



HAL
open science

Static analysis via abstract interpretation of multithreaded programs

Pietro Ferrara

► **To cite this version:**

Pietro Ferrara. Static analysis via abstract interpretation of multithreaded programs. Software Engineering [cs.SE]. Ecole Polytechnique X, 2009. English. NNT: . tel-00417502

HAL Id: tel-00417502

<https://pastel.hal.science/tel-00417502>

Submitted on 16 Sep 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE
PRÉSENTÉE À
L'ÉCOLE POLYTECHNIQUE

POUR OBTENIR LE TITRE DE
DOCTEUR EN SCIENCES DE L'ÉCOLE POLYTECHNIQUE

Discipline
Informatique

par

PIETRO FERRARA
22 mai 2009

ANALYSE STATIQUE DE LOGICIELS MULTITÂCHES
PAR INTERPRÉTATION ABSTRAITE

Static analysis via abstract interpretation of multithreaded programs

President

Manuel Hermenegildo

Professeur, Universidad Politécnica Madrid, Espagne

Rapporteurs

Manuel Hermenegildo

Professeur, Universidad Politécnica Madrid, Espagne

Helmut Seidl

Professeur, Technische Universität München, Allemagne

Examineurs

Eric Goubault

Directeur de Recherche, CEA, France

Francesco Logozzo

Chercheur, Microsoft Research, Etats Unis

Directeurs de thèse

Radhia Cousot

Directeur de recherche, CNRS/ENS, France

Agostino Cortesi

Professeur, Università Ca' Foscari di Venezia, Italie

Résumé

Le but de cette thèse est de présenter une analyse statique générique pour des programmes multitâche écrits en Java.

Les programmes multitâche exécutent plusieurs tâches en parallèle. Ces tâches communiquent implicitement par le biais de la mémoire partagée et elles se synchronisent sur des moniteurs (les primitives `wait` – `notify`, etc, ...). Il y a quelques années, les architectures avec double processeurs ont commencé à être disponibles sur le marché à petit prix. Aujourd'hui, presque tous les ordinateurs ont au moins deux noyaux, la tendance actuelle du marché étant de mettre de plus en plus de processeurs par puce. Cette révolution amène également de nouveaux défis en matière de programmation, car elle demande aux développeurs d'implanter des programmes multitâche. Le multitâche est supporté en natif par la plupart des langages de programmation courants, comme Java et C#.

Le but de l'analyse statique est de calculer des informations sur le comportement d'un programme, de manière conservative et automatique. Une application de l'analyse statique est le développement d'outils qui aident au débogage des programmes. Plusieurs méthodes d'analyse statique ont été proposées. Nous suivons le cadre de l'interprétation abstraite, une théorie mathématique permettant de définir des approximations correctes de sémantiques de programmes. Cette méthode a déjà été utilisée pour un large spectre de langages de programmation.

L'idée fondamentale des analyseurs statiques génériques est de développer un outils qui puissent être interfacé avec différents domaines numériques et différentes propriétés. Pendant ces dernières années, beaucoup de travaux se sont attaqués à cet enjeu, et ils ont été appliqués avec succès pour déboguer des logiciels industriels. La force de ces analyseurs réside dans le fait qu'une grande partie de l'analyse peut être réutilisée pour vérifier plusieurs propriétés. L'utilisation de différents domaines numériques permet le développement d'analyses plus rapides mais moins précises, ou plus lentes mais plus précises.

Dans cette thèse, nous présentons la conception d'un analyseur générique pour des programmes multitâche. Avant tout, nous définissons le modèle mémoire, appelé *happens-before memory model*. Puis, nous approximons ce modèle mémoire en une sémantique calculable. Les modèles mémoire définissent les comportements autorisés pendant l'exécution d'un programme multitâche. Commencant par la définition (informelle) de ce modèle mémoire particulier, nous définissons une sémantique qui construit toutes les exécutions finies selon ce modèle mémoire. Une exécution d'un programme multitâche

est décrite par une fonction qui associe les tâches à des séquences (ou traces) d'états. Nous montrons comment concevoir une sémantique abstraite calculable, et nous montrons formellement la correction des résultats de cette analyse.

Ensuite, nous définissons et approximons une nouvelle propriété qui porte sur les comportements non déterministes causés par le multitâche, c'est à dire ceux qui sont dus aux entrelacements arbitraires pendant l'exécution de différentes instructions de lecture. Avant tout, le non déterminisme d'un programme multitâche se définit par une différence entre plusieurs exécutions. Si deux exécutions engendrent des comportements différents dus aux valeurs qui sont lues ou écrites en mémoire partagée, alors le programme est non déterministe. Nous approximons cette propriété en deux étapes : dans un premier temps, nous regroupons, pour chaque tâche, la valeur (abstraite) qui peut être écrite dans la mémoire partagée à un point de programme donné. Dans un deuxième temps, nous résumons toutes les valeurs pouvant être écrites en parallèle, tout en nous rappelant l'ensemble des tâches qui pourraient les avoir écrites. À un premier niveau d'approximation, nous introduisons un nouveau concept de déterminisme faible. Nous proposons par ailleurs d'autres manières d'affaiblir la propriété de déterminisme, par exemple par projection des traces et des états, puis nous définissons une hiérarchie globale de ces affaiblissements. Nous étudions aussi comment la présence de conflit sur les accès des données peut affecter le déterminisme du programme.

Nous appliquons ce cadre de travail théorique à Java. En particulier, nous définissons une sémantique du langage objet de Java, selon sa spécification. Ensuite, nous approximons cette sémantique afin de garder uniquement l'information qui est nécessaire pour l'analyse des programmes multitâche. Le cœur de cette abstraction est une analyse d'alias qui approxime les références afin d'identifier les tâches, de vérifier les accès en mémoire partagée, et de détecter quand deux tâches ont un moniteur commun afin d'en déduire quelles parties du code ne peuvent pas être exécutées en parallèle.

L'analyseur générique qui est décrit ci-dessus a été entièrement implanté, dans un outil appelé Checkmate. Checkmate est ainsi le premier analyseur générique pour des programmes multitâche écrits en Java. Des résultats expérimentaux sont donnés et analysés en détails. En particulier, nous étudions la précision de l'analyse lorsqu'elle est appliquée à des schémas courants de la programmation concurrente, ainsi qu'à d'autres exemples. Nous observons également les performances de l'analyse lorsqu'elle est appliquée à une application incrémentale, ainsi qu'à des exemples de référence bien connus.

Une autre contribution de cette thèse est l'extension d'un analyseur générique existant qui s'appelle Clousot et qui permet de vérifier le non débordement des mémoires tampons. Il s'avère que cette analyse passe à l'échelle des programmes industriels et qu'elle est précise. En résumé, nous présentons une application d'un analyseur statique générique industriel existant pour détecter et prouver une propriété présentant un intérêt pratique, ce qui montre la puissance de cette approche dans le développement d'outils qui soient utiles pour les développeurs.

Riassunto

L'obiettivo di questa tesi è di presentare un'analisi statica generica per programmi Java multithread.

Un programma multithread esegue molteplici task, chiamati thread, in parallelo. I thread comunicano implicitamente attraverso una memoria condivisa, e si sincronizzano attraverso monitor, primitive wait-notify, etc... Le prime architetture dual-core sono apparse sul mercato a prezzi contenuti alcuni anni fa; oggi praticamente tutti i computer sono almeno dual-core. L'attuale trend di mercato è addirittura quello del many-core, ovvero di aumentare sempre di più il numero di core presenti su una CPU. Alcune nuove sfide sono state introdotte da questa rivoluzione multicore a livello di linguaggi di programmazione, dal momento che gli sviluppatori software devono implementare programmi multithread. Questo pattern di programmazione è supportato nativamente dalla maggior parte dei linguaggi di programmazione moderni come Java e C#.

Lo scopo dell'analisi statica è di calcolare automaticamente e in maniera conservativa una serie di informazioni sul comportamento a tempo di esecuzione di un programma; una sua applicazione è lo sviluppo di strumenti che aiutino a trovare e correggere errori software. In questo campo svariati approcci sono stati proposti: nel corso della tesi verrà seguita la teoria dell'interpretazione astratta, un approccio matematico che permette di definire e approssimare correttamente la semantica dei programmi. Questa metodologia è già stata utilizzata con successo per l'analisi di un vasto insieme di linguaggi di programmazione. Gli analizzatori generici possono essere istanziati con diversi domini numerici e applicati a svariate proprietà. Negli ultimi anni numerosi lavori sono stati centrati su questo approccio, e alcuni di essi sono stati utilizzati con successo in contesto industriale. Il loro punto di forza è il riutilizzo della maggior parte dell'analizzatore per verificare molteplici proprietà, e l'utilizzo di diversi domini numerici permette di ottenere analisi più veloci ma più approssimate, oppure più precise ma più lente.

Nel corso di questa tesi presenteremo un analizzatore generico per programmi multithread.

Definiremo innanzitutto il modello di memoria happens-before sotto forma di punto fisso e lo approssimeremo con una semantica che sia calcolabile. Un modello di memoria definisce quali comportamenti di un programma multithread sono consentiti durante la sua esecuzione. A partire da una definizione informale del modello di memoria happens-before, introdurremo una semantica che costruisca tutte le esecuzioni finite che rispettino tale modello di memoria; in tale contesto un'esecuzione è rappresentata come una fun-

zione che associa ciascun thread ad una traccia di stati che rappresenta la sua esecuzione. Introduciamo infine una semantica astratta che può essere calcolata, provandone la correttezza formalmente.

Definiremo e approssimeremo quindi una nuova proprietà focalizzata sui comportamenti non deterministici causati dall'esecuzione multithread (ad esempio dall'intercalarsi arbitrario durante l'esecuzione in parallelo di diversi thread). Prima di tutto, il non determinismo di un programma multithread è definito come differenza tra esecuzioni. Un programma è non deterministico se due diverse esecuzioni espongono comportamenti differenti a causa dei valori letti e scritti sulla memoria condivisa. Astrarremo quindi tale proprietà su due livelli: inizialmente tratteremo per ogni thread il valore astratto che potrebbe aver scritto sulla o letto dalla memoria condivisa. Al successivo passo di astrazione tratteremo un solo valore, che approssimerà tutti i possibili valori scritti in parallelo, e l'insieme dei thread che potrebbero aver fatto ciò. Sul primo livello di astrazione definiremo poi il concetto di determinismo debole. Proporrò quindi diverse modalità di rilassamento di tale proprietà, in particolare proiettandola su un sottoinsieme delle tracce di esecuzione e degli stati, definendo una gerarchia complessiva. Infine studieremo come la presenza di data race possa influenzare il determinismo di un programma.

Tutto questo lavoro teorico verrà quindi applicato a programmi Java. In particolare definiremo una semantica concreta del linguaggio Java bytecode seguendo la sua specifica. Quindi lo approssimeremo in maniera da astrarre precisamente le informazioni richieste per poter analizzare un programma multithread. Il fulcro di ciò è l'approssimazione degli indirizzi di memoria per poter identificare i diversi thread, per controllare gli accessi alla memoria condivisa e per poter scoprire quando due thread sono sempre sincronizzati su uno stesso monitor e quindi quali parti di codice non possono essere eseguite in parallelo. L'analizzatore generico definito fin qui formalmente è stato implementato in *Checkmate*, il primo analizzatore generico di programmi Java multithread. Riporteremo e studieremo approfonditamente i risultati sperimentali: in particolare verrà studiata la precisione dell'analisi quando utilizzata su alcuni pattern comuni di programmazione concorrente e alcuni casi di studio, e le sue prestazioni quando eseguita su un'applicazione incrementale e su un insieme di benchmark esterni.

L'ultimo contributo della tesi sarà l'estensione di un analizzatore generico industriale esistente (*Clousot*) all'analisi degli accessi effettuati tramite puntatori diretti alla memoria. In questa parte finale presenteremo l'applicazione di un analizzatore generico ad una proprietà di interesse pratico su codice industriale, mostrando quindi la forza di questo tipo di approccio allo scopo di costruire strumenti utili per sviluppare software.

Abstract

The goal of this thesis is to present a generic static analysis of Java multithreaded programs.

Multithreaded programs execute many task, called threads, in parallel. Threads communicate through the shared memory implicitly, and they synchronize on monitors, wait-notify primitives, etc... Some years ago dual core architectures started being distributed on the broad market at low price. Today almost all the computers are at least dual core. Many-core, i.e. putting more and more cores on the same CPU, is now the current trend of CPU market. This multicore revolution yields to new challenges on the programming side too, asking the developers to implement multithreaded programs. Multithreading is supported natively by the most common programming languages, e.g. Java and C#.

The goal of static analysis is to compute behavioral information about the executions of a program, in a safe and automatic way. An application of static analysis is the development of tools that help to debug programs. In the field of static analysis, many different approaches have been proposed. We will follow the framework of abstract interpretation, a mathematical theory that allows to define and soundly approximate semantics of programs. This methodology has been already applied to a wide set of programming languages.

The basic idea of generic analyzers is to develop a tool that can be plugged with different numerical domains and properties. During the last years many works addressed this issue, and they were successfully applied to debug industrial software. The strength of these analyzers is that the most part of the analysis can be re-used in order to check several properties. The use of different numerical domains allows to develop faster and less precise or slower and more precise analyses.

In this thesis, the design of a generic analyzer for multithreaded programs is presented. First of all, we define the happens-before memory model in fixpoint form and we abstract it with a computable semantics. Memory models define which behaviors are allowed during the execution of a multithreaded program. Starting from the (informal) definition of the happens-before memory model, we define a semantics that builds up all the finite executions following this memory model. An execution of a multithreaded program is represented as a function that relates threads to traces of states. We show how to design a computable abstract semantics, and we prove the correctness of the resulting analysis, in a formal way.

Then we define and abstract a new property focused on the non-deterministic behaviors

due to multithreading, e.g. the arbitrary interleaving during the execution of different threads. First of all, the non-determinism of a multithreaded program is defined as difference between executions. If two executions expose different behaviors because of values read from and written to the shared memory, then that program is not deterministic. We abstract it in two steps: in the first step we collect, for each thread, the (abstract) value that it may write into a given location of the shared memory. At the second level we summarize all the values written in parallel, while tracking the set of threads that may have written it. At the first level of abstraction, we introduce the new concept of weak determinism. We propose other ways in order to relax the deterministic property, namely by projecting traces and states, and we define a global hierarchy. We formally study how the presence of data races may afflict the determinism of the program.

We apply this theoretical framework to Java. In particular, we define a concrete semantics of bytecode language following its specification. Then we abstract it in order to track the information required by the analysis of multithreaded programs. The core is an alias analysis that approximates references in order to identify threads, to check the accesses to the shared memory, and to detect when two threads own a common monitor thereby inferring which parts of the code cannot be executed in parallel.

The generic analyzer described above has been fully implemented, leading to **Checkmate**, the first generic analyzer of Java multithreaded programs. We report and deeply study some experimental results. In particular, we analyze the precision of the analysis when applied to some common pattern of concurrent programming and some case studies, and its performances when applied to an incremental application and to a set of well-known benchmarks.

An additional contribution of the thesis is about the extension of an existing industrial generic analyzer, **Clousot**, to the checking of buffer overrun. It turns out that this analysis is scalable and precise. In summary, we present an application of an existing, industrial, and generic static analyzer to a property of practical interest, showing the strength of this approach in order to develop useful tools for developers.

Acknowledgments

First of all, I would like to thank my PhD advisors, Radhia Cousot and Agostino Cortesi, to have introduced me to abstract interpretation, and to have strongly supported my work throughout all my thesis. Their encouragements, suggestions, and enthusiasm were very helpful to me.

Manuel Hermenegildo and Helmut Seidl accepted to be the reviewers of my thesis: I am proud of that, and they deserve my biggest thanks for the time spent to read, comment, and discuss the critical points of my work. I would like to thank also Eric Goubault for taking part in my jury.

I met Francesco Logozzo about 5 years ago. At that time, I was amazed by his passion and his strong principles. His enthusiasm as young researcher touched me. I had the pleasure and the honor of being one of his interns at Microsoft Research, and to have him in my jury. For all these things, I am particularly grateful to him.

Patrick Cousot deserves a special thank for his great work. His course at École Normale Supérieure was the best way to learn the deepest concepts of abstract interpretation. I want to thank all the actual and former members of Cousots' *equipe* that I met during my thesis: first of all, Guillaume Capron and Elodie-Jane Sims that were my co-PhD at Ecole Polytechnique, and also Julien Bertrane, Bruno Blanchet, Liqian Chen, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival, and Axel Simon.

Many thanks go also to all my friends, and in particular to Yasmina, Nicolas, Cesar, Carolina, and China. It is impossible to remember all of them, so I chose to cite only the ones that were at my PhD defense. These thanks extend to all other my friends of course. I am particularly grateful to my family, that supported me all along my life, and thus during the three years spent on my PhD thesis. In particular, I want to mention my mother Luisella, my father Pino, my brother Jacopo, and my grandmother Lidia.

Last but not least, my deepest thank goes to Francesca, that strongly encouraged and sustained me and my work. The time spent together in Paris will be one of the best *souvenirs* of my life.

Contents

1	Introduction	1
1.1	Motivation	2
1.1.1	Multicore Architectures: Why, Where, When	2
1.1.2	Multithreading	4
1.1.3	Static Analysis	4
1.2	Context	5
1.2.1	Memory Models	5
1.2.2	Static Analyses of Multithreaded Programs	6
1.2.3	Generic Static Analyzers	7
1.3	Contribution	7
1.3.1	Static Analysis of the Happens-Before Memory Model	7
1.3.2	Determinism of Multithreaded Programs	8
1.3.3	A Generic Static Analyzer of Java Multithreaded Programs	8
1.3.4	An Industrial Case Study: Unsafe Code	8
1.4	Overview of the Thesis	9
2	Preliminaries	11
2.1	Notation	11
2.1.1	Sets	11
2.1.2	Partial Orders and Lattices	12
2.1.3	Functions	12
2.1.4	Fixpoints	13
2.1.5	Traces	14
2.2	Abstract Interpretation	14
2.2.1	Galois Connections	15
2.2.2	Fixpoint Approximation	16
2.2.3	Widening	16
2.3	Running Example	17
3	Static Analysis of the Happens-Before Memory Model	21
3.1	Memory Models	21
3.1.1	An Example	22
3.2	The Happens-Before Memory Model	23
3.2.1	Reasoning Statically	24
3.2.2	The Example	25
3.3	Multithreaded Concrete Semantics	25
3.3.1	Assumptions	25

3.3.2	Thread-Partitioning Concrete Domain	26
3.3.3	Single Step Function	28
3.3.4	Fixpoint Semantics	30
3.3.5	Launching a Thread	32
3.3.6	The Example	32
3.4	Multithreaded Abstract Semantics	33
3.4.1	Assumptions	33
3.4.2	Thread-partitioning Abstract Domain	35
3.4.3	Upper Bound Operators	35
3.4.4	Partial Order Operators	36
3.4.5	Abstraction Functions	41
3.4.6	\overline{step} Function	42
3.4.7	Fixpoint Semantics	43
3.4.8	Launching a Thread	47
3.4.9	The Example	47
3.5	Related work	47
3.6	Discussion	49
3.6.1	Thread Identifiers	49
3.6.2	Monitors	49
3.6.3	Modular Analysis of Multithreaded Programs	49
4	Determinism of Multithreaded Programs	51
4.1	Analyzing Multithreaded Programs	51
4.1.1	Data Races	51
4.1.2	Model of Execution	52
4.1.3	An Example	52
4.2	Syntax and Concrete Semantics	53
4.2.1	Syntax	53
4.2.2	Concrete Domain	53
4.2.3	Transfer Function	54
4.2.4	An Example	54
4.3	A Value for Each Thread (Abstraction 1)	54
4.3.1	Abstract Domain (First Level)	54
4.3.2	Upper Bound Operator	55
4.3.3	Abstraction Function	55
4.3.4	Transfer Function	58
4.3.5	The Example	59
4.4	Just one Value (Abstraction 2)	60
4.4.1	Abstract Domain (Second Level)	60
4.4.2	Upper Bound Operator	60
4.4.3	Abstraction Function	61
4.4.4	Transfer Function	63
4.4.5	The Example	65

4.5	The Deterministic Property	65
4.5.1	Determinism	65
4.5.2	Formal Definition of Determinism on the Concrete Domain . . .	66
4.5.3	First Level of Abstraction	66
4.5.4	Second Level of Abstraction	67
4.5.5	The Example	68
4.6	Weak Determinism	69
4.6.1	Approximating Numerical Values	69
4.6.2	Formal Definition	69
4.6.3	Example 2	70
4.7	Tracing Nondeterminism	70
4.7.1	Modifying a Value	70
4.7.2	An Example	71
4.7.3	Writing on the Shared Memory	72
4.7.4	Discussion	72
4.8	Projecting Traces and States	73
4.8.1	Concrete States	73
4.8.2	Abstract States	74
4.8.3	Concrete Traces	75
4.8.4	Abstract States	76
4.8.5	Projecting both States and Traces	76
4.8.6	Hierarchy	77
4.8.7	An example	78
4.8.8	Discussion	79
4.9	SQL Phenomena	79
4.9.1	The SQL Approach	79
4.9.2	SQL Phenomena in our Framework	80
4.9.3	Effects of Phenomena on the Determinism	80
4.9.4	Phenomena and Deterministic Property	81
4.9.5	In the Abstract	82
4.10	Data Race Condition	82
4.10.1	Synchronization	82
4.10.2	Data Races and SQL Phenomena	83
4.10.3	Deterministic Property	83
4.10.4	Abstract States	84
4.11	From Determinism to Semi-Automatic Parallelization	84
4.11.1	Motivation	84
4.11.2	Determinism and Parallelism	84
4.11.3	Relaxing the Deterministic Property	84
4.11.4	An example	85
4.12	Related Work	85
4.13	Discussion	86
4.13.1	Relational Domains	86

4.13.2	States in Traces	87
4.13.3	Thread Identifiers	87
5	Concrete and Abstract Domain and Semantics of Java Bytecode	89
5.1	Notation	89
5.2	Supported Language	90
5.3	An Example	91
5.4	Concrete Domain	91
5.5	Concrete Operational Semantics	93
5.5.1	Load and Store	93
5.5.2	Monitors	93
5.5.3	Objects	93
5.5.4	Arrays	94
5.5.5	Arithmetic Expressions	94
5.5.6	Constants	95
5.5.7	Jumps	95
5.5.8	Method Invocation	95
5.5.9	Applying it to the Example	96
5.6	Control Flow Graph	98
5.6.1	Formal Definition	98
5.6.2	Soundness with respect to $\langle \wp(\Sigma^{\mp}), \subseteq \rangle$	99
5.7	Method Calls	100
5.8	Abstract Domain	100
5.8.1	Alias Analysis	101
5.8.2	Domain	103
5.9	Abstract Operational Semantics	104
5.9.1	Load and Store	104
5.9.2	Monitors	104
5.9.3	Objects	105
5.9.4	Arrays	106
5.9.5	Arithmetic Expressions	106
5.9.6	Constants	106
5.9.7	Jumps, If and Method Calls	107
5.9.8	Applying it to the Example	107
5.10	Soundness	107
5.10.1	Domain	107
5.10.2	Semantics	109
5.10.3	Objects	109
5.11	Related Work	111
5.12	Application to the Happens-Before Memory Model	112
5.12.1	Concrete Domain	112
5.12.2	Abstract Thread Identifiers	113
5.12.3	Abstract Domain	113

5.13	Application to the Deterministic Property	114
5.13.1	Concrete Domain	114
5.13.2	Abstract Domain	114
5.13.3	Second Level of Abstraction	115
5.14	Discussion	115
6	Checkmate: a Generic Static Analyzer of Java Multithreaded Programs	117
6.1	Generic Analyzers	117
6.2	On Native Methods	118
6.3	An Example	119
6.4	Structure	120
6.4.1	Property	120
6.4.2	Numerical Domain	122
6.4.3	Memory Model	122
6.4.4	An Example of Interaction	123
6.5	Parameters	125
6.5.1	Properties	125
6.5.2	Numerical Domain	125
6.5.3	Memory Models	126
6.6	User Interfaces	127
6.6.1	Command Line	127
6.6.2	Eclipse Plugin	129
6.7	Experimental Results	130
6.7.1	Common Patterns of Multithreaded Programs	130
6.7.2	Weak Memory Model	132
6.7.3	Incremental Example	134
6.7.4	Benchmarks	138
6.8	Related Work	140
6.8.1	Concurrency Properties	140
6.8.2	Other properties	143
6.9	Discussion	143
7	Static Analysis of Unsafe Code	145
7.1	What is Unsafe Code	145
7.2	Design by Contracts	146
7.2.1	Foxtrot	147
7.3	Our Contribution	147
7.3.1	Clousot	147
7.3.2	Applying Clousot to the Analysis of Unsafe Code	149
7.4	Examples	150
7.4.1	From Source Code to MSIL	151
7.4.2	Array Initialization	152
7.4.3	Callee Checking	152

7.4.4	Interaction with the Operating System	153
7.5	Syntax and Concrete Semantics	153
7.5.1	Syntax	154
7.5.2	Concrete Domain	155
7.5.3	Concrete Transition Semantics	155
7.6	Abstract Semantics	157
7.6.1	Abstracting Away the Values	157
7.6.2	Generic Memory Access Analysis	159
7.7	The Right Numerical Abstract Domain	161
7.8	The Stripes Abstract Domain	165
7.8.1	Constraints	165
7.8.2	Abstract Domain Structure	165
7.8.3	Refinement of the Abstract State	167
7.8.4	Transfer Functions	168
7.8.5	Representation of Strp	168
7.9	Refined Abstract Semantics	168
7.9.1	Checking Lower Bounds of Accesses	169
7.9.2	Compilation of fixed	171
7.10	Experiments	172
7.10.1	System.Drawing Case Study	173
7.10.2	Summary	174
7.11	Related Work	174
7.12	Discussion	176
8	Conclusions	177
A	Source Code of Examples Taken from [85]	179
A.1	ExpandableArray	179
A.2	LinkedCell	181
A.3	Document	183
A.4	Dot	184
A.5	Cell	185
A.6	TwoLockQueue	186
A.7	Account package	188
B	Incremental application	193
B.1	Account	193
B.2	ATM	193
B.3	Bank	194
B.4	BankAccount	194
B.5	Card	195
B.6	Cheque	196
B.7	Money	197

B.8 Person	197
B.9 ThreadATM	198
B.10 ThreadDeposit	199
B.11 ThreadInterests	199
B.12 ThreadWithdraw	200
B.13 TransferFunds	200
B.14 Test	201
Bibliography	203

1

Introduction

In this thesis we present a generic approach to the static analysis of multithreaded programs based on abstract interpretation. In particular, we present a generic framework for the static analysis of object oriented programs containing multithreading and we instantiate it to the analysis of `Java`. Finally, we show the effectiveness of designing generic static analyzers by extending an industrial one in order to analyze buffer overruns.

Static analysis of programs proves properties that are satisfied by all the possible executions at compile time. Since all the executions of a program cannot be computed (because of inputs, arbitrary interleaving of threads, etc..), static analysis needs to introduce approximation. Because of this abstraction, we may not be able to prove the correctness of a program even if all the executions respect the property of interest. In this case, our analysis would produce false alarms. The optimal goal is to build a static analysis that is precise enough to capture properties of interest, while producing as few as possible false alarms, and coarse enough to be computable in an efficient way. Until now, static analysis has been applied only to a relatively small part of the software developed worldwide, while the most part is only tested on a finite number of cases. The situation seems to be changing [37]: tools which automatically analyze programs are becoming of practical use for almost all developers that have to deal with larger and mostly critical programs.

