weg75], verification and certification of resource consumption [CW00], cost analysis opens up interesting applications in the context of concurrent programming. In general, having anticipated knowledge on the resource consumption of the different components which constitute a system is useful for distributing the load of work. Upper bounds can be used to predict that one component may receive a large amount of remote requests, while other siblings are idle most of the time. Also, our framework allows instantiating the different component?s with the particular features of the infrastructure on which they are deployed. Then, analysis can be used to detect the components that consume more resources and may introduce bottlenecks. Lower bounds on the resource usage can be used to decide if it is worth executing locally a task or requesting remote execution.

In order to develop our analysis, we consider a concurrency model based on the notion of concurrently running (groups of) objects, similar to the actor-based and active-objects approaches [SPH10, SM08]. These models take advantage of the concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. Concurrent objects communicate via *asynchronous* method calls. Intuitively, each concurrent object is a monitor and allows at most one *active* process to execute within the object. Scheduling among the processes of an object is *cooperative*, i.e., a process has to release the monitor lock explicitly, except for termination. Each object has an unbounded set of pending processes. In case the lock of a concurrent object is free, any process in the set of pending processes can grab the lock and start to execute.

6.1.1 Organization of the Chapter

The remainder of the chapter is organized as follows. Section 6.2 presents the concurrency model we consider in order to develop our analysis. Section 6.3 defines the notion of cost for the concurrent distributed programs that we aim at approximating by means of the resource analysis.

In Section 6.4, we describe the *field-sensitive* size analysis for the concurrent setting presented at [AAG+11]. Section 6.5 adapts the object-sensitive points-to analysis of Milanova [MRR05, SBL11] to our setting. Then, the points-to information gathered by the analysis allows us to define in Section 6.6 object-sensitive recurrence relations which, together with the size abstractions, constitute the core of our analysis. Section 6.7 presents the experimental results obtained by applying our approach over a set of typical applications of concurrent and distributed programming. Finally, Section 6.8 reviews the related work.

6.2 A Language with Concurrent Objects

The concurrency model of Java and C# is based on threads that share memory and are scheduled preemptively, i.e., they can be suspended or activated at any time. To avoid undesired interleavings, low-level synchronization mechanisms such as locks have to be used. Thread-based programs are error-prone, difficult to debug, verify and maintain. In order to overcome these problems, several higher-level concurrency models that take advantage of the inherent concurrency implicit in the notion of object have been developed [SPH10, SM08, JO07, dBCJ07, Mey97]. They provide simple language extensions that allow programming concurrent applications with relatively little effort. Concurrent objects [JO07, dBCJ07] form today a well established high-level model for distributed concurrent systems.

6.2.1 The Concurrency Model

We develop our analysis on the imperative subset of the ABS language [JHS+12], a simple imperative language with concurrent objects. However, our techniques

work for some other languages that use actors (e.g., there are implementations of actor libraries for Scala, Java, Erlang, among others). The central concept of this concurrency model is that of *concurrent object*. Conceptually, each object has a dedicated processor and encapsulates a *local heap* which is not accessible from outside this object, i.e., all fields are always accessed using the *this* object, and any other object can only access such fields through method calls. Concurrent objects live in a distributed environment with asynchronous and unordered communication by means of asynchronous method calls. Thus, an object has a set of tasks (i.e., calls) to execute and, among them, at most one task is *active* and the others are *suspended* on a task queue.

Process scheduling is by default non-deterministic, but controlled by *processor release points* and *future variables* in a cooperative way. After asynchronously calling $f := o \,!\, m(\overline{e})$ (object o invokes asynchronously method m with arguments $\overline{e}$), the caller may proceed with its execution without blocking on the call. Here f is a future variable which refers to a return value that has yet to be computed. There are two operations on future variables, which control external synchronization. First, `await` f? suspends the active task (allowing other tasks in the object to be scheduled) until the future variable f has been assigned a value. Second, the value stored in f can be retrieved using f.`get`, which blocks all execution in the object until f gets a value (in case it has not been assigned a value yet). It is possible to unconditionally release the processor by means of a `release` instruction (not used in the running example) which suspends the current task and lets a pending task in.

Example 6.2.1. *Figure 6.1 shows the source code of our running example which implements a simple* file input stream *(defined in class* FileIS*) that provides two different ways of processing a file. The class contains three fields (defined as class parameters) which represent, respectively, the name of the file* fp*, the length of the file* lth *and the size of the block to be read from the field* blockS*. Method* readBlock *reads file* fp *block by block (of sizes* blockS*) and sums the values retrieved using* `get`*. Method* readOnce *reads the whole file in just one invocation to* readContent*. The latter method calls method* process *of class* Reader *which reads and processes* elems *elements of the file starting at position* pos*. Method* hdRead *represents the*

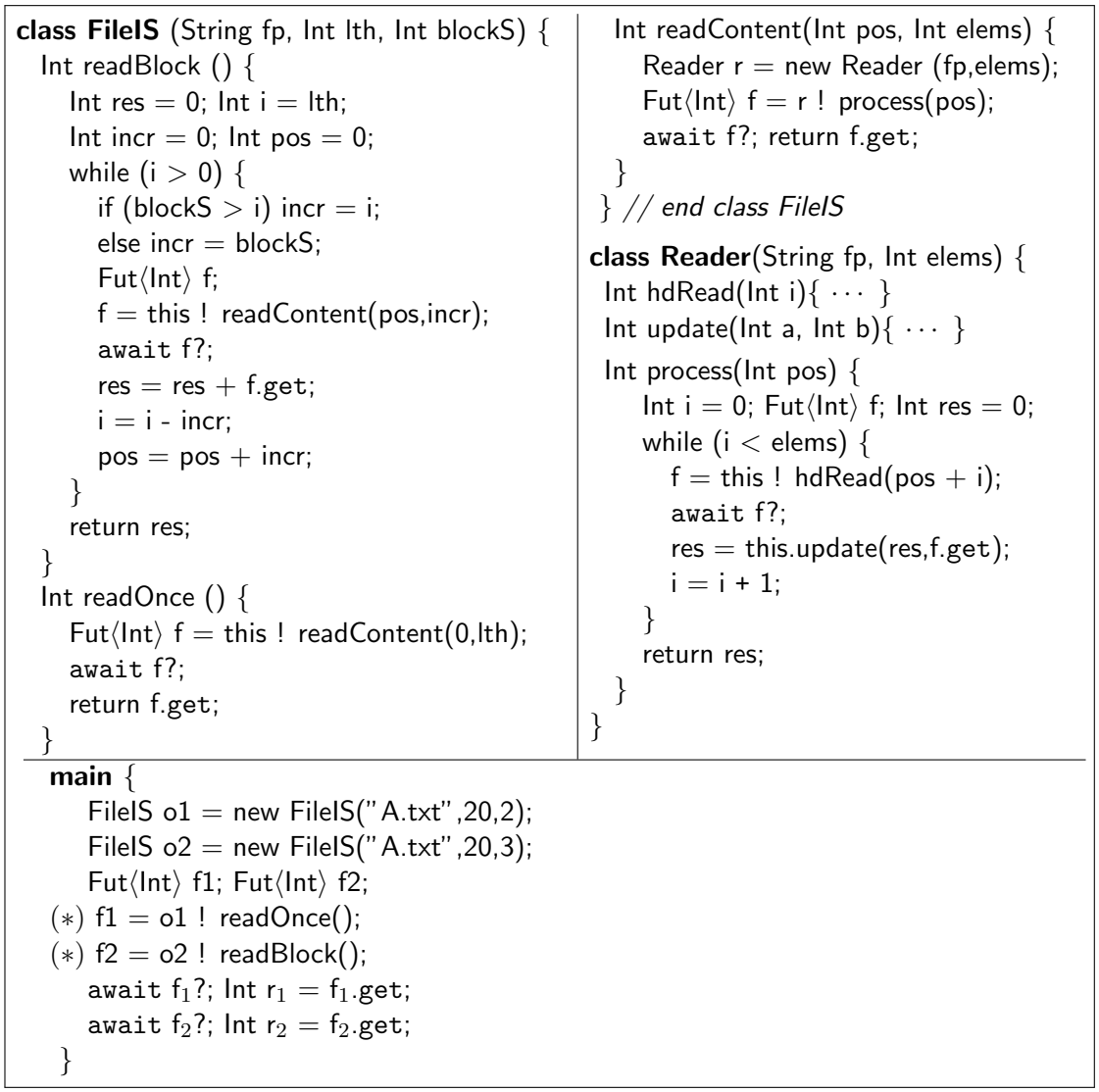


Figure 6.1: Running Example

low-level access to the hard-disk and method update *performs some arithmetic operation on its arguments and returns an integer value. We do not show the code of these methods as they are not relevant for the purpose of the analysis presentation.*

6.2.2 A Rule-based Representation for Concurrent Objects

To formalize the resource analysis of concurrent objects, we use an intermediate representation similar to the RBR of Chapter 2. In the sake of clarity, we ignore some rules that are not relevant for our purpose (like arrays and some arithmetic and comparison operators) and we include the concurrency primitives. The new grammar is:

r ::= $m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n.$

b ::= $x{:=}e \mid this.f{:=}e \mid x{:=}\mathsf{new}\ C \mid \mathsf{call}(ct, m(rec, \bar{x}, \bar{y})) \mid \mathtt{await}\ g \mid \mathtt{release} \mid x{:=}y.\mathsf{get}$

g ::= $true \mid g \wedge g \mid x? \mid e\ op\ e$

e ::= $\mathsf{null} \mid a$

a ::= $x \mid n \mid a{-}a \mid a{+}a \mid a{*}a \mid a/a$

where $op{\in}\{>, =, \geq\}$, $m(this, \bar{x}, \bar{y})$ is the *head* of the rule, *this* is the identifier of the object on which the method is executing, g specifies the conditions for the rule to be applicable, and, $b_1, \ldots, b_n$ is the rule's *body*. One aspect that is different from the syntax of Chapter 2 is that calls are of the form $\mathsf{call}(ct, m(rec, \bar{x}, \bar{y}))$ where $ct \in \{\mathtt{m}, \mathtt{b}\}$ which allows us to distinguish between calls to methods and intermediate blocks (like *while* or *if-then-else* blocks); *rec* is a variable that refers to the receiver object; the variables $\bar{x}$ (resp. $\bar{y}$) are the formal parameters (resp. return values). For intermediate blocks, *rec* is always *this*. For methods, $\bar{y}$ is either empty or contains a single output variable. Future variables ($x?$) can be used in `await` instructions but not in rule guards. An instruction $\mathsf{new}\ \mathsf{C}(\bar{\mathsf{t}})$ is represented in the RBR by $\mathsf{new}\ C$ followed by a call to the class constructor with the corresponding parameters $\bar{\mathsf{t}}$. The translation from the high-level programs to the RBR is (almost) identical to the translation of Java (bytecode) to the RBR explained in Section 2.2, and thus we skip the details and illustrate it by an example.