Multithreading consists in partitioning a large application into many different sub-tasks, each of them possibly running in parallel. Threads can communicate through shared memory and they can synchronize each other through monitors, rendez-vous style constructs (e.g. semaphores), etc.. The arbitrary interleaving occurring during the parallel execution of different threads may lead to nondeterministic behaviors. These may be difficult to reproduce, as they may depend upon a particular sequence of interleaving, a specific compiler's optimization, etc. It is therefore generally known that developing multithreaded applications is strictly more difficult than designing sequential programs.

The development of bug-free sequential programs is proven impossible in the practice. In addition, multithreaded programs are particularly bug-prone. In this context, tools able to discover bugs and provide useful informations on them are particularly welcomed. This motivates why the design of static analyses for multithreaded programs is not only a challenging topic at theoretical level, but also an appealing issue from a practical and industrial point of view.

1.1 Motivation

Multithreading appears to be the most common way to build parallel applications in commercial programming languages. Parallelism cannot be avoided: it is the only immediate and native way in order to take advantage from multicore architectures, that represent the most important trend of CPU market today. On the other hand, it is hard to debug multithreaded programs. In fact, arbitrary interleaving and compiler's optimizations may expose unexpected behaviors. In addition, these are often difficult to reproduce.

1.1.1 Multicore Architectures: Why, Where, When

The focus of researchers on better models and tools for multithreaded programs are induced by the multicore revolution.

Why should we be more interested in multicore architectures today than 10 years ago?

“Manufacturers have found themselves unable to effectively continue improving microprocessor performance the old-fashioned way – by shrinking transistors and packing more of them onto single-core chips. (...) Vendors are increasing performance by building chips with multiple cores. (...) 16-core chips will be available by the end of this decade. Intel has already developed a research chip with 80 cores.” [52] (D. Geer, 2007)

In this context, multicore hardware seems to be the only way in order to extend Moore's law into the future. Increasing single-core performances today is too expensive and difficult: it's better to start thinking about multiple cores on the same computer. In addition, many-core will be the trend in the next future. In fact, prototype of 80 cores (Intel © Teraflops Research Chip [72]) appeared in 2006, and 8 and 16 cores processors (e.g. Intel © Xeon) are already available on the market. On the other hand, another question arises: are we still interested in Moore's law? Processors today are enough powerful in order to satisfy all the main needs of common people.

“Moore's law holds because it is profitable for semiconductor manufacturers; for forty years, large numbers of consumers have paid high prices for the highest performance designs. The revenue from this market has enabled the research and development needed to move along the exponential growth path. If multicore microprocessors fail to impress consumers, sales will fall flat, and revenue will be reduced” [12] (M. C. Bell and P. H. Madden, 2006)

A great economic interest is behind the multicore revolution. In addition, developing more and more powerful processors often opens up new applications. Until some years ago, computers were mostly used to write and store documents. Then Internet arrived, and they were used to write and send email, download and see textual documents, communicate with other people. Today they are often used for multimedia documents, as watching videos and listening music. New applications that fully exploit powerful graphic cards

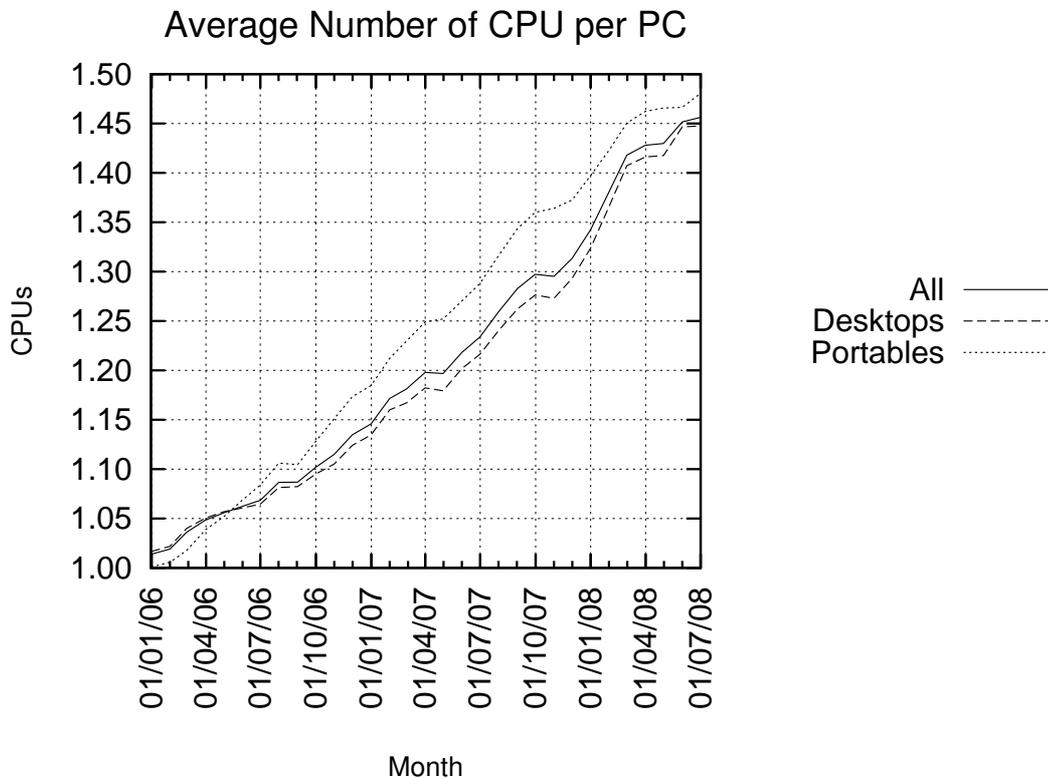


Figure 1.1: CPU per PC (© 2008 PC Pitstop Research)

are commonplace on most of personal computers. As more powerful processors will be released, new applications will exploit them.

The last question to be considered is: when and by whom will multicore architectures be bought and exploited?

“Dual-core processors first appeared on the market in 2001. (...) The greatest change in processor architecture came with the dual-core processors that AMD and Intel introduced in 2005. (...) A desktop computer with a dual-core processor can today be bought for less than \$500” [96] (A. Marowka, 2007)

It suffices to look at the market of PCs to understand that almost all the computers sold today are multicore. In the next future the most part of common people (i.e. not only researchers or specialists of information technology) will exploit multicore architecture. Figure 1.1 depicts the trend in the market of multicore PC from January 2006 to July 2008. While until 3 years ago almost all PCs had one CPU (in fact the average CPU per PC was about 1.00), at July 2008 the average was about 1.50 (so, since the most part of multicore PCs were dual core, about the 50% was multicore). In 2 years and half, about the 50% of PCs passed from single-core to multicore.

Multicore architectures cannot be avoided and are already sold to the masses; what we need is to exploit them as much as possible. If we will not take advantage of these architectures, people will not be interested anymore in buying multicore computers.

1.1.2 Multithreading

“Any application that will run on a single-core Intel processor will run on an Intel dual-core processor. However, in order for an application to take advantage of the dual-core capabilities, the application should be optimized for multithreading” [79] (G. Koch, 2005)

“Multithreaded programming breaks an application into subtasks, or “threads”, that run concurrently and independently. To take advantage of multi-core processors, applications must be redesigned for the processor to be able to run them as multiple threads” [96] (A. Marowka, 2007)

Arguing about threads is intrinsically problematic [86, 109], because of the arbitrary interleaving during the execution of different threads that causes unexpected and sometimes counterintuitive behaviors. Other patterns and styles of parallelism exist, e.g. Software Transactional Memory [127] and message passing [57]. On the other hand, popular programming languages, e.g. Java and C#, support threads natively. Runtime environments can implement threads without too much overhead, while other solutions may require more resources, or limit the parallelism.

In this context, multithreading appears to be today the most common way in order to exploit multicore architectures.

1.1.3 Static Analysis

“Parallel programming, because of its unfamiliarity and intrinsic difficulty, is going to require better programming tools to systematically find defects, help debug programs, find performance bottlenecks, and aid in testing. Without these tools, concurrency will become an impediment that reduces developer and tester productivity and makes concurrent software more expensive and of lower quality. (...) Conventional methods of debugging, such as re-executing a program with a breakpoint set earlier in its execution, do not work well for concurrent programs, whose execution paths and behaviors may vary from one execution to the next. Systematic defect detection tools are extremely valuable in this world. These tools use static program analysis to systematically explore all possible executions of a program, and so can catch errors that are impossible to reproduce” [134] (H. Sutter and J. Larus, 2005)

The paragraph above fully explains the main motivations for applying static analysis to multithreaded applications. Since it is not possible to compute all the executions of a

Thread 1	Thread 2
<code>i = 1;</code> <code>j = 1;</code>	<code>if(j == 1 && i == 0)</code> <code>throw new Exception();</code>

Figure 1.2: Why multithreading is subtle

program, the key idea of static analysis is to approximate the semantics of a program in order to focus on a particular observational property of the behaviors of the program, and check if such a property always holds in all of its possible executions.

Abstract interpretation [25, 27] is a mathematical theory that allows to build up completely automatic static analyses that may apply directly on the source code. This is not the case of model checking, another static analysis approach, as it requires a model of the program (usually a Kripke structure) provided by the user as an input. It also differs from theorem proving techniques, as they often require an interaction with a specialized user (i.e. someone that thoroughly knows how the theorem prover works) to generate the proofs.

In this context, applying abstract interpretation to the analysis of multithreaded programs appears to be particularly appealing.

1.2 Context

“For many years parallel computers have been used by an exclusive scientific niche. Only rich universities and research institutions backed by government budgets or by multibillion-dollar corporations could afford state-of-the-art parallel machines. Multiprocessor machines are very expensive and demand highly specialized expertise in systems administration and programming skills.” [96] (A. Marowka, 2007)

Even if researchers have worked on parallel computing during almost the last 30 years, there are still important shortcomings in formal methods and static analysis with respect to multithreading.

Research on multithreading is still ongoing. On one hand, the specification of programming languages has been longtime flawed [113], and only recent works tried to fix this problem [95]. On the other hand, because of the inherently difficulty when dealing with multithreading (both when developing applications and static analysis), most existing static analyses are focused on specific properties.

1.2.1 Memory Models

What is legal during the execution of a program and what is not? Consider the example depicted by Figure 1.2. If at the beginning of the computation the values of variables `i` and

j are both equal to zero, it seems impossible that Thread 2 raises an exception. However, this is a possible and even acceptable behavior. Why?

For instance, a common optimization performed by compilers is to reorder independent statements. As the two actions performed by Thread 1 apply on disjoint sets of variables, they are independent, and so the compiler may reorder them. This optimization does not expose any new behavior at the single thread level, but it may cause Thread 2 to raise the exception.

Memory models define which behaviors are allowed during the execution of a multithreaded programs. In particular, they specify which values written in parallel may be read from shared memory. The interest in this topic has increased recently: for instance, the first specification of the Java Virtual Machine [89] was flawed [113], and only recent work [95] has revised it. This solution is quite complex, especially from a static analysis point of view. Other memory models have been proposed in the past: [83] formalized the sequentially consistency rule. It is quite simple, but too restrictive, as for instance it does not allow the behavior presented in Figure 1.2.

1.2.2 Static Analyses of Multithreaded Programs

The problem of static analysis of multithreaded programs has already been partially investigated in the literature.

A large amount of work has been dedicated to specific properties, and, in particular, races [106]. A general race happens when two threads access the same location of the shared memory in parallel, and at least one of the two accesses performs a write operation. A data race additionally requires that the two threads are not synchronized at that moment, e.g. they do not own a common monitor. The absence of general races guarantees the determinism of a multithreaded programs, but it is too restrictive: all communication between different threads must be strictly synchronized. In contrast, data races allow some nondeterministic behaviors. For instance, if two threads are synchronized on a monitor they can be executed in different orders in different executions. In addition, assuming freedom of data races may sometimes require additional synchronization, thereby restricting the parallelism of the program.

Since considering all the possible executions of a multithreaded program is particularly difficult, due to both arbitrary interleaving and compiler optimizations, not so many generic static analyses have been proposed in this context. In the last years a huge amount of work has been revolved around context bound analysis [114]. As verifying a program is undecidable [118], a multithreaded program is analyzed until a given context bound, that is until the number of context switchings has reached a given bound n . A context switch happens when the control passes from a thread to another. In this way, these analyses are not sound for all the possible executions, but only for the ones with at most n context switchings. Furthermore, they take arbitrary interleavings into account, and not, for instance, compiler optimizations. This approach is therefore closer to testing than to static analysis. For instance, if the analysis proves a property, there may exist an execution with a bigger number of context switchings for which the property is not validated.

1.2.3 Generic Static Analyzers

Some generic static analyzers based on abstract interpretation have already been proposed in the recent years. These static analyzers support the use of different domains (in order to obtain faster and more approximated or slower and more refined analyses) and they can analyze different properties. Some example of these analyzers are [92, 91, 130, 112].

The main advantage of this approach is that the most part of an analyzer can be reused to analyze different properties and tuned at different levels of efficiency and precision through the numerical domain. However, as far as we know, at the moment no existing generic analyzers supports multithreading in non-trivial way.

1.3 Contribution

The main contribution of our work is to formalize and develop a generic static analyzer of multithreaded programs. In order to achieve this goal, we define a generic static analysis of memory models by applying it to the happens-before model, and we introduce a new property yielding a formal model of non-determinism. Then we develop an ad-hoc semantics of Java bytecode language. We develop and implement a generic static analyzer that can be fitted with different memory models, numerical domains, and in order to check several properties. Finally, we extend an existing industrial generic analyzer in order to check buffer overruns, obtaining a scalable and precise analysis, then showing the practical impact of generic analyzers.

1.3.1 Static Analysis of the Happens-Before Memory Model

In order to formally argue about concurrency, we need to define a static analysis sound with respect to a memory model. But which model? Sequential consistency [83] has been proved to be too much restrictive with respect to the needs of modern programming languages. On the other hand, the Java memory model [95] strongly relies on some runtime informations. How to apply a static analysis to it is not at all clear, and it seems quite difficult to trace all the informations formalized by the definitions on executions at static level. A good compromise is the happens-before memory model [82]: it is an over-approximation of the Java one¹, and it is simple enough to base a static analysis on it. Tuning a static analysis at this level will allow us to obtain an analysis that is sound with respect to Java execution, even if more approximated than the Java model.

We define the happens-before memory model in a way that is amenable to a fixpoint computation, and then we abstract it to a computable semantics. This approach is completely modular with respect to the semantics of statements of the programming language, the domains used in order to trace information on numerical values, references, etc., and the property of interest.

¹Without considering *out-of-thin-air* values

1.3.2 Determinism of Multithreaded Programs

In the literature, the proposed properties (e.g. general and data races) and models (e.g. transactions) try to limit nondeterminism by defining a restricted model of execution or by avoiding some types of uncontrolled communications through shared memory. We think that studying directly the effects of this nondeterminism can allow us to achieve more interesting results. Intuitively, we want to trace information on the effects of unordered communications through shared memory, rather than working on the reasons that cause them. To this end, we define and approximate the deterministic property of multithreaded programs, and propose also a new property called weak determinism. We present some different ways of projection on states and traces, thereby building up a global hierarchy. We relate determinism to data races. This approach is strictly more flexible than existing ones, as it can be easily restricted only on a part of the shared memory, on a subset of the active threads, on some statements, etc..

1.3.3 A Generic Static Analyzer of Java Multithreaded Programs

If generic static analyzers have been successfully applied to the analysis of single-thread programs, their application to multithreaded programs should be at least as successful as for single-threading, since multithreading seems to be particularly interested in tools that help to debugging.

We define and abstract the semantics of Java bytecode on a low-level domain, developing an alias analysis particularly focused on multithreading issues, i.e. identification of threads, synchronization through monitors, and accesses on the shared memory. This semantics is parameterized by

- a numerical domain,
- a property of interest,
- a memory model.

This work has resulted in a static analyzer called **Checkmate**. In this context, we implemented also the happens-before memory model, and the deterministic property of multithreaded programs. The experimental results of **Checkmate** are quite promising.

1.3.4 An Industrial Case Study: Unsafe Code

Finally, we apply an industrial generic static analyzer (**Clousot**) to a specific property, showing the effectiveness of this type of static analyzers.

In particular, we check the absence of buffer overruns in unsafe code (i.e. code containing pointers) of MSIL (i.e. bytecode of .NET framework). In order to obtain a fast and precise analysis, we develop **Strp**, a new relational domain, and combine it with some existing numerical domains in order to increase its precision. The analysis is particularly fast and

precise: in average we are able to analyze about 20.000 methods in about 3 minutes, with a precision of 58% on average, i.e. we validate automatically the 58% of unsafe accesses.

1.4 Overview of the Thesis

The results of Chapters 3, 4, 5, 6, and 7 have been published in the proceedings of international conferences with program committee [45, 44, 43, 46, 47].

Chapter 2 introduces the notation and some basic concepts about abstract interpretation.

Chapter 3 defines the happens-before memory model in fixpoint form and abstracts it with a computable semantics. Starting from the (informal) definition of the happens-before memory model [82, 95], we define a fixpoint semantics that builds up all the finite executions following this memory model. An execution of a multithreaded program is represented as a function that relates threads to traces of states. Then we abstract it with a computable semantics before proving the correctness of our approach formally.

Chapter 4 defines and abstracts the deterministic property. First of all, the non-determinism of a multithreaded program is defined as difference between executions. If two executions expose different behaviors because of values read from and written to the shared memory, then that program is considered as non-deterministic. Then we abstract it in two steps: in the first step we collect, for each thread, the (abstract) value that it may write into a given location of the shared memory. At the second level we summarize all the values written in parallel, while tracking the threads that may have written it. At the first level of abstraction, we introduce the new concept of weak determinism. We propose other ways in order to relax the deterministic property, namely by projecting traces and states, and we define a global hierarchy. We formally study how the presence of data races may afflict the determinism of the program.

Chapter 5 defines a concrete semantics of `Java bytecode` language following its official specification [89]. Then we abstract it in order to precisely track the information required by the framework developed in the two previous chapters. The core is an alias analysis that precisely approximates references in order to identify threads, to check the accesses to the shared memory, and to detect when two threads own a common monitor thereby inferring which parts of the code cannot be executed in parallel.

Chapter 6 presents `Checkmate`, a generic static analyzer of multithreaded programs that implements the theoretical framework developed in Chapters 3, 4, and 5. We report and deeply study some experimental results. In particular, we analyze the precision of the analysis when applied to some common patterns of concurrent programming [85], certain case studies presented in [94], and its performances when applied to an incremental application, and to a set of well-known benchmarks [73, 138].

Chapter 7 presents the application of an industrial generic static analyzer, `Clousot`, to the analysis of unsafe code (i.e. code containing pointers) in the `.NET` framework. It turns out that this analysis is scalable and precise when applied to industrial code. In summary, we present an application of an existing, industrial, and generic static analyzer

to a property of practical interest, showing the strength of this approach in order to develop tools useful for developers.

Finally, Chapter 8 concludes and suggests the future work.

2

Preliminaries

This chapter introduces the mathematical background used throughout the thesis. In particular, we introduce some basic notation, and some well-known theoretical results on lattices, fixpoints, and abstract interpretation theory. In addition, we present a running example that will be used in the following chapters to explain the concepts in practice.

2.1 Notation

In this section, we introduce some basic mathematical concepts and notations on sets, lattices, functions, and traces.

2.1.1 Sets

We denote sets with sans serif strings beginning always with a capital letter, and elements by sans serif strings with only lower case characters. Let Set be a set, and e an element, we denote by $e \in \text{Set}$ the fact the e is a member of Set .

Let \mathbb{N} be the set of natural numbers (where $0 \in \mathbb{N}$). Let \mathbb{Z} be the set of integer numbers, and let $[a..b]$ be the set $\{i \in \mathbb{Z} : i \geq a \wedge i \leq b\}$.

Given two sets X and Y , their Cartesian product is denoted by $X \times Y$. This set contains all the possible pairs composed by an element in X as first component and by an element in Y as second component. Formally: $X \times Y = \{(x, y) : x \in X \wedge y \in Y\}$.

A relation r between X and Y is a subset of their Cartesian product (i.e. $r \subseteq X \times Y$), while a relation on X is a subset of the Cartesian product $X \times X$. We denote relations by italic strings beginning with a lower case letter.

The set containing all the elements of X that are defined at least on one $y \in Y$ in a relation r is called domain and is denoted by $\text{dom}(r)$. Formally $\text{dom}(r) = \{x : x \in X \wedge \exists y \in Y : (x, y) \in r\}$. In a similar way, the co-domain of a relation, $\text{codom}(r)$, is defined as $\text{codom}(r) = \{y \in Y \wedge \exists x \in X : (x, y) \in r\}$. We write xry to mean that $(x, y) \in r$.

2.1.2 Partial Orders and Lattices

A partial order \leq on a set X is a relation on X such that it is:

- reflexive: $\forall x \in X : x \leq x$
- antisymmetric: $\forall x_1, x_2 \in X : x_1 \leq x_2 \wedge x_2 \leq x_1 \Rightarrow x_1 = x_2$
- transitive: $\forall x_1, x_2, x_3 \in X : x_1 \leq x_2 \wedge x_2 \leq x_3 \Rightarrow x_1 \leq x_3$

A partially ordered set (poset) is a set equipped with a partial order; we denote it by $\langle X, \leq \rangle$. A poset $\langle X, \leq \rangle$ has a top element \top iff $\top \in X \wedge \forall x \in X : x \leq \top$. Dually, it has a bottom element \perp iff $\perp \in X \wedge \forall x \in X : \perp \leq x$.

Given $X_1 \subseteq X$, $x \in X$ is an upper bound of X_1 iff $\forall x' \in X_1 : x' \leq x$. It is the least upper bound (lub) if $\forall x_1 \in X$ such that x_1 is an upper bound of X_1 , then $x \leq x_1$; we denote it by $x = \sqcup X_1$.

Symmetrically, we define lower bounds and greatest lower bounds (glb).

A lattice is a poset such that any two elements belonging to X have a least upper bound and a greatest lower bound. A complete lattice is a poset such that every subset of X has a least upper bound and a greatest lower bound.

A chain C in a poset $\langle X, \leq \rangle$ is a subset of X such that $\forall c_1, c_2 \in C : c_1 \leq c_2 \vee c_2 \leq c_1$. An ascending chain is an ordered subset $\{x_i : i \in [a..b]$ where $a, b \in \mathbb{N} \cup \{-\infty, +\infty\}\}$ of X such that $\forall j, k \in [a..b] : j \leq k \Rightarrow x_j \leq x_k$. Dually, a descending chain is an ordered subset of elements such that each element is less or equal than the previous ones.

2.1.3 Functions

A function is a relation r such that if $(x, y_1) \in r$ and $(x, y_2) \in r$, then $y_1 = y_2$. In other words, a function is a relation that relates each element of the domain to at most one element of the co-domain. Thus, given an element $x \in \text{dom}(r)$, we denote the element in the co-domain by $r(x)$. In order to define functions, we use the λ notation. By $f = \lambda x. \text{Expr}$ we denote a function f that relates the evaluation of the expression Expr , which depends on x , to the element x of its domain. We denote sets of functions by capital Greek letters. Let f be a function, x an element in its domain and y an element in its co-domain. We use the notation $f[x \mapsto y]$ to represent a function that behaves as f except for the input x , for which it returns y . Similarly, $[x \mapsto y]$ denotes an elements (x, y) of a function.

By the notation $f : [X \rightarrow Y]$ we mean that the domain of the function f is included in X , and its co-domain is included in Y . Let $f : [X \rightarrow Y]$ and $g : [Y \rightarrow Z]$, then $g \circ f : [X \rightarrow Z]$ represents the composition of functions f and g , i.e. $\lambda x. g(f(x))$.

Given two posets $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$, a function $f : [X \rightarrow Y]$ is:

- monotonic iff $\forall x_1, x_2 \in X : x_1 \leq_X x_2 \Rightarrow f(x_1) \leq_Y f(x_2)$
- join preserving iff $\forall x_1, x_2 \in X : f(x_1 \sqcup_X x_2) = f(x_1) \sqcup_Y f(x_2)$ where \sqcup_X and \sqcup_Y are, respectively, the lub on $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$

- complete join preserving iff $\forall X_1 \subseteq X$ such that $\bigsqcup_X X_1$ exists, then $f(\bigsqcup_X X_1) = \bigsqcup_Y f(X_1)$
- continuous iff for all chains $C \subseteq X$ we have that $f(\bigsqcup_X C) = \bigsqcup_Y \{f(c) : c \in C\}$

A poset $\langle X, \leq \rangle$ satisfies the ascending chain condition (ACC) if every ascending chain $c_1 \leq c_2 \leq \dots$ of elements in X is eventually stationary, i.e. $\exists i \in \mathbb{N} : \forall j > i : c_j = c_i$. Dually, a poset satisfies the descending chain condition (DCC) if there is not any infinite decreasing chain.

2.1.4 Fixpoints

Let f be a function on a poset $\langle X, \leq \rangle$. The set of fixpoints of f is $P = \{x \in X : f(x) = x\}$. An element $x \in X$ is:

- a pre-fixpoint iff $x \leq f(x)$;
- a post-fixpoint iff $f(x) \leq x$;

A fixpoint $x \in P$ of f is a least fixpoint of f if $\forall p \in P : x \leq p$. If f has a least fixpoint, this is unique, and if it exists, we denote by $lfp_d^{\leq} f$ the least fixpoint of f greater than d with respect to the order \leq .

A fixpoint $x \in P$ of f is a greatest fixpoint of f if $\forall p \in P : p \leq x$. If f has a greatest fixpoint, this is unique, and if it exists, we denote by $gfp_d^{\leq} f$ the greatest fixpoint of f smaller than d with respect to the order \leq .

The existence of the least and greatest fixpoints on a monotonic map is guaranteed by the following theorem.

THEOREM 2.1.1 (TARSKI'S THEOREM [136]) *Let $\langle X, \leq, \perp, \top, \sqcup, \sqcap \rangle$ be a complete lattice. Let $f : [X \rightarrow X]$ be a monotonic function on this lattice. Then the set of fixpoints is a not-empty complete lattice, and:*

$$\begin{aligned} lfp_{\perp}^{\leq} f &= \sqcap \{x \in X : f(x) \leq x\} \\ gfp_{\top}^{\leq} f &= \sqcup \{x \in X : x \leq f(x)\} \end{aligned}$$

This result is not constructive.

THEOREM 2.1.2 (CONSTRUCTIVE VERSION OF TARSKI'S THEOREM [26]) *Let $\langle X, \leq, \perp, \top, \sqcup, \sqcap \rangle$ be a complete lattice. Let $f : [X \rightarrow X]$ be a monotonic function on this lattice. Define the following sequence:*

$$\begin{aligned} f^0 &= \perp \\ f^\delta &= f(f^{\delta-1}) \quad \text{for every successor ordinal } \delta \\ f^\delta &= \bigsqcup_{\alpha < \delta} f^\alpha \quad \text{for every limit ordinal } \delta \end{aligned}$$

Then the ascending chain $\{f^i : 0 \leq i \leq \delta\}$ (where δ is an ordinal) is ultimately stationary for some $\rho \in \mathbb{N}$ that is $f^\rho = lfp_{\perp}^{\leq} f$.

2.1.5 Traces

DEFINITION 2.1.3 (TRACE) *Given a set S , a trace τ is a partial function $[\mathbb{N} \rightarrow S]$ such that*

$$\forall i \in \mathbb{N} : i \notin \text{dom}(\tau) \Rightarrow \forall j > i : j \notin \text{dom}(\tau)$$

This definition implies that the domain of all non-empty traces is a segment of \mathbb{N} . Intuitively, a trace is an ordered sequence of elements such that it is defined on the first k elements. The empty trace (i.e. the trace τ such that $\text{dom}(\tau) = \emptyset$) is denoted by ϵ . Let be S a generic set of elements, we denote by S^\ddagger the set of all the finite traces composed of elements in S .

$\text{len} : [S^\ddagger \rightarrow \mathbb{N}]$ is the function that, given a trace, returns its length. Formally: $\text{len}(\tau) = i + 1 : i \in \text{dom}(\tau) \wedge i + 1 \notin \text{dom}(\tau)$. If $\tau = \epsilon$, then $\text{len}(\tau) = 0$.

Usually, we represent a trace as a sequence of states, i.e. $\sigma_0 \rightarrow \sigma_1 \rightarrow \dots$ represents the trace $\{(0, \sigma_0), (1, \sigma_1), \dots\}$.

S_{\rightarrow}^\ddagger represents the set of traces in S^\ddagger ending with a final state with respect to the transition \xrightarrow{T} , i.e. $S_{\rightarrow}^\ddagger = \{\sigma_0 \rightarrow \dots \rightarrow \sigma_i : \sigma_0 \rightarrow \dots \rightarrow \sigma_i \in S^\ddagger, \nexists \sigma_j \in S : \sigma_i \xrightarrow{T} \sigma_j\}$.

The concatenation of two traces τ_1 and τ_2 is written as $\tau_1 \rightarrow_\tau \tau_2$ and represents the trace $\tau_1 \cup \{i \mapsto \sigma : \exists j \in \text{dom}(\tau_2) : i = j + \text{len}(\tau_1), \sigma = \tau_2(j)\}$.

Given a set of initial elements S_0 and a transition relation $\xrightarrow{T} \subseteq \Sigma \times \Sigma$, the partial trace semantics builds up all the traces that can be obtained by starting from traces containing only a single element from S_0 and then iteratively applying the transition relation until a fixpoint is reached.

DEFINITION 2.1.4 (PARTIAL TRACE SEMANTICS [27]) *Let Σ be a set of states, $S_0 \subseteq \Sigma$ a set of initial elements, and $\xrightarrow{T} \subseteq \Sigma \times \Sigma$ a transition relation. Let $f : [\emptyset(\Sigma) \rightarrow [S^\ddagger \rightarrow \Sigma^\ddagger]]$ be the function defined as:*

$$F(S_0) = \lambda T. \{0 \mapsto \sigma_0 : \sigma_0 \in S_0\} \cup \cup \{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_{i-1} \xrightarrow{T} \sigma_i\}$$

The partial trace semantics is defined as

$$\text{PT}[\mathbf{S}_0] = \text{lfp}_\emptyset^{\subseteq} F(\mathbf{S}_0)$$

2.2 Abstract Interpretation

Abstract interpretation is a mathematical theory of approximation of semantics developed by P. and R. Cousot about 30 years ago [23, 25, 27]. Applied to static analysis of programs, abstract interpretation allows to approximate an uncomputable concrete semantics with a computable abstract one. Approximation is required, then the result is correct but incomplete. The inferred properties are satisfied by all the possible results of the concrete

semantics, but if a property is not inferred in the abstract it may still be satisfied by the concrete semantics.

The main idea of abstract interpretation is to define the semantics of a program as the fixpoint of a monotonic function.

2.2.1 Galois Connections

DEFINITION 2.2.1 (GALOIS CONNECTION) Let $\langle C, \leq \rangle$ and $\langle \bar{C}, \sqsubseteq \rangle$ be two posets. Two functions $\alpha : [C \rightarrow \bar{C}]$ and $\gamma : [\bar{C} \rightarrow C]$ form a Galois connection if and only if

$$\forall x \in C : \forall \bar{x} \in \bar{C} : \alpha(x) \sqsubseteq \bar{x} \Rightarrow x \leq \gamma(\bar{x})$$

We denote this fact by writing

$$\langle C, \leq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{C}, \sqsubseteq \rangle$$

Alternatively, $\langle C, \leq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{C}, \sqsubseteq \rangle$ holds iff

- α and γ are monotone;
- $\alpha \circ \gamma$ is reductive (i.e. $\forall \bar{x} \in \bar{C} : \alpha \circ \gamma(\bar{x}) \sqsubseteq \bar{x}$);
- $\gamma \circ \alpha$ is extensive (i.e. $\forall x \in C : x \leq \gamma \circ \alpha(x)$).

Usually, we call the left part of the Galois as the concrete poset, and the right one as the abstract one. Similarly, γ is called the concretization and α is the abstraction. The concrete sets and elements are denoted as defined in Section 2.1.1, while the abstract ones are over-lined. For instance, if S is a concrete set, the respective abstract set is denoted by \bar{S} . If fun is the concrete one, its abstract counterpart is denoted by \bar{fun} .

A Galois connection can be induced by an abstraction function that is a complete \sqcap -morphism, or dually by a function that is a complete \sqcup -morphism (where \sqcap and \sqcup are respectively the lower bound operator on the abstract lattice and the upper bound operator on the concrete lattice), as proved by Proposition 7 of [28].

THEOREM 2.2.2 (GALOIS CONNECTION INDUCED BY LUB PRESERVING MAPS) Let $\alpha : [A \rightarrow \bar{A}]$ be a complete join preserving maps between posets $\langle A, \leq \rangle$ and $\langle \bar{A}, \sqsubseteq \rangle$. Define:

$$\gamma = \lambda \bar{y}. \bigvee \{z : \alpha(z) \sqsubseteq \bar{y}\}$$

If γ is well-defined then

$$\langle A, \leq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{A}, \sqsubseteq \rangle$$

THEOREM 2.2.3 (GALOIS CONNECTION INDUCED BY GLB PRESERVING MAPS) Let $\gamma : [\bar{A} \rightarrow A]$ be a complete join preserving maps between posets $\langle A, \leq \rangle$ and $\langle \bar{A}, \sqsubseteq \rangle$. Define:

$$\alpha = \lambda y. \bigwedge \{\bar{z} : y \leq \gamma(\bar{z})\}$$

If α is well-defined then

$$\langle A, \leq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{A}, \sqsubseteq \rangle$$

An interesting property of Galois connections is that they are compositional, i.e. the composition of two Galois connections is still a Galois connection.

THEOREM 2.2.4 (COMPOSITION OF GALOIS CONNECTIONS) *Suppose that $\langle \mathbb{A}, \leq_1 \rangle \xleftrightarrow[\alpha_1]{\gamma_1} \langle \overline{\mathbb{A}}, \leq_2 \rangle$ and $\langle \overline{\mathbb{A}}, \leq_2 \rangle \xleftrightarrow[\alpha_2]{\gamma_2} \langle \overline{\mathbb{A}'}, \leq_3 \rangle$. Then*

$$\langle \mathbb{A}, \leq_1 \rangle \xleftrightarrow[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} \langle \overline{\mathbb{A}'}, \leq_3 \rangle$$

2.2.2 Fixpoint Approximation

Usually in abstract interpretation the concrete and abstract semantics are defined as the fixpoint computation of monotonic functions. The ultimate goal is to prove the correctness of the abstract semantics with respect to the concrete one, i.e. that the concretization of the results of the abstract semantics over-approximates the output of the concrete semantics.

Formally, let $\mathbb{A} : [\mathbb{C} \rightarrow \mathbb{C}]$ and $\overline{\mathbb{A}} : [\overline{\mathbb{C}} \rightarrow \overline{\mathbb{C}}]$ be the concrete and the abstract semantics respectively, where $\langle \mathbb{C}, \leq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \overline{\mathbb{C}}, \sqsubseteq \rangle$. The abstract semantics is sound iff for all the prefixpoints $\overline{p} \in \overline{P} \subseteq \overline{\mathbb{C}}$ of $\overline{\mathbb{A}}$, we have that $\gamma \circ \overline{\mathbb{A}}[\overline{p}] \geq \mathbb{A}[\gamma(\overline{p})]$.

When applied to the static analysis of programs, the transfer function depends on a program P .

There are many different ways in order to prove that an abstract semantics is sound, relying on some different properties of transfer functions, concrete and abstract lattices, concretization and abstraction functions. We refer the interested reader to [29] for a complete overview on this topic. In this thesis, we will rely on the following theorem.

THEOREM 2.2.5 (KLEENE-LIKE, JOIN-MORPHISM-BASED FIXPOINT APPROXIMATION [25]) *Let $\langle \mathbb{L}, \sqsubseteq, \sqcup \rangle$ and $\langle \overline{\mathbb{L}}, \overline{\sqsubseteq}, \overline{\sqcup} \rangle$ be complete lattices. Let $F : [\mathbb{L} \rightarrow \mathbb{L}]$ and $\overline{F} : [\overline{\mathbb{L}} \rightarrow \overline{\mathbb{L}}]$ be two monotone functions with respect to \sqsubseteq and $\overline{\sqsubseteq}$ respectively. Let $\alpha : [\mathbb{L} \rightarrow \overline{\mathbb{L}}]$ be a join-morphism such that $\alpha \circ F \overline{\sqsubseteq} \overline{F} \circ \alpha$, where $\overline{\sqsubseteq}$ is the lifting of the ordering operator $\overline{\sqsubseteq}$ to functions. Let $\mathbf{a} \in \mathbb{L}$ be a prefixpoint of F . Then $\alpha(\text{lfp}_{\mathbf{a}}^{\sqsubseteq} F) \overline{\sqsubseteq} \text{lfp}_{\alpha(\mathbf{a})}^{\overline{\sqsubseteq}} \overline{F}$.*

2.2.3 Widening

If the abstract domain respects the ACC, the abstract semantics can be computed in a finite time. Otherwise we need a widening operator in order to make the analysis convergent.

DEFINITION 2.2.6 (WIDENING) *Given an ascending chain $\mathbf{d}_0 \leq \mathbf{d}_1 \leq \mathbf{d}_2 \leq \dots$ in a poset $\langle \mathbb{C}, \leq \rangle$, a widening operator $\nabla : [\mathbb{C} \rightarrow \mathbb{C}]$ is an upper bound operator such that the chain*

$$\mathbf{w}_0 = \mathbf{d}_0, \mathbf{w}_1 = \mathbf{w}_0 \nabla \mathbf{d}_1, \dots, \mathbf{w}_i = \mathbf{w}_{i-1} \nabla \mathbf{d}_i$$

is ultimately stationary, i.e. $\exists j \in \mathbb{N} : \forall k \in \mathbb{N} : k > j \Rightarrow \mathbf{w}_j = \mathbf{w}_k$

The use of widening on abstract domains not satisfying the ACC makes the analysis convergent still obtaining sound (even if more approximated) results.

THEOREM 2.2.7 (WIDENING SOUNDNESS) *Let $\langle A, \leq \rangle$ and $\langle \bar{A}, \sqsubseteq \rangle$ be two complete lattices, $\gamma : [\bar{A} \rightarrow A]$ and $\alpha : [A \rightarrow \bar{A}]$ be two functions such that $\langle A, \leq \rangle \xrightarrow[\alpha]{\gamma} \langle \bar{A}, \sqsubseteq \rangle$. Let $F : [A \rightarrow A]$ and $\bar{F} : [\bar{A} \rightarrow \bar{A}]$ be two monotonic function such that $\text{lfp}(F) \leq \gamma \circ \text{lfp}(\bar{F})$. Then the sequence defined by*

$$\begin{aligned} \bar{x}_0 &= \perp \\ \bar{x}_i &= \begin{cases} \bar{F}(\bar{x}_{i-1}) & \text{if } \bar{F}(\bar{x}_{i-1}) \sqsubseteq \bar{x}_{i-1} \\ \bar{F}(\bar{x}_{i-1}) \nabla \bar{x}_{i-1} & \text{otherwise} \end{cases} \end{aligned}$$

is ultimately stationary and its limit \bar{x}_k is a post-fixpoint of \bar{F} . Hence, it soundly approximates the concrete semantics, i.e. $\text{lfp}(F) \leq \gamma \circ \text{lfp}(\bar{F}) \leq \gamma(\bar{x}_k)$.

2.3 Running Example

We will often recur to some examples in order to explain the ideas and the formalizations presented throughout the next chapters. Chapter 7 would be an exception, as we will use some code taken from .NET shipped libraries. Figure 2.1 contains the code of a class `Account`. It is aimed at simulating some of the most common operations performed on a bank account, i.e. deposits and withdraws. Class `Cheque` is presented in Figure 2.2 and represents a signed cheque that can be used to withdraw an amount. We can also check if a cheque is valid, i.e. if it contains money that can be withdrawn, or if it is empty.

We will use these classes in order to illustrate how actions are performed in parallel and thereby explain the formally defined concepts in practice.

```
public class Account {
    private int amount=0;
    private final double interestRate=0.015;
    public final Signature signature;

    public Account(int am, Signature sign) {
        this.amount=am;
        this.signature=sign;
    }
    public void withdraw(int money) {
        synchronized(this) {
            this.amount-=money;
        }
    }
    public void withdrawNoDebts(int money) {
        synchronized(this) {
            int temp=this.amount-money;
            if (temp>0)
                this.amount=temp;
        }
    }
    public void deposit(int money) {
        synchronized(this) {
            this.amount+=money;
        }
    }
    public void calculateInterests () {
        synchronized(this) {
            this.amount*=1+this.interestRate;
        }
    }
    public int getAmount() {
        return this.amount;
    }
    public void printAmount() {
        ATM.screen.print(this.amount);
    }
}
```

Figure 2.1: The class Account

```
public class Cheque {
    private Signature sign;
    private int amount;
    private Account account;

    public Cheque(Signature s, int am, Account acc) {
        this.sign=s;
        this.amount=am;
        this.account=acc;
    }
    public boolean withdraw() {
        if (account!=null && this.sign.equals(account.signature)) {
            this.account.withdraw(amount);
            this.amount=0;
            account=null;
            return true;
        }
        else return false;
    }
    public boolean isValid() {
        if (this.amount>0 && this.account==null)
            return false;
        else return true;
    }
}
```

Figure 2.2: The class Cheque

3

Static Analysis of the Happens-Before Memory Model

In this chapter, we will define the happens-before memory model in a fixpoint form and we approximate it with a computable semantics.

Memory models define which executions of multithreaded programs are legal. The happens-before one was formalized about 30 years ago [82], and it is an over-approximation of the one adopted by Java [95]. Our approach is completely independent of both the programming language and the analyzed property. It appears to be a promising framework to define, compare and statically analyze other memory models.

This chapter is based on the published paper [43].

3.1 Memory Models

The semantics of a programming language supporting multithreading must be defined well enough that developers can fully and easily understand which behaviors are allowed during an execution. A common approach in the literature has been to consider all the programs containing data races as incorrect [120], and to let unspecified the semantics in this case. Many static analyses have been aimed at proving the absence of data races [106, 121]. Leaving the semantics of these programs completely unspecified is unsatisfying for modern programming languages, particularly those that are focused on security issues.

On the other hand, guaranteeing the sequential consistency [83] for programs containing data races is not possible, as this would forbid the most part of compilers optimizations. In this context, weak memory models have been introduced [35, 55]. These models offer greater performances [2].

The interest in this topic has increased during the last few years: for instance, the first specification of the Java Virtual Machine [89], corrected in [95], was flawed [113]. Nowadays, the specification of the memory model appears to be the “lingua franca” to define which behaviors of multithreaded programs are allowed. In this context, two different approaches are considered:

- to restrict the non-deterministic behaviors in order to provide a simple reference to

the developers,

- to allow as many compiler optimizations as possible, introducing non-deterministic behaviors.

On this topic, the debate is still in progress [19], and different ideas and solutions have been proposed [126, 17]. In particular, the Java memory model seems to allow undesired behaviors and to prohibit desirable executions [5].

Most state-of-the-art static analyses do not support multithreading, or they deal only with the possible interleavings of instructions. This is why they are not sound with respect to the memory model, as it usually allows more behaviors than the ones exposed by sequentially consistent executions.

Contribution: Given the current state of the art, a static analysis able to approximate all the possible runtime behaviors of a multithreaded program with respect to a memory model seems to be particularly appealing, as it would help developers to reason about the parallel execution of multiple threads [134]. Moreover, since threads communicate implicitly through shared memory, particularly subtle and unwanted interactions may arise, and a static analysis may detect and provide useful information about them. Some examples of these interactions, like the one depicted by Figure 1.2, are presented in [95].

In the definition of our concrete and abstract semantics we will focus on the consistency condition [133]. In particular, it specifies which is the output of the shared memory when a read is performed on it. This output is represented as a set of values written in parallel with respect to the read action.

We first define the concrete trace semantics in a fixpoint form, aimed at formalizing the happens-before memory model. Then we abstract it and we prove the soundness of our analysis.

The semantics of statements does not take multithreaded executions into account, i.e. parallel writes on the shared memory and synchronization actions. In this way, we may reuse the semantics on single-thread programs and apply them to the analysis of multithreaded programs. About synchronizations, we focus only on mutual exclusion and launching of threads. Other synchronization patterns, e.g. `Thread.end()` in Java, may be easily added to our framework.

Our analysis is generic on the programming language, as the happens-before memory model is. The only restrictions we applied are that the small step semantics of statements is atomic, and that some functions, that returns a part of information on a given state, are provided. Thus, our framework may be used to formalize, compare, and statically analyze other memory models.

3.1.1 An Example

Figure 3.1 depicts an example that will be used to illustrate the formal concepts introduced in this chapter. It is composed of two threads that operate on a shared cheque. The code

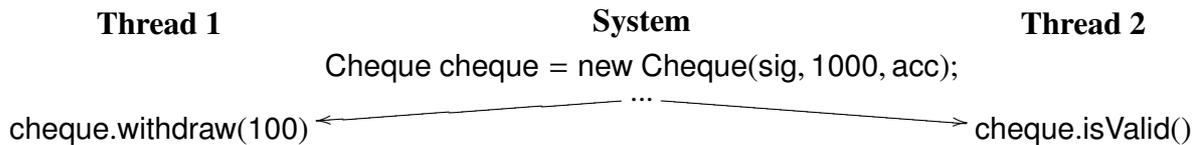


Figure 3.1: Checking if a cheque is valid

of the called methods is presented in Section 2.3. We are interested in checking if the method `isValid()` may return `false` when executed by Thread 2.

3.2 The Happens-Before Memory Model

In the recent literature, memory models have been aimed at formalizing the behaviors that are allowed during the execution of a multithreaded program.

The Java Memory Model was presented in [113]. Its formalization involves many different run-time components. In the same paper the happens-before memory model is formalized as an over-approximation of the Java memory model without considering *out-of-thin-air* values. It allows a larger number of runtime behaviors. Its formalization is simpler, and it allows us to reason in terms of static analysis. In addition, an abstract analysis on this model will allow us to obtain results sound but more approximate with respect to the Java memory model.

The main components of this model are (we denote some rules with a specific name which will be used during the formalization):

- the program order, that, for each thread, totally orders the actions performed during its execution;
- a synchronizes-with relation that relates two synchronized actions. For instance, the acquisition of a monitor synchronizes-with all the previous releases of the same monitor. Moreover the first action of the execution of a thread is synchronized-with the action that launched it (rule IN);
- the happens-before order initially introduced in [82]. An action a_1 happens-before another action a_2 (rule HB) if
 - a_1 appears before a_2 in the program order;
 - a_2 synchronizes-with a_1 ;
 - if you can reach a_2 by following happens-before edges starting from a_1 , i.e. the happens-before order is transitive.

Through the happens-before order, a consistency rule is defined. In particular, it states that a read r of a variable v is allowed to see a write w on v if:

Thread 1	Thread 2
lock(o)	lock(o)
var = v1	temp = var
var = v2	unlock(o)
unlock(o)	

Figure 3.2: An example

- r is not in happens-before relation with w , i.e. a read can not see a write that has to be executed after it,
- there is not a write w' on v that happens-before r and w happens-before it, i.e. there is no write on the same variable that is executed between the observed write and the read, thereby overwriting it (rule OW).

The happens-before memory model says nothing about what a variable is and its granularity (an object, a field, an array, a primitive value, ...) is.

3.2.1 Reasoning Statically

One point is not clear in these definitions: on one hand the definition of the happens-before consistency appears to be a static rule. On the other hand, the program order talks about a total order covering all the actions of an execution: this concept is clearly dynamic. Our approach is parameterized by the abstract intra-thread transition relation. So we suppose that it approximates this program order. In this way if a state is before another one in the trace produced through this relation, it means that it will always be executed before it.

With respect to the synchronizes-with relation, threads generally synchronize on some elements (for instance in Java they synchronize on monitors defined on objects), and the mutual exclusion during the execution of some parts of the code is guaranteed. In this way, they acquire a synchronizable element, perform some actions, and finally release it. In a static context, we do not know which thread acquires the synchronizable element first. For instance, consider the multithreaded program of Figure 3.2.

Which values may thread two read? Before the read action, we acquire the same monitor owned when executing both the writes of thread one. It may read the initial value stored in `var`, or `v2`, but not `v1`, as the acquisition of the monitor of thread two synchronizes-with the release of thread one or, vice versa, its release synchronizes-with the acquisition of thread one.

This consideration leads us to the following conclusion. A read r synchronized on a set S of synchronizable elements can see a value written by an action w performed by another thread if it is not:

- overwritten by a following action w'
- such that all the actions between w and w' are synchronized at least on a common element in S , e.g. a monitor.

This is a consequence of the mutual exclusion principle, i.e. that some instructions of two parallel threads cannot be executed in parallel if they are synchronized.

The formalization of the concrete semantics takes into account these considerations.

3.2.2 The Example

Let us apply these concepts to the example depicted in Section 3.1.1, and in particular to the question if method `isValid()` may return `false`. To answer this question, we evaluate which values may be read by the condition of the `if` statement of `isValid()` method when executed by Thread 2.

First of all, since there is no synchronization, the `synchronize-with` order is empty, and all the actions of Thread 1 do not happen-before the evaluation of the condition. Then this instruction is allowed to see the initial value of field `this.amount` which is equal to 1.000 and the value written by method `withdraw` to the field `account` which assigns `null` to it. Therefore, it is consistent to evaluate this condition to `true`. So `isValid()` method may return `false`.

For instance, suppose that the two statements that assign values to fields `amount` and `account` of method `withdraw` are switched by the compiler. Then a single-core processor may execute `account = null`, before the control switches to Thread 2, where the condition of the `if` statement evaluates to `true`.

3.3 Multithreaded Concrete Semantics

In this section we present the multithreaded concrete semantics. This semantics is aimed at formalizing the happens-before memory model in a fixpoint form. It is parameterized by the concrete operational semantics that defines the behaviors of intra-thread computational steps and on some functions that return a part of a given state. In this way we separate the semantics of the language from its memory model.

Since the happens-before memory model only refers to finite executions, we consider finite traces. Our multithreaded concrete semantics produces all complete executions, i.e. executions in which all the threads end in a blocking state.

3.3.1 Assumptions

In order to define the happens-before memory model on the concrete semantics, we need to introduce some sets and functions that extract information from states.

For the sets, we denote by `TId` the set of the thread identifiers, by `Sh` the set of shared memories, by `Loc` the locations, by `Val` the values (e.g. numerical values and references), by `Sync` all the shared elements on which a thread can synchronize, and by `St` the states containing both memory and control state of a single thread.

We suppose that a transition function $\overset{\circ}{\rightarrow} : [St \times St \rightarrow \{\text{true}, \text{false}\}]$ is provided, and that defines the single step behavior of the program. We require that these steps are atomic at

thread level, i.e. it is not possible for another thread to see an intermediate state during a single intra-thread transition.

We also require that the following functions are provided:

- *shared* : $[\text{St} \rightarrow \text{Sh}]$. Given a state, *shared* returns the shared memory contained in it;
- *action* : $[\text{St} \rightarrow \perp_a \cup (\{\text{r}, \text{w}\} \times \text{Loc} \times (\text{Val} \cup \perp_v))]$. Given a state, *action* returns the operation it is going to perform (reading from or writing on the shared memory), the shared location on which it operates and the written value or \perp_v if it is a read action, or \perp_a if it has performed another type of operation;
- *synchronized* : $[\text{St} \rightarrow \wp(\text{Sync})]$. Given a state, *synchronized* returns all the elements on which it is synchronized, e.g. the set of all the monitors previously locked and not yet released;
- *assign* : $[\text{Sh} \times \text{Loc} \times \text{Val} \rightarrow \text{Sh}]$. Given a state of the shared memory, a location and a value to be written, *assign* returns the shared memory obtained by assigning the given value to the given location in the given shared memory;
- *setshared* : $\text{St} \times \text{Sh} \rightarrow \text{St}$. Given a state and a shared memory, *setshared* returns a state equal to the given one but in which the shared memory is replaced by the given one.

3.3.2 Thread-Partitioning Concrete Domain

The concrete domain is aimed at collecting information about the parallel execution of different threads. To this end, we partition the trace, which represents one interleaving of the global execution of different threads, relating each active thread to the trace containing only its execution.

In current programming languages, threads are created and launched (i.e. their parallel execution is started) by other threads during their execution. Then for each thread we track also the thread that has launched it, and the index in its trace of the state that is produced after the launch. For the main thread, that is launched by the system, we use a special value \perp_Ω . We collect the number of the state in order to restrict the execution trace to the states after the launch of the thread, and so to respect the rule IN. Thus, our concrete domain is composed of two functions where the second one is aimed at maintaining some information on the launches of threads:

$$\begin{aligned} \Psi &: [\text{TId} \rightarrow \text{St}^{\dagger}] \\ \Omega &: [\text{TId} \rightarrow ((\text{TId} \times \mathbb{N}) \cup \perp_\Omega)] \end{aligned}$$

Our thread-partitioning domain is aimed at formalizing the executions of multithreaded programs. In particular, it is generic with respect to the hardware architecture on which the programs are executed. In this way, it abstracts away some information with respect to real executions.

Single-core architectures

When executing multithreaded programs on a single-core architecture, we have a total order on the actions executed by all the threads. Let MSt be the set of multithreaded states. Then a trace in MSt^{\ddagger} represents an execution on a single-code architecture. Let $projectMSt : [MSt \times TId \rightarrow St]$ be the function that given a multithreaded state and a thread identifier returns the local state of this thread. Let $whichThread : [MSt \rightarrow TId]$ be the function that, given a multithreaded state, returns the thread that has executed the last computational step. The abstraction function that, given a single-core architecture, returns a state of our thread-partitioning concrete domain, is defined as follows.

$$\begin{aligned}
& \alpha_{MSt} : [MSt^{\ddagger} \rightarrow \Psi] \\
& \alpha_{MSt}(\sigma_0 \rightarrow \dots \rightarrow \sigma_i) = \{[t \mapsto \{\sigma'_0 \rightarrow \dots \rightarrow \sigma'_i\}] : \\
& \quad (i) \sigma'_0 = projectMSt(\sigma_0, t), \\
& \quad (ii) \forall k \in [1..j-1], \exists w \in [1..i] : \sigma'_k = projectMSt(\sigma_w, t), whichThread(\sigma_w) = t, \\
& \quad (iii) \exists z \in [1..w-1] : \sigma'_{k-1} = projectMSt(\sigma_z, t) : whichThread(\sigma_z) = t, \\
& \quad \quad \quad \forall i' \in [z+1..w-1] : whichThread(\sigma_{i'}) \neq t \\
& \quad (iv) \exists h \in [w+1..i] : \sigma'_{k+1} = projectMSt(\sigma_h, t) : whichThread(\sigma_h) = t, \\
& \quad \quad \quad \forall i'' \in [w+1..h-1] : whichThread(\sigma_{i''}) \neq t \\
& \quad \quad \quad \}
\end{aligned}$$

Note that two different single-core executions may be abstracted into the same element of the thread-partitioning concrete domain. For instance, consider the two following executions (where \xrightarrow{T} means that the transition is executed by thread T):

$$\begin{array}{ccc}
& T1 & T2 \\
\sigma_0 & \xrightarrow{T1} \sigma_1 & \xrightarrow{T2} \sigma_2 \\
& T2 & T1 \\
\sigma_0 & \xrightarrow{T2} \sigma'_1 & \xrightarrow{T1} \sigma'_2
\end{array}$$

The two executions start with the same state σ_0 , and the two threads execute the same statements. If the threads do not communicate, their intra-thread state is not influenced by the order of execution and we will obtain that

$$\begin{aligned}
projectMSt(\sigma_2, T2) &= projectMSt(\sigma'_1, T2) \\
projectMSt(\sigma_1, T1) &= projectMSt(\sigma'_2, T1)
\end{aligned}$$

and so they will be abstracted into the same element of our thread-partitioning domain.