Example 6.2.2. *Figure 6.2 depicts the RBR (left) and the CFG (right) of method* readBlock*. Loops are extracted in separate CFGs to enable compositional cost analysis (e.g., the CFG at the bottom is the one for the while loop). The method is represented by four procedures, readBlock, while, if and if^c, which have a correspondence with blocks in the CFG and the entry to the loop. Each procedure*

$readBlock(\langle this\rangle, \langle r\rangle) \leftarrow$
$\quad res := 0, i := this.lth,$
$\quad incr := 0, pos := 0,$
$\quad \mathsf{call}(\mathtt{b}, while(\overline{inp}, \overline{out})),$
$\quad r := res.$
$while(\overline{inp}, \overline{out}) \leftarrow i \leq 0.$
$while(\overline{inp}, \overline{out}) \leftarrow i > 0,$
$\quad \mathsf{call}(\mathtt{b}, if(\overline{inp}, \overline{out})).$
$if(\overline{inp}, \overline{out}) \leftarrow this.blockS > i,$
$\quad incr := i,$
$\quad \mathsf{call}(\mathtt{b}, if^c(\overline{inp}, \overline{out})).$
$if(\overline{inp}, \overline{out}) \leftarrow this.blockS \leq i,$
$\quad incr := this.blockS,$
$\quad \mathsf{call}(\mathtt{b}, if^c(\overline{inp}, \overline{out})).$
$if^c(\overline{inp}, \overline{out}) \leftarrow$
$\quad \mathsf{call}(\mathtt{m}, readContent(\langle this, pos, incr\rangle, \langle f\rangle)),$
$\quad \mathtt{await}\ f?, v := f.\mathtt{get},$
$\quad res := res + v, i := i - incr,$
$\quad pos := pos + incr,$
$\quad \mathsf{call}(\mathtt{b}, while(\overline{inp}, \overline{out})).$

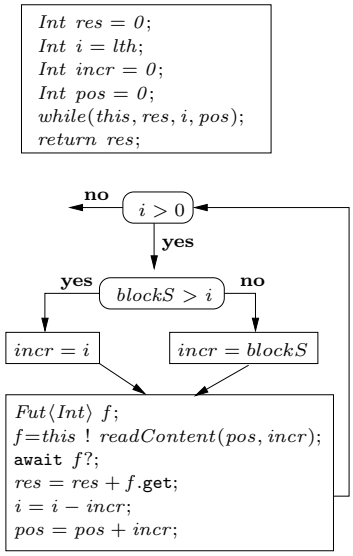


Figure 6.2: The RBR and CFG for method readBlock

is defined by means of guarded rules. $\overline{inp}$ *stands for* $\langle this, res, i, incr, pos\rangle$ *and* $\overline{out}$ *for* $\langle res, i, incr, pos\rangle$*. Guards in rules state the conditions under which the corresponding blocks in the CFG can be executed. When there is more than one successor in the CFG, we create a* continuation procedure *and a corresponding call in the rule. Blocks in the continuation will in turn be defined by means of (mutually exclusive) guarded rules. As a result of the translation, observe that all forms of iteration in the program are represented by means of* recursive *calls. The unique parameter of the procedure* readBlock *is the reference to this object. When calling a block, we pass as arguments all local variables that are needed in the block. The heap remains implicit.*

6.2.3 Operational Semantics

An execution state (or *configuration* S) has the form $\{\!|a_1, \ldots, a_n|\!\}$, where a_i can be either an *object*, a *future* event, or a method invocation. An object is of the form $\mathtt{ob}(o, C, h, \langle tv, s\rangle, \mathcal{Q})$, where o is the *object identifier*, C is its class name, h is its *heap*, tv is its *table of local variables*, s is a sequence of instructions to be executed by the current task, and $\mathcal{Q}$ is the set of pending tasks. A heap h maps *field names* (declared in C) to $\mathbb{V} = \mathbb{Z} \cup \{\mathsf{null}\} \cup \mathit{Objects}$, where *Objects* denotes the set of object identifiers. A table of variables tv maps local variables to $\mathbb{V}$. It contains the special entry `destiny` to associate the return variable of a method to the corresponding future variable.

Future events have the form $\mathtt{fut}(\mathtt{fn}, v)$ where $v \in \mathbb{V} \cup \bot$. The symbol $\bot$ indicates that `fn` does not have a value yet. A method invocation is of the form $\mathsf{invoke}\ (m(o, \bar{x}), tv, \mathtt{fn})$, where `fn` is the future variable in which the return value should be stored. For simplicity, we assume that all methods return a value. The *operational semantics* is given in a rewriting-based style, where, at each step, a subset of the state is rewritten according to the following rules:

$$(1)\ \frac{v = \mathit{eval}(e, h, tv, \mathit{Obj}),\ x \in \mathit{dom}(tv)}{\{\!|\mathtt{ob}(o, C, h, \langle tv, x := e \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, \langle tv[x \mapsto v], s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\}}$$

$$(2)\ \frac{v = \mathit{eval}(e, h, tv, \mathit{Obj})}{\{\!|\mathtt{ob}(o, C, h, \langle tv, \mathit{this}.f := e \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h[f \mapsto v], \langle tv, s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\}}$$

$$(3)\ \frac{o' \text{ is fresh, } h' \text{ is an empty heap}}{\begin{array}{c}\{\!|\mathtt{ob}(o, C, h, \langle tv, x := \mathsf{new}\ D \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\} \rightsquigarrow \\ \{\!|\mathtt{ob}(o, C, h, \langle tv[x \mapsto o'], s\rangle, \mathcal{Q}), \mathtt{ob}(o', D, h', \epsilon, \emptyset) | \mathit{Obj}|\!\}\end{array}}$$

$$(4)\ \frac{\begin{array}{c}ct \in \{\mathtt{m}, \mathtt{b}\}, r \equiv p(o, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \ll_{tv} P, \\ tv' \text{ is a default mapping over } \mathit{vars}(r) \setminus (\bar{x} \cup \bar{y}),\ \mathit{eval}(g, h, tv \cup tv', \mathit{Obj}) = \mathit{true}\end{array}}{\begin{array}{c}\{\!|\mathtt{ob}(o, C, h, \langle tv, \mathsf{call}(ct, p(o, \bar{x}, \bar{y})) \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\} \rightsquigarrow \\ \{\!|\mathtt{ob}(o, C, h, \langle tv \cup tv', b_1, \ldots, b_n \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\}\end{array}}$$

$$(5)\ \frac{o'' = \mathit{eval}(o', h, tv, \mathit{Obj}), \bar{v} = \mathit{eval}(\bar{x}, h, tv, \mathit{Obj}), \mathtt{fn} \text{ is a fresh future name}}{\begin{array}{c}\{\!|\mathtt{ob}(o, C, h, \langle tv, \mathsf{call}(\mathtt{m}, m(o', \bar{x}, y)) \cdot s\rangle, \mathcal{Q}) | \mathit{Obj}|\!\} \rightsquigarrow \\ \{\!|\mathtt{ob}(o, C, h, \langle tv[y \mapsto \mathtt{fn}], s\rangle, \mathcal{Q}), \mathsf{invoke}\ (m(o'', \bar{v}), \mathtt{fn}), \mathtt{fut}(\mathtt{fn}, \bot) | \mathit{Obj}|\!\}\end{array}}$$

$$(6)\ \frac{\begin{array}{c} r \equiv m(this, \bar{x}, y) \leftarrow g, b_1, \ldots, b_n \ll P,\ tv' \text{ is a default mapping over } vars(r), \\ tv'' = tv'[this \mapsto o, \bar{x} \mapsto \bar{v}], eval(g, h, tv'', Obj) = true \end{array}}{\begin{array}{c} \{\!|\mathtt{ob}(o, C, h, \langle tv, s\rangle, \mathcal{Q}), \mathsf{invoke}\ (m(o, \bar{v}), \mathtt{fn})|Obj|\!\} \rightsquigarrow \\ \{\!|\mathtt{ob}(o, C, a, \langle tv, s\rangle, \{\langle tv'', b_1, \ldots, b_n\rangle|\mathcal{Q}\})|Obj|\!\} \end{array}}$$

$$(7)\ \frac{tv(\mathtt{destiny}) = (y, \mathtt{fn})}{\{\!|\mathtt{ob}(o, C, h, \langle tv, \epsilon\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, \bot)|Obj|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, \epsilon, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v)|Obj|\!\}}$$

$$(8)\ \frac{\mathtt{fn} = tv(y),\ v \neq \bot}{\{\!|\mathtt{ob}(o, C, h, \langle tv, x := y.\mathsf{get}\ \cdot s\rangle, \mathcal{Q}), \mathtt{fut}(\mathtt{fn}, v)|Obj|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, \langle tv, x := v \cdot s\rangle, \mathcal{Q})|Obj|\!\}}$$

$$(9)\ \frac{eval(g, h, tv, Obj) = true}{\{\!|\mathtt{ob}(o, C, h, \langle tv, \mathtt{await}\ g \cdot s\rangle, \mathcal{Q})|Obj|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, \langle tv, s\rangle, \mathcal{Q})|Obj|\!\}}$$

$$(10)\ \frac{eval(g, h, tv, Obj) = false}{\{\!|\mathtt{ob}(o, C, h, \langle tv, \mathtt{await}\ g \cdot s\rangle, q)|Obj|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, \langle tv, \mathtt{release} \cdot \mathtt{await}\ g \cdot s\rangle, \mathcal{Q})|Obj|\!\}}$$

$$(11)\ \frac{}{\{\!|\mathtt{ob}(o, C, h, \langle tv, \mathtt{release} \cdot s\rangle, \mathcal{Q})|Obj|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, \mathsf{idle}, \{\!|\langle tv, s\rangle|\mathcal{Q}|\!\})|Obj|\!\}}$$

$$(12)\ \frac{b = \epsilon\ or\ b = \mathsf{idle}}{\{\!|\mathtt{ob}(o, C, h, b, \mathcal{Q})|Obj|\!\} \rightsquigarrow \{\!|\mathtt{ob}(o, C, h, s, \mathcal{Q} - \{\!|s|\!\})|Obj|\!\}}$$

Let us intuitively explain the semantics. Function $eval(e, h, tv, Obj)$ evaluates an expression e using the heap h and the table of variables tv in the standard way. If e is a future, then it uses Obj to see if it has a value. The notation $tv[x \mapsto v]$ (resp. $h[f \mapsto v]$) in the first two rules stands for storing v in the local variable x (resp. field f). In rule 3, it can be observed that an instruction new creates a new object initially empty. We assume that, after creating the object, the scheduler always picks the task associated to the constructor before selecting any other task. In rule 4, a call to a block is resolved by finding a matching rule and adding (a renamed apart version of) its body to the sequence of instructions to be executed.

The most important points in the semantics are: (a) the treatment of method invocations and future variables (rules 5-7), and, (b) synchronization operations

(rules 8-12). As regards to (a), when we find an asynchronous call in rule 5, we create a (fresh) future variable `fn` on which the result will be returned and establish the link between the return variable y of the method and `fn` by means of an assignment instruction. The fact that the asynchronous call reaches the object occurs in rule 6, where the asynchronous call is dequeued for execution, and the destiny future variable is stored in the table of local variables. Then in rule 7, when the corresponding method finishes execution, the future variable is updated with the returned value. As regards (b), the instruction `get` blocks the execution until the future variable has a value in 8. If the evaluation of the guard in an `await` instruction succeeds, the execution continues in rule 9. If it fails, the processor is released (rule 10) to allow another task to become active. This can be seen in rule 11 in which the task becomes idle. In rule 12 another task is dequeued (because the current one terminated or released the processor).

We assume that executions start from a main method. Thus, the *initial configuration* is of the form $\{\!|\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv, \bar{b}\rangle, \emptyset)|\!\}$ where local variables in tv are initialized to default values. The execution ends in a *final configuration* S in which all events are either future events or objects of the form $\mathtt{ob}(o, C, h, \epsilon, \emptyset)$. Execution proceeds non-deterministically by applying the above execution steps. When there is no rule to apply the execution stops. Executions can be regarded as *traces* of the form $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_n$ where S_n is a final configuration.

Example 6.2.3. *Consider the* main *method of the running example (Figure 6.1). After executing the constructors we reach a configuration with three objects:*
$\{\!|\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}, bc\rangle, \emptyset), \mathtt{ob}(o_1, FileIS, h_{o_1}, \epsilon, \emptyset), \mathtt{ob}(o_2, FileIS, h_{o_2}, \epsilon, \emptyset)|\!\}$
where bc corresponds to the sequence of instructions from (*) *on. After processing both asynchronous calls consecutively, the new state takes the form:*

$\{\!|$ invoke $(readOnce(o_1, f_1), \mathtt{fn}_1), \mathtt{ob}(o_1, FileIS, h_{o_1}, \epsilon, \emptyset), \mathtt{fut}(\mathtt{fn}_1, \bot),$
invoke $(readBlock(o_2, f_2), \mathtt{fn}_2), \mathtt{ob}(o_2, FileIS, h_{o_2}, \epsilon, \emptyset), \mathtt{fut}(\mathtt{fn}_2, \bot),$
$\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}[f_1 \mapsto \mathtt{fn}_1, f_2 \mapsto \mathtt{fn}_2], bc'\rangle, \emptyset)$ $|\!\}$

The application of rule (6) to the first two elements of the above state removes the invoke event and introduces $\langle tv_{o_1}, body\rangle$ in the queue of o_1, where body is the body (possibly renamed) of method readOnce. *Furthermore, tv_{o_1} stores the assignment $tv(\mathtt{destiny}) = (f_1, \mathtt{fn}_1)$. When this event is extracted from the queue of o_1 (rule*

(12)), its complete processing will replace $\mathtt{fut}(\mathtt{fn}_1, \bot)$ *by* $\mathtt{fut}(\mathtt{fn}_1, v)$ *(rule(7)), where* v *is the value returned by the method* readOnce*. Note then that rule (9) can be used to process the instruction* **await** f_1? *of the object* `main`*. At this point the new state will take this form:*

$$\{\!|\ \mathtt{fut}(\mathtt{fn}_1, v), \mathtt{ob}(o_1, \mathit{FileIS}, h_{o_1}, \epsilon, \emptyset),$$
$$\mathsf{invoke}\ (\mathit{readBlock}(o_2, f_2), \mathtt{fn}_2), \mathtt{ob}(o_2, \mathit{FileIS}, h_{o_2}, \epsilon, \emptyset), \mathtt{fut}(\mathtt{fn}_2, \bot),$$
$$\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv_{\mathtt{main}}[f_1 \mapsto \mathtt{fn}_1, f_2 \mapsto \mathtt{fn}_2], bc'' \rangle, \emptyset)\ |\!\}$$

6.3 Cost and Cost Models for Concurrent Programs

We now define the notion of cost for concurrent programs that we aim at approximating. An execution step is annotated as $S \rightsquigarrow_o^b S'$, which denotes that we move from a state S to a state S' by executing instruction b in object o. Note that from a given state there may be several possible execution steps that can be taken since we have no assumptions on task scheduling. In order to quantify the cost of an execution step, we use a generic cost model $\mathcal{M} : \mathit{Ins} \mapsto \mathbb{R}$ which maps instructions built using the grammar in Section 6.2.2 to real numbers. The cost of an execution step is defined as $\mathcal{M}(S \rightsquigarrow_o^b S') = \mathcal{M}(b)$.

In the execution of sequential programs, the *cumulative cost of a trace* is obtained by applying a given cost model to each step of the trace. In our setting, this has to be extended because, rather than considering a single machine in which all steps are performed, we have a potentially distributed setting, with multiple objects possibly running concurrently on different CPUs. Thus, rather than aggregating the cost of all executing steps, it is more useful to treat execution steps which occur on different computing infrastructures separately. With this aim, we adopt the notion of *cost centers* [RGM98], proposed for profiling functional programs. Since the concurrency unit of our language is the object, cost centers are used to charge the cost of each step to the cost center associated to the object where the step is performed. For a given set of objects identifiers O and a trace t, we use $t|_O = \{S_i \rightsquigarrow_{o_i}^{b_i} S_{i+1} \mid S_i \rightsquigarrow_{o_i}^{b_i} S_{i+1} \in t,\ o_i \in O\}$ to denote the set of execution steps that are performed on objects from O. The cost of t w.r.t. a cost

model $\mathcal{M}$ and a cost center O is $\mathcal{C}(t, O, \mathcal{M}) = \sum_{e \in t|_O} \mathcal{M}(e)$. O might contain multiple object identifiers, but also only one. Observe that it is also possible to apply different cost models to different cost centers.

Cost Models.

As it is mentioned in Section 2.3.3, we consider platform independent cost models (e.g., worst-case execution time or energy consumption are excluded). The cost models $\mathcal{M}_i$, $\mathcal{M}_m$ and $\mathcal{M}_o$ of Section 2.3.3 can be directly used for our concurrent programs. A cost model that counts $\mathsf{call}(\mathtt{m}, \_)$, can be used to infer the number of tasks that are spawned along an execution. By '_', we mean any (valid) expression. We can also count the number of calls to specific methods or objects, e.g., by counting $\mathsf{call}(\mathtt{m}, \_(o, \_, \_))$ we obtain bounds on the number of requests to a remote component o. This is useful for approximating the components' load and finding optimal deployment configurations (e.g., group objects according to the amount of tasks they receive to execute, by also taking into account the infrastructure on which they are deployed). The above cost models can also be used to prove termination of the program by setting the underlying solver [AAGP11] to only bound the number of iterations in loops (see Section 2.4).

6.4 Field-Sensitive Size Analysis for Concurrent OO Programs

The objective of size analysis is to infer *size abstractions* which allow reasoning on how the sizes of data change along a program's execution, which is fundamental for bounding the number of iterations that loops perform.

6.4.1 The Basic Size Analysis

We present the size analysis in two steps: we first recall the notion of size measure of Section 2.3; and we then present an abstraction which compiles instructions into size constraints, trying to keep as much information on global data (i.e., fields) as possible, while still being sound in concurrent executions.

Size Measures. The language on which we develop our analysis is deliberately

simplified so that it only considers numerical and reference types. In the case of numbers, the size is the actual numerical value. References however require a more sophisticated treatment. A commonly used size measure is the *path-length* decribed in Section 2.2 which counts the number of elements of the longest chain of references that can be traversed through the initial object (e.g., length of a list, depth of a tree, etc.). However, in our context, since objects are intended to simulate concurrent computing entities and not data structures, it is hence not common for them to directly affect loop iterations. Therefore, ignoring their sizes is sound and precise enough in most cases. A slightly more precise abstraction distinguishes between the case in which a reference variable points to an object (size 1) or to null (size 0). The size of a future variable is the same as the size of the value it holds. This is sound since such variables can be used only through `get`, which blocks until the variable has a value. Our actual implementation allows choosing among those size measures. Also, since it has support for strings and for functional (parametric) types, it uses other size measures. Namely, for strings, their length, and, for functional terms, the so-called *term-size* measure, which counts the number of type constructors in a given term.

Abstract Compilation. Modeling shared memory is a main challenge in static analysis of OO programs. Our starting point is [AAG+10, RD11], which models fields as local variables when the field to be tracked satisfies: (1) its memory location does not change; and (2) it is always accessed through the same reference (i.e., not through aliases). Both conditions can often be proven statically and the transformation of fields into local variables can then be applied for many fragments of the program. If we ignore concurrency, this approach could be directly adopted for our language. However, concurrency introduces new challenges.

Example 6.4.1. *Consider the loop in the* readBlock *method in Figure 6.1. Ignoring the* `await` *instruction, the above soundness conditions (1) and (2) hold for the field* blockS*, and hence, we can track it as if it was a local variable. In a concurrent setting, however, while* readBlock *is executing, another task in the same object might modify* blockS*. Therefore, when analyzing* readBlock*, we cannot assume that the value of* blockS *is locally trackable. For instance,* readBlock *might introduce non-termination if we add a method* void p() {blockS = blockS − 2;} *to*

	b	$\alpha_\rho(b)$	ρ'
1	$e\ op\ e$	$\alpha_\rho(e)\ op\ \alpha_\rho(e)$	$\rho'{=}\rho$
2	$e\ op'\ e$	_	$\rho'{=}\rho$
3	`null` $\mid x \mid this.f$	$0 \mid \rho(x) \mid \rho(f)$	$\rho'{=}\rho$
4	`release`	$true$	$\rho'{=}\rho[\bar{f}_C \mapsto \rho(\bar{f}_C)']$
5	`await` g	$\alpha_{\rho'}(g)$	$\rho'{=}\rho[\bar{f}_C \mapsto \rho(\bar{f}_C)']$
6	$x := y.$`get` $\mid x := e$	$\rho'(x){=}\rho(y) \mid \rho'(x){=}\alpha_\rho(e)$	$\rho'{=}\rho[x \mapsto \rho(x)']$
7	$this.f := e$	$\rho'(f){=}\alpha_\rho(e)$	$\rho'{=}\rho[f \mapsto \rho(f)']$
8	$x :=$ `new` C	$\rho'(x){=}1$	$\rho'{=}\rho[x \mapsto \rho(x)']$
9	`call`$(\mathtt{b}, q(rec, \bar{x}, \bar{y}))$	$q(\rho(rec), \rho(\bar{x}{\cdot}\bar{f}_C), \rho'(\bar{y}{\cdot}\bar{f}_C))$	$\rho'{=}\rho[\bar{y}{\cdot}\bar{f}_C{\mapsto}\rho(\bar{y}{\cdot}\bar{f}_C)']$
10	`call`$(\mathtt{m}, q(rec, \bar{x}, \bar{y}))$	$q(\rho(rec), \rho(\bar{x}), \rho'(\bar{y}))$	$\rho'{=}\rho[\bar{y} \mapsto \rho(\bar{y})']$
11	$otherwise$	$true$	$\rho'{=}\rho$

where in case (1) $op \in \{\wedge, >, \geq, =, +, -\}$ and in (2) $op' \in \{*, /\}$

Figure 6.3: Abstract compilation. $\mathtt{ABST}(b_{k:i}, \rho){=}\langle\alpha_\rho(b_{k:i}), \rho'\rangle$

class FileIS. *When the* **`await`** *is executed inside the loop, method* p *might change the value of* blockS *to a non-positive value, and thus the loop counter* i *would not decrement.*

Handling fields requires identifying program points at which the shared memory might be modified by other tasks. This can happen when: (1) `release` or `await` are explicitly executed, and thus allow other tasks (of the same object) to run; and (2) an asynchronous invocation is issued, and until the called method starts to execute, the fields of the called object might be changed by other tasks. We refer to such program points as *release points*. The above observation suggests that in a sequence of instructions not including `release` or `await`, the shared memory can be tracked locally. However, the values in the shared memory when a method starts to execute may not be identical to those when it was called. We first present a safe abstraction which loses all information at release points and at method entries. In a second step we handle these points.

An abstract state is a set of linear constraints whose solutions define possible concrete states. This representation allows describing relations that are essential for inferring cost and proving termination, e.g., the size of x decreases by 1 in two consecutive states. The building blocks for this representation are constraints that describe the effect of each instruction b on a given state. We refer to such constraints as the *abstraction* of b. Figure 6.3 depicts these abstractions. In order

to abstract an instruction b, we use a mapping ρ from variables and field names to constraint variables that represent their sizes in the state before executing b. The result of abstracting b w.r.t. ρ are the constraints $\alpha_\rho(b)$, and a new mapping ρ' that refers to the sizes in the state after executing b.

Let us describe the abstraction of some instructions. In Line 6, the instruction $x := e$ is abstracted into the equality $\rho'(x) = \alpha_\rho(e)$, where $\alpha_\rho(e)$ is the size of e w.r.t. ρ. Note that $\rho'(x)$ (resp. $\rho(x)$) refers to the size of x *after* (resp. *before*) executing the instruction. The abstraction of `release` at Line 4 "forgets" sizes of the fields $\bar{f}_C$. This is because they might be updated by other methods that take the control when the current task suspends. The abstraction of `await` is similar, though we add to the abstract state the information that the guard g is satisfied upon completion of `await` g. When abstracting a call to a block in Line 9, the class fields are added as arguments in order to track their values. However, when abstracting calls to methods (Line 10) the fields are not added. For methods, they are not added because their values at call time might not be the same as when the method actually starts to execute. Since we use linear constraints only, non-linear arithmetic expressions (Line 2) are abstracted to a fresh constraint variable "_" that represents any value. A program P is transformed into an abstract program P^α, that approximates its behavior w.r.t. a size measure, by abstracting its rules as follows.

Definition 6.4.2 (abstract compilation)**.** *Given* $r \equiv m(this, \bar{x}, \bar{y}) \leftarrow g, b_1, \ldots, b_n \in P$*, and an identity map* ρ_0 *over* $vars(r) \cup \bar{f}_C$*, the abstract compilation of* r *is* $r^\alpha \equiv m(this, \bar{I}, \rho_{n+1}(\bar{O})) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$ *where:*

- $\langle g^\alpha, \rho_1\rangle$=$\mathit{ABST}(g, \rho_0)$, $\langle b_i^\alpha, \rho_{i+1}\rangle$=$\mathit{ABST}(b_i, \rho_i)$ *; and*
- $\bar{I}$=$\bar{x}\cdot\bar{f}_C$ *and* $\bar{O}$=$\rho_{n+1}(\bar{y}\cdot\bar{f}_C)$ *if* m *is a block; otherwise* $\bar{I}$=$\bar{x}$ *and* $\bar{O}$=$\rho_{n+1}(\bar{y})$*. The* size abstraction *for the rule* r *is* $g^\alpha \wedge b_1^\alpha \wedge \ldots \wedge b_n^\alpha$*.*

Example 6.4.3. *The following is the abstract compilation of the rule for* if_c *in Figure 6.2, where* $\overline{inp}$*,* $\overline{out}$ *and* $\overline{F}$ *denote, respectively, the input parameters* $this, res, i, incr, pos$*, the output parameters* $res'', i'', incr', pos''$ *and the fields* $fp, lth, blockS$*. The substitution* ρ_0 *stands for the identity mapping.*

$$
\begin{array}{lll}
if_c(\langle\overline{inp}, \overline{F}\rangle, \langle\overline{out}, \overline{F}''\rangle) \leftarrow & & \rho_0 \\
\text{(a)}\ \ readContent(\langle this, pos, incr\rangle, \langle f'\rangle), & & \rho_1 = \rho_0[f \mapsto f'] \\
\text{(b)}\ \ true, & & \rho_2 = \rho_1[\overline{F} \mapsto \overline{F}'] \\
\quad v' = f', & & \rho_3 = \rho_2[v \mapsto v'] \\
\quad res' = res + v' & & \rho_4 = \rho_3[res \mapsto res'] \\
\quad i' = i - incr & & \rho_5 = \rho_4[i \mapsto i'] \\
\quad pos' = pos + incr & & \rho_6 = \rho_5[pos \mapsto pos'] \\
\text{(c)}\ \ while(\langle this, res', i', incr, pos', \overline{F}'\rangle, & & \rho_7 = \rho_6[res' \mapsto res'', i' \mapsto i'', \\
\qquad \langle res'', i'', incr', pos'', \overline{F}''\rangle). & & \qquad incr \mapsto incr', pos' \mapsto pos'', \bar{F}' \mapsto \bar{F}'']
\end{array}
$$

Note that at (b) `await` *is abstracted to true and the information on fields is lost, at* (c) *the fields are added to the call in order to keep track of their values, however, when calling a method at* (a)*, the abstraction "forgets" this information.*

6.4.2 Class Invariants in Cost Analysis

The accuracy of the size analysis can be improved by using a generalization of *class invariants* (see, e.g., [Mey97]). As discussed above, release points are problematic since at these points other task(s) may modify the values of shared fields. However, it is often possible to gather useful information about shared variables, in the form of class invariants, which must hold at those points. In sequential programs, class invariants have to be established by constructors and must hold on termination of all (public) methods of the class. They can be assumed at (public) method entry but may not hold temporarily at intermediate states not visible outside the object. In our context, we need such invariants to hold on method termination and also at all release points of all methods. This way, we can use them to improve the abstraction at the release points. In the following, given a class C, Ψ_C denotes the class invariant for class C, which is a set of linear constraints over the fields of C and possibly some constant symbols.