In the case of dual core architectures we have no longer a total order on the executions of all the threads, as the two cores may execute different threads in parallel. Intuitively, we obtain two traces of executions (one for each core) and a partial (temporal) order between statements executed by different cores. Without going into formal details deeply, we abstract these executions in a similar way as for a single-core architecture, collecting a trace of execution for each active thread. As above, two different executions may be abstracted into the same element in our thread-partitioning concrete domain.

Discussion

We can deal with architectures with more than two cores in a similar way. As we pointed out, our thread-partitioning concrete domain is already an abstraction of real executions. Since we want to build up a static analysis that is generic with respect to hardware architectures, this abstraction is mandatory. In particular, we abstract away the inter-thread order of execution. On the other hand, this is exactly what developers do when writing multithreaded applications: they think about threads separately, and they do not take the inter-thread order of execution into account. Then they add some synchronization actions in order to avoid that some part of different threads are executed in parallel. The definition of our fixpoint semantics will consider these actions, in order to discard the computational steps that cannot be executed in parallel and whose effects are not visible by another thread in a given point of its execution. Note that our approach is focused only on synchronizations on monitors, and so we do not consider other synchronization patterns like `wait()` and `notify()` in Java. With respect to these primitives our analysis is sound but imprecise, as it abstracts away the fact that the two threads are synchronized using them, and so that some parts of the code may be not executed in parallel.

3.3.3 Single Step Function

We define a *step* function that performs a single intra-thread step, that is consistent with the happens-before memory model, and which returns the set of the possible states that results from executing the step.

DEFINITION 3.3.1 (*step FUNCTION*) *Given the identifier of the active thread, a multithreaded state containing the traces of the executions of all the threads, and an element of Ω , the step function returns the set of all the possible resulting states.*

If the thread does not read from the shared memory, it computes the step given by the intra-thread semantics (point (1)). Otherwise it may either

- *perform the step given by the intra-thread semantics (point (1)),*
- *or select one visible value following the happens-before consistency rule and perform the step assigning this value in the shared memory (point (2)).*

Formally,

$$\text{step} : [\text{TId} \times \Psi \times \Omega \times \text{St} \rightarrow \wp(\text{St})]$$

$$\text{step}(t, f, s, \sigma_i) = \{\sigma\} \text{ such that}$$

$$(1) \sigma_i \xrightarrow{\circ} \sigma$$

∨

$$(2) \text{ if } \text{action}(\sigma_i) = (r, l, \perp_v) :$$

$$\exists v \in \text{vis}(t, l, \text{synchronized}(\sigma_i), f, s(t)) :$$

$$\sigma' = \text{setshared}(\sigma_i, \text{assign}(\text{shared}(\sigma_i), l, v)), \sigma' \xrightarrow{\circ} \sigma$$

DEFINITION 3.3.2 (*vis FUNCTION*) *The vis function returns the values written in parallel to a given location following the happens-before memory model. This set is built up by the values produced by the thread that launched the one that is reading, restricting it only on the part of the trace executed after the launch (point (1), rule IN), and the values produced by other threads (point (2)).*

$$\begin{aligned} \text{vis} &: [\text{Tid} \times \text{Loc} \times \wp(\text{Sync}) \times \Psi \times ((\text{Tid} \times \mathbb{N}) \cup \perp_{\Omega}) \rightarrow \wp(\text{Val})] \\ \text{vis}(t, l, \mathbf{S}, f, (t', i')) &= \\ &\quad \text{project}(l, \text{suffix}(f(t'), i'), \mathbf{S}) \cup \\ &\quad \{v : v \in \text{project}(l, f(t''), \mathbf{S}) : t'' \in \text{dom}(f) \setminus \{t, t'\}\} \end{aligned} \tag{1} \tag{2}$$

DEFINITION 3.3.3 (*suffix FUNCTION*) *The suffix function, given a trace and an index, cuts the trace at the i -th element and returns the suffix of the trace.*

$$\begin{aligned} \text{suffix} &: [\text{St}^{\ddagger} \times \mathbb{N} \rightarrow \text{St}^{\ddagger}] \\ \text{suffix}(\sigma_0 \rightarrow \dots \rightarrow \sigma_j, i) &= \begin{cases} \sigma_i \rightarrow \dots \rightarrow \sigma_j & \text{if } i \geq 0 \wedge i \leq j \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

DEFINITION 3.3.4 (*project FUNCTION*) *The project function, given a location, a trace, a set of synchronizable elements locked previously and not yet released, and the thread that is currently analyzed, returns the set of visible values in the given trace following the happens-before consistency.*

$$\begin{aligned} \text{project} &: [\text{Loc} \times \text{St}^{\ddagger} \times \wp(\text{Sync}) \rightarrow \wp(\text{Val})] \\ \text{project}(l, \sigma_0 \rightarrow \dots \rightarrow \sigma_i, \mathbf{S}) &= \{v : \exists j \in [0..i] : \text{action}(\sigma_j) = (w, l, v) \wedge \\ &\quad \text{notsynchronized}(\sigma_j \rightarrow \dots \rightarrow \sigma_i, \mathbf{S})\} \end{aligned}$$

The first part of the condition ($\text{action}(\sigma_j) = (w, l, v)$) excludes the transitions that do not write to shared memory. The second part the ones whose values are overwritten by a successive action following the happens-before order (rule OW).

DEFINITION 3.3.5 (*notsynchronized FUNCTION*) *Given a trace and a set of synchronizable elements, the notsynchronized function returns true if and only if*

- *the first state of the trace does not own one of the given synchronizable elements (case (1)),*
- *or if there is not a write action that writes on the same location of the first action of the given trace and that is synchronized-with it (case (2)).*

$$\begin{aligned} \text{notsynchronized} &: [\text{St}^{\ddagger} \times \wp(\text{Sync}) \rightarrow \{\text{true}, \text{false}\}] \\ \text{notsynchronized}(\sigma_0 \rightarrow \dots \rightarrow \sigma_i, \mathbf{S}) &= \text{true if and only if} \\ &\quad (1) \mathbf{S} \cap \text{synchronized}(\sigma_0) = \emptyset \vee \\ &\quad (2) \nexists \sigma_j \in \text{cut}(\sigma_0 \rightarrow \dots \rightarrow \sigma_i, \mathbf{S}) : \text{action}(\sigma_j) = (w, l, v), \\ &\quad \quad \text{action}(\sigma_0) = (w, l_0, v_0), l = l_0 \end{aligned}$$

DEFINITION 3.3.6 (cut FUNCTION) *Given a trace and a set of synchronizable elements, the cut function project the given trace on the n initial states that own at least one of the given synchronizable elements.*

$$\begin{aligned} \text{cut} &: [\text{St}^\tau \times \wp(\text{Sync}) \rightarrow \text{St}^\tau] \\ \text{cut}(\sigma_0 \rightarrow \dots \rightarrow \sigma_i, \mathbf{S}) &= \begin{cases} \epsilon & \text{if } \text{synchronized}(\sigma_0) \cap \mathbf{S} = \emptyset \\ \sigma_0 \rightarrow_\tau \text{cut}(\sigma_1 \rightarrow \dots \rightarrow \sigma_i, \mathbf{S}) & \text{otherwise} \end{cases} \end{aligned}$$

3.3.4 Fixpoint Semantics

By using the *step* function we define the fixpoint concrete semantics in order to compute all the possible finite traces of a given multithreaded program.

Single-thread Semantics

Given a thread and an element of the thread-partitioning domain, the single-thread semantics returns the traces of all possible partial finite executions, following the happens-before memory model, when the parallel executions of other threads are the ones represented by the given element of the thread-partitioning domain. It is the basic step that will be used to define the multithread semantics. This approach is classic in literature, see for instance the example 7.2.0.6.3 of [27].

DEFINITION 3.3.7 (SINGLE-THREAD SEMANTICS \mathbb{S}°) *Let σ_0 be the initial state of computation.*

$$\begin{aligned} \mathbb{S}^\circ &: [(\Psi \times \Omega \times \text{Tld}) \rightarrow \wp(\text{St}^\tau)] \\ \mathbb{S}^\circ \llbracket \mathbf{f}, \mathbf{r}, \mathbf{t} \rrbracket &= \text{lfp}_\emptyset^{\subseteq} F^\circ \end{aligned}$$

where

$$\begin{aligned} F^\circ &: [\wp(\text{St}^\tau) \rightarrow \wp(\text{St}^\tau)] \\ F^\circ &= \lambda \mathbb{T}. \{\sigma_0\} \cup \{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in \mathbb{T} \wedge \sigma_i \in \text{step}(\mathbf{t}, \mathbf{f}, \mathbf{r}, \sigma_{i-1})\} \end{aligned}$$

Multithread Semantics

The multithreaded fixpoint semantics computes all the possible executions of a multithreaded program following the happens-before memory model.

It starts from an element of the thread-partitioning domain that relates each thread that is active at the beginning of the computation to a trace containing only its initial state σ_0 ($\mathbf{f}_0 = \{\llbracket \mathbf{t} \mapsto \{[0 \rightarrow \sigma_0] : \mathbf{t} \text{ is the identifier of an active thread} \rrbracket\}$), and in the second component each active thread to \perp_Ω ($\mathbf{r}_0 = \{\llbracket \mathbf{t} \mapsto \perp_\Omega : \mathbf{t} \text{ is the identifier of an active thread and } \sigma_0 \text{ is its initial state} \rrbracket\}$). At each iteration it computes the semantics using the multithreaded element obtained at the previous step. In particular we have to discard all the elements that are overwritten during a set of transitions that are synchronized-with the analyzed read action. To do that, we need to consider only the traces that are complete, i.e. restricting the traces only on the ones belonging to set $\text{St}_{\rightarrow}^\tau$.

DEFINITION 3.3.8 (MULTITHREAD SEMANTICS \mathbb{S}^{\parallel})

$$\begin{aligned} \mathbb{S}^{\parallel} &: [\Psi \times \Omega \rightarrow \wp(\Psi \times \Omega)] \\ \mathbb{S}^{\parallel}[[f_0, r_0]] &= \text{lf}_{\emptyset}^{\subseteq} F^{\parallel} \end{aligned}$$

where

$$\begin{aligned} F^{\parallel} &: [\wp(\Psi \times \Omega) \rightarrow \wp(\Psi \times \Omega)] \\ F^{\parallel} &= \lambda \Phi. \{(f_0, r_0)\} \cup \{(f_i, r_{i-1}) : \exists (f_{i-1}, r_{i-1}) \in \Phi : \forall t \in \text{dom}(f_{i-1}) : \\ &\quad \tau \in \mathbb{S}^{\circ}[[f_{i-1}, r_{i-1}, t]], \tau \in \text{St}_{\rightarrow}^{\dagger}, f_i(t) = \tau\} \end{aligned}$$

The intuition behind this fixpoint definition is as follows:

- at the first iteration it computes the complete semantics of each thread “in isolation” since the trace of the other threads is empty, and then the *step* function performs a step using the last state of the given thread following $\overset{\circ}{\rightarrow}$;
- at the second (or *i*-th) iteration it computes the complete semantics of each thread in which the visible values have been modified at most one (or *i*-1) times by different threads.

Discussion

Computing this fixpoint may seem to be useless. A common (but unsound) intuition is that as all the active threads are exposed, we can check which values they write to the shared memory, then compute the semantics using this information, obtaining the result of the analysis. The problem is that the values written by one thread may cause other threads to write new values, and so on. Consider for instance the following example:

Thread 1	Thread 2
if(a == 0){	if(a == 1){
a = 1;	a = 2;
if(a == 2){	if(a == 3){
a = 3;	a = 4;
if(a == 4){	if(a == 5){
...	...

Even if we are considering a multithreaded program without loops and which is composed only of two threads, there is not a clear number of iterations after which all the possible behaviors would be exposed. In fact, at the first iteration thread 1 would expose the value 1, at the second iteration thread 2 would expose 2, and so on. The number of iterations required in order to expose all the possible behaviors relies on the structure of the program, and so we need to compute a fixpoint in order to be sound.

Context bound analysis

Context bound analysis [114] is a novel approach that obtained a huge amount of both theoretical and practical results during the last years. Starting from the premise that verifying a concurrent program (with a context and synchronization sensitive analysis, when dealing with rendezvous style synchronization primitives) is undecidable [118], a multithreaded program is analyzed until a given context bound, i.e. the number of context switches is limited to n . A context switch happens when the control passes from one thread to another. In this way, these analyses are not sound for all possible executions, but only to those with at most n context switches. During the last years many analyses in this field have been proposed, e.g. [102, 80, 16, 115].

Our approach relies on the idea of abstraction. We are able to build up an analysis sound with respect to all the possible multithreaded executions, with an unbounded number of context switches, executing on all the possible multi-core architectures, and considering also compiler optimizations. This requires two nested fixpoints computation. Intuitively, at the n -th iteration our multithread semantics computes all the executions with at most n context switches. Iterating this process until a fixpoint is reached accumulates all possible multithreaded executions. We will be able to build up a computable approximation of this semantics using an abstraction.

3.3.5 Launching a Thread

The *step* function is not in the position to launch a new thread, as it concerns intra-thread steps only. Thus, the multithread semantics must be extended to support the dynamic creation and launching of threads. Since we are generic with respect to the programming language, we do not present the details. On the other hand, it is important to define the launching of threads in order to explain how the relations between threads are traced by the thread-partitioning concrete domain.

In this context, we suppose that a function $launch : [St \rightarrow (Tid \times St \times St) \cup \perp_l]$ is provided. Given a state, if its next action is the launch of a thread, it returns the identifier of the new thread, its initial state and the next state of the execution. The computational multithreaded step may be defined in the following way, where (f, r) is the previous state:

$$\begin{aligned} (f', r') : \quad & t \in dom(f), f(t) = \sigma_0 \rightarrow \dots \rightarrow \sigma_i, launch(\sigma_i) = (t', \sigma'_0, \sigma_{i+1}), \\ & f' = f[t \mapsto (\sigma_0 \rightarrow \dots \rightarrow \sigma_i \rightarrow \sigma_{i+1}), t' \mapsto (\sigma'_0)], r' = r[t' \mapsto (t, i)] \end{aligned}$$

3.3.6 The Example

We apply these definitions to the example presented in Section 3.1.1. We focus only on the analysis of the condition in Thread 2.

The result obtained by the first iteration of the computation of \mathbb{S}^{\parallel} is depicted by Figure 3.3. We only represent the state of shared memory fields of cheque, ignoring the control state and the private memory.

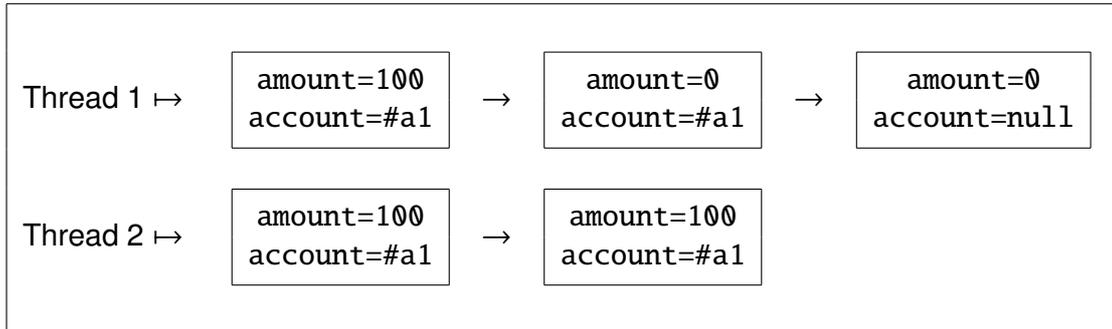


Figure 3.3: The result of the first iteration of the multithread semantics computation

Which are the values returned by the *vis* function when we are evaluating the condition of Thread2 at the second iteration? In order to compute them, we need to consider which values are returned by the *project* function. We ignore the first use of this function, as we suppose there are two parallel threads at the beginning of the execution, such that both are launched by the system. In the second case, we use the *project* function only with the execution trace of Thread1, as it is the only thread in the domain of our multithreaded state that is not the current thread. Note that there is no synchronization action. In this situation, the read of the field *amount* may retrieve both 0 and 100, while reading *account* may return both #a1 and null.

Finally, we are in position to check if, in this situation, the condition may be evaluated to true. The condition to be evaluated is `this.amount > 0 && this.account == null`. If the read action on *amount* sees 100, and it sees the value written by the second instruction of Thread1 and returned by the *vis* function on *account*, the condition would be evaluated to true. So `isValid()` method may return false. This behavior is sound with respect to the happens-before memory model, as pointed out in Section 3.2.2.

3.4 Multithreaded Abstract Semantics

In order to develop a static analysis via abstract interpretation, we define an abstract semantics aimed at computing an approximation of the concrete one. We also prove the soundness of our approach.

3.4.1 Assumptions

As we did for the concrete semantics, in order to be generic with respect to the programming language we need that some sets and functions are provided.

In particular, the required sets are the following, with the same semantics of the ones introduced by the concrete semantics: $\overline{\text{Sh}}$, $\overline{\text{Loc}}$, $\overline{\text{Sync}}$, and $\overline{\text{St}}$.

In the same way the function $\overline{\rightarrow} : [\overline{\text{St}} \times \overline{\text{St}} \rightarrow \{\text{true}, \text{false}\}]$ defines the abstract single step relation.

We require the following functions are provided, with the same meaning as the concrete semantics: $\overline{shared} : [\overline{St} \rightarrow \overline{Sh}]$, $\overline{action} : [\overline{St} \rightarrow \overline{\mathbb{I}}_a \cup (\{r, w\} \times \overline{Loc} \times (\overline{Val} \cup \perp_{\overline{v}}))]$, $\overline{synchronized} : [\overline{St} \rightarrow \wp(\overline{Sync})]$, $\overline{assign} : [\overline{Sh} \times \overline{Loc} \times \overline{Val} \rightarrow \overline{Sh}]$, and $\overline{setshared} : [\overline{St} \times \overline{Sh} \rightarrow \overline{St}]$. We suppose that all these functions are sound with respect to their concrete counterparts.

Abstract Values

While in the concrete context it is quite easy to understand what values are (usually references and numerical values), we need to go into more detail for abstract values. In particular, we need to understand what may be an abstract numerical value when we assign it.

Nonrelational domains: A first type of numerical domains are the nonrelational ones, as for instance boxed intervals [25], i.e. a domain that relates each variable to an interval, that approximates all the possible numerical values that variable may have in the concrete executions. This approach is well-known [24]: a value is nothing more than the interval assigned to a variable in the boxed domain or the one obtained evaluating an expression. For instance, the abstract value of the expression $x + 1$ in the boxed intervals domain $\{[y \mapsto [0..2], x \mapsto [3..5], z \mapsto [-\infty..0]]\}$ is the interval $[3..5] \oplus [1..1] = [4..6]$ (where \oplus is the sum operator on intervals). This value can be assigned to a variable or a location.

Relational domains: Usually, relational domains such as polyhedra [31], octagons [101], pentagons [93], and stripes [47] define the semantics of assignments, evaluating the assigned expression, i.e. the right part of the statement, together with the assignment of its value to the variable. This is an obvious consequence of the fact that they trace relations between variables, and so they cannot deal separately with values but they must deal with variables directly. On the other hand, we can define a value as the set of relational constraints that holds for an expression.

For instance, imagine using the octagon domain (i.e. a domain that trace relations of type $\pm x \pm y \leq k$) when evaluating $i = j + 1$ when the following constraints hold: $+j \leq 10$, $-j \leq 0$, $j + x \leq 0$. In this context, we can infer the set of constraints that holds for the expression $j + 1$ (introducing a special variable e representing the abstract value of the expression), and assign this value to i . So we obtain the constraints $+e \leq 11$, $-e \leq 1$, $e + x \leq 1$, and the final result is $+i \leq 11$, $-i \leq 1$, $i + x \leq 1$ (note that we only need to replace e with i).

In a sequential context, these two steps has to be performed together, as in this way we can infer the constraints on the assigned variable directly. Instead, in a multithreaded program we have that a variable may be assigned in parallel by multiple threads, we need to take the upper bound between all the possibly assigned values (i.e. the approximation of the constraints that hold for all the values), and so we have to distinguish the value obtained from evaluating the expression and the assignment of that value.

Note that classical relational domains are defined on a finite set of variables. Since we deal with location of shared memory, we may be in a context in which locations are dynamically created (which is the case for Java multithreading). Logozzo [91] already applied the octagon domain in such a context, and other relational domains may work

adopting a similar approach.

3.4.2 Thread-partitioning Abstract Domain

The abstract domain is similar to the concrete one: the only difference is that it deals with abstract elements, while the meaning is exactly the same.

$$\begin{aligned}\bar{\Psi} &: [\text{TId} \rightarrow \bar{\text{St}}^\dagger] \\ \bar{\Omega} &: [\text{TId} \rightarrow ((\text{TId} \times \mathbb{N}) \cup \perp_\Omega)]\end{aligned}$$

3.4.3 Upper Bound Operators

We require that the upper bound operator between two single-thread states (\sqcup_{St}) and between two values (\sqcup_{Val}) are provided.

PROPOSITION 3.4.1 *We assume that $\langle \bar{\text{St}}, \sqsubseteq_{\text{St}}, \sqcup_{\text{St}}, \sqcap_{\text{St}} \rangle$ is a complete lattice. Let $\alpha_{\text{St}} : [\wp(\text{St}) \rightarrow \bar{\text{St}}]$ be the abstraction function on set of concrete states such that:*

$$\alpha_{\text{St}}(\mathbf{S}) = \bigsqcup_{\substack{\text{St} \\ \sigma \in \mathbf{S}}} \alpha'_{\text{St}}(\sigma)$$

where $\alpha'_{\text{St}} : [\text{St} \rightarrow \bar{\text{St}}]$ is the abstraction of a single concrete state.

We suppose that α'_{St} function is a join-morphism, i.e. $\alpha'_{\text{St}}(\bigcup_{i \in I} \mathbf{S}_i) = \bigsqcup_{i \in I} \alpha'_{\text{St}}(\mathbf{S}_i)$ for any interval $I \subseteq \mathbb{N}$.

DEFINITION 3.4.2 (UPPER BOUND OPERATOR ON $\bar{\text{St}}^\dagger$) *The upper bound on traces is defined as*

$$\begin{aligned}\sqcup_\tau &: [\bar{\text{St}}^\dagger \times \bar{\text{St}}^\dagger \rightarrow \bar{\text{St}}^\dagger] \\ (\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_j) \sqcup_\tau (\bar{\sigma}'_0 \rightarrow \dots \rightarrow \bar{\sigma}'_i) &= \\ &= (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}'_0) \rightarrow \dots \rightarrow (\bar{\sigma}_j \sqcup_{\text{St}} \bar{\sigma}'_j) \rightarrow \bar{\sigma}'_{j+1} \rightarrow \bar{\sigma}'_i\end{aligned}$$

supposing that $j \leq i$.

DEFINITION 3.4.3 (UPPER BOUND OPERATOR ON $\bar{\Psi}$)

$$\begin{aligned}\sqcup_f &: [\bar{\Psi} \times \bar{\Psi} \rightarrow \bar{\Psi}] \\ \bar{f}_1 \sqcup_f \bar{f}_2 &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2), \\ \bar{\tau} &= \begin{cases} \bar{f}_1(t) \sqcup_\tau \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1) \end{cases} \}\end{aligned}$$

3.4.4 Partial Order Operators

DEFINITION 3.4.4 (PARTIAL ORDER ON $\overline{\text{St}}^{\vec{\tau}}$)

$$\begin{aligned} \sqsubseteq_{\tau}: [\overline{\text{St}}^{\vec{\tau}} \times \overline{\text{St}}^{\vec{\tau}} \rightarrow \{\text{true}, \text{false}\}] \\ \bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_2 \Leftrightarrow \bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_2 = \bar{\tau}_2 \end{aligned}$$

LEMMA 3.4.5 \sqcup_{τ} is well defined

PROOF. Since we suppose that $\langle \overline{\text{St}}, \sqsubseteq_{\text{St}}, \sqcup_{\text{St}} \rangle$ is a complete lattice, then \sqcup_{St} is well-defined. By definition of \sqcup_{τ} :

$$\begin{aligned} (\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_j) \sqcup_{\tau} (\bar{\sigma}'_0 \rightarrow \dots \rightarrow \bar{\sigma}'_i) = \\ = (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}'_0) \rightarrow \dots \rightarrow (\bar{\sigma}_j \sqcup_{\text{St}} \bar{\sigma}'_j) \rightarrow \bar{\sigma}'_{j+1} \rightarrow \bar{\sigma}'_i \end{aligned}$$

supposing that $j \leq i$ (since \sqcup_{St} is commutative this supposition is not restrictive). In this way each state of the trace is well-defined (as or it is composed by an element of the second trace, or it is the result of \sqcup_{St}), and so also \sqcup_{τ} is well defined. ■

LEMMA 3.4.6 \sqcup_f is well-defined

PROOF. By definition of \sqcup_f we have that:

$$\begin{aligned} \bar{f}_1 \sqcup_f \bar{f}_2 = \{ [t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2), \\ \bar{\tau} = \begin{cases} \bar{f}_1(t) \sqcup_{\tau} \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1) \end{cases} \} \end{aligned}$$

So for each thread identifier in $\text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2)$ we have exactly one of the three cases, as the three sets $(\text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2))$, $(\text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2))$, and $(\text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1))$ are a partition of the set $\text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2)$ by basic set properties. Each of these cases is well-defined, as it keeps the thread related to thread in one of the two functions (and by hypothesis of the case the function is defined on it), or it applies \sqcup_{τ} between the two traces contained by the two given function (and by hypothesis of the case the two functions are defined on it, and \sqcup_{τ} is well-defined as proved by lemma 3.4.5).

So \sqcup_f is well-defined. ■

LEMMA 3.4.7 $\langle \overline{\text{St}}^{\vec{\tau}}, \sqsubseteq_{\tau}, \sqcup_{\tau} \rangle$ is a complete lattice

PROOF. First of all we prove that $\langle \overline{\text{St}}^{\vec{\tau}}, \sqsubseteq_{\tau} \rangle$ is a partially ordered set. To do it, we need to prove that \sqsubseteq_{τ} is a partial order operator, i.e. that it is:

- reflexive: $\bar{\tau} \sqsubseteq_{\tau} \bar{\tau}$.