Definition 6.4.4. *We extend Definition 6.4.2 as follows: (1) when abstracting a method rule, we add Ψ_C to the abstract rule (just before g^α); and (2) we abstract* `release` *(resp.* `await`*) to $\Psi_C[\bar{f}_C \mapsto \rho'(\bar{f}_C)]$ (resp. $\alpha_\rho(g) \wedge \Psi_C[\bar{f}_C \mapsto \rho'(\bar{f}_C)]$).*

Example 6.4.5. *The following invariants will be required in order to obtain the cost of all methods of our running example: (1) In class* Reader*, we need to*

know that field elem is bounded, i.e., $0 \leq elems \leq elems_{max}$ where $elems_{max}$ is a constant symbol that bounds the value of elems. Besides, in order to bound the number of iterations of the loop in method process*, we need an invariant that states $elems = elems_{init}$, i.e., field elems is initialized in the constructor and it is never modified again; and (2) In class* FileIS*, we also need to know that fields blockS and lth are bounded. As before, we need an invariant $blockS = blockS_{init}$ for the loop in method* readBlock*. Such invariants can be inferred automatically by means of a syntactic analysis that simply checks that the corresponding fields are initialized and never updated again.*

6.5 Points-to Analysis for Concurrent Programs

The aim of the points-to analysis is to approximate the set of objects which each reference variable may point to during program execution. An analysis is *object-sensitive* [MRR05, SBL11] if methods may be analyzed separately for different (sets of) objects on which they are invoked. More precisely, the analysis uses a finite set of *object names* to partition the (possibly infinite) set of objects allocated at runtime into *contexts* which are analyzed separately.

This section presents a flow-sensitive object-sensitive points-to analysis for concurrent programs. It is based on Milanova's analysis framework [MRR05] for Java. As Milanova's analysis is flow-insensitive, it is sound for concurrent programs because it implicitly considers all possible interactions and interleavings between tasks that may happen in a concurrent program. However, our proposed analysis is flow-sensitive since for the inference of the object-sensitive recurrence relations, it is fundamental to track flow-sensitive relations among objects.

It is known that flow-sensitive analysis of concurrent programs is challenging due to the complexity of their flow. All possible task interleavings must be considered in order to develop a sound analysis. As our contribution in this regard, we extend the analysis of [MRR05] to make it flow-sensitive in the presence of concurrent behaviours. The main idea is to keep abstractions for local variables and for fields separate such that, when the processor is released, only the state of the fields is affected since, as we have already discussed in Section 6.4, the values

of local variables cannot be modified at release points. Besides, as we will see in the analysis, not all information on fields has to be lost. By keeping track of the values of *this*, we can notably reduce information loss.

6.5.1 The Abstract Domain

The abstraction of each object created in the program is a syntactic construction of the form $o_{ij...pq}$ that represents all run-time objects that were created at program point q when the enclosing instance method was invoked on an object represented by $o_{ij...p}$, which was in turn created at allocation site p, that is program point where the "new" is executed.

Let S be the set of all allocation sites in a program. Given a constant $k \geq 1$, the analysis considers a finite set of object names, denoted $\mathcal{N}$, which is defined as: $\mathcal{N} = \{\epsilon\} \cup S \cup S^2 \ldots S^k$. Note that k defines the maximum size of sequences of allocations, and it allows controlling the precision of the analysis. S^k represents the set of allocation sequences of length k. Allocation sequences have in principle unbounded length and thus it is sometimes necessary to lose precision during analysis. This is done by just keeping the k rightmost positions in sequences whose length is greater than k. We use $|s|$ to denote the length of a sequence s. We define the operation $\langle i, j, \ldots, p\rangle \oplus q$ which returns $\langle i, j, \ldots, p, q\rangle$ if $|\langle i, j, \ldots, p, q\rangle| \leq k$ and $\langle j, \ldots, p, q\rangle$, otherwise. A variable can be assigned objects with different object names. In order to represent all possible objects pointed to by a variable, sets of object names are used.

$\mathcal{V}$ represents the set of all possible *reference* local variables that may occur in a program. $\mathcal{F}$ represents all possible pairs (o, f) which denote all possible accesses to the *reference* fields f through the objects o in $\mathcal{N}$. In what follows, such pairs are represented as $o.f$. Following Milanova's approach, context sensitivity is achieved by maintaining multiple replicas of each reference variable x for each possible context in which x may be used for calling a method. Let x be a local variable and l an object name to which *this* may point to, we use the fresh variable name x^l to store the analysis information for x and context l. We drop the superscript l when it is not relevant. The set of replicas is defined by $map : \mathcal{V} \times \mathcal{N} \mapsto \mathcal{V}'$. An abstract state is a tuple $\langle \phi, \theta \rangle$ where:

- ϕ is a mapping $\phi : \mathcal{V}' \mapsto \wp(\mathcal{N})$, s.t. $\phi(x^l)$ is the set of object names that represents all possible objects that may be assigned to local variable x when *this* points to object name l;

- θ is a mapping $\theta : \mathcal{F} \mapsto \wp(\mathcal{N})$, s.t. $\theta(o.f)$ is the set of object names that represents all possible objects that may be assigned to the field f for the object name o;

The abstract domain is the lattice $\langle AS, \top, \bot, \sqcup, \sqsubseteq \rangle$, where AS is the set of abstract states, $\top$ is the top of the lattice which is equal to $\langle \phi_\top, \theta_\top \rangle$ s.t. $\forall v, \phi_\top(v) = \mathcal{N}$, and $\forall o.f, \theta_\top(o.f) = \mathcal{N}$, and $\bot$ is the bottom of the lattice, $\forall v, \phi_\bot(v) = \emptyset$, $\forall o.f, \theta_\bot(o.f) = \emptyset$. Given two abstract states $\langle \phi_1, \theta_1 \rangle$ and $\langle \phi_2, \theta_2 \rangle$, we use $\langle \phi, \theta \rangle = \langle \phi_1, \theta_1 \rangle \sqcup \langle \phi_2, \theta_2 \rangle$ to denote that $\langle \phi, \theta \rangle$ is their least upper bound. It is defined as $\forall v, \phi(v) = \phi_1(v) \cup \phi_2(v)$ and $\forall o.f, \theta(o.f) = \theta_1(o.f) \cup \theta_2(o.f)$. In the same way, $\langle \phi_1, \theta_1 \rangle \sqsubseteq \langle \phi_2, \theta_2 \rangle$ holds iff $\forall v, \phi_1(v) \subseteq \phi_2(v)$ and $\forall o.f, \theta_1(o.f) \subseteq \theta_2(o.f)$.

6.5.2 The Transfer Function

Our proposed analysis is a standard forward analysis that assigns an abstract state to each program point by relying on a transfer function $\tau : \wp(\mathcal{N}) \times \mathit{Instr} \times AS \mapsto AS$, where Instr is the set of basic instructions and AS the set of abstract states, as defined in Figure 6.4. We use *This* to represent the set of object names which currently approximate the value of *this*. We assume the considered instruction is located at program point q, that x, y are reference (local) variables and that f, g are reference fields. It is important to note that modifications to local variables (rows 1-4) affect ϕ in a flow-sensitive way (i.e., updates on variables overwrite the previous abstract value). However, updates on reference fields (rows 5-7) modify θ in a flow-insensitive way (i.e., the information is added to the previous values for such field). Method calls (rows 9-10) are handled by $\mathit{interp}(\langle \phi, \theta \rangle, \mathit{This}, m(\bar{z}, y))$, which (a) looks up the method definition for method m and projects $\bar{z}$ using ϕ and θ to fit the calling context of m, resulting in a new mapping ϕ', (b) uses ϕ' and *This*, the set of possible values for *this* to analyze m, and (c) after the analysis, which can modify θ, gets the analysis output ϕ'' and modifies ϕ to set the new value for y, namely $\phi[y \mapsto \phi''(ret)]$. In row 10, the only difference with

	$q : instr$	$\tau(This, instr, \langle\phi, \theta\rangle)$	
(1)	$x = \mathsf{new}\ C$	$\langle\phi[x^l \mapsto l \oplus q], \theta\rangle$	$\forall l \in This$
(2)	$x = y$	$\langle\phi[x^l \mapsto \phi(y^l)], \theta\rangle$	$\forall l \in This$
(3)	$x = this.f$	$\langle\phi[x^l \mapsto \theta(l.f)], \theta\rangle$	$\forall l \in This$
(4)	$x = \mathsf{null}$	$\langle\phi[x^l \mapsto \emptyset], \theta\rangle$	$\forall l \in This$
(5)	$this.f = y$	$\langle\phi, \theta[l.f \mapsto (\theta(l.f) \cup \phi(y^l))]\rangle$	$\forall l \in This$
(6)	$this.f = this.g$	$\langle\phi, \theta[l.f \mapsto (\theta(l.f) \cup \theta(l.g))]\rangle$	$\forall l \in This$
(7)	$this.f = \mathsf{null}$	$\langle\phi, \theta\rangle$	
(8)	$\mathsf{return}\ \ x$	$\langle\phi[ret^l \mapsto \phi(x^l)], \theta\rangle$, ret^l fresh	$\forall l \in This$
(9)	$y = this!m(\bar{z})$	$interp(\langle\phi, \theta\rangle, This, m(this, \bar{z}, y))$	
(10)	$y = x!m(\bar{z})$	$interp(\langle\phi, \theta\rangle, \phi(x), m(x, \bar{z}, y))$	
(11)	$otherwise$	$\langle\phi, \theta\rangle$	

Figure 6.4: Transfer Function (where $l \equiv o_{i\ldots p}$, and $l \oplus q \equiv o_{i\ldots p \oplus q}$).

row 9 is that the abstract value of x, i.e., $\phi(x)$, is used instead of *This*. The analysis of loops requires iterating the corresponding code several times until a fixpoint is reached (convergence is guaranteed because the domain is finite). The analysis merges abstract states at convergence points (i.e., after if and at loop entries) using the join operation $\sqcup$.

Example 6.5.1. *Figure 6.5 shows (part of) the result, at the end of the fixpoint, of applying the points-to analysis to the running example with $k = 2$. This is the smallest k for which no information is lost when handling object names. Keeping track of the value of the this reference is crucial for the precision of the points-to analysis. All object creations use the object name(s) pointed to by this to generate new object names by adding the current allocation site. E.g., at p.p. ③, this may be either this $\mapsto o_1$ or this $\mapsto o_2$; the new object names created are o_{13} and o_{23}, respectively. Observe that we keep the calling context as a superscript to the variable such that r^{o_1} denotes the abstract value for* r *when this is o_1. The value of this within a method comes from the object name(s) for the variable used to call the method. The example only shows ϕ from the transfer function, since only local variables are changed. Assignments to field variables would affect θ accordingly, as mentioned above.*

The next theorem states the soundness of the analysis, which can be eas-

Int readBlock () {	$\phi = \{this \mapsto o_2\}$
... f = this ! readContent(pos,incr); ...	$\phi = \{this \mapsto o_2\}$
}	
Int readOnce () {	$\phi = \{this \mapsto o_1\}$
Fut⟨Int⟩ f = this ! readContent(0,lth);	$\phi = \{this \mapsto o_1\}$
await f?;	$\phi = \{this \mapsto o_1\}$
return f.get;	$\phi = \{this \mapsto o_1\}$
}	
Int readContent(Int pos, Int elems) {	ϕ=$\{this \mapsto \{o_1, o_2\}$
③ Reader r = new Reader (fp,elems);	ϕ=$\{this \mapsto \{o_1, o_2\}, \mathsf{r}^{o_1} \mapsto o_{13}, \mathsf{r}^{o_2} \mapsto o_{23}\}$
Fut⟨Int⟩ f = r ! process(pos);	ϕ=$\{this \mapsto \{o_1, o_2\}, \mathsf{r}^{o_1} \mapsto o_{13}, \mathsf{r}^{o_2} \mapsto o_{23}\}$
await f?; return f.get;	ϕ=$\{this \mapsto \{o_1, o_2\}, \mathsf{r}^{o_1} \mapsto o_{13}, \mathsf{r}^{o_2} \mapsto o_{23}\}$
}	
main {	$\phi = \{this \mapsto \epsilon\}$
① FileIS o1 = new FileIS("A.txt",20,2);	$\phi = \{this \mapsto \epsilon, \mathsf{o1} \mapsto o_1\}$
② FileIS o2 = new FileIS("A.txt",20,3);	$\phi = \{this \mapsto \epsilon, \mathsf{o1} \mapsto o_1, \mathsf{o2} \mapsto o_2\}$
Fut⟨Int⟩ f1; Fut⟨Int⟩ f2;	$\phi = \{this \mapsto \epsilon, \mathsf{o1} \mapsto o_1, \mathsf{o2} \mapsto o_2\}$
f1 = o1 ! readOnce();	$\phi = \{this \mapsto \epsilon, \mathsf{o1} \mapsto o_1, \mathsf{o2} \mapsto o_2\}$
f2 = o2 ! readBlock(); ...	$\phi = \{this \mapsto \epsilon, \mathsf{o1} \mapsto o_1, \mathsf{o2} \mapsto o_2\}$

Figure 6.5: Points-to analysis results for the running example.

ily proven correct by following the same proof scheme as in Milanova's analysis framework [Mil03], since it only differs in the flow-sensitive aspect.

Theorem 6.5.2. *Given a program P, the transfer function τ generated for P provides a safe approximation of the objects that can be pointed to by variables and fields in any execution of P.*

In what follows, given a reference variable (respectively a reference field) x, we use $pt(q, x, o_t)$ to refer to the set of values $\phi(x)$ (resp. $\theta(x)$) computed by the points-to analysis at program point q when *this* points to the object name o_t.

6.6 Object-Sensitive Resource Analysis

Our analysis follows the approach described in Chapter 2, a program is first transformed into a set of cost relations [AAG+12b] which can then be solved into

closed-form upper/lower bounds [AAGP11]. This section focuses exclusively on the cost relation system (CRS) generation phase as the equations can be solved using the approach described in Chapter 2 without requiring any change. We illustrate in Section 6.6.1 how an object-insensitive analysis can be defined as in sequential programming, by using the size abstraction computed in Section 6.4, and point out its limitations. Then, Section 6.6.2 defines the object-sensitive analysis which, by relying on the object-sensitive points-to information of Section 6.5, overcomes the limitations of the insensitive analysis.

6.6.1 Object-Insensitive Analysis

The generation of *object-insensitive* cost relations from our concurrent and distributed programs, for a generic cost model $\mathcal{M}$, can be done exactly as for sequential programs (see Section 2.3 and [AAG+12b]), by using the size abstractions which already take the concurrent behaviour into account, and then simply applying the generic cost model to each instruction of each rule. This object insensitive approach has a main drawback: it is not capable of distinguishing the different distributed components. Instead, the resource usage contributed by all objects (which represent potentially distributed components) is simply accumulated in a single cost center which corresponds to the whole execution of the distributed system.

Example 6.6.1. *By applying the CRS generation techniques described in Section 2.3.5 to the RBR of our running example (partly shown in Figure 6.2), the size relations (partly shown in Example 6.4.3) and the invariants in Example 6.4.5, the following CRS for the cost model $\mathcal{M}_i$ is obtained:*

$$
\begin{array}{lcll}
readBlock() & = & 8{+}while(i, blockS) & \{i{=}lth, 0 \leq lth \leq lth_{max}\} \\
while(i, blockS) & = & 1 & \{i \geq 0\} \\
while(i, blockS) & = & 2{+}if(i, blockS) & \{i{<}0\} \\
if(i, blockS) & = & 3{+}if_0(i, blockS, incr) & \{incr{=}i\} \\
if(i, blockS) & = & 3{+}if_0(i, blockS, incr) & \{incr = blockS\} \\
if_0(i, blockS, incr) & = & 11{+}readContent()+ & \{i' = i{-}incr, \\
 & & while(i', blockS') & blockS' = blockS_{init}\} \\
readContent() & = & 10{+}process() & \{\} \\
readOnce() & = & 5{+}readContent() & \\
process() & = & 4 + while_1(i, elems) & \{i{=}0, 0 \leq elems{\leq}elems_{max}\} \\
while_1(i, elems) & = & 2 & \{i{\geq}elems\} \\
while_1(i, elems) & = & 15{+}while_1(i', elems') & \{i{<}elems, elems'{=}elems_{init}, i'{=}i + 1\}
\end{array}
$$

where, for method process *we assume that the execution of methods* hdRead *and* update *has a constant cost, which is included in the constant* 15 *of the second equation for* $while_1$*. Likewise, the constant* 10 *in the equation for readContent includes the cost of executing the constructor of class* Reader *which is assumed to be constant. This CRS is solved using [AAGP11] into closed-form upper bounds:*

$$
\begin{array}{lcl}
UB_{readBlock}() & = & 9{+}\mathsf{nat}(lth_{max}){*}(34{+}15{*}\mathsf{nat}(elems_{max})) \\
UB_{process}() & = & 6 + 15 * \mathsf{nat}(elems_{max}) \\
UB_{readContent}() & = & 16 + 15 * \mathsf{nat}(elems_{max}) \\
UB_{readOnce}() & = & 21{+}15 * \mathsf{nat}(elems_{max})
\end{array}
$$

Let us explain the different parts of the upper bound computed for readBlock*. The constant* 9 *comes from* 8 *(the constant in the equation readBlock) plus* 1 *(the constant in the first equation of* $while_1$ *that corresponds to the exit of the loop). The cost of the loop is the following quadratic expression*

$$\mathsf{nat}(lth_{max}){*}(34{+}15{*}\mathsf{nat}(elems_{max})),$$

where $\mathsf{nat}(lth_{max})$ *is an upper bound on the number of iterations of the loop and* $34{+}15{*}\mathsf{nat}(elems_{max})$ *is the worst-case cost of each iteration. Note that the loop invokes method readContent which contains a loop whose cost is linear on* $elems_{max}$*.*

6.6.2 Adding Cost Centers to the Equations

As the main novelty of this work, CRS in the object-sensitive resource analysis use cost centers in order to keep the resource usage assigned to the different components separate. The main idea is to take advantage of the object-sensitive points-to information to generate cost equations for all possible contexts (and thus objects). In particular, the object-sensitive equations will allow us to count separately the cost that corresponds to different instances of objects that are created at the same allocation site but correspond to different object names and may belong to different distributed components. We use a symbolic expression $c(o_l)$ per object name l returned by the points-to analysis in order to denote the cost center associated to o_l. We assume all rules are annotated as follows:

$$R \equiv [p(\bar{x}, \bar{y})]^{This} \leftarrow g, [b_1]^{O_1}, \ldots, [b_n]^{O_j} \in P$$

where the head of the rule is annotated with the set of object names $This = \{t_1, \ldots, t_k\}$ and each method call b_i of the form $q : \mathsf{call}(\_, m(x, \bar{w}, y))$ is annotated with the set of sets $O_i = \{O_i^{t_1}, \ldots, O_i^{t_k}\}$, where $O_i^{t_m} = pt(q, x, t_m)$, i.e., the set of object names that x may point to when *this* points to the object name t_m. Given a rule R, we use the following functions: $methods(R)$ to obtain the annotated set of elements $[p(x, \overline{w}, y)]^O$ which are calls to methods of the form $[\mathsf{call}(\mathtt{m}, p(x, \overline{w}, y))]^O$ in the body of R, $blocks(R)$ to refer to the set of elements $b(\overline{w}, y)$ which are calls to intermediate rules of the form $\mathsf{call}(\mathtt{b}, b(this, \overline{w}, y))$ in the body of R, and $instr(R)$ to refer to the set of elements in the body of R that are other instructions.

Definition 6.6.2 (object-sensitive resource analysis)**.** *Given an annotated rule* $R \equiv [p(\bar{x}, \bar{y})]^{This} \leftarrow \_ \in P$ *where* $methods(R) = \{[m_1(y_1, \overline{z}_1, w_1)]^{O_1}, \ldots, [m_j(y_j, \overline{z}_j, w_j)]^{O_j}\}$*, and its size abstraction* φ*, the following set of equations define its cost: for each* $o \in This$*, and for each* $\langle o_1, \ldots, o_j \rangle \in O_1^o \times \cdots \times O_j^o$ *such that* $O_i^o \in O_i$*, we generate the equation*

$$p\_o(\overline{x}) = \sum_{b \in instr(R)} c(o) * \mathcal{M}(b) + \sum_{b(\overline{y}, w) \in blocks(R)} b\_o(\overline{y}) + m_1\_o_1(\overline{z}_1) + \cdots + m_j\_o_j(\overline{z}_j)$$

where $m_i\_o_i$ is the name of the equation that represents a call to method m_i from object o_i.

Intuitively, the above definition generates, from one rule, as many equations as needed for defining its cost such that all possible contexts (i.e., object names of callees) are considered. The new names are obtained by concatenating the corresponding object name to the rule name. Besides, as regards to method invocations, all combinations have to be generated. This is done in the definition by means of the cartesian product $O_1^o \times \cdots \times O_j^o$ which gives us all possible combinations for the elements in the sets. The cost expressions we accumulate are multiplied by a symbolic expression $c(o)$ which denotes the cost center of the object on which the call is performed. As an example, if we have a rule:

$$[m_1(x,y)]^{\{o_1,o_2\}} \leftarrow [m_2(x,u)]^{\{\{o_3\}^{o_1},\{o_4,o_5\}^{o_2}\}} + [m_3(u,y)]^{\{\{o_6,o_7\}^{o_1},\{o_8\}^{o_2}\}}$$

These four equations are generated to cover all cases:

$$m_1\_o_1(x) = m_2\_o_3(x,u) + m_3\_o_6(u,y) \qquad m_1\_o_2(x) = m_2\_o_4(x,u) + m_3\_o_8(u,y)$$
$$m_1\_o_1(x) = m_2\_o_3(x,u) + m_3\_o_7(u,y) \qquad m_1\_o_2(x) = m_2\_o_5(x,u) + m_3\_o_8(u,y)$$

Multiple rules for the same procedure are interpreted as multiple choices and the upper bound solver computes the maximum over them. Therefore, the fact that multiple rules are introduced (e.g., two rules for $m_1\_o_1(x)$) does not degrade the quality of the upper bound obtained. If we replace $c(o)$ by 1 (for all object names o), the accuracy of object-insensitive CRS coincides with that of object-sensitive CRS. The upper bound for a set of objects $\mathcal{O}$, $UB_p|_{\mathcal{O}}$, is obtained by setting $c(o)$ to 1 for all object names $o \in \mathcal{O}$ and to 0 for the remaining ones.