By definition of \sqsubseteq_{τ} , we have that $\bar{\tau} \sqsubseteq_{\tau} \bar{\tau} \Leftrightarrow \bar{\tau} \sqcup_{\tau} \bar{\tau} = \bar{\tau}$, so we need to prove that $\bar{\tau} \sqcup_{\tau} \bar{\tau} = \bar{\tau}$.

$$\begin{aligned}
\bar{\tau} \sqcup_{\tau} \bar{\tau} &= \\
(\text{supp. } \tau = \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i) &= (\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i) \sqcup_{\tau} (\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i) \\
(\text{by definition of } \sqcup_{\tau}) &= (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}_0) \rightarrow \dots \rightarrow (\bar{\sigma}_i \sqcup_{\text{St}} \bar{\sigma}_i) \\
(\text{by idempotence of } \sqcup_{\text{St}}) &= \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i \\
(\text{by definition of } \bar{\tau}) &= \bar{\tau}
\end{aligned}$$

We proved that $\bar{\tau} \sqcup_{\tau} \bar{\tau} = \bar{\tau}$, and so, by definition of \sqsubseteq_{τ} , that $\bar{\tau} \sqsubseteq_{\tau} \bar{\tau}$

- antisymmetric: $\bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_2 \wedge \bar{\tau}_2 \sqsubseteq_{\tau} \bar{\tau}_1 \Rightarrow \bar{\tau}_1 = \bar{\tau}_2$.

By definition of \sqsubseteq_{τ} , we have that $\bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_2 \Leftrightarrow \bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_2 = \bar{\tau}_2(1)$ and $\bar{\tau}_2 \sqsubseteq_{\tau} \bar{\tau}_1 \Leftrightarrow \bar{\tau}_2 \sqcup_{\tau} \bar{\tau}_1 = \bar{\tau}_1(2)$. Supposing that $\bar{\tau}_1 = \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i$, $\bar{\tau}_2 = \bar{\sigma}'_0 \rightarrow \dots \rightarrow \bar{\sigma}'_j$, and $i \leq j$ (the proof is similar if $i > j$), we have that:

$$\begin{aligned}
\bar{\tau}_2 &= \\
(\text{by (1)}) &= \bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_2 \\
(\text{by definition of } \sqcup_{\tau}) &= (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}'_0) \rightarrow \dots \rightarrow (\bar{\sigma}_i \sqcup_{\text{St}} \bar{\sigma}'_i) \rightarrow \\
&\quad \rightarrow \bar{\sigma}'_{i+1} \rightarrow \dots \rightarrow \bar{\sigma}'_j \\
(\text{by commutative property of } \sqcup_{\text{St}}) &= (\bar{\sigma}'_0 \sqcup_{\text{St}} \bar{\sigma}_0) \rightarrow \dots \rightarrow (\bar{\sigma}'_i \sqcup_{\text{St}} \bar{\sigma}_i) \rightarrow \\
&\quad \rightarrow \bar{\sigma}'_{i+1} \rightarrow \dots \rightarrow \bar{\sigma}'_j \\
(\text{by definition of } \sqcup_{\tau}) &= \bar{\tau}_2 \sqcup_{\tau} \bar{\tau}_1 \\
(\text{by (2)}) &= \bar{\tau}_1
\end{aligned}$$

So we proved that $\bar{\tau}_2 = \bar{\tau}_1$.

- transitive: $\bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_2 \wedge \bar{\tau}_2 \sqsubseteq_{\tau} \bar{\tau}_3 \Rightarrow \bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_3$.

By definition of \sqsubseteq_{τ} , we have that $\bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_2 \Rightarrow \bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_2 = \bar{\tau}_2(1)$ and $\bar{\tau}_2 \sqsubseteq_{\tau} \bar{\tau}_3 \Rightarrow \bar{\tau}_2 \sqcup_{\tau} \bar{\tau}_3 = \bar{\tau}_3(2)$.

Supposing that $\bar{\tau}_1 = \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i$, $\bar{\tau}_2 = \bar{\sigma}'_0 \rightarrow \dots \rightarrow \bar{\sigma}'_j$, $\bar{\tau}_3 = \bar{\sigma}''_0 \rightarrow \dots \rightarrow \bar{\sigma}''_k$, by (1) and (2) we have that $j \geq i \wedge k \geq j$, and by definition of \sqcup_{τ} we have that $\bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_2 = (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}'_0) \rightarrow \dots \rightarrow (\bar{\sigma}_i \sqcup_{\text{St}} \bar{\sigma}'_i) \rightarrow \bar{\sigma}'_{i+1} \rightarrow \dots \rightarrow \bar{\sigma}'_j = \bar{\sigma}'_0 \rightarrow \dots \rightarrow \bar{\sigma}'_j(3)$, and $\bar{\tau}_2 \sqcup_{\tau} \bar{\tau}_3 = (\bar{\sigma}'_0 \sqcup_{\text{St}} \bar{\sigma}''_0) \rightarrow \dots \rightarrow (\bar{\sigma}'_j \sqcup_{\text{St}} \bar{\sigma}''_j) \rightarrow \bar{\sigma}''_{j+1} \rightarrow \dots \rightarrow \bar{\sigma}''_k = \bar{\sigma}''_0 \rightarrow$

$$\begin{aligned}
& \dots \rightarrow \bar{\sigma}_k''(4). \\
\bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_3 &= \\
\text{(by def. of } \sqcup_{\tau}) &= (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}_0'') \rightarrow \dots \rightarrow (\bar{\sigma}_i \sqcup_{\text{St}} \bar{\sigma}_i'') \rightarrow \bar{\sigma}_{i+i}'' \rightarrow \dots \rightarrow \bar{\sigma}_k'' \\
&\text{(by (4))} = (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}_0' \sqcup_{\text{St}} \bar{\sigma}_0'') \rightarrow \dots \rightarrow (\bar{\sigma}_i \sqcup_{\text{St}} \bar{\sigma}_i' \sqcup_{\text{St}} \bar{\sigma}_i'') \rightarrow \\
&\quad \rightarrow (\bar{\sigma}_{i+1}' \sqcup_{\text{St}} \bar{\sigma}_{i+1}'') \rightarrow \dots \rightarrow (\bar{\sigma}_j' \sqcup_{\text{St}} \bar{\sigma}_j'') \rightarrow \\
&\quad \rightarrow \bar{\sigma}_{j+i}'' \rightarrow \dots \rightarrow \bar{\sigma}_k'' \\
&\text{(by (2))} = (\bar{\sigma}_0 \sqcup_{\text{St}} \bar{\sigma}_0' \sqcup_{\text{St}} \bar{\sigma}_0'') \rightarrow \dots \rightarrow (\bar{\sigma}_i \sqcup_{\text{St}} \bar{\sigma}_i' \sqcup_{\text{St}} \bar{\sigma}_i'') \rightarrow \\
&\quad \rightarrow \bar{\sigma}_{i+i}'' \rightarrow \dots \rightarrow \bar{\sigma}_k'' \\
&\text{(by (3))} = (\bar{\sigma}_0' \sqcup_{\text{St}} \bar{\sigma}_0'') \rightarrow \dots \rightarrow (\bar{\sigma}_i' \sqcup_{\text{St}} \bar{\sigma}_i'') \rightarrow \bar{\sigma}_{i+i}'' \rightarrow \dots \rightarrow \bar{\sigma}_k'' \\
&\text{(by (4))} = \bar{\sigma}_0'' \rightarrow \dots \rightarrow \bar{\sigma}_i'' \rightarrow \bar{\sigma}_{i+i}'' \rightarrow \dots \rightarrow \bar{\sigma}_k'' \\
&\text{(by def. of } \bar{\tau}_3) = \bar{\tau}_3
\end{aligned}$$

We proved that $\bar{\tau}_1 \sqcup_{\tau} \bar{\tau}_3 = \bar{\tau}_3$, and so $\bar{\tau}_1 \sqsubseteq_{\tau} \bar{\tau}_3$ by definition of \sqsubseteq_{τ} .

The fact that every subset of $\overline{\text{St}}^{\vec{\tau}}$ has a least upper bound is a trivial consequence of the hypothesis that every subset of $\overline{\text{St}}$ has a least upper bound, as we suppose that $\langle \overline{\text{St}}, \sqsubseteq_{\text{St}}, \sqcup_{\text{St}} \rangle$ is a complete lattice, and of the definition of \sqcup_{τ} .

We proved that $\langle \overline{\text{St}}^{\vec{\tau}}, \sqsubseteq_{\tau} \rangle$ is a partial ordered set and, that every subset of $\overline{\text{St}}^{\vec{\tau}}$ has a least upper bound, so that $\langle \overline{\text{St}}^{\vec{\tau}}, \sqsubseteq_{\tau}, \sqcup_{\tau} \rangle$ is a complete lattice. ■

DEFINITION 3.4.8 (ORDERING OPERATOR ON $\overline{\Psi}$)

$$\begin{aligned}
\sqsubseteq_f: [\overline{\Psi} \times \overline{\Psi} \rightarrow \{\text{true}, \text{false}\}] \\
\bar{f}_1 \sqsubseteq_f \bar{f}_2 \Leftrightarrow \bar{f}_1 \sqcup_f \bar{f}_2 = \bar{f}_2
\end{aligned}$$

LEMMA 3.4.9 (\sqsubseteq_f IS REFLEXIVE) \sqsubseteq_f is reflexive

PROOF. \sqsubseteq_f is reflexive iff $\bar{f} \sqsubseteq_f \bar{f}$. By definition of \sqsubseteq_f , $\bar{f} \sqsubseteq_f \bar{f} \Rightarrow \bar{f} \sqcup_f \bar{f} = \bar{f}$

$$\begin{aligned}
\bar{f} \sqcup_f \bar{f} &= \\
\text{(by def. of } \sqcup_f \text{ and set properties)} &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}), \bar{\tau} = \bar{f}(t) \sqcup_{\tau} \bar{f}(t)\} \\
\text{(by idempotence property of } \sqcup_{\tau}) &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}), \bar{\tau} = \bar{f}(t)\} \\
&= \bar{f}
\end{aligned}$$

We proved that $\bar{f} \sqcup_f \bar{f} = \bar{f}$, and so that \sqsubseteq_f is reflexive ■

LEMMA 3.4.10 (\sqsubseteq_f IS ANTISYMMETRIC) \sqsubseteq_f is antisymmetric

PROOF. We have to prove that $\bar{f}_1 \sqsubseteq_f \bar{f}_2 \wedge \bar{f}_2 \sqsubseteq_f \bar{f}_1 \Rightarrow \bar{f}_1 = \bar{f}_2$.

By definition of \sqsubseteq_f we have that $\bar{f}_1 \sqsubseteq_f \bar{f}_2 \Rightarrow \bar{f}_1 \sqcup_f \bar{f}_2 = \bar{f}_2$ (1) and $\bar{f}_2 \sqsubseteq_f \bar{f}_1 \Rightarrow \bar{f}_2 \sqcup_f \bar{f}_1 = \bar{f}_1$

(2).

By definition of \sqcup_f and (1) we have that:

$$\begin{aligned} \bar{f}_2 &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2), \\ \bar{\tau} &= \begin{cases} \bar{f}_1(t) \sqcup_{\tau} \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1) \end{cases} \} (3) \end{aligned}$$

By definition of \sqcup_f and (2) we have that:

$$\begin{aligned} \bar{f}_1 &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_2) \cup \text{dom}(\bar{f}_1), \\ \bar{\tau} &= \begin{cases} \bar{f}_2(t) \sqcup_{\tau} \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_2) \cap \text{dom}(\bar{f}_1) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2) \end{cases} \} (4) \end{aligned}$$

By commutative property of \cup , \sqcup_{τ} , and \cap we can rewrite (4) in the following way:

$$\begin{aligned} \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2), \\ \bar{\tau} &= \begin{cases} \bar{f}_1(t) \sqcup_{\tau} \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2) \end{cases} \} \end{aligned}$$

that is exactly the same results obtained in (3). Then $\bar{f}_1 = \bar{f}_2$.

We proved that if $\bar{f}_1 \sqsubseteq_f \bar{f}_2 \wedge \bar{f}_2 \sqsubseteq_f \bar{f}_1$ then $\bar{f}_1 = \bar{f}_2$, and so that \sqsubseteq_f is antisymmetric. ■

LEMMA 3.4.11 (\sqsubseteq_f IS TRANSITIVE) \sqsubseteq_f is transitive

PROOF. We have to prove that $\bar{f}_1 \sqsubseteq_f \bar{f}_2 \wedge \bar{f}_2 \sqsubseteq_f \bar{f}_3 \Rightarrow \bar{f}_1 \sqsubseteq_f \bar{f}_3$.

By definition of \sqsubseteq_f we have that $\bar{f}_1 \sqsubseteq_f \bar{f}_2 \Rightarrow \bar{f}_1 \sqcup_f \bar{f}_2 = \bar{f}_2$ (1) and $\bar{f}_2 \sqsubseteq_f \bar{f}_3 \Rightarrow \bar{f}_2 \sqcup_f \bar{f}_3 = \bar{f}_3$ (2).

By definition of \sqcup_f and (1) we have that:

$$\begin{aligned} \bar{f}_2 &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2), \\ \bar{\tau} &= \begin{cases} \bar{f}_1(t) \sqcup_{\tau} \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \setminus \text{dom}(\bar{f}_2) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_1) \end{cases} \} (3) \end{aligned}$$

By definition of \sqcup_f and (2) we have that:

$$\begin{aligned} \bar{f}_3 &= \{[t \mapsto \bar{\tau}] : t \in \text{dom}(\bar{f}_2) \cup \text{dom}(\bar{f}_3), \\ \bar{\tau} &= \begin{cases} \bar{f}_2(t) \sqcup_{\tau} \bar{f}_3(t) & \text{if } t \in \text{dom}(\bar{f}_2) \cap \text{dom}(\bar{f}_3) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \setminus \text{dom}(\bar{f}_3) \\ \bar{f}_3(t) & \text{if } t \in \text{dom}(\bar{f}_3) \setminus \text{dom}(\bar{f}_2) \end{cases} \} (4) \end{aligned}$$

Note that in (3) we have that $dom(\bar{f}_1) \subseteq dom(\bar{f}_2)$ and in (4) $dom(\bar{f}_2) \subseteq dom(\bar{f}_3)$; so by transitive property of \subseteq we have that $dom(\bar{f}_1) \subseteq dom(\bar{f}_3)$.

By (3) we have that if $t \in dom(\bar{f}_1)$ then $\bar{f}_2(t) = \bar{f}_1(t) \sqcup_{\tau} \bar{f}_2(t)$ (5). By (4) we have that if $t \in dom(\bar{f}_2)$ then $\bar{f}_3(t) = \bar{f}_2(t) \sqcup_{\tau} \bar{f}_3(t)$ (6).

If $t \in dom(\bar{f}_1) \subseteq dom(\bar{f}_2)$ then by (5) and (6) we obtain that $\bar{f}_3(t) = \bar{f}_1(t) \sqcup_{\tau} \bar{f}_2(t) \sqcup_{\tau} \bar{f}_3(t) = \bar{f}_1(t) \sqcup_{\tau} \bar{f}_3(t)$ (7).

So by (7) and (4) we obtain that:

$$\bar{f}_3 = \{[t \mapsto \bar{\tau}] : t \in dom(\bar{f}_3), \bar{\tau} = \begin{cases} \bar{f}_1(t) \sqcup_{\tau} \bar{f}_3(t) & \text{if } t \in dom(\bar{f}_1) \cap dom(\bar{f}_3) \\ \bar{f}_3(t) & \text{if } t \notin dom(\bar{f}_1) \cap dom(\bar{f}_3) \end{cases}\} (8)$$

Since $dom(\bar{f}_1) \subseteq dom(\bar{f}_3)$, by basic set properties we have that $dom(\bar{f}_3) = dom(\bar{f}_1) \cup dom(\bar{f}_3)$, $dom(\bar{f}_1) \setminus dom(\bar{f}_3) = \emptyset$, and $t \notin dom(\bar{f}_1) \cap dom(\bar{f}_3)$ is equivalent to $t \in dom(\bar{f}_3) \setminus dom(\bar{f}_1)$. So (8) can be rewritten as:

$$\bar{f}_3 = \{[t \mapsto \bar{\tau}] : t \in dom(\bar{f}_1) \cup dom(\bar{f}_3), \bar{\tau} = \begin{cases} \bar{f}_1(t) \sqcup_{\tau} \bar{f}_3(t) & \text{if } t \in dom(\bar{f}_1) \cap dom(\bar{f}_3) \\ \bar{f}_1(t) & \text{if } t \in dom(\bar{f}_1) \setminus dom(\bar{f}_3) \\ \bar{f}_3(t) & \text{if } t \in dom(\bar{f}_3) \setminus dom(\bar{f}_1) \end{cases}\} (9)$$

By (9) and by definition of \sqcup_f follows that $\bar{f}_3 = \bar{f}_1 \sqcup_f \bar{f}_3$, and by definition of \sqsubseteq_f we obtain that $\bar{f}_1 \sqsubseteq_f \bar{f}_3$.

We proved that $\bar{f}_1 \sqsubseteq_f \bar{f}_2 \wedge \bar{f}_2 \sqsubseteq_f \bar{f}_3 \Rightarrow \bar{f}_1 \sqsubseteq_f \bar{f}_3$, and so that \sqsubseteq_f is transitive. \blacksquare

LEMMA 3.4.12 $\langle \bar{\Psi}, \sqsubseteq_f, \sqcup_f \rangle$ is a complete lattice

PROOF. First of all we prove that $\langle \bar{\Psi}, \sqsubseteq_f \rangle$ is a partially ordered set. To do it, we proved that \sqsubseteq_f is a partial order operator, alias that \sqsubseteq_f is:

- reflexive by lemma 3.4.9;
- antisymmetric by lemma 3.4.10;
- transitive by lemma 3.4.11.

The fact that every subset of $\bar{\Psi}$ has a least upper bound is a trivial consequence of the fact that every subset of $\bar{\text{St}}^{\bar{\tau}}$ has a least upper bound as guaranteed by lemma 3.4.7, and by definition of \sqcup_f .

We proved that $\langle \bar{\Psi}, \sqsubseteq_f \rangle$ is a partially ordered set and that every subset of $\bar{\Psi}$ has a least upper bound, so $\langle \bar{\Psi}, \sqsubseteq_f, \sqcup_f \rangle$ is a complete lattice. \blacksquare

3.4.5 Abstraction Functions

Note that in the following definitions and in the soundness proofs we focus only on the first component of the domain Ψ , as the second component just traces some relations between threads.

DEFINITION 3.4.13 (ABSTRACTION FUNCTION OF $\wp(\text{St}^\ddagger)$)

$$\begin{aligned} \alpha_\tau &: [\wp(\text{St}^\ddagger) \rightarrow \overline{\text{St}^\ddagger}] \\ \alpha_\tau(\mathbb{T}) &= \bigsqcup_{\tau \in \mathbb{T}} \alpha'_\tau(\tau) \end{aligned}$$

where

$$\begin{aligned} \alpha'_\tau &: [\text{St}^\ddagger \rightarrow \overline{\text{St}^\ddagger}] \\ \alpha'_\tau(\sigma_0 \rightarrow \dots \rightarrow \sigma_i) &= \alpha'_{\text{St}}(\sigma_0) \rightarrow \dots \rightarrow \alpha'_{\text{St}}(\sigma_i) \end{aligned}$$

DEFINITION 3.4.14 (ABSTRACTION FUNCTION OF $\wp(\Psi)$)

$$\begin{aligned} \alpha_f &: [\wp(\Psi) \rightarrow \overline{\Psi}] \\ \alpha_f(\Phi) &= \bigsqcup_f \alpha'_f(f) \end{aligned}$$

where

$$\begin{aligned} \alpha'_f &: [\Psi \rightarrow \overline{\Psi}] \\ \alpha'_f(f) &= \{[t \mapsto \bar{t}] : \exists t \in \text{dom}(f) : \bar{t} = \alpha'_\tau(f(t))\} \end{aligned}$$

LEMMA 3.4.15 α_τ is a join-morphism

PROOF. α_τ is a join-morphism iff $\alpha_\tau\left(\bigcup_{i \in I} \mathbb{T}_i\right) = \bigsqcup_{i \in I} \alpha_\tau(\mathbb{T}_i)$ for any interval $I \subseteq \mathbb{N}$.

$$\begin{aligned} \alpha_\tau\left(\bigcup_{i \in I} \mathbb{T}_i\right) &= \\ \text{(by def. of } \alpha_\tau) &= \bigsqcup_{\sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \bigcup_{i \in I} \mathbb{T}_i} \alpha'_\tau(\sigma_0 \rightarrow \dots \rightarrow \sigma_i) \\ \text{(by def. of } \alpha'_\tau) &= \bigsqcup_{\sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \bigcup_{i \in I} \mathbb{T}_i} \bigsqcup_{\text{St}} \alpha'_{\text{St}}(\sigma_0) \rightarrow \dots \rightarrow \bigsqcup_{\text{St}} \alpha'_{\text{St}}(\sigma_i) \rightarrow \dots \\ \text{(by prop. 3.4.1)} &= \bigsqcup_{i \in I} \bigsqcup_{\sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \mathbb{T}_i} \alpha'_{\text{St}}(\sigma_0) \rightarrow \dots \rightarrow \bigsqcup_{i \in I} \bigsqcup_{\sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \mathbb{T}_i} \alpha'_{\text{St}}(\sigma_i) \rightarrow \dots \\ \text{(by def. of } \bigsqcup_\tau) &= \bigsqcup_{i \in I} \bigsqcup_{\sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \mathbb{T}_i} (\alpha'_{\text{St}}(\sigma_0) \rightarrow \dots \rightarrow \alpha'_{\text{St}}(\sigma_i)) \\ \text{(by def. of } \alpha'_\tau) &= \bigsqcup_{i \in I} \bigsqcup_{\sigma_0 \rightarrow \dots \rightarrow \sigma_i \in \mathbb{T}_i} \alpha'_\tau(\sigma_0 \rightarrow \dots \rightarrow \sigma_i) \\ \text{(by def. of } \alpha_\tau) &= \bigsqcup_{i \in I} \alpha_\tau(\mathbb{T}_i) \end{aligned}$$

We prove that $\alpha_\tau\left(\bigcup_{i \in I} \mathbb{T}_i\right) = \bigsqcup_{i \in I} (\alpha_\tau(\mathbb{T}_i))$, and so that α_τ is a join-morphism. ■

LEMMA 3.4.16 α_f is a join-morphism

PROOF. α_f is a join-morphism iff $\alpha_f\left(\bigcup_{i \in I} \Phi_i\right) = \bigsqcup_{i \in I} (\alpha_f(\Phi_i))$ for any interval $I \subseteq \mathbb{N}$.

$$\begin{aligned} \alpha_f\left(\bigcup_{i \in I} \Phi_i\right) &= \\ \text{(by def. of } \alpha_f) &= \bigsqcup_{f \in \bigcup_{i \in I} \Phi_i} \alpha'_f(f) \\ \text{(by def. of } \alpha'_f) &= \bigsqcup_{f \in \bigcup_{i \in I} \Phi_i} \{[t \mapsto \bar{\tau}] : \exists t \in \text{dom}(f) : \bar{\tau} = \alpha'_\tau(f(t))\} \end{aligned}$$

\bigsqcup_f by definition makes the least upper bound (through \sqcup_τ) of the traces of all the abstract functions defined on the same thread identifier. As $\langle \overline{\text{St}}^\dagger, \sqsubseteq_\tau, \bigsqcup_\tau \rangle$ is a complete lattice by lemma 3.4.7 and α_τ is a join-morphism by lemma 3.4.15, for each Φ_i with $i \in I$ we obtain that the least upper bound between all its functions is equal to $\bigsqcup_{f \in \Phi_i} \alpha'_f$, that, by

definition of α_f is equal to $\alpha_f(\Phi_i)$.

So we obtain that $\alpha_f(\bigcup_{i \in I} \Phi_i) = \bigsqcup_{i \in I} (\alpha_f(\Phi_i))$, i.e. that α_f is a join-morphism. \blacksquare

3.4.6 $\overline{\text{step}}$ Function

The $\overline{\text{step}}$ function is quite similar to the concrete one. If the action is not a read it just performs the step through the $\overset{\circ}{\rightarrow}$ function. Otherwise it computes the next step injecting the least upper bound of all the values returned by the $\overline{\text{vis}}$ function into the read value and considering also the sequentially consistent case. The $\overline{\text{vis}}$ function is obtained as the canonical abstraction of the vis function.

DEFINITION 3.4.17 ($\overline{\text{step}}$ FUNCTION)

$$\begin{aligned} \overline{\text{step}} : [\text{TId} \times \overline{\Psi} \times \overline{\Omega} \times \overline{\text{St}} \rightarrow \overline{\text{St}}] \\ \overline{\text{step}}(t, \bar{f}, \bar{r}, \overline{\sigma}_i) = \overline{\sigma} \text{ such that} \\ \overline{\sigma}_i \overset{\circ}{\rightarrow} \overline{\sigma} & \text{ if } \pi_1(\overline{\text{action}}(\overline{\sigma}_i)) \neq r \\ \\ \overline{\sigma}'_i \overset{\circ}{\rightarrow} \overline{\sigma} : & \text{ if } \overline{\text{action}}(\overline{\sigma}_i) = (r, \bar{l}, \perp_v) \\ \overline{V} = \overline{\text{vis}}(t, \bar{l}, \overline{\text{synchronized}}(\overline{\sigma}_i), \bar{f}, \bar{r}(t)) \\ \overline{v} = \bigsqcup_{\overline{v} \in \overline{V}} \overline{v}' \\ \overline{\text{sh}} = \overline{\text{shared}}(\overline{\sigma}_i), \overline{\text{sh}}' = \overline{\text{assign}}(\overline{\text{sh}}, \bar{l}, \overline{v}) \\ \overline{\sigma}'_i = \overline{\text{setshared}}(\overline{\sigma}, \overline{\text{sh}}') \sqcup \overline{\sigma}_i \end{aligned}$$

DEFINITION 3.4.18 (\overline{vis} FUNCTION)

$$\begin{aligned} \overline{vis} &: [\text{Tld} \times \overline{\text{Loc}} \times \wp(\overline{\text{Sync}}) \times \overline{\Psi} \times (\text{Tld} \times \mathbb{N}) \rightarrow \wp(\overline{\text{Val}})] \\ \overline{vis}(t, \bar{l}, \bar{S}, \bar{f}, (t', i')) &= \\ &= \overline{project}(\bar{l}, \overline{suffix}(\bar{f}(t'), i'), \bar{S}) \cup \\ &\{\bar{v} : \bar{v} \in \overline{project}(\bar{l}, \bar{f}(t''), \bar{S}) : t'' \in \text{dom}(\bar{f}) \setminus \{t, t'\}\} \end{aligned}$$

DEFINITION 3.4.19 (\overline{suffix} FUNCTION)

$$\begin{aligned} \overline{suffix} &: [\overline{\text{St}}^{\dagger} \times \mathbb{N} \rightarrow \overline{\text{St}}^{\dagger}] \\ \overline{suffix}(\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_j, i) &= \begin{cases} \bar{\sigma}_i \rightarrow \dots \rightarrow \bar{\sigma}_j & \text{if } i \geq 0 \wedge i \leq j \\ \bar{\epsilon} & \text{otherwise} \end{cases} \end{aligned}$$

DEFINITION 3.4.20 ($\overline{project}$ FUNCTION)

$$\begin{aligned} \overline{project} &: [\overline{\text{Loc}} \times \overline{\text{St}}^{\dagger} \times \wp(\overline{\text{Sync}}) \rightarrow \wp(\overline{\text{Val}})] \\ \overline{project}(\bar{l}, \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i, \bar{S}) &= \{\bar{v} : \exists j \in [0..i] : \overline{action}(\bar{\sigma}_j) = (w, \bar{l}, \bar{v}) \wedge \\ \overline{notsynchronized}(\bar{\sigma}_j \rightarrow \dots \rightarrow \bar{\sigma}_i, \bar{S})\} \end{aligned}$$

DEFINITION 3.4.21 ($\overline{notsynchronized}$ FUNCTION)

$$\begin{aligned} \overline{notsynchronized} &: [\overline{\text{St}}^{\dagger} \times \wp(\overline{\text{Sync}}) \rightarrow \{\text{true}, \text{false}\}] \\ \overline{notsynchronized}(\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i, \bar{S}) &= \text{true if and only if} \\ \bar{S} \cap \overline{synchronized}(\bar{\sigma}_0) &= \emptyset \vee \\ \nexists \bar{\sigma}_j \in \overline{cut}(\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i, \bar{S}) : \overline{action}(\bar{\sigma}_{j-1}) &= (w, \bar{l}, \bar{v}), \\ \overline{action}(\bar{\sigma}_0) &= (w, \bar{l}_0, \bar{v}_0), \bar{l} = \bar{l}_0 \end{aligned}$$

DEFINITION 3.4.22 (\overline{cut} FUNCTION)

$$\begin{aligned} \overline{cut} &: [\overline{\text{St}}^{\dagger} \times \wp(\overline{\text{Sync}}) \rightarrow \overline{\text{St}}^{\dagger}] \\ \overline{cut}(\bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i, \bar{S}) &= \begin{cases} \bar{\epsilon} & \text{if } \overline{synchronized}(\bar{\sigma}_0) \cap \bar{S} = \emptyset \\ \bar{\sigma}_0 \rightarrow_{\tau} \overline{cut}(\bar{\sigma}_1 \rightarrow \dots \rightarrow \bar{\sigma}_i, \bar{S}) & \text{otherwise} \end{cases} \end{aligned}$$

3.4.7 Fixpoint Semantics

We proceed as in Section 3.3.4: we define the single-thread semantics in fixpoint form based on the \overline{step} function just presented. Then we present the multithread semantics.