Example 6.6.3. *The cost equation for* main *takes the form* $main() = 19 + readOnce() + readBlock()$. *The context-insensitive resource analysis is not able to distinguish between the cost centers* ① *and* ② *of Figure 6.5 (see Example 6.6.1) where methods* readOnce *and* readBlock *execute and accumulates both costs together. Using the context-sensitive resource analysis of Definition 6.6.2, the an-*

notated rules for $k = 2$ are as follows (only relevant calls are shown):

$$[main()]^{\{\epsilon\}} \leftarrow ..., [\mathsf{call}(\mathtt{m}, readOnce())]^{\{\{o_1\}\}}, [\mathsf{call}(\mathtt{m}, readBlock())]^{\{\{o_2\}\}}, ...$$

$$[readBlock(\langle this\rangle, \langle r\rangle)]^{\{o_2\}} \leftarrow ..., [\mathsf{call}(\mathsf{b}, while(\langle \overline{in}\rangle, \langle \overline{out}\rangle))]^{\{\{o_2\}\}}, ...$$

$$[if^c(\langle \overline{in}\rangle, \langle \overline{out}\rangle)]^{\{o_2\}} \leftarrow ..., [\mathsf{call}(\mathtt{m}, readContent(\langle this, pos, incr\rangle, \langle f\rangle))]^{\{\{o_2\}\}}, ...$$

$$[readOnce(\langle this\rangle, \langle r\rangle)]^{\{o_1\}} \leftarrow ..., [\mathsf{call}(\mathtt{m}, readContent(\langle this, pos, incr\rangle, \langle f\rangle))]^{\{\{o_1\}\}}, ...$$

$$[readContent(\langle this, pos, incr\rangle, \langle f\rangle)]^{\{o_1,o_2\}} \leftarrow$$
$$..., [\mathsf{call}(\mathtt{m}, process(\langle this, pos\rangle, \langle r\rangle))]^{\{\{o_{13}\}^{o_1}, \{o_{23}\}^{o_2}\}}, ...$$

$$[process(\langle this, pos\rangle, \langle r\rangle)]^{\{o_{13},o_{23}\}} \leftarrow ...$$

Some blocks of method readBlock *are omitted since all of them are annotated with* $\{o_2\}$. *Note that rules for* readContent *and* process *are annotated with* $\{o_1, o_2\}$ *and* $\{o_{13}, o_{23}\}$, *respectively, since they can be invoked using two different object names for this. By applying Definition 6.6.2, the equations for each element in the annotated sets are generated by replicating the equations for* readContent *and* process. *For example, the (replicated) equations for* readContent *are as follows:*

$$\begin{aligned} readContent\_o_1() &= c(o_1) * 10 + process\_o_{13}() \\ readContent\_o_2() &= c(o_2) * 10 + process\_o_{23}() \end{aligned}$$

The closed-form upper bounds now keep separate the resource consumption associated to each cost center o_i by means of a symbolic constant $c(o_i)$. From the above equations for readContent *the solver obtains the upper bounds:*

$$\begin{aligned} UB_{readContent\_o_1}() &= c(o_1) * 10 + c(o_{13}) * (6 + 15 * \mathsf{nat}(elems_{max})) \\ UB_{readContent\_o_2}() &= c(o_2) * 10 + c(o_{23}) * (6 + 15 * \mathsf{nat}(elems_{max})) \end{aligned}$$

In contrast to the upper bound obtained in Example 6.6.1, the closed-form upper bound for main *keeps the number of instructions executed on each object separate:*

$$\begin{aligned} UB_{main}() = \; & c(\epsilon) * 19 + c(o_1) * 15 + \\ & c(o_{13}) * (6 + 15 * \mathsf{nat}(elems_{max})) + \\ & c(o_2) * (9 + 28 * \mathsf{nat}(lth_{max})) + \\ & c(o_{23}) * \mathsf{nat}(lth_{max}) * (6 + 15 * \mathsf{nat}(elems_{max})) \end{aligned}$$

E.g., the upper bound for $\mathcal{O} = \{o_{13}\}$ is $UB_{main}|_{\mathcal{O}} = 6 + 15 * \mathsf{nat}(elems_{max})$. The main observation is that the accuracy of the upper bound for main is significantly better when the analysis is performed with $k = 2$ than with $k = 1$ (or with an object-insensitive analysis). If the points-to analysis is performed for $k = 1$, the object names o_{13} and o_{23} would collapse in a single object name o_3. Therefore, it would not be possible to distinguish between the objects created from o_1 and from o_2 and the costs are aggregated together resulting in a much less precise upper bound that accumulates the expressions for $c(o_{13})$ and $c(o_{23})$.

Since the length of object names is limited to a length k, allocation sequences of length greater than k do not appear as such in the results of points-to analysis. Instead, they are represented by object names that cover them. Therefore, we need some means for relating allocation sequences to the object name that approximates them. We now define such notion. Given an allocation sequence l and a set of object names $\mathcal{O}$, the *approximation* of l in $\mathcal{O}$ is the longest object name in $\mathcal{O}$ which covers l. I.e., an object name $o_{l'} \in \mathcal{O}$ is the approximation of l in $\mathcal{O}$ iff $o_l \leq o_{l'}$ and $\forall o_{l''} \in \mathcal{O} \; . \; o_l \leq o_{l''} \; \rightarrow \; |l''| < |l'|$ or $l'' = l'$.

Theorem 6.6.4 (soundness)**.** *Let P be a program, t a trace that includes a single object with a single task scheduled for execution (not started yet) that corresponds to a method p, and $\bar{v}$ be the values of the input parameters of p and fields in the initial state. If O is a set of objects in t, and $\mathcal{O}$ is a set of object names computed by a points-to analysis such that each $o \in O$ is approximated by a name $l \in \mathcal{O}$, then $\mathcal{C}(t, O, \mathcal{M}) \leq UB_p(\bar{v})|_{\mathcal{O}}$.*

Proof. We sketch the main ideas of the proof for the object-insensitive analysis, and then we comment on the straightforward changes required to handle the object-sensitive case. The proof sketch consists of two parts:

- In the first one, we define an abstract operational semantics for the abstract (size) program, in which abstract states are annotated with the amount of resources consumed so far; and then we show that it can be used to approximate the resource consumption behaviour of the original program.
- Then, in a second part, we show that the cost relations generated from the

$$(1_a)\frac{C \equiv [\varphi \cdot C', \bar{y}, \bar{y}']}{\langle \{C|Cs\}, \Psi, e\rangle \dot{\rightsquigarrow} \langle \{[C', \bar{y}, \bar{y}']|Cs\}, \varphi \wedge \Psi, \mathcal{M}(\varphi) + e\rangle}$$

$$(2_a)\frac{\begin{array}{c} C \equiv [p(\bar{w}, \bar{z}) \cdot C', \bar{y}, \bar{y}'],\ p \text{ is a block},\ p(\bar{w}', \bar{z}') \leftarrow \varphi_g, C'' \in P_a, \\ \Psi \wedge \bar{w} = \bar{w}' \wedge \varphi_g \not\models false,\ \Psi' \equiv \bar{w} = \bar{w}' \wedge \bar{z} = \bar{z}' \wedge \varphi_g \wedge \Psi \end{array}}{\langle \{C|Cs\}, \Psi, e\rangle \dot{\rightsquigarrow} \langle \{[C'' \cdot C', \bar{y}, \bar{y}']|Cs\}, \Psi', \mathcal{M}(p(\bar{w}, \bar{z})) + e\rangle}$$

$$(3_a)\frac{\begin{array}{c} C \equiv [p(\bar{w}, \bar{z}) \cdot C', \bar{y}, \bar{y}'],\ p \text{ is a method},\ p(\bar{w}', \bar{z}') \leftarrow \varphi_g, C'' \in P_a \\ \Psi' \equiv \bar{w} = \bar{w}' \wedge \varphi_g \wedge \Psi,\ \Psi' \not\models false, \end{array}}{\langle \{C|Cs\}, \Psi, e\rangle \dot{\rightsquigarrow} \langle \{[C', \bar{y}, \bar{y}'], [C'', \bar{z}, \bar{z}']|Cs\}, \Psi', \mathcal{M}(p(\bar{w}, \bar{z})) + e\rangle}$$

$$(4_a)\frac{C \equiv [\epsilon, \bar{y}, \bar{y}']}{\langle \{C|Cs\}, \Psi, e\rangle \dot{\rightsquigarrow} \langle Cs, \bar{y} = \bar{y}' \wedge \Psi, e\rangle}$$

Figure 6.6: Abstract (cost) operational semantics

abstract program indeed approximate the resource consumption behaviour of the abstract program.

The abstract operational semantics is depicted in Figure 6.6. An abstract state is of the form $\langle Cs, \Psi, e\rangle$ where Cs is a set of abstract tasks, Ψ is a conjunction of linear constraints with integer constraint variables (the store), and e is a real number that represents the resources consumed so far. An abstract task $C \in Cs$ has the form $[b_1^\alpha \cdots b_n^\alpha, \bar{y}, \bar{y}']$, where each b_i^α is an (abstract) instruction, i.e., a linear constraint or a call, and $\bar{y}$ and $\bar{y}'$ represent an association of actual and formal output variables (for some call). These variables will be matched when the execution of $b_1^\alpha \cdots b_n^\alpha$ is completed to simulate returning a value through future variables. For simplicity, we assume that $\mathcal{M}(b_i^\alpha)$ equals to $\mathcal{M}(b_i)$, and that $\delta(C)$ returns the mapping ρ that was used to abstract b_1 into b_1^α (usually with renamed constraint variables).

Let us now explain the different rules of Figure 6.6. First note that all rules select a task C from Cs and make one execution step, depending on the type of the first abstract instruction in C:

- Rule (1_a) is used for executing an abstract instruction φ that corresponds to a simple instruction which is not a call. Recall that φ is a conjunction of linear constraints. In this case φ is simply added to the store, to simulate

the execution of the corresponding instruction, and its associated resource consumption is accumulated.

- Rule (2_a) is used for calling a block. It selects a matching and *applicable* block rule for p, and then adds its instructions to the current task. Note that the rule's guard is added to the store. The store is also modified to match the actual and formal input and output variables. It is essential that the selected rule for p uses fresh variables that were never used before during the execution.

- Rule (3_a) is used for calling a method. It selects a matching method rule and creates a new abstract task initialized with the corresponding instructions. The store is modified to match the actual and formal input variables. Note that the association of actual and formal output variables is kept in the state in order to be matched later, upon completion of that task. It is essential that the selected rule for p uses fresh variables that were never used before during the execution.

- Rule (4_a) corresponds to returning a value through future variables. Simply the corresponding actual and formal output parameters are matched.

We say that an abstract state $S^\alpha = \langle Cs, \Psi, e \rangle$ approximates a concrete state S, denoted by $S^\alpha \approx S$, if every concrete task is *covered* by a *different* abstract task, formally stated: For any $\mathtt{ob}(o, ClassTag, h, \langle tv, s \rangle, \mathcal{Q})$, if we pick up a task s' from $\{s\} \cup \mathcal{Q}$ (which is different from idle), then there exists an abstract task $C \in Cs$ (which is different for each s') such that:

$\mathcal{A}$) The sequence of abstract instructions C corresponds to the abstract compilation of the instructions in s' (if s' has a "meta" `release` that was introduced in Rule (10) of the operational semantics then we ignore it);

$\mathcal{B}$) Let ρ be the renaming that corresponds to $\delta(C)$, then there exists a model σ of Ψ (i.e., a solution that maps constraint variables to integers) such that for any $x \in dom(tv) \cup dom(h)$ it holds that $\sigma(\rho(x))$ equals to the size measure of $eval(x, h, tv, S)$. Note that if the variable $\rho(x)$ does not appear

syntactically in Ψ, then we assume that $\sigma(\rho(x))$ can be any value since the variable $\rho(x)$ is not involved in any constraint.

Now we state the first correctness claim, i.e., that the abstract program correctly simulates the resource consumption of the original ABS program.

Let $S_0 = \{\!|\mathtt{ob}(\mathtt{main}, \bot, \bot, \langle tv, \bar{b}\rangle, \emptyset)|\!\}$ where $\bar{b}$ is the body of method `main`, and let $S_0^\alpha = \langle [\bar{b}^\alpha, \langle\rangle, \langle\rangle], true, 0\rangle$ where $\bar{b}^\alpha$ is the abstract compilation of $\bar{b}$. We claim that if $S_0 \rightsquigarrow^* S$ then there is $S^\alpha = \langle C, \Psi, e\rangle$ such that $S_0^\alpha \dot{\rightsquigarrow}^* S^\alpha$, $S^\alpha \approx S$, and $e = \mathcal{M}(S \rightsquigarrow^* S)$ where $\mathcal{M}(S_0 \rightsquigarrow^* S)$ is the sum of the resource consumption of all transitions in $S_0 \rightsquigarrow^* S$. The proof can be done by induction on the length of the concrete trace. We explain the essentials.

Base case Straightforward since clearly $S_0^\alpha \approx S_0$.