DEFINITION 3.4.23 ($\overline{\mathbb{S}}^{\circ}$)

$$\begin{aligned} \overline{\mathbb{S}}^{\circ} &: [(\overline{\Psi} \times \overline{\Omega} \times \text{Tld}) \rightarrow \overline{\text{St}}^{\dagger}] \\ \overline{\mathbb{S}}^{\circ}[[\bar{f}, \bar{r}, t]] &= \text{lfp}_{\bar{\epsilon}}^{\tau} F^{\circ} \end{aligned}$$

where

$$\begin{aligned} \overline{F}^\circ &: [\overline{\text{St}}^\ddagger \rightarrow \overline{\text{St}}^\ddagger] \\ \overline{F}^\circ &= \lambda \bar{\tau}. \{\overline{\sigma}_0\} \sqcup_\tau \{\overline{\sigma}_0 \rightarrow \dots \rightarrow \overline{\sigma}_{i-1} \rightarrow \overline{\sigma}_i : \overline{\sigma}_0 \rightarrow \dots \rightarrow \overline{\sigma}_{i-1} = \bar{\tau} \wedge \\ &\quad \overline{\sigma}_i = \overline{\text{step}}(\bar{t}, \bar{f}, \bar{r}, \overline{\sigma}_{i-1})\} \end{aligned}$$

DEFINITION 3.4.24 ($\overline{\mathbb{S}}^\parallel$)

$$\begin{aligned} \overline{\mathbb{S}}^\parallel &: [\overline{\Psi} \times \overline{\Omega} \rightarrow \overline{\Psi} \times \overline{\Omega}] \\ \overline{\mathbb{S}}^\parallel \llbracket \bar{f}_0, \bar{r}_0 \rrbracket &= \text{lfp}_\emptyset^{\sqsubseteq_f} \overline{F}^\parallel \end{aligned}$$

where

$$\begin{aligned} \overline{F}^\parallel &: [\overline{\Psi} \times \overline{\Omega} \rightarrow \overline{\Psi} \times \overline{\Omega}] \\ \overline{F}^\parallel &= \lambda (\bar{f}, \bar{r}). \{(\bar{f}_0, \bar{r}_0)\} \sqcup_f \{(\bar{f}_i, \bar{r}) : \forall t \in \text{dom}(\bar{f}) : \bar{f}_i(t) = \overline{\mathbb{S}}^\circ \llbracket \bar{f}, \bar{r}, t \rrbracket\} \end{aligned}$$

The intuition of these definitions is exactly the same of the concrete semantics: $\overline{\mathbb{S}}^\circ$ computes the semantics of a single thread given a multithreaded state (from which the $\overline{\text{step}}$ function extrapolates the visible values of the shared memory through the $\overline{\text{vis}}$ function), while $\overline{\mathbb{S}}^\parallel$ iterates this computation using the previous multithreaded state for each thread until a fixpoint is reached.

The definition of the multithread semantics may be straightforwardly extended in order to support widening and narrowing operators [25], which are required to guarantee the convergence of the analysis and refine the results when the abstract domain is of infinite height.

LEMMA 3.4.25 (SOUNDNESS OF $\overline{\text{step}}$) *Supposing that $\overset{\circ}{\rightarrow}$ is sound, then $\forall t \in \text{TId}, (f, r) \in \Psi \times \Omega, \sigma \in \text{St} : \alpha_{\text{St}}(\text{step}(t, f, r, \sigma_i)) \sqsubseteq_{\text{St}} \overline{\text{step}}(t, \alpha_f(f), \alpha_r(r), \alpha_{\text{St}}(\sigma_i))$*

PROOF. We reason by case:

- if $\pi_1(\text{action}(\sigma_i)) \neq r$, then, by definition of step , $\text{step}(t, f, r, \sigma_i) = \sigma : \sigma_i \overset{\circ}{\rightarrow} \sigma$. By definition of $\overline{\text{step}}$, $\overline{\text{step}}(t, \alpha_f(f), \alpha_r(r), \alpha_{\text{St}}(\sigma_i)) = \overline{\sigma} : \alpha_{\text{St}}(\sigma_i) \overset{\circ}{\rightarrow} \overline{\sigma}$. As by hypothesis $\overset{\circ}{\rightarrow}$ is sound, then $\alpha_{\text{St}}(\sigma) \sqsubseteq_{\text{St}} \overline{\sigma}$. Finally, we proved that in this case $\forall t \in \text{TId}, (f, s) \in \Psi \times \Omega, \sigma_i \in \text{St} : \alpha_{\text{St}}(\text{step}(t, f, r, \sigma_i)) \sqsubseteq_{\text{St}} \overline{\text{step}}(t, \alpha_f(f), \alpha_r(r), \alpha_{\text{St}}(\sigma_i))$ where $\alpha'_r : \Omega \mapsto \overline{\Omega}$ is the abstraction function on the concrete set of functions Ω .
- if $\pi_1(\text{action}(\sigma_i)) = r$, then, by definition of step , we can have two cases:
 - $\text{step}(t, f, r, \sigma_i) = \sigma : \sigma_i \overset{\circ}{\rightarrow} \sigma$. It is the same situation of the first case. The $\overline{\text{step}}$ function makes the least upper bound between all the visible values, and it considers also the value exposed at single-thread level. In this way the soundness is preserved

- otherwise the *step* function takes a value returned by the *vis* function, injects it in the state, executes the step through the $\overset{\circ}{\rightarrow}$ function, and returns the obtained state. Indeed, the \overline{step} function makes the least upper bound between all the values returned by the abstraction of *vis* function. Since we suppose that the functions that operates on abstract states are a sound approximation of the concrete ones, and \overline{step} performs the same operations of its concrete counterpart, it soundly approximates *step* for all the possible values provided by *vis*. As \overline{step} makes the least upper bound between all the values, it is sound with respect to any value that may be returned by *vis* function, and so the result of \overline{step} is sound.

We proved that in all the possible cases \overline{step} is sound with respect to *step*. ■

LEMMA 3.4.26 $\forall T \in \wp(\text{St}^{\ddagger}) : \alpha_{\tau}(F^{\circ}(T)) \sqsubseteq_{\tau} \overline{F^{\circ}}(\alpha_{\tau}(T))$

PROOF.

$$\begin{aligned}
\alpha_{\tau}(F^{\circ}(T)) &= \text{(by definition of } F^{\circ}\text{)} \\
&\quad \alpha_{\tau}(\lambda T. \{\sigma_0\} \cup \{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\
&\quad \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r, \sigma_i)\}) \\
&= \text{(by functional lifting)} \\
&\quad \lambda \alpha_{\tau}(T). \alpha_{\tau}(\{\sigma_0\} \cup \{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\
&\quad \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r, \sigma_i)\}) \\
&= \text{(as } \alpha_{\tau} \text{ is a join-morphism as proved in Section 3.4.15)} \\
&\quad \lambda \alpha_{\tau}(T). \alpha_{\tau}(\{\sigma_0\}) \sqcup_{\tau} \alpha_{\tau}(\{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \\
&\quad \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r, \sigma_i)\}) \\
&= \text{(by definition of } \alpha_{\tau}\text{)} \\
&\quad \lambda \alpha_{\tau}(T). \alpha'_{\tau}(\sigma_0) \sqcup_{\tau} \{\alpha'_{\tau}(\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i) : \\
&\quad \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_i \in \text{step}(t, f, r, \sigma_i)\} \\
&\sqsubseteq_{\tau} \text{(by lemma 3.4.25 and definition of } \sqsubseteq_{\tau}\text{)} \\
&\quad \lambda \alpha_{\tau}(T). \alpha'_{\tau}(\sigma_0) \sqcup_{\tau} \{\overline{\sigma_0} \rightarrow \dots \rightarrow \overline{\sigma_{i-1}} \rightarrow \overline{\sigma_i} : \\
&\quad \overline{\sigma_0} \rightarrow \dots \rightarrow \overline{\sigma_{i-1}} = \alpha_{\tau}(T) \wedge \\
&\quad \overline{\sigma_i} \in \overline{\text{step}}(t, \alpha_f(f), \alpha_r(r), \alpha_{\text{St}}(\sigma_i))\} \\
&= \text{(by definition of } \overline{F^{\circ}}\text{)} \\
&\quad \overline{F^{\circ}}(\alpha_{\tau}(T))
\end{aligned}$$

■

LEMMA 3.4.27 *Let be $\bar{f} \in \overline{\Psi}$, $\bar{r} \in \overline{\Omega}$, and $t \in \text{Tld}$. Supposing that $\overline{\text{St}}_{pre}$ is the set of all the prefixpoints of $\overline{F^{\circ}}$, then $\forall \overline{\sigma_0} \in \overline{\text{St}}_{pre} : \alpha_{\tau}(\mathbb{S}^{\circ})[[\bar{f}, \bar{r}, t]] \sqsubseteq_{\tau} \overline{\mathbb{S}^{\circ}}[[\bar{f}, \bar{r}, t]]$.*

PROOF. Note that:

- F° is monotonic for \sqsubseteq as trivial consequence of its definition

- $\overline{F^\circ}$ is monotonic for \sqsubseteq_τ as trivial consequence of its definition
- $\langle \wp(\text{St}), \subseteq, \cup \rangle$ is a complete lattice by properties of set union operator and subset ordering
- $\langle \overline{\text{St}}, \sqsubseteq_\tau, \sqcup_\tau \rangle$ is a complete lattice as proved by lemma 3.4.7
- $\overline{\sigma}$ is a prefixpoint of $\overline{F^\circ}$ by hypothesis
- α_τ is a join-morphism as proved by lemma 3.4.15
- $\alpha_\tau(F^\circ) \sqsubseteq_\tau \overline{F^\circ}(\alpha_\tau)$ as proved by lemma 3.4.26

Then, by theorem 2.2.5, $\forall \overline{\sigma} \in \overline{\text{St}}_{pre} : \alpha_\tau(\mathbb{S}^\circ) \llbracket \bar{f}, \bar{r}, t \rrbracket \sqsubseteq_\tau \overline{\mathbb{S}^\circ} \llbracket \bar{f}, \bar{r}, t \rrbracket$. ■

LEMMA 3.4.28 $\forall \Phi \in \wp(\Psi) : \alpha_f(F^\parallel(\Phi)) \sqsubseteq \overline{F^\parallel}(\alpha_f(\Phi))$

PROOF.

$$\begin{aligned}
\alpha_f(F^\parallel(\Phi)) &= \text{(by definition of } F^\parallel) \\
&\alpha_f(\lambda \Phi. \{(f_0, r_0)\} \cup \{(f_i, r_{i-1}) : \exists (f_{i-1}, r_{i-1}) \in \Phi : \\
&\quad \forall t \in \text{dom}(f_{i-1}) : \tau \in \mathbb{S}^\circ \llbracket f_{i-1}, r_{i-1}, t \rrbracket, \tau \in \text{St}_{\rightarrow}^{\ddagger}, f_i(t) = \tau\}) \\
&= \text{(by functional lifting)} \\
&\lambda \alpha_f(\Phi). \alpha_f(\{(f_0, r_0)\} \cup \{(f_i, r_{i-1}) : \exists (f_{i-1}, r_{i-1}) \in \Phi : \\
&\quad \forall t \in \text{dom}(f_{i-1}) : \tau \in \mathbb{S}^\circ \llbracket f_{i-1}, r_{i-1}, t \rrbracket, \tau \in \text{St}_{\rightarrow}^{\ddagger}, f_i(t) = \tau\}) \\
&= \text{(as } \alpha_f \text{ is a join-morphism by lemma 3.4.16)} \\
&\lambda \alpha_f(\Phi). \alpha_f(\{(f_0, r_0)\}) \sqcup_f \alpha_f(\{(f_i, r_{i-1}) : \exists (f_{i-1}, r_{i-1}) \in \Phi : \\
&\quad \forall t \in \text{dom}(f_{i-1}) : \tau \in \mathbb{S}^\circ \llbracket f_{i-1}, r_{i-1}, t \rrbracket, \tau \in \text{St}_{\rightarrow}^{\ddagger}, f_i(t) = \tau\}) \\
&= \text{(by definition of } \alpha_f) \\
&\lambda \alpha_f(\Phi). \alpha'_f(f_0, r_0) \sqcup_f \{\alpha'_f(f_i, r_{i-1}) : \exists (f_{i-1}, r_{i-1}) \in \Phi : \\
&\quad \forall t \in \text{dom}(f_{i-1}) : \tau \in \mathbb{S}^\circ \llbracket f_{i-1}, r_{i-1}, t \rrbracket, \tau \in \text{St}_{\rightarrow}^{\ddagger}, f_i(t) = \tau\} \\
&\sqsubseteq_f \text{(by lemma 3.4.27)} \\
&\lambda \alpha_f(\Phi). \alpha'_f(f_0, r_0) \sqcup_f \{(\bar{f}_i, \bar{r}_{i-1}) : \exists (\bar{f}_{i-1}, \bar{r}_{i-1}) = \alpha_f(\Phi) : \\
&\quad \forall t \in \text{dom}(\bar{f}_{i-1}) : \bar{f}_i(t) = \overline{\mathbb{S}^\circ} \llbracket \bar{f}_{i-1}, \bar{r}_{i-1}, t \rrbracket\} \\
&= \text{(by definition of } \overline{F^\parallel}) \\
&\overline{F^\parallel}(\alpha_f(\Phi))
\end{aligned}$$

■

THEOREM 3.4.29 *Supposing that $\overline{\Psi}_{pre} \times \overline{\Omega}_{pre}$ is the set of all the prefixpoints of $\overline{F^\parallel} \times \overline{\Omega}$, then $\forall (\bar{f}, \bar{r}) \in \overline{\Psi}_{pre} \times \overline{\Omega}_{pre} : \alpha_f(\mathbb{S}^\parallel) \llbracket \bar{f}, \bar{r} \rrbracket \sqsubseteq_f \overline{\mathbb{S}^\parallel} \llbracket \bar{f}, \bar{r} \rrbracket$.*

PROOF. Note that:

- F^\parallel is monotonic for \subseteq as trivial consequence of its definition

- $\overline{F^{\parallel}}$ is monotonic for \sqsubseteq_f as trivial consequence of its definition
- $\langle \wp(\Psi), \subseteq, \cup \rangle$ is a complete lattice by properties of set union operator and subset ordering
- $\langle \overline{\Psi}, \sqsubseteq_f, \sqcup_f \rangle$ is a complete lattice as proved by lemma 3.4.12
- \bar{f} is a prefixpoint of $\overline{F^{\parallel}}$ by hypothesis
- α_f is a join-morphism as proved by lemma 3.4.16
- $\alpha_f(F^{\parallel}) \sqsubseteq_f \overline{F^{\parallel}}(\alpha_f)$ as proved by lemma 3.4.28

Then, by theorem 2.2.5, $\forall(\bar{f}, \bar{r}) \in \overline{\Psi}_{pre} \times \overline{\Omega}_{pre} : \alpha_f(\mathbb{S}^{\parallel})[\bar{f}, \bar{r}] \sqsubseteq_f \overline{\mathbb{S}^{\parallel}}[\bar{f}, \bar{r}]$. ■

3.4.8 Launching a Thread

The launch of a thread can be directly abstracted from the concrete definition presented in Section 3.3.5.

3.4.9 The Example

We analyze the example presented in Section 3.1.1 using the Interval domain in order to infer information about numerical values.

For the first iteration we obtain the same results as the concrete semantics, completely described in Section 3.3.6. The only difference is that now we deal with abstract values, and so we relate each numerical variable to an interval instead of an integer, and the concrete reference would be abstracted into an abstract one. Note that Chapter 5 will introduce an ad-hoc analysis in order to abstract references.

We analyze which abstract values are returned by the *visible* function when reading amount and account during the second iteration of the fixpoint computation. In particular, both the initial values (amount = [100..100] and account = #aba1) and the values written by Thread 1 (amount = [0..0] and account = null) are visible. The least upper bound of these elements returns amount = [0..100] and account = {#aba1, null}. Then we check the condition of Thread 2 `this.amount > 0 && this.account == null` may be evaluated to true, and we conclude that method `isValid()` when executed by Thread 2 may return false. This result is sound with respect to the concrete semantics, and so to the happens-before memory model.

3.5 Related work

Many approaches have been developed in order to statically analyze multithreaded programs. Most of them deal with deadlock and data race detection [121]. In the last few years other approaches, analyzing other and more generic properties, have been proposed

[125, 20, 141, 41]. Usually these approaches suppose that the execution is sequentially consistent, but this assumption is not legal under, for instance, the Java Memory Model. On the other hand, many definitions and some static analyses with respect to a memory model have been proposed.

Roychoudhury and Mitra [122] present a semantics for Java multithreaded programs that respects on an earlier version of the Java Memory Model. In particular, it presents an executable semantics that is sound and complete with respect to the Java Memory Model, and it verifies programs on it using model checking techniques. It is specific to the Java programming language, and it is affected by the state space explosion problem.

In a similar way, Huynh et al. [71] develop a model checker for the .NET memory model [36]. It is specific to the C# language, and it suffers from the state explosion problem.

Cenciarelli, Knapp and Sibilio [19] propose a formalization of the Java Memory Model through a semantics that combines operational, denotational, and axiomatic approaches. The authors build up a subset of the legal executions under the Java Memory Model. This approach is different from ours, since we compute a superset of these executions. However this approach is specific to the Java programming language, and it does not introduce any static analysis.

Boudol and Petri [17] propose an operational approach to the definition of a relaxed memory model. In particular, this work is focused on the formalization of the behaviors of buffers and how communications through shared memory use them.

Steinke and Nutt [133] propose an unifying approach to the definition of consistency rules on multithreaded executions. All consistency rules previously proposed can be defined in this framework. We think that this generality may be achieved also in our framework tuning on different memory models the *vis* function.

Rakamaric and Hu [116] propose a new memory model for low-level code. In particular, previous memory models require some checks before each memory access at runtime. This approach is not scalable. So the authors propose a static analysis to validate some properties of interest. So many runtime checks can be safely removed, and the resulted memory model is scalable. On the other hand, this static analysis is focused on some particular properties, and so it is not a generic static analysis sound with respect to a memory model.

Given this context, our work appears to be

- the first definition in a fixpoint form of a memory model,
- the first static analysis sound with respect to the happens-before memory model,
- the first static analysis of a memory model based on abstract interpretation theory,
- the first work that combines together a generic definition of a memory model and its static analysis.

3.6 Discussion

In this chapter we presented a whole-program analysis of multithreaded programs. Our analysis is as generic as possible with respect to the programming language. It requires only that some functions and a small step semantic atomic at multithreaded level are provided. In addition, it is generic with respect to the abstract numerical domain, and the property of interest. In order to apply it to a real programming language, and in particular Java, we need to study in the details some issues related with this approach.

3.6.1 Thread Identifiers

In Java threads are objects. In order to create and launch a thread, developers have to create a class that extends `java.lang.Thread`, override the method `run()` with the code that has to be executed in parallel, instantiate an object of this class, and then invoke the method `start()` on it. In this context, threads are objects. In the Java virtual machine objects are stored in the heap, and thus they are identified by reference.

In our approach, we considered that the set of thread identifiers is the same in our concrete and abstract domain. This means that the set of active threads is statically determined at the beginning of the computation, and it is finite. This is not the case when analyzing Java programs, as in the concrete context we may have an unbounded allocation of references, and we need to approximate it in the abstract. Chapter 5 will present an ad-hoc may-alias analysis in order to soundly abstract concrete references.

3.6.2 Monitors

In a similar way, Java bytecode has two primitives (`monitorenter` and `monitorexit`) in order to lock and unlock monitors. Each object is related to a monitor: in order to manage it, the Java Virtual Machine uses the reference that identifies the object.

In our approach, the abstract semantics supposes that if two threads own a common abstract synchronizable element then they are synchronized in all the possible executions. As in Java monitors are identified by reference, we need to develop a must-alias analysis. Chapter 5 will present it.

3.6.3 Modular Analysis of Multithreaded Programs

When developing programs, we often want to reason modularly, i.e. to partition an application into different tasks, reason on and develop separately each of them, and finally compose them through some well-defined interfaces. Modularity is a key objective of many programming languages, and indeed it is one of the main motivations of object oriented languages [100]. One of the consequences of modular programming is that a practical analysis must fit the toolchain the developer uses, that is, an analysis for Java must be modular. For instance, Logozzo [90] analyzes each class separately from the others, inferring properties that are valid in all possible environments of execution.

However, modular reasoning on multithreaded programs is not possible. For instance, when we develop the code of a thread we cannot specify some constraints on how other threads would access shared data in parallel, and they may read and write them freely. Comparing multithreading to object oriented languages, while in the second case we have `public`, `protected`, and `private` fields, and through these modifiers we may avoid other classes to access them, in multithreading there is no way to locally forbid parallel accesses or put some constraints on them. In addition in object oriented programs other constraints may be specified at runtime, for instance using contracts. Instead, usually there is no way to check at runtime what other threads are doing (e.g. if they have written a shared variables without owning a given monitor).

Some extensions of contracts to multithreaded programs have been proposed recently [108, 87]. On one hand, the authors propose some restrictions (e.g. through ownership types) on how threads interacts to avoid data races and deadlocks [107]. On the other hand, they propose some automatic synchronization actions, called wait-semantics, to enforce assertions, class invariants, pre- and post-conditions also in multithreaded programs. Finally, monitor invariants hold when no thread is executing within the monitor, that is, between acquire and release operations [11]. Our approach is slightly different with respect to this one, as we want to

- analyze all Java multithreaded programs, and not to restrict programs in order to avoid data races and deadlocks,
- check properties for all the possible executions without requiring additional automatic synchronizations.

In this context, we had to develop a whole-program analysis. On the other hand, it seems evident that something more is required semantically in order to put developers in position to modularly think about threads. For instance, a different approach has been adopted by the Software Transactional Memory [127]. However, STM has not been adopted by common programming languages (e.g. Java and C#) which, so far, offer threads rather than transactions.

4

Determinism of Multithreaded Programs

In this chapter, we will define and abstract a deterministic property focused on multi-threaded programs. Our property is aimed at checking the nondeterministic behavior that is due to the arbitrary interleaving during the execution of different threads. We define it as difference among concrete traces. Then we abstract it in two separate steps in order to statically analyze it. At the intermediate level of abstraction, we propose the new idea of weak determinism. We discuss how nondeterminism may flow. We relax the deterministic property projecting it on a part of traces and states. We introduce how data races and SQL phenomena affect the determinism of multithreaded programs. Finally, we sketch how the proposed property may be used in order to semi-automatically parallelize sequential programs.

This chapter is based on previously published work [44].

4.1 Analyzing Multithreaded Programs

A topic thoroughly studied has been how parallel threads can communicate in order to limit nondeterministic behavior. In this context, many different approaches have been developed.

4.1.1 Data Races

The first way is the static or dynamic checking on threads' actions like general and data races [106]. When two threads access the shared memory, and at least one performs a write operation, they form a general race. The data race requires also that the concurrent accesses are not synchronized. The idea is that races cause nondeterminism and that are a symptom of bugs. The efforts in the static analysis of races have been huge: Rinard [121] has presented a complete overview until some years ago. Many other approaches have been proposed during the last few years, e.g. [1, 21, 41, 49, 61, 63, 81, 104, 137, 141]. Nevertheless, it seems that the data race condition is not expressive and flexible enough. In fact, the absence of general races is a too restrictive condition for multithreaded programs

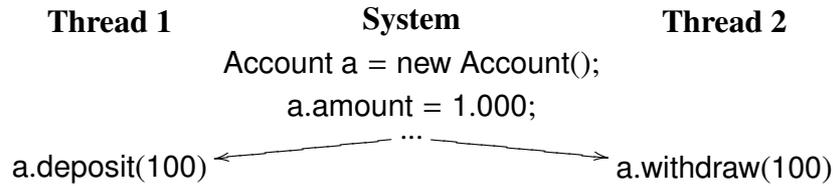


Figure 4.1: Depositing and withdrawing in parallel

that communicate asynchronously through shared memory. On the other hand, avoiding data races does not guarantee the determinism of a program.

In addition, a race does not take into account

- which statements and threads cause it,
- which area of the shared memory it deals with,
- if the informations written in parallel are different and how much.

Our idea is to directly study determinism. The aim is to relax the deterministic property on a critical subset of the shared memory, considering only some statements and threads, looking also at the (abstract) values written and read in parallel. Our approach is thus more flexible than general and data races.

4.1.2 Model of Execution

Another large amount of work has focused on consistency conditions that multithreaded programs have to provide. The best known model is sequential consistency [83], that has been shown to be too restrictive for modern programming languages. For instance, the Java memory model [95] is more relaxed than it. Our approach is orthogonal with respect to these consistency conditions that define which states of shared memory are visible during the execution of different threads.

A similar approach is the definition of a specific programming model that restricts the non-determinism of multithreaded programs by allowing certain interactions between threads. The Software Transactional Memory (STM) [127, 60] allows to specify that the execution of a given part of a program is atomic, i.e. it is viewed as a unique operation by other threads. A large work on the semantics of parallel languages has been developed by trace theory [98], e.g. [51]. In this case, we distinguish the model of execution from the determinism of a program. Our deterministic property may be applied to analyses sound with respect to sequential consistency, happens-before memory model, Java memory model, software transactional memory, etc..

4.1.3 An Example

In order to illustrate the concepts presented throughout this chapter, we will always refer to the following example.

Figure 4.1 depicts a multithreaded execution that uses the class `Account` introduced in Section 2.3. The system is going to execute in parallel a deposit of 100 and a withdrawal of the same amount of money. At the beginning, the value contained in account is 1.000.

4.2 Syntax and Concrete Semantics

In this section, we present

- a syntax focused on write and read operations on shared memory,
- the concrete domain and semantics. They are augmented in order to trace, for each value, the thread that wrote it in the shared memory.

4.2.1 Syntax

For the sake of readability, we consider a very restricted syntax. It is focused on the interactions with the shared memory, i.e. read and write actions. In this way we consider only statements of the form `sh_var = value` and `read(sh_var)`.

Statement `sh_var = value` writes on a shared variable `sh_var` a given value. `read(sh_var)` reads a shared variable `sh_var` and returns its value. Note that this syntax can be easily extended with other statements (if, while, etc..).

4.2.2 Concrete Domain

The concrete domain we consider is strictly focused on the shared memory. For each value, we trace the thread that wrote it. As we are in the concrete context, we know exactly which thread wrote a value at each point of the computation. Thus, we relate each shared variable to a pair composed of its value and an identifier of the thread that has written the values.

Formally, let TId be the set of the identifiers of threads, Var be the set of shared variables, and V be the set of concrete values. The shared memory is a function that relates each variable to a pair composed by its value and the identifier of the thread that wrote it ($S : [Var \rightarrow (V \times TId)]$). Note that this representation of shared memory is quite different with respect to the approach usually adopted by current programming languages. For instance, in Java the shared memory is the heap, and values are accessed by reference. At the static level this approach would lead to multiple issues that are orthogonal with respect to our goal, e.g. alias analysis. These will be considered in Chapter 5.

The concrete domain is the thread-partitioning one introduced in Section 3.3.2. In particular, the states are the shared memories ($\Psi : [TId \rightarrow S^\ddagger]$).

4.2.3 Transfer Function

In order to define the concrete semantics of write actions on shared memory, we introduce $\mathbb{W} : [(S \times \text{TId}) \rightarrow S]$ applied to $\text{sh_var} = \text{value}$. It is defined as $\mathbb{W}[\text{sh_var} = \text{value}](s, t) = s[\text{sh_var} \mapsto (\text{value}, t)]$.

The value read from a shared memory s is given by the function $\mathbb{R} : [S \rightarrow V]$ applied to $\text{eval}(\text{sh_var})$. It is defined as $\mathbb{R}[\text{eval}(\text{sh_var})](s) = \pi_1(s(\text{sh_var}))$, where π_1 is the projection on the first component.

4.2.4 An Example

<p>Thread 1 : [a.amount \mapsto (1.000, System)] \rightarrow [a.amount \mapsto (1.100, Thread 1)]</p> <p>Thread 2 : [a.amount \mapsto (1.100, Thread 1)] \rightarrow [a.amount \mapsto (1.000, Thread 2)]</p>
<p>Thread 1 : [a.amount \mapsto (900, Thread 2)] \rightarrow [a.amount \mapsto (1.000, Thread 1)]</p> <p>Thread 2 : [a.amount \mapsto (1.000, System)] \rightarrow [a.amount \mapsto (900, Thread 2)]</p>

Figure 4.2: The concrete semantics

Figure 4.2 presents the two multithreaded concrete executions obtained using the domain and the semantics just introduced. The first element represents the case in which Thread 1 is executed before Thread 2. The second element depicts the opposite situation. Note that other executions are not possible as both the methods are synchronized on the same monitor.

4.3 A Value for Each Thread (Abstraction 1)

In this section, we present the first level of abstraction where each shared variable is related to an abstract value for each thread that may write on that. Our analysis is parameterized by the abstract non-relational domain that approximates numerical values.

4.3.1 Abstract Domain (First Level)

We consider the following Galois connection between the concrete domain of values and its abstract counterpart.

$$\langle \wp(V), \subseteq, \emptyset, \vee, \cup, \cap \rangle \xleftrightarrow[\alpha_V]{\gamma_V} \langle \widehat{V}, \sqsubseteq_{\widehat{V}}, \perp_{\widehat{V}}, \top_{\widehat{V}}, \sqcup_{\widehat{V}}, \sqcap_{\widehat{V}} \rangle$$

As we work pointwisely on values, this means that the non-relational abstract domain has to soundly approximate the concrete values. In the concrete context, different executions

may contain values written by different threads because of their arbitrary interleavings. The first level of abstraction approximates all the concrete executions with a unique abstract trace. For each shared variable it gathers all the values written by the same thread on the same shared variable. So it collects an abstract value for each thread.

Formally, a shared memory \widehat{S} relates each variable to a function that maps a thread to the abstraction of the values that it may have written, i.e. $\widehat{S} : [\text{Var} \rightarrow [\text{TId} \rightarrow \widehat{V}]]$.

As in the concrete semantics, we adopt the thread-partitioning trace domain ($\widehat{\Psi} : [\text{TId} \rightarrow \widehat{S}^+]$).

4.3.2 Upper Bound Operator

The upper bound operator on shared memories keeps all the values written by different threads. It relies on the join operator of the abstract numerical domain.

$$\widehat{s}_1 \sqcup_{\widehat{S}} \widehat{s}_2 = \widehat{s} : \forall \text{var} \in \text{dom}(\widehat{s}_1) \cup \text{dom}(\widehat{s}_2), \forall t \in \text{dom}(\widehat{s}_1(\text{var})) \cup \text{dom}(\widehat{s}_2(\text{var})),$$

$$\widehat{s}(\text{var})(t) = \begin{cases} \widehat{s}_1(\text{var})(t) & \text{if } \text{var} \notin \text{dom}(\widehat{s}_2) \vee t \notin \text{dom}(\widehat{s}_2(\text{var})) \\ \widehat{s}_2(\text{var})(t) & \text{if } \text{var} \notin \text{dom}(\widehat{s}_1) \vee t \notin \text{dom}(\widehat{s}_1(\text{var})) \\ \widehat{s}_1(\text{var})(t) \sqcup_{\widehat{V}} \widehat{s}_2(\text{var})(t) & \text{otherwise} \end{cases}$$

The upper bound operator on traces simply applies the upper bound operator of shared memories on all traces' states.

$$\widehat{\tau}_1 \sqcup_{\widehat{S}^+} \widehat{\tau}_2 = \widehat{\sigma}_0 \rightarrow \dots \rightarrow \widehat{\sigma}_i : i = \max(\text{len}(\widehat{\tau}_1), \text{len}(\widehat{\tau}_2)), \forall j \in [0..i] :$$

$$\widehat{\sigma}_j = \begin{cases} \widehat{\tau}_1(j) \sqcup_{\widehat{S}} \widehat{\tau}_2(j) & \text{if } j < \text{len}(\widehat{\tau}_1) \wedge j < \text{len}(\widehat{\tau}_2) \\ \widehat{\tau}_2(j) & \text{if } j \geq \text{len}(\widehat{\tau}_1) \\ \widehat{\tau}_1(j) & \text{if } j \geq \text{len}(\widehat{\tau}_2) \end{cases}$$

The upper bound operator on the multithreaded state is the pointwise extension of the upper bound of traces on all the elements of the codomain.

$$\widehat{f}_1 \sqcup_{\widehat{\Psi}} \widehat{f}_2 = \widehat{f} : \forall t \in \text{dom}(\widehat{f}_1) \cup \text{dom}(\widehat{f}_2) :$$

$$\widehat{f}(t) = \begin{cases} \widehat{f}_1(t) \sqcup_{\widehat{S}^+} \widehat{f}_2(t) & \text{if } t \in \text{dom}(\widehat{f}_1) \cap \text{dom}(\widehat{f}_2) \\ \widehat{f}_1(t) & \text{if } t \in \text{dom}(\widehat{f}_1) \wedge t \notin \text{dom}(\widehat{f}_2) \\ \widehat{f}_2(t) & \text{if } t \in \text{dom}(\widehat{f}_2) \wedge t \notin \text{dom}(\widehat{f}_1) \end{cases}$$

4.3.3 Abstraction Function

The abstraction function maps a set of concrete shared memories into an abstract memory.

$$\alpha_S : [\wp(S) \rightarrow \widehat{S}]$$

$$\alpha_S(S) = \widehat{f} : \forall \text{var} \in \bigcup_{s \in S} \text{dom}(s), \forall t \in \bigcup_{s \in S} \text{dom}(s(\text{var})) :$$

$$\widehat{f}(\text{var})(t) = \alpha_V(\{v : \exists s \in S : s(\text{var}) = (v, t)\})$$

The abstraction of a set of traces produces an abstract trace such that its i -th element is the abstraction of the i -th elements of all the given concrete traces.

$$\begin{aligned} \alpha_{S^\ddagger} &: [\wp(S^\ddagger) \rightarrow \widehat{S}^\ddagger] \\ \alpha_{S^\ddagger}(\mathbf{T}) &= \widehat{\sigma}_0 \rightarrow \cdots \rightarrow \widehat{\sigma}_i : i = \max(\bigcup_{\tau \in \mathbf{T}} \text{len}(\tau)), \\ &\quad \forall j \in [0..i] : \widehat{\sigma}_j = \alpha_S(\bigcup_{\tau \in \mathbf{T} : \text{len}(\tau) > j} \tau(j)) \end{aligned}$$

When considering traces, the abstraction function returns a unique function. This abstracts together all the traces produced by the same thread in the different concrete executions.

$$\begin{aligned} \alpha_\Psi &: [\wp(\Psi) \rightarrow \widehat{\Psi}] \\ \alpha_\Psi(\Phi) &= \widehat{f} : \forall t \in \bigcup_{f \in \Phi} \text{dom}(f) : \widehat{f}(t) = \alpha_{S^\ddagger}(\bigcup_{f \in \Phi : t \in \text{dom}(f)} f(t)) \end{aligned}$$

PROPOSITION 4.3.1 (α_V IS A JOIN PRESERVING MAP) *We suppose that α_V is a join preserving map, i.e. $\forall V_1, V_2 \subseteq V : \alpha_V(V_1 \cup V_2) = \alpha_V(V_1) \sqcup_V \alpha_V(V_2)$*

LEMMA 4.3.2 (α_S IS A JOIN PRESERVING MAP) *α_S is a join preserving map, i.e. $\forall S_1, S_2 \in S : \alpha_S(S_1 \cup S_2) = \alpha_S(S_1) \sqcup_S \alpha_S(S_2)$*

PROOF.

$$\begin{aligned} &\alpha_S(S_1 \cup S_2) \\ &\quad (\text{by definition of } \alpha_S) \\ &= \widehat{f} : \forall \text{var} \in \bigcup_{s \in S_1 \cup S_2} \text{dom}(s), \forall t \in \bigcup_{s \in S_1 \cup S_2} \text{dom}(s(\text{var})) : \\ &\quad \widehat{f}(\text{var})(t) = \alpha_V(\{v \in V : \exists s \in S_1 \cup S_2 : s(\text{var}) = (v, t)\}) \\ &\quad (\text{by basic set properties}) \\ &= \widehat{f} : \forall \text{var} \in \bigcup_{s \in S_1} \text{dom}(s) \cup \bigcup_{s \in S_2} \text{dom}(s), \\ &\quad \forall t \in \bigcup_{s \in S_1} \text{dom}(s(\text{var})) \cup \bigcup_{s \in S_2} \text{dom}(s(\text{var})) : \widehat{f}(\text{var})(t) = \\ &\quad = \alpha_V(\{v \in V : \exists s \in S_1 : s(\text{var}) = (v, t)\} \cup \{v \in V : \exists s \in S_2 : s(\text{var}) = (v, t)\}) \\ &\quad (\text{by proposition 4.3.1}) \\ &= \widehat{f} : \forall \text{var} \in \bigcup_{s \in S_1} \text{dom}(s) \cup \bigcup_{s \in S_2} \text{dom}(s), \\ &\quad \forall t \in \bigcup_{s \in S_1} \text{dom}(s(\text{var})) \cup \bigcup_{s \in S_2} \text{dom}(s(\text{var})) : \widehat{f}(\text{var})(t) = \\ &\quad = \alpha_V(\{v \in V : \exists s \in S_1 : s(\text{var}) = (v, t)\}) \sqcup_V \alpha_V(\{v \in V : \exists s \in S_2 : s(\text{var}) = (v, t)\}) \\ &\quad (\text{by definition of } \alpha_S \text{ the abstract result is defined on the union of all the domains} \\ &\quad (\text{of all the concrete elements})) \\ &= \widehat{f} : \forall \text{var} \in \text{dom}(\alpha_S(S_1)) \cup \text{dom}(\alpha_S(S_2)), \\ &\quad \forall t \in \text{dom}(\alpha_S(S_1)(\text{var})) \cup \text{dom}(\alpha_S(S_2)(\text{var})) : \\ &\quad \widehat{f}(\text{var})(t) = \alpha_V(\{v \in V : \exists s \in S_1 : s(\text{var}) = (v, t)\}) \sqcup_V \\ &\quad \alpha_V(\{v \in V : \exists s \in S_2 : s(\text{var}) = (v, t)\}) \end{aligned}$$

We reason by case:

- if $\text{var} \notin \text{dom}(\alpha_S(S_2)) \vee \nexists s \in S_2 : s(\text{var}) = (\text{val}, t)$, then $\widehat{f}(\text{var})(t) = \alpha_V(\{\text{val} : \exists s \in S_1 : s(\text{var}) = (\text{val}, t)\})$. By definition of α_S we have that $\alpha_V(\{\text{val} : \exists s \in S_1 :$

$s(\text{var}) = (\text{val}, t) \} = \alpha_S(S_1)(\text{var})(t)$ and that $\nexists s \in S_2 : s(\text{var}) = (\text{val}, t) \Rightarrow t \notin \text{dom}(\alpha_S(S_2)(\text{var}))$. Combining them, we obtain that

$$\text{var} \notin \text{dom}(\alpha_S(S_2)) \vee \nexists t \notin \text{dom}(\alpha_S(S_2)(\text{var})) \Rightarrow \widehat{f}(\text{var})(t) = \alpha_S(S_1)(\text{var})(t) \quad (4.3.1)$$

- in the same way but inverting S_1 and S_2 we obtain that

$$\text{var} \notin \text{dom}(\alpha_S(S_1)) \vee \nexists t \notin \text{dom}(\alpha_S(S_1)(\text{var})) \Rightarrow \widehat{f}(\text{var})(t) = \alpha_S(S_2)(\text{var})(t) \quad (4.3.2)$$

- otherwise, the value would be the least upper bound between the values contained in S_1 and S_2 , and so we obtain that

$$\widehat{f}(\text{var})(t) = \alpha_S(S_1)(\text{var})(t) \sqcup_{\widehat{S}} \alpha_S(S_2)(\text{var})(t) \quad (4.3.3)$$

Combining 4.3.1, 4.3.2, and 4.3.3 we obtain that

$$\widehat{f} : \forall \text{var} \in \text{dom}(\alpha_S(S_1)) \cup \text{dom}(\alpha_S(S_2)), \forall t \in \text{dom}(\alpha_S(S_1)(\text{var})) \cup \text{dom}(\alpha_S(S_2)(\text{var})),$$

$$\widehat{f}(\text{var})(t) = \begin{cases} \alpha_S(S_1)(\text{var})(t) & \text{if } \text{var} \notin \text{dom}(\alpha_S(S_2)) \vee \nexists t \notin \text{dom}(\alpha_S(S_2)(\text{var})) \\ \alpha_S(S_2)(\text{var})(t) & \text{if } \text{var} \notin \text{dom}(\alpha_S(S_1)) \vee \nexists t \notin \text{dom}(\alpha_S(S_1)(\text{var})) \\ \alpha_S(S_1)(\text{var})(t) \sqcup_{\widehat{S}} \alpha_S(S_2)(\text{var})(t) & \text{otherwise} \end{cases}$$

and so by definition of $\sqcup_{\widehat{S}}$ that $\widehat{f} = \alpha_S(S_1) \sqcup_{\widehat{S}} \alpha_S(S_2)$.

We proved that $\widehat{f} = \alpha_S(S_1 \cup S_2)$ and that $\widehat{f} = \alpha_S(S_1) \sqcup_{\widehat{S}} \alpha_S(S_2)$. So, by transitive property of equivalence, we proved that $\alpha_S(S_1 \cup S_2) = \alpha_S(S_1) \sqcup_{\widehat{S}} \alpha_S(S_2)$. ■

LEMMA 4.3.3 (α_{S^\ddagger} IS A JOIN PRESERVING MAP) α_{S^\ddagger} is a join preserving map, i.e. $\forall T_1, T_2 \in S^\ddagger : \alpha_{S^\ddagger}(T_1 \cup T_2) = \alpha_{S^\ddagger}(T_1) \sqcup_{\widehat{S}^\ddagger} \alpha_{S^\ddagger}(T_2)$

PROOF.

$$\begin{aligned} & \alpha_{S^\ddagger}(T_1 \cup T_2) \\ & \quad (\text{by definition of } \alpha_{S^\ddagger}) \\ & = \widehat{\sigma}_0 \rightarrow \cdots \rightarrow \widehat{\sigma}_i : i = \max(\bigcup_{\tau \in T_1 \cup T_2} \text{len}(\tau)), \forall j \in [0..i] : \\ & \quad \widehat{\sigma}_j = \alpha_S(\bigcup_{\tau \in T_1 \cup T_2 : \text{len}(\tau) > j} \tau(j)) \\ & \quad (\text{by basic set properties}) \\ & = \widehat{\sigma}_0 \rightarrow \cdots \rightarrow \widehat{\sigma}_i : i = \max(\bigcup_{\tau \in T_1} \text{len}(\tau), \bigcup_{\tau \in T_2} \text{len}(\tau)), \\ & \quad \forall j \in [0..i] : \widehat{\sigma}_j = \alpha_S(\bigcup_{\tau \in T_1 : \text{len}(\tau) > j} \tau(j) \cup \bigcup_{\tau \in T_2 : \text{len}(\tau) > j} \tau(j)) \\ & \quad (\text{by Lemma 4.3.2}) \\ & = \widehat{\sigma}_0 \rightarrow \cdots \rightarrow \widehat{\sigma}_i : i = \max(\bigcup_{\tau \in T_1} \text{len}(\tau), \bigcup_{\tau \in T_2} \text{len}(\tau)), \\ & \quad \forall j \in [0..i] : \widehat{\sigma}_j = \alpha_S(\bigcup_{\tau \in T_1 : \text{len}(\tau) > j} \tau(j)) \sqcup_{\widehat{S}} \alpha_S(\bigcup_{\tau \in T_2 : \text{len}(\tau) > j} \tau(j)) \\ & \quad (\text{by definition of } \sqcup_{\widehat{S}^\ddagger}) \\ & = \alpha_{S^\ddagger}(T_1) \sqcup_{\widehat{S}^\ddagger} \alpha_{S^\ddagger}(T_2) \end{aligned}$$
■

LEMMA 4.3.4 (α_Ψ IS A JOIN PRESERVING MAP) α_Ψ is a join preserving map, i.e. $\forall \Phi_1, \Phi_2 \in \Psi : \alpha_\Psi(\Phi_1 \cup \Phi_2) = \alpha_\Psi(\Phi_1) \sqcup_{\widehat{\Psi}} \alpha_\Psi(\Phi_2)$

PROOF.

$$\begin{aligned}
& \alpha_\Psi(\Phi_1 \cup \Phi_2) \\
& \quad (\text{by definition of } \alpha_\Psi) \\
& = \widehat{f} : \forall t \in \bigcup_{\Phi_1 \cup \Phi_2} \text{dom}(f) : \widehat{f}(t) = \alpha_{S^\#}(\bigcup_{f \in \Phi_1 \cup \Phi_2, t \in \text{dom}(f)} f(t)) \\
& \quad (\text{by basic set properties}) \\
& = \widehat{f} : \forall t \in \bigcup_{\Phi_1 \cup \Phi_2} \text{dom}(f) : \widehat{f}(t) = \alpha_{S^\#}(\bigcup_{f \in \Phi_1, t \in \text{dom}(f)} f(t) \cup \bigcup_{f \in \Phi_2, t \in \text{dom}(f)} f(t)) \\
& \quad (\text{by Lemma 4.3.3}) \\
& = \widehat{f} : \forall t \in \bigcup_{\Phi_1 \cup \Phi_2} \text{dom}(f) : \widehat{f}(t) = \alpha_{S^\#}(\bigcup_{f \in \Phi_1, t \in \text{dom}(f)} f(t)) \sqcup_{S^\#} \alpha_{S^\#}(\bigcup_{f \in \Phi_2, t \in \text{dom}(f)} f(t)) \\
& \quad (\text{by definition of } \sqcup_{S^\#}) \\
& = \alpha_\Psi(\Phi_1) \sqcup_{\widehat{\Psi}} \alpha_\Psi(\Phi_2)
\end{aligned}$$

■

THEOREM 4.3.5 Let $\leq_{\widehat{\Psi}}$ be the partial order induced by $\sqcup_{\widehat{\Psi}}$, and γ_Ψ be defined as

$$\gamma_\Psi = \lambda \widehat{y}. \bigcup \{z : \alpha_\Psi(z) \sqsubseteq_{\widehat{\Psi}} \widehat{y}\}$$

Then

$$\langle \wp(\Psi), \subseteq, \emptyset, \Psi, \cup, \cap \rangle \xrightleftharpoons[\alpha_\Psi]{\gamma_\Psi} \langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}}, \perp_{\widehat{\Psi}}, \top_{\widehat{\Psi}}, \sqcup_{\widehat{\Psi}}, \sqcap_{\widehat{\Psi}} \rangle$$

PROOF. Lemma 4.3.4 proved that α_Ψ is a join preserving map.

Then by Theorem 2.2.2 we proved that

$$\langle \wp(\Psi), \subseteq, \emptyset, \Psi, \cup, \cap \rangle \xrightleftharpoons[\alpha_\Psi]{\gamma_\Psi} \langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}}, \perp_{\widehat{\Psi}}, \top_{\widehat{\Psi}}, \sqcup_{\widehat{\Psi}}, \sqcap_{\widehat{\Psi}} \rangle$$

■

4.3.4 Transfer Function

In correspondence to the semantic functions \mathbb{W} and \mathbb{R} , we introduce their abstract counterpart $\widehat{\mathbb{W}}$ and $\widehat{\mathbb{R}}$ as follows.

$\widehat{\mathbb{W}} : [(\widehat{S} \times \text{Tld}) \rightarrow \widehat{S}]$ of `sh_var = value` is defined as

$$\widehat{\mathbb{W}}[\text{sh_var} = \text{value}](\widehat{s}, t) = \widehat{s}[\text{sh_var} \mapsto [t \mapsto \alpha_V(\text{value})]]$$

$\widehat{\mathbb{R}} : [\widehat{S} \rightarrow \widehat{V}]$ of `eval(sh_var)` is defined as

$$\widehat{\mathbb{R}}[\text{eval}(\text{sh_var})](\widehat{s}) = \bigsqcup_{t \in \text{dom}(\widehat{s}(\text{sh_var}))} \widehat{s}(\text{sh_var})(t)$$

LEMMA 4.3.6 (SOUNDNESS OF $\widehat{\mathbb{W}}$) $\widehat{\mathbb{W}}$ is the abstraction of \mathbb{W} , i.e.

$$\forall s \in S, \forall t \in \text{Tld} : \alpha_S(\mathbb{W}[\text{sh_var} = \text{value}](s, t)) = \widehat{\mathbb{W}}[\text{sh_var} = \text{value}](\alpha_S(\{s\}), t)$$

PROOF.

$$\begin{aligned}
& \alpha_S(\mathbb{W}[\text{sh_var} = \text{value}]({s}, t)) \\
& \quad (\text{by definition of } \mathbb{W}[\text{sh_var} = \text{value}]) \\
& = \alpha_S(\{s[\text{sh_var} \mapsto (\text{value}, t)]\}) \\
& \quad (\text{by definition of } \alpha_S) \\
& = \alpha_S(\{s\})[\text{sh_var} \mapsto (\alpha_V(\text{value}), t)] \\
& \quad (\text{by definition of } \widehat{\mathbb{W}}[\text{sh_var} = \text{value}]) \\
& = \widehat{\mathbb{W}}[\text{sh_var} = \text{value}](\alpha_S(\{s\}), t)
\end{aligned}$$

■

LEMMA 4.3.7 (SOUNDNESS OF $\widehat{\mathbb{R}}$) $\widehat{\mathbb{R}}$ is the abstraction of \mathbb{R} , i.e.

$$\forall s \in S : \alpha_V(\{\mathbb{R}[\text{eval}(\text{sh_var})](s)\}) = \widehat{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_S(\{s\}))$$

PROOF.

$$\begin{aligned}
& \alpha_V(\{\mathbb{R}[\text{eval}(\text{sh_var})](s)\}) \\
& \quad (\text{by definition of } \{\mathbb{R}[\text{eval}(\text{sh_var})]\}) \\
& = \alpha_V(\{\pi_1(s(\text{sh_var}))\}) \\
& \quad (\text{by definition of } \alpha_S) \\
& = \pi_1(\alpha_S(\{s\})(\text{sh_var})) \\
& \quad (\text{by definition of } \widehat{\mathbb{R}}[\text{eval}(\text{sh_var})]) \\
& = \widehat{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_S(\{s\}))
\end{aligned}$$

■

4.3.5 The Example

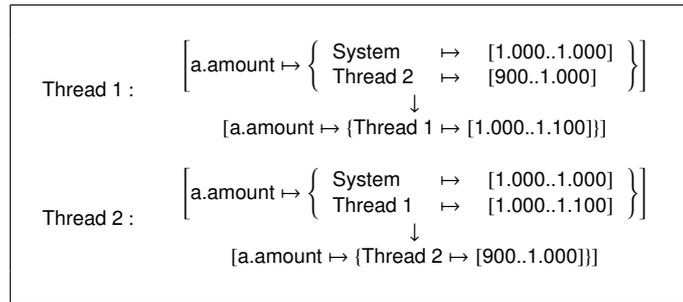


Figure 4.3: The abstract semantics

Figure 4.3 depicts the results of the abstract semantics. In order to capture numerical information we use the Interval domain.

From these results, we discover that the value read from the field `a.amount` by Thread 1 may have been written by System or Thread 2. The value read by Thread 2 may have been written by System or Thread 1. This makes evidence that some nondeterministic behaviors due to arbitrary interleaving of threads arise.

4.4 Just one Value (Abstraction 2)

In this section we present the second level of abstraction: all values written by different threads collapse into the same abstract element.

4.4.1 Abstract Domain (Second Level)

A shared memory $\bar{s} \in \bar{\mathcal{S}}$ relates each variable var to a pair composed by a value and the set of threads that may have written it. Formally $\bar{\mathcal{S}} : [\text{Var} \rightarrow (\widehat{\mathcal{V}} \times \wp(\text{TId}))]$.

Then our abstract domain is represented as a function in $\bar{\Psi} : [\text{TId} \rightarrow \bar{\mathcal{S}}^{\mp}]$.

4.4.2 Upper Bound Operator

The upper bound operator on shared memories produces for each variable a pair composed by

- the upper bound between the abstract numerical values assigner to the given variable,
- and the set union of all the threads that may have written on it.

$$\bar{s}_1 \sqcup_{\bar{\mathcal{S}}} \bar{s}_2 = \bar{s} : \forall \text{var} \in \text{dom}(\bar{s}_1) \cup \text{dom}(\bar{s}_2) :$$

$$\bar{s}(\text{var}) = \begin{cases} (\pi_1(\bar{s}_1(\text{var})) \sqcup_{\widehat{\mathcal{V}}} \pi_1(\bar{s}_2(\text{var})), \pi_2(\bar{s}_1(\text{var})) \cup \pi_2(\bar{s}_2(\text{var}))) & \text{if } \text{var} \in \text{dom}(\bar{s}_1) \cap \text{dom}(\bar{s}_2) \\ \bar{s}_1(\text{var}) & \text{if } \text{var} \in \text{dom}(\bar{s}_1) \wedge \text{var} \notin \text{dom}(\bar{s}_2) \\ \bar{s}_2(\text{var}) & \text{if } \text{var} \in \text{dom}(\bar{s}_2) \wedge \text{var} \notin \text{dom}(\bar{s}_1) \end{cases}$$

The upper bound operator on traces is the pointwise application of the upper bound operator of shared memories.

$$\bar{\tau}_1 \sqcup_{\bar{\mathcal{S}}^{\mp}} \bar{\tau}_2 = \bar{\sigma}_0 \rightarrow \dots \rightarrow \bar{\sigma}_i : i = \max(\text{len}(\bar{\tau}_1), \text{len}(\bar{\tau}_2)),$$

$$\forall j \in [0..i] : \bar{\sigma}_j = \begin{cases} \bar{\tau}_1(j) \sqcup_{\bar{\mathcal{S}}} \bar{\tau}_2(j) & \text{if } j < \text{len}(\bar{\tau}_1) \wedge j < \text{len}(\bar{\tau}_2) \\ \bar{\tau}_1(j) & \text{if } j < \text{len}(\bar{\tau}_1) \wedge j \geq \text{len}(\bar{\tau}_2) \\ \bar{\tau}_2(j) & \text{if } j < \text{len}(\bar{\tau}_2) \wedge j \geq \text{len}(\bar{\tau}_1) \end{cases}$$

The upper bound operator on the multithreaded state is the pointwise application of the upper bound of traces on all the elements of the codomain.

$$\bar{f}_1 \sqcup_{\bar{\Psi}} \bar{f}_2 = \bar{f} : \forall t \in \text{dom}(\bar{f}_1) \cup \text{dom}(\bar{f}_2) :$$

$$\bar{f}(t) = \begin{cases} \bar{f}_1(t) \sqcup_{\bar{\mathcal{S}}^{\mp}} \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_1) \cap \text{dom}(\bar{f}_2) \\ \bar{f}_1(t) & \text{if } t \in \text{dom}(\bar{f}_1) \wedge t \notin \text{dom}(\bar{f}_2) \\ \bar{f}_2(t) & \text{if } t \in \text{dom}(\bar{f}_2) \wedge t \notin \text{dom}(\bar{f}_1) \end{cases}$$

4.4.3 Abstraction Function

For each variable the abstraction of shared memories takes the join all the values written by different threads.

$$\begin{aligned} \alpha_{\widehat{\mathcal{S}}} &: [\widehat{\mathcal{S}} \rightarrow \overline{\mathcal{S}}] \\ \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}) &= \overline{\mathcal{S}} : \forall \text{var} \in \text{dom}(\widehat{\mathcal{S}}) : \overline{\mathcal{S}}(\text{var}) = (\bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}(\text{var}))} \widehat{\mathcal{S}}(t), \text{dom}(\widehat{\mathcal{S}}(\text{var}))) \end{aligned}$$

The abstraction of traces is the pointwise application of the abstraction of the shared memory.

$$\begin{aligned} \alpha_{\widehat{\mathcal{S}}^\ddagger} &: [\widehat{\mathcal{S}}^\ddagger \rightarrow \overline{\mathcal{S}}^\ddagger] \\ \alpha_{\widehat{\mathcal{S}}^\ddagger}(\widehat{\sigma}_0 \rightarrow \dots \rightarrow \widehat{\sigma}_i) &= \alpha_{\widehat{\mathcal{S}}}(\widehat{\sigma}_0) \rightarrow \dots \rightarrow \alpha_{\widehat{\mathcal{S}}}(\widehat{\sigma}_i) \end{aligned}$$

The abstraction on the multithreaded state is the pointwise application of the abstraction of traces.

$$\begin{aligned} \alpha_{\widehat{\Psi}} &: [\widehat{\Psi} \rightarrow \overline{\Psi}] \\ \alpha_{\widehat{\Psi}}(\widehat{f}) &= \overline{f} : \forall t \in \text{dom}(\widehat{f}) : \overline{f}(t) = \alpha_{\widehat{\mathcal{S}}^\ddagger}(\widehat{f}(t)) \end{aligned}$$

LEMMA 4.4.1 ($\alpha_{\widehat{\mathcal{S}}}$ IS A JOIN PRESERVING MAP) $\alpha_{\widehat{\mathcal{S}}}$ is a join preserving map, i.e. $\forall \widehat{\mathcal{S}}_1, \widehat{\mathcal{S}}_2 \in \widehat{\mathcal{S}} : \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2) = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_1) \sqcup_{\overline{\mathcal{S}}} \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_2)$

PROOF.

$$\begin{aligned} &\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2) \\ &\quad (\text{by definition of } \alpha_{\widehat{\mathcal{S}}}) \\ &= \overline{\mathcal{S}} : \forall \text{var} \in \text{dom}(\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2) : \\ &\quad \overline{\mathcal{S}}(\text{var}) = (\bigsqcup_{t \in \text{dom}((\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2)(\text{var}))} \widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2(t), \text{dom}((\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2)(\text{var}))) \\ &\quad (\text{by definition of } \sqcup_{\widehat{\mathcal{S}}} \text{ we have that }) \\ &\quad (\text{dom}(\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2) = \text{dom}(\widehat{\mathcal{S}}_1) \cup \text{dom}(\widehat{\mathcal{S}}_2) \text{ and } \forall \text{var} \in \text{dom}(\widehat{\mathcal{S}}_1) \cup \text{dom}(\widehat{\mathcal{S}}_2) :) \\ &\quad (\text{dom}((\widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2)(\text{var})) = \text{dom}(\widehat{\mathcal{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathcal{S}}_2(\text{var}))) \\ &= \overline{\mathcal{S}} : \forall \text{var} \in \text{dom}(\widehat{\mathcal{S}}_1) \cup \text{dom}(\widehat{\mathcal{S}}_2) : \\ &\quad \overline{\mathcal{S}}(\text{var}) = (\bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathcal{S}}_2(\text{var}))} \widehat{\mathcal{S}}_1 \sqcup_{\widehat{\mathcal{S}}} \widehat{\mathcal{S}}_2(t), \text{dom}(\widehat{\mathcal{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathcal{S}}_2(\text{var}))) \end{aligned}$$

We reason by case:

- if $\text{var} \notin \text{dom}(\widehat{\mathcal{S}}_2)$ we have that $\overline{\mathcal{S}}(\text{var}) = (\bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}_1(\text{var}))} \widehat{\mathcal{S}}_1(t), \text{dom}(\widehat{\mathcal{S}}_1(\text{var})))$. By definition $\alpha_{\widehat{\mathcal{S}}}$ we have that $(\bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}_1(\text{var}))} \widehat{\mathcal{S}}_1(t), \text{dom}(\widehat{\mathcal{S}}_1(\text{var}))) = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_1)$, and, by transitive property of the equivalence operator, $\overline{\mathcal{S}}(\text{var}) = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_1)$. So we obtain that

$$\text{var} \notin \text{dom}(\widehat{\mathcal{S}}_2) \Rightarrow \overline{\mathcal{S}}(\text{var}) = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_1) \quad (4.4.1)$$

- in the same way but inverting $\widehat{\mathcal{S}}_1$ and $\widehat{\mathcal{S}}_2$ we obtain that

$$\text{var} \notin \text{dom}(\widehat{\mathcal{S}}_1) \Rightarrow \overline{\mathcal{S}}(\text{var}) = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}_2) \quad (4.4.2)$$

- otherwise, we have that

$$\begin{aligned}\bar{s}(\text{var}) &= (\bigsqcup_{t \in \text{dom}(\widehat{\mathbf{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathbf{S}}_2(\text{var}))} \widehat{\mathbf{S}}_1 \sqcup_{\widehat{\mathbf{S}}} \widehat{\mathbf{S}}_2(t), \text{dom}(\widehat{\mathbf{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathbf{S}}_2(\text{var}))) \\ &= (\bigsqcup_{t \in \text{dom}(\widehat{\mathbf{S}}_1(\text{var}))} \widehat{\mathbf{S}}_1(t) \sqcup_{\widehat{\mathbf{V}}} \bigsqcup_{t \in \text{dom}(\widehat{\mathbf{S}}_2(\text{var}))} \widehat{\mathbf{S}}_2(t), \text{dom}(\widehat{\mathbf{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathbf{S}}_2(\text{var})))\end{aligned}$$

by definition of $\sqcup_{\widehat{\mathbf{S}}}$. By definition of $\alpha_{\widehat{\mathbf{S}}}$ we have that

$$\begin{aligned}(\bigsqcup_{t \in \text{dom}(\widehat{\mathbf{S}}_1(\text{var}))} \widehat{\mathbf{S}}_1(t) \sqcup_{\widehat{\mathbf{V}}} \bigsqcup_{t \in \text{dom}(\widehat{\mathbf{S}}_2(\text{var}))} \widehat{\mathbf{S}}_2(t), \text{dom}(\widehat{\mathbf{S}}_1(\text{var})) \cup \text{dom}(\widehat{\mathbf{S}}_2(\text{var}))) &= \\ = (\pi_1(\alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1)(\text{var})) \sqcup_{\widehat{\mathbf{V}}} \pi_1(\alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2)(\text{var})), \pi_2(\widehat{\mathbf{S}}_1(\text{var})) \cup \pi_2(\widehat{\mathbf{S}}_2(\text{var})))\end{aligned}$$

So we obtain that

$$\begin{aligned}\text{var} &\in \text{dom}(\widehat{\mathbf{S}}_1) \cap \text{dom}(\widehat{\mathbf{S}}_2) \\ &\quad \downarrow \\ \bar{s}(\text{var}) &= (\pi_1(\alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1)(\text{var})) \sqcup_{\widehat{\mathbf{V}}} \pi_1(\alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2)(\text{var})), \pi_2(\widehat{\mathbf{S}}_1(\text{var})) \cup \pi_2(\widehat{\mathbf{S}}_2(\text{var})))\end{aligned}\tag{4.4.3}$$

Combining 4.4.1, 4.4.2, and 4.4.3 we obtain that

$$\begin{aligned}\bar{s} : \forall \text{var} \in \text{dom}(\widehat{\mathbf{S}}_1) \cup \text{dom}(\widehat{\mathbf{S}}_2) : \\ \widehat{\mathbf{S}}(\text{var}) = \begin{cases} \alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1) & \text{if } \text{var} \notin \text{dom}(\widehat{\mathbf{S}}_2) \\ \alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2) & \text{if } \text{var} \notin \text{dom}(\widehat{\mathbf{S}}_1) \\ (\pi_1(\alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1)(\text{var})) \sqcup_{\widehat{\mathbf{V}}} \pi_1(\alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2)(\text{var})), \pi_2(\widehat{\mathbf{S}}_1(\text{var})) \cup \pi_2(\widehat{\mathbf{S}}_2(\text{var}))) & \text{if } \text{var} \in \text{dom}(\widehat{\mathbf{S}}_1) \cap \text{dom}(\widehat{\mathbf{S}}_2) \end{cases}\end{aligned}$$

and so by definition of $\sqcup_{\widehat{\mathbf{S}}}$ that $\bar{s} = \alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1) \sqcup_{\widehat{\mathbf{S}}} \alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2)$.

We proved that $\alpha_{\widehat{\mathbf{S}}}(\widehat{\mathbf{S}}_1 \sqcup_{\widehat{\mathbf{S}}} \widehat{\mathbf{S}}_2) = \bar{s}$ and that $\bar{s} = \alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1) \sqcup_{\widehat{\mathbf{S}}} \alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2)$. So, by transitive property of equivalence, we get $\alpha_{\widehat{\mathbf{S}}}(\widehat{\mathbf{S}}_1 \sqcup_{\widehat{\mathbf{S}}} \widehat{\mathbf{S}}_2) = \alpha_{\widehat{\mathbf{S}}_1}(\widehat{\mathbf{S}}_1) \sqcup_{\widehat{\mathbf{S}}} \alpha_{\widehat{\mathbf{S}}_2}(\widehat{\mathbf{S}}_2)$. ■

LEMMA 4.4.2 ($\alpha_{\widehat{\mathbf{S}}^\ddagger}$ IS A JOIN PRESERVING MAP) $\alpha_{\widehat{\mathbf{S}}^\ddagger}$ is a join preserving map, i.e. $\forall \widehat{\tau}_1, \widehat{\tau}_2 \in \widehat{\mathbf{S}}^\ddagger : \alpha_{\widehat{\mathbf{S}}^\ddagger}(\widehat{\tau}_1 \sqcup_{\widehat{\mathbf{S}}^\ddagger} \widehat{\tau}_2) = \alpha_{\widehat{\mathbf{S}}^\ddagger}(\widehat{\tau}_1) \sqcup_{\widehat{\mathbf{S}}^\ddagger} \alpha_{\widehat{\mathbf{S}}^\ddagger}(\widehat{\tau}_2)$

PROOF. Supposing that $\widehat{\tau}_1 = \widehat{\sigma}_0 \rightarrow \dots \rightarrow \widehat{\sigma}_i$, $\widehat{\tau}_2 = \widehat{\sigma}'_0 \rightarrow \dots \rightarrow \widehat{\sigma}'_j$, and $i > j$ we obtain that:

$$\begin{aligned}\alpha_{\widehat{\mathbf{S}}^\ddagger}((\widehat{\sigma}_0 \rightarrow \dots \rightarrow \widehat{\sigma}_i) \sqcup_{\widehat{\mathbf{S}}^\ddagger} (\widehat{\sigma}'_0 \rightarrow \dots \rightarrow \widehat{\sigma}'_j)) & \\ & \text{(by definition of } \sqcup_{\widehat{\mathbf{S}}^\ddagger} \text{)} \\ = \alpha_{\widehat{\mathbf{S}}^\ddagger}((\widehat{\sigma}_0 \sqcup_{\widehat{\mathbf{S}}} \widehat{\sigma}'_0) \rightarrow \dots \rightarrow (\widehat{\sigma}_j \sqcup_{\widehat{\mathbf{S}}} \widehat{\sigma}'_j) \rightarrow \widehat{\sigma}_{j+1} \rightarrow \dots \rightarrow \widehat{\sigma}_i) & \\ & \text{(by definition of } \alpha_{\widehat{\mathbf{S}}^\ddagger} \text{)} \\ = \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_0 \sqcup_{\widehat{\mathbf{S}}} \widehat{\sigma}'_0) \rightarrow \dots \rightarrow \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_j \sqcup_{\widehat{\mathbf{S}}} \widehat{\sigma}'_j) \rightarrow \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_{j+1}) \rightarrow \dots \rightarrow \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_i) & \\ & \text{(by Lemma 4.4.1)} \\ = \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_0) \sqcup_{\widehat{\mathbf{S}}} \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}'_0) \rightarrow \dots \rightarrow \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_j) \sqcup_{\widehat{\mathbf{S}}} \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}'_j) \rightarrow \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_{j+1}) \rightarrow \dots \rightarrow \alpha_{\widehat{\mathbf{S}}}(\widehat{\sigma}_i) & \\ & \text{(by definition of } \sqcup_{\widehat{\mathbf{S}}^\ddagger} \text{)} \\ = \alpha_{\widehat{\mathbf{S}}^\ddagger}(\widehat{\tau}_1) \sqcup_{\widehat{\mathbf{S}}^\ddagger} \alpha_{\widehat{\mathbf{S}}^\ddagger}(\widehat{\tau}_2) & \end{aligned}$$

The proof is the same when the situation is opposite, i.e. when $j \geq i$. ■

LEMMA 4.4.3 ($\alpha_{\widehat{\Psi}}$ IS A JOIN PRESERVING MAP) $\alpha_{\widehat{\Psi}}$ is a join preserving map, i.e. $\forall \widehat{f}_1, \widehat{f}_2 \in \widehat{\Psi} :$
 $\alpha_{\widehat{\Psi}}(\widehat{f}_1 \sqcup_{\widehat{\Psi}} \widehat{f}_2) = \alpha_{\widehat{\Psi}}(\widehat{f}_1) \sqcup_{\overline{\Psi}} \alpha_{\widehat{\Psi}}(\widehat{f}_2)$

PROOF.

$$\begin{aligned}
& \alpha_{\widehat{\Psi}}(\widehat{f}_1 \sqcup_{\widehat{\Psi}} \widehat{f}_2) \\
& \quad (\text{by definition of } \alpha_{\widehat{\Psi}}) \\
& = \bar{f} : \forall t \in \text{dom}(\widehat{f}_1 \sqcup_{\widehat{\Psi}} \widehat{f}_2) : \bar{f}(t) = \alpha_{\widehat{\mathcal{S}}^\#}((\widehat{f}_1 \sqcup_{\widehat{\Psi}} \widehat{f}_2)(t)) \\
& \quad (\text{by definition of } \sqcup_{\widehat{\Psi}}) \\
& = \bar{f} : \forall t \in \text{dom}(\widehat{f}_1) \cup \text{dom}(\widehat{f}_2) : \bar{f}(t) = \alpha_{\widehat{\mathcal{S}}^\#}(\widehat{f}_1(t) \sqcup_{\widehat{\mathcal{S}}^\#} \widehat{f}_2(t)) \\
& \quad (\text{by Lemma 4.4.2}) \\
& = \bar{f} : \forall t \in \text{dom}(\widehat{f}_1) \cup \text{dom}(\widehat{f}_2) : \bar{f}(t) = \alpha_{\widehat{\mathcal{S}}^\#}(\widehat{f}_1(t)) \sqcup_{\widehat{\mathcal{S}}^\#} \alpha_{\widehat{\mathcal{S}}^\#}(\widehat{f}_2(t)) \\
& \quad (\text{by definition of } \sqcup_{\widehat{\mathcal{S}}^\#}) \\
& = \alpha_{\widehat{\Psi}}(\widehat{f}_1) \sqcup_{\overline{\Psi}} \alpha_{\widehat{\Psi}}(\widehat{f}_2)
\end{aligned}$$

■

THEOREM 4.4.4 Let $\leq_{\overline{\Psi}}$ be the partial order induced by $\sqcup_{\overline{\Psi}}$, and $\gamma_{\widehat{\Psi}}$ be defined as

$$\gamma_{\widehat{\Psi}} = \lambda \bar{y}. \bigsqcup_{\widehat{\Psi}} \{ \widehat{z} : \alpha_{\widehat{\Psi}}(\widehat{z}) \sqsubseteq_{\overline{\Psi}} \bar{y} \}$$

Then

$$\langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}} \rangle \xleftrightarrow[\alpha_{\widehat{\Psi}}]{\gamma_{\widehat{\Psi}}} \langle \overline{\Psi}, \sqsubseteq_{\overline{\Psi}} \rangle$$

PROOF. Lemma 4.4.3 proved that $\alpha_{\widehat{\Psi}}$ is a join preserving map. So by Theorem 2.2.2 we proved that

$$\langle \widehat{\Psi}, \sqsubseteq_{\widehat{\Psi}} \rangle \xleftrightarrow[\alpha_{\widehat{\Psi}}]{\gamma_{\widehat{\Psi}}} \langle \overline{\Psi}, \sqsubseteq_{\overline{\Psi}} \rangle$$

■

4.4.4 Transfer Function

The transfer function $\overline{\mathbb{W}} : [(\overline{\mathcal{S}} \times \text{Tld}) \rightarrow \overline{\mathcal{S}}]$ of `sh_var = value` is defined as

$$\overline{\mathbb{W}}[\text{sh_var} = \text{value}](\bar{s}, t) = \bar{s}[\text{sh_var} \mapsto (\widehat{\text{value}}, \{t\})]$$

The function $\overline{\mathbb{R}} : [\overline{\mathcal{S}} \rightarrow \widehat{\mathcal{V}}]$ of `eval(sh_var)` is defined as

$$\overline{\mathbb{R}}[\text{eval}(\text{sh_var})](\bar{s}) = \pi_1(\bar{s}(\text{sh_var}))$$

LEMMA 4.4.5 (SOUNDNESS OF $\overline{\mathbb{W}}$) $\overline{\mathbb{W}}$ is the abstraction of $\widehat{\mathbb{W}}$, i.e.

$$\forall \widehat{s} \in \widehat{\mathcal{S}}, \forall t \in \text{Tld} : \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathbb{W}}[\text{sh_var} = \text{value}](\widehat{s}, t)) = \overline{\mathbb{W}}[\text{sh_var} = \text{value}](\alpha_{\widehat{\mathcal{S}}}(\widehat{s}), t)$$

PROOF.

$$\begin{aligned}
& \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathbb{W}}[\text{sh_var} = \text{value}](\widehat{\mathcal{S}}, t)) \\
& \quad (\text{by definition of } \widehat{\mathbb{W}}[\text{sh_var} = \text{value}]) \\
& = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}[\text{sh_var} \mapsto [t \mapsto \alpha_V(\text{value})]]) \\
& \quad (\text{by definition of } \alpha_{\widehat{\mathcal{S}}}) \\
& = \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}[\text{sh_var} \mapsto (\alpha_V(\text{value}), \{t\})]) \\
& \quad (\text{by definition of } \widehat{\mathbb{W}}[\text{sh_var} = \text{value}]) \\
& = \widehat{\mathbb{W}}[\text{sh_var} = \text{value}](\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}), t)
\end{aligned}$$

■

LEMMA 4.4.6 (SOUNDNESS OF $\overline{\mathbb{R}}$) $\overline{\mathbb{R}}$ is equal to $\widehat{\mathbb{R}} \circ \alpha_{\widehat{\mathcal{S}}}$, i.e.

$$\forall \widehat{\mathcal{S}} \in \widehat{\mathcal{S}} : \alpha_V(\widehat{\mathbb{R}}[\text{eval}(\text{sh_var})](\widehat{\mathcal{S}})) = \overline{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}}))$$

PROOF. By definition of $\widehat{\mathbb{R}}[\text{eval}(\text{sh_var})]$ we have that

$$\forall \widehat{\mathcal{S}} \in \widehat{\mathcal{S}} : \widehat{\mathbb{R}}[\text{eval}(\text{sh_var})](\widehat{\mathcal{S}}) = \bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}(\text{sh_var}))} \widehat{\mathcal{S}}(\text{sh_var})(t) \quad (4.4.4)$$

In the same way, by definition of $\overline{\mathbb{R}}[\text{eval}(\text{sh_var})]$ we have that

$$\forall \widehat{\mathcal{S}} \in \widehat{\mathcal{S}} : \overline{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})) = \pi_1(\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})(\text{sh_var})) \quad (4.4.5)$$

By definition of $\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})$ we have that

$$\forall \text{var} \in \text{dom}(\widehat{\mathcal{S}}) : \alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})(\text{var}) = \left(\bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}(\text{var}))} \widehat{\mathcal{S}}(t), \text{dom}(\widehat{\mathcal{S}}(\text{var})) \right) \quad (4.4.6)$$

Combining 4.4.5 and 4.4.6, we obtain that

$$\overline{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})) = \pi_1\left(\left(\bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}(\text{sh_var}))} \widehat{\mathcal{S}}(t), \text{dom}(\widehat{\mathcal{S}}(\text{sh_var}))\right)\right)$$

So by definition of the projection function, we finally obtain that

$$\overline{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})) = \bigsqcup_{t \in \text{dom}(\widehat{\mathcal{S}}(\text{sh_var}))} \widehat{\mathcal{S}}(t) \quad (4.4.7)$$

Combining 4.4.4 and 4.4.7, we prove that

$$\overline{\mathbb{R}}[\text{eval}(\text{sh_var})](\alpha_{\widehat{\mathcal{S}}}(\widehat{\mathcal{S}})) = \widehat{\mathbb{R}}[\text{eval}(\text{sh_var})](\widehat{\mathcal{S}})$$

by transitive property of equivalence. ■

Thread 1 : [a.amount \mapsto ([900..1.000], {System, Thread 2})] \rightarrow [a.amount \mapsto ([1.000..1.100], {Thread 1})]

Thread 2 : [a.amount \mapsto ([1.000..1.100], {System, Thread 1})] \rightarrow [a.amount \mapsto ([900..1.000], {Thread 2})]

Figure 4.4: The abstract semantics on the example

4.4.5 The Example

Figure 4.4 presents the results of the abstract semantics. Also in this case we use the Interval domain in order to capture numerical information.

This level of abstraction infers that the value read by Thread 1 in a.amount may have been previously written by both System and Thread 2. The value read by Thread 2 may have been written by System or Thread 1.

4.5 The Deterministic Property

In the last three sections we presented the concrete domain and semantics, and the two levels of abstraction. In this section, we define the deterministic property on them.

4.5.1 Determinism

A program is said to be deterministic “if given an input it returns always the same output” [38]. This definition does not specify what the input and the output of a program is. Someone may think that the output is what is written on the screen, by it may also be the state of the memory at the end of the execution, or during it.

A too restrictive application of the concept of determinism may lead to a definition where a multithreaded program is deterministic if all the statements are always executed in the same total order. We face this problem through the thread-partitioning domain that abstracts away the inter-thread order in which the statements are executed.

Moreover we focus on the non-determinism induced by the arbitrary interleavings of the execution of different threads. As we do not make any supposition or restriction on the programming language and its semantics, many different forms of non-determinism may exist, e.g. a random number produced by the system, or the inputs received by a user. We want to catch and analyze only the non-deterministic behaviors due to the parallel execution of threads.

This goal can be reached by using the information collected by our analysis. We trace for each value on the shared memory which thread may have written it. Then it is sufficient to check if a value could have been written by two different threads at a given point, i.e. for a given state of execution of a given thread.

4.5.2 Formal Definition of Determinism on the Concrete Domain

We first define the determinism as difference between two shared memories, and then we apply it to a set of elements of the concrete multithread semantics.

DEFINITION 4.5.1 (DETERMINISM ON SHARED MEMORY) *Given two shared memories, they underline a nondeterministic behavior due to the multithreaded execution if they relate the same variable to values written by different threads.*

$$\begin{aligned}
 ds &: [\mathbb{S} \times \mathbb{S} \rightarrow \{\text{true}, \text{false}\}] \\
 ds(s_1, s_2) &= \text{false} \\
 &\Downarrow \\
 \exists \text{var} \in \text{dom}(s_1) \cap \text{dom}(s_2) &: s_1(\text{var}) = (\text{val}_1, t_1), s_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2
 \end{aligned}$$

DEFINITION 4.5.2 (DETERMINISM ON MULTITHREADED STATE)

$$\begin{aligned}
 d &: [\wp(\Psi) \rightarrow \{\text{true}, \text{false}\}] \\
 d(\theta) &= \text{false} \\
 &\Downarrow \\
 \exists f_1, f_2 \in \theta &: \exists t \in \text{dom}(f_1) \cap \text{dom}(f_2) : \tau_1 = f_1(t), \tau_2 = f_2(t), \\
 \exists i \in [0.. \min(\text{len}(\tau_1), \text{len}(\tau_2))] &: ds(\tau_1(i), \tau_2(i)) = \text{false}
 \end{aligned}$$

4.5.3 First Level of Abstraction

DEFINITION