Induction step We assume that the above claim holds when the length of $S_0 \rightsquigarrow^* S$ is n, and we show that it holds for a trace $S_0 \rightsquigarrow^* S \rightsquigarrow S'$ of length $n+1$. By the induction hypothesis, there is $S^\alpha = \langle C, \Psi, e\rangle$ such that $S_0^\alpha \dot{\rightsquigarrow}^* S^\alpha$, $S^\alpha \approx S$ and $e = \mathcal{M}(S_0 \rightsquigarrow^* S)$. Now we show that either we make one more step $S_0^\alpha \dot{\rightsquigarrow}^* S^\alpha \dot{\rightsquigarrow} S'^\alpha$ such that $S'^\alpha = \langle C', \Psi', e'\rangle$, $S^\alpha \approx S$, and $e' = \mathcal{M}(S_0 \rightsquigarrow^* S \rightsquigarrow S')$, or simply we have that $S' \approx S^\alpha$ and that the transition $S \rightsquigarrow S'$ does not consume resources, i.e., we still have $e = \mathcal{M}(S_0 \rightsquigarrow^* S \rightsquigarrow S')$. We briefly explain the cases by considering each of the semantic rules of Section 6.2.3 for the transition $S \rightsquigarrow S'$ as follows:

- Assume S' is obtained by applying Rule (1), (2), (3), (8) or (9). First note that these rules modify only one task in S by executing a corresponding simple instruction b. In such case, we use Rule (1_a) to obtain S'^α in a similar way. Clearly $S' \approx S'^\alpha$ since (i) φ is the abstract compilation of b; and (ii) the variables in φ do not appear in any other abstract task in S^α, so adding them to the store does not violate the covering of any other task. Moreover:

$$e' = e + \mathcal{M}(\varphi) = \mathcal{M}(S_0 \rightsquigarrow^* S) + \mathcal{M}(b) = \mathcal{M}(S_0 \rightsquigarrow^* S \rightsquigarrow S')$$

- Assume S' is obtained by applying Rule (4). Note that this rule also modifies only one task in S. In such case, we use Rule (2_a), and the abstract version of the same rule for p that has been used in (4), to obtain S'^{α}. Clearly we have $S' \approx S'^{\alpha}$ since we have used the abstract version of the same rule for p and also added $\bar{w} = \bar{w}' \wedge \bar{z} = \bar{z}'$ to the store in order to bind the actual and formal input and output variables. Moreover, we have $e' = \mathcal{M}(S_0 \rightsquigarrow^* S \rightsquigarrow S')$ as in the previous case.

- Assume S' is obtained by first applying Rule (5) and then immediately Rule (6) – we assume that both are applied just for simplicity, actually they are separated in the semantics just for simplifying the notation. Note that this case modifies one task in S and adds a new task that corresponds to the invoked method. In such case, we use Rule (3_a), in a similar way to the above case in which we used (2_a), to obtain the required S'^{α}. The only difference is that the formal input and output variables are not matched, they will be matched later when this new task terminates.

- Assume S' is obtained by applying Rule (7). Note that no task is modified, we only modify the value of a future variable. This modification might affect the value of local variables that are associated to this specific future variable. In the abstract setting, this can be obtained by applying Rule (4_a) which propagates the value of the future variable in the abstract setting by matching the associated actual and formal output variables.

- The rest of the rules do not modify the tasks in S, except adding some meta information such as `release` when the await fails. Here, clearly $\mathcal{A}$ and $\mathcal{B}$ still hold without making any execution step in the abstract setting.

Next we briefly explain why the cost relations generated from the abstract rules approximate the resource consumption of the abstract program, and thus the resource consumption of the original program. We do this by starting from the abstract program and the abstract semantics of Figure 6.6, and then modify them several times until we obtain the corresponding cost relations and the corresponding semantics [AAGP11].

In the first step, we consider a program that is obtained from the abstract program by removing all output variables, we refer to this program as output-free program. Clearly, any trace obtained using the abstract program has a corresponding trace that is obtained using the output-free program with the same resource consumption. This is true since the only difference is that in each step we might add less constraints to the store (we do not add those that match the formal and actual output parameters).

In the second step, we change the abstract semantics such that instead of accumulating the resource consumption of each execution step, it accumulates the resource consumption of all abstract instructions immediately when they are added to the abstract state in rules (2_a) and (3_a). This change amounts to: (i) changing rule (1_a) such that it does not accumulate $\mathcal{M}(\varphi)$, and (ii) changing rules (2_a) and (3_a) to accumulate also $c = \mathcal{M}(b_1^\alpha) + \cdots + \mathcal{M}(b_n^\alpha)$ where $C'' = b_1^\alpha, \ldots, b_n^\alpha$. Clearly, this change only anticipates the consumption of resources, and thus for any abstract trace that is obtained using the output-free program and the abstract semantics of Figure 6.6, we can generate a corresponding abstract trace using the same program and the modified abstract semantics such that it consumes at least the same amount of resources.

In the third step, we eliminate Rule (1_a) from the abstract semantics and modify rules (2_a) and (3_a) such that (i) they add all constraints that appear in the body of the selected rules (let us call them $\varphi = \varphi_1 \wedge \cdot \varphi_k$) to the store, and the rest, which are calls, are added as usual to the corresponding task. It is still guaranteed that using this abstract semantics we can reproduce the resource consumption of any trace generated in the above step. This is because the constraints in the body are obtained by applying a single static assignment transformation, thus for any $i > j$ the constraint φ_i does not restrict the values of variable in φ_j.

Now let us consider an equation $\langle p(\bar{x}) = c + \Sigma q_i(\bar{w}_i), \varphi \rangle$ in the cost relation. Here c and φ are the total resource consumptions and the constraints of a given rule respectively (as above). It is easy to see that this equation is just a denotational form of the resource consumption as developed in the third step above. Thus, any upper-bound of the cost relation is also an upper bound in the resource consumption of the corresponding abstract traces.

The correctness for the object-sensitive case is straightforward given the soundness of the points-to analysis. The above proof can be adapted to the object-sensitive case by: (i) modifying the abstract program such that it includes corresponding points-to annotations; and (ii) change the abstract semantics in order to accumulate expressions of the form $c(o) * \mathcal{M}(b)$. The correctness of the points-to analysis guarantees that if in the concrete setting we accumulate $\mathcal{M}(b)$ when executing within object o', then in the abstract setting we accumulate $c(o) * \mathcal{M}(b)$ where o is the approximation of the o' inferred by the points-to analysis. Finally, cloning the equation as done in Definition 6.6.2 just makes the points-to information explicit in the rules names.

□

The use of cost centers easily allows us to instantiate our analysis with different deployment strategies. Such strategies determine the groups of objects that share the processor (see, e.g., JCobox [SPH10]). The resource consumption of each group can be easily obtained by our approach.

6.7 Experimental Evaluation

We have developed COSTABS [AAG+11], a cost analyzer of ABS programs implemented as an extension of COSTA. An experimental evaluation has been carried out using several typical concurrent applications: PeerToPeer, a peer to peer protocol implementation; BBuffer, a classical bounded-buffer for communicating several producers; Chat, a chat application; Mail, a simple model of a Mail server; and DistHT, a distributed implementation of a hash table. The experiments have been performed on an Intel Core 2 Duo at 2.53GHz with 4GB of RAM, running Lunix 3.2.0.

Table 6.1 summarizes the main results. For each application, three different analyses have been performed: object-insensitive analysis (under the heading **Obj. Ins.**), and object-sensitive analyses using a points-to analysis with constant k=1, and with constant k=2. The experiments have been also performed for k=3, but they are not shown because no relevant improvement has been noticed. Column **#M** and **#in** show, respectively, the number of methods and the number

				Obj. Ins.			k=1			k=2			%
Bench.	**#M**	**#in**	**#I**	$\mathbf{T}_{cr}$	**#E**	$\mathbf{T}_{ub}$	**#o**	**#E**	$\mathbf{T}_{ub}$	**#o**	**#E**	$\mathbf{T}_{ub}$	**Impr.**
BBuffer	6	53	2	30	44	100	8	56	190	13	87	640	62.5 %
Chat	26	125	6	30	76	160	12	78	170	17	143	410	25.0 %
Mail	6	51	4	31	45	230	8	52	370	13	68	580	62.5 %
DistHT	10	74	12	40	69	490	6	69	380	9	93	510	50.0 %
P2P	14	143	8	120	101	2380	11	188	4350	19	472	10600	75.0 %

Table 6.1: Statistics about the Object-Sensitive Resource Analysis (times in ms.)

of instructions (in the intermediate representation) for each benchmark. Column **#I** shows the number of invariants needed to obtain an upper bound. $\mathbf{T}_{cr}$ shows the time taken to generate the CRS of the program. Column **#E** shows the number of equations generated by the analysis, and column $\mathbf{T}_{ub}$ shows the time taken to solve the CRS and obtain the closed-form upper bound. The time to generate the equations for the object-sensitive analysis has not been shown as it is almost identical to the object-insensitive one. However, it can be observed that the time to solve the equations notably grows with the number of equations to be solved (in [AAGP11] it is shown experimentally that this time grows almost linearly with the number of equations).

Column **#o** shows the number of different cost centers obtained by the points-to analysis for the corresponding k. Column **Impr.** aims at showing the further accuracy of k=2 w.r.t. k=1. This is done by comparing, for each cost center, the upper bound obtained for this cost center with k=1 with the upper bounds found for k=2 in the corresponding cost centers (possibly more than one) created at such allocation site. **% Impr** shows the percentage of the cost centers for which the obtained expressions for $k = 2$ are smaller than the expression obtained for $k = 1$. Note that the upper bounds are not constant, but rather are cost expressions (see, e.g., the upper bounds in Example 6.6.1). Thus, cost expressions can be easily compared to see which one is larger, but it is often not easy to quantify the gain (as such gain might in turn not be constant).

Let us summarize the main conclusions of the experiments. The object-insensitive columns show that cost analysis for concurrent objects is feasible and efficient if we assume a single cost center for the whole application. Also, as

expected, the number of different objects identified by the points-to analysis is larger when we apply the analysis with $k = 2$ than for $k = 1$. This increment in precision with $k = 2$ results in more precise upper bounds than those obtained with k=1. In almost all cases, more than the 50% of the cost centers improve their results, only Chat shows a lower percentage. There is an efficiency vs. precision tradeoff as achieving further precision requires generating a larger number of equations and hence the process of inferring the upper bounds is less efficient. Our system lets the user set up the value of k.

6.8 Related Work

Our work is closely related to other resource usage analysis frameworks [GMC09, HH10]. Most of such frameworks assume a sequential execution model and thus do not deal with the main challenges addressed in this chapter. Notable exceptions are [KPJ10, FM95]. A live heap space analysis for a concurrent language is proposed in [KPJ10] for a simple model of shared memory which only considers a particular type of resource (memory). A completely different approach to ours is the use of *dynamic matrices* for modeling cost analysis of concurrent programs as introduced in [FM95]. The use of cost centers has been proposed in the context of profiling, but to our knowledge, its use in the context of static analysis is new.

The termination of multi-threaded programs presented in [CPR07] is based on inferring conditions on the global state that are sufficient to guarantee termination and are similar to our class invariants. Observe that such conditions are only one component within our cost analysis framework, which additionally requires the generation of a new form of recurrence relations and the definition of cost models for the concurrent setting. The particular case of occurrence counting analysis in mobile systems of processes, which in our proposal can be obtained using a particular cost model, has been addressed by several contributions in the literature, although they focus on high-level models, such as the π-calculus and BioAmbients [Fer01, RL05].

When considering cumulative cost models, as we do in this thesis, asynchronous calls can be handled exactly as synchronous calls without sacrifying

precision. This is because, in such cost models, what is important is to approximate the number of times a method is executed (i.e., called), and not how many of them might be running in parallel. In contrast, when considering noncumulative cost models, information on the lifetime of each task is important, since it might directly affect the peak consumption of the corresponding resource. As future work, we plan to integrate in our framework cost models that are noncumulative [AAGZ11].

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The main motivation of this thesis has been to develop novel techniques that help improve the efficiency, accuracy and reliability of the resource analysis for Java-like programs. All the techniques described in this thesis have been implemented and evaluated experimentally proving that our theoretical results can be applied in practice.

Let us summarize the most relevant conclusions that can be drawn from this thesis:

(1) We have presented a practical approach to heap-sensitive cost analysis that is able to infer the aliasing conditions under which the termination of loops over heap-allocated data can be ensured. The main strengths of this approach are:

It handles object fields and arrays in an uniform way.

Our approach handles in the same way all heap-allocated data. We have generalized the reference constancy analysis for inferring the access paths used not only for accessing object fields, but also for accessing array elements. Heap accesses can be replaced by non heap-allocated variables when such heap accesses are local in the considered scope.

We introduce the notion of locality partition.

There are cases in which the field is not local unconditionally because we do not have enough information for tracking the values stored in the heap. Our technique can infer automatically locality partitions and generate the conditions needed to track the heap locations. Given the locality conditions, the transformation of the heap accesses into local variables is sound if particular conditions are satisfied.

It infers aliasing preconditions on the input arguments.

Proving the global termination of the program requires the composition of all locality conditions inferred for the involved scopes. Our technique combines and propagates the aliasing conditions obtained for each scope to obtain the aliasing conditions in terms of the input arguments of the scopes that invoke it. If the conditions hold in the initial state, termination of the whole program is guaranteed.

(2) We have presented a novel incremental resource usage analysis technique that can handle modifications in the program in an incremental way. After a modification in the program, our method takes the previously computed analysis results and only recomputes the analysis of those parts affected by the modification. The main strengths of our work are:

It handles several abstract domains simultaneously.

Our incremental approach is multi-domain in the sense that it interleaves the computation of all pre-analysis domains and takes care of dependencies among them, invalidating and recomputing only partial pre-computed information.

It presents the notion of cost summary.

Any change might modify the overall cost of the program, forcing the recomputation of the UB previously obtained. Our work shows that keeping information about the cost subcomponents associated to each method enables the efficient recomputation of those subcomponents affected by the modification.

It is efficient in practice.

Experiments performed on real programs have shown that our solution is efficient in comparison to analyzing of the whole program from scratch after each modification. We have applied systematic experiments over the evaluated programs, and the results show that our incremental analysis performs much more efficiently in practice than non-incremental cost analysis.

(3) We have presented a framework for the generation of verified resource guarantees by combining the capabilities of a cost analyzer and a verification tool. The main strengths of our approach are:

It proves the correctness of the cost analysis results automatically.

Our work describes the combination of a resource analyzer and a formal verification tool to infer and verify resource guarantees in a fully automatic way. The resource analyzer generates the parts of an UB and outputs them as JML annotations in the analyzed program. Then, the verification tool verifies the correctness of the annotated program to produce verified resource guarantees.

It handles integers and heap-allocated data structures.

Our work shows that the verification of the UBs obtained for both integer manipulating programs and heap manipulating programs is feasible in practice. Heap manipulating programs require us to keep track and verify the size relations among heap-allocated data. To this aim, our framework extends the JML language with new annotations to declare structural heap properties, like path-length, cyclicity, reachability and sharing.

(4) We have presented a novel object-sensitive resource usage analysis for concurrent programs by relying on the information gathered by a points-to analysis. The main novelties of the object-sensitive approach presented in this thesis are:

It separates the cost of distributed components in cost centers.

Standard cost analysis for sequential programs assumes a single cost center which accumulates the cost of the whole execution. In our work we propose a novel form of object-sensitive CRS which keeps the cost of the diverse distributed components separate. The idea of having cost center in the equations is of general applicability.

It generates CRSs that can be solved by standard solvers.

The CRSs generated by our object-sensitive approach can still be solved into closed form upper/lower-bounds using standard solvers for cost analysis of sequential programs.

7.2 Future Work

We plan to extend the work presented in this thesis along the following directions:

(1) Heap accesses are already the bane of resource analysis for sequential programs. The problem is even more complex in the context of concurrent programs because different threads can access to shared memory in a interleaved way. We want to consider the thread-based concurrency of Java language and study how our heap-sensitive analysis has to be adapted to produce sound and precise results.

(2) The incremental resource usage analysis described in this thesis has been developed for sequential programs. We plan to study how to adapt our incremental algorithms to use them in a concurrent and distributed context in order to recompute only those analysis results of the distributed system that might be affected by the change.

(3) Software product lines[PBL05], or software product line development, refers to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production. It is quite natural to apply incremental analysis on such adaptative software. In such a way that every time a new product

is selected we only need to recompute the new features added, and still can reuse some precomputed information. The application of our incremental analysis to this context is an open and interesting problem that we plan to address in the near future. Cost analysis of adaptative software and product derivation may help to detect performance and safety issues in software. We plan to study the applicability of the incremental cost analysis to software product lines.

(4) We plan to improve the accuracy of the object-sensitive resource analysis by using the results gathered by a may-happen-in-parallel (MHP) analysis [AFMG12]. A MHP approximates the set of pairs of program points that may be executing in parallel. This information can prevent the size analysis from losing all information on object fields (see Section 6.4). Intuitively we can track a field if it surely cannot be modified by another task running in parallel with the current one.

Bibliography

[AAC+08] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 2–18. Springer, 2008.

[AAG+08] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *Procs. of SAC*, pages 368–375. ACM Press, 2008.

[AAG+09] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. *Electronic Notes of Theoretical Computer Science*, 258(1):109–121. Elsevier, 2009.

[AAG+10] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *Procs. of SAS 2010*, volume 6337 of LNCS, pages 100-116. Springer, 2010.

[AAG+11] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Procs. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, 2011.

[AAG+12a] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *Procs. of PEPM'12*, pages 151–154. ACM Press, 2012.

[AAG+12b] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[AAGP09] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *Procs. of FM'09*, volume 5850 of LNCS, pages 370–386. Springer, 2009.

[AAGP11] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[AAGZ11] E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *Procs. of LCTES'11*, pages 21–30. ACM Press, 2011.

[ABG+11a] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. Verified resource guarantees using COSTA and KeY. In *Procs. of PEPM'11*, pages 73–76. ACM Press, 2011.

[ABG+11b] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. *10th KeY Symposium*, August 2011.

[ABG+12] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Procs. of FASE'12*, volume 7212 of *LNCS*, pages 130–145. Springer, March 2012.

[ACPRD11] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Towards Incremental Resource Usage Analysis. In *APLAS'11*. Poster Presentation. December 2011.

[ACPRD12] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Incremental Resource Usage Analysis. In *Procs. of PEPM 2012*, pages 25–34. ACM Press, January 2012.

[AFKT03] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *Procs. of PLDI'03*, pages 129–140. ACM, 2003.

[AFMG12] E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of may-happen-in-parallel in concurrent objects. In *Formal Techniques for Distributed Systems*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.

[AGGZ10] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM Press, 2010.

[AGRD12] E. Albert, S. Genaim, and G. Román-Díez. Conditional Termination of Loops over Arrays. In *BYTECODE'12*, March 2012.

[All70] F. Allen. Control flow analysis. In *Procs. of a Symposium on Compiler Optimization*, pages 1–19, 1970.

[BBL+10] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier research platform. In *Proc. of CAV'10*, volume 6174 of *LNCS*, pages 119–122. Springer, 2010.

[BCDO06] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Procs. of CAV*, volume 4144 of *LNCS*, pages 386–400. Springer, 2006.

[Ben01] R. Benzinger. Automated Complexity Analysis of Nuprl Extracted Programs. *Journal Functional Programming*, 11(1):3–31, 2001.

[BHS06] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.

[BIK12] M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination, In *Procs. of TACAS'12*, volume 7214 of *LNCS*, pages 252–266. Springer, 2012.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Procs. of ECOOP'08*, volume 5142 of *LNCS*, pages 387–411, 2008.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Procs. of POPL'77*, pages 238–252. ACM Press, 1977.

[CC02] P. Cousot and R. Cousot. Modular Static Program Analysis, invited paper. In *Procs. of CC'02*, volume 2304 of *LNCS*, pages 159–179. Springer, 2002.

[CGLA+08] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Procs. of CAV'08*, volume 5123 of *LNCS*, pages 328–340, 2008.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Procs. of POPL'78*, pages 84–97. ACM Press, 1978.

[CL05] B. Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Procs. of VMCAI'05*, volume 3385 in *LNCS*, pages 147–163. Springer, 2005.

[CPR07] B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *Procs. of PLDI'07*, pages 320–330. ACM Press, 2007.

[CW00] K. Crary and S. Weirich. Resource Bound Certification. In *Procs. of POPL'00*, pages 184–198. ACM Press, 2000.

[dBCJ07] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Procs. of ESOP'07*, volum 4421 of *LNCS*, pages 316–330. Springer, 2007.

[DDA08] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Procs. of PLDI'08*, pages 270–280. ACM Press, 2008.

[DL93] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[DP11] J. De Dios and R. Peña. Certification of Safe Polynomial Memory Bounds. In *Procs. of FM'11*, pages 184–199. ACM Press, 2011.

[Fer01] J. Feret. Occurrence counting analysis for the pi-calculus. *Electronic Notes of Theoretical Computer Science*, 39(2):1–18, 2001.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. In *Procs. of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[FM95] G. L. Ferrari and U. Montanari. Dynamic Matrices and the Cost Analysis of Concurrent Programs. In *Procs. of AMAST'95*, volume 936 of *LNCS*, pages 307–321. Springer, 1995.

[FM07] J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Procs. of CAV'07*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[GL98] C. A. Lakos G. Lewis. Towards incremental analysis. In *Workshop on Formal Methods for Dependable Systems (FMDS)*, 1998.

[GKP12] S. Ganov, S. Khurshid, and D. Perry. Annotations for alloy: Automated incremental analysis using domain specific solvers. In *Procs. of ICFEM*, volume 7635 of *LNCS*, pages 414–429. Springer, 2012.

[GMC09] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Procs. of POPL'09*, pages 127–139. ACM Press, 2009.

[GS08] S. Genaim and F. Spoto. Constancy Analysis. In *10th Workshop on Formal Techniques for Java-like Programs*, 2008.

[GZ10] S. Genaim and D. Zanardini. The acyclicity inference of COSTA. In *11th International Workshop on Termination*, 2010.

[Hed89] G. Hedin. An object-oriented notation for attribute grammars. In *Procs. of ECOOP'89*, pages 329–345, 1989.

[HH10] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Procs. of ESOP'10*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, 2000.

[JH11] M. Hofmann J. Hoffmann, K. Aehlig. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.

[JHS+12] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Post Proc. of FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

[JO07] E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

[KMSS97] A. Kelly, K. Marriott, H. Søndergaard, and P.J. Stuckey. A generic object oriented incremental analyser for constraint logic programs. In *Procs. of the 20th Australasian Computer Science Conference*, pages 92–101, 1997.

[KPJ10] M. Kero, P. Pietrzak, and Nordlander J. Live Heap Space Bounds for Real-Time Systems. In *Proc. of APLAS'10*, volume 6461 of *LNCS*, pages 287–303. Springer, 2010.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. of LPAR'10*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[Log11] F. Logozzo. Practical verification for the working programmer with codecontracts and abstract interpretation - (invited talk). In *Procs. of VMCAI'11*, volume 6538 of *LNCS*, pages 19–22. Springer, 2011.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 1997.

[Mil03] A. Milanova. *Precise and practical flow analysis of object-oriented software*. PhD thesis, Rutgers University, 2003.

[Min06] A. Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *Procs. of LCTES'06*, pages 54–63. ACM, 2006.

[MRR05] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology*, 14:1–41, 2005.

[Nec97] G. Necula. Proof-Carrying Code. In *Procs. of (POPL'97)*, pages 106–119, ACM Press, 1997.

[NMLGH07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *Procs. of ICLP'07*, volume 4670 of *LNCS*, pages 348–363. Springer, 2007.

[NNH99] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[OBvEG09] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Termination Analysis of Java Bytecode by Term Rewriting. In *International Workshop on Termination WST'09*, Leipzig, Germany, June 2009.

[PYRK11] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Procs. of PLDI'11*, pages 504–515. ACM, 2011.

[PBL05] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PR04] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *Procs. of VMCAI'04*, volume 2937 of *LNCS*, pages 239–251, 2004.

[Pro] Apache Commons Project. http://commons.apache.org/.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proc. of TACAS'98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[RCP12] D. Ramírez, J. Correas, and G. Puebla. Modular Termination Analysis of Java Bytecode and its Application to phoneME Core Libraries. In *Procs. of FACS'10*, volume 6921 of *LNCS*, pages 218–236, 2012.

[RD11] D. Ramírez-Deantes. *Modular and field-sensitive termination analysis of java bytecode.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, 2011.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Procs. of LICS'02*, pages 55–74, 2002.

[RGM98] S. A. Jarvis R. G. Morgan. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing*, 8(3):201–237, 1998.

[RHS95] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Procs. POPL'95*, pages 49–61. ACM Press, 1995.

[RL05] R.Gori and F. Levi. A new occurrence counting analysis for bioambients. In *APLAS'05*, volume 3780 of *LNCS*, pages 381–400. Springer, 2005.

[Ros89] M. Rosendahl. Automatic Complexity Analysis. In *FPCA'89*. ACM Press, 1989.

[RS06] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 95–110. Springer, 2006.

[Ryd88] B. Ryder. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.

[SBL11] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *Procs. of POPL'11*, pages 17–30. ACM, 2011.

[SHP06] F. Spoto, P.M. Hill, and E. Payet. Path-Length Analysis of Object-Oriented Programs. In *Procs. of EAAI'06*, 2006.

[SJ03] F. Spoto and T. P. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Transactions on Programming Languages and Systems*, 25(5):578–630, 2003.

[SJPW08] J. Smans, B. Jacobs, F. Piessens, and Schulte W. An automatic verifier for Java-like programs based on dynamic frames. In *Procs. of FASE'08*, volume 4961 of *LNCS*, pages 261–275. Springer, 2008.

[SM08] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008.

[SMP10] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode based on Path-Length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[SPH10] J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing Active Objects to Concurrent Components. In *Procs. of ECOOP'10*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.

[SRW99] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Procs. of POPL'99*, pages 105–118, 1999.

[SS05] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In *Procs. of SAS'05*, volume 3672 of *LNCS*, pages 320–335. Springer, 2005.

[Sui] JOlden Suite. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[Weg75] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.

[Wei11] B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[WG97] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *Procs. of PLDI'97*, pages 31–43, 1997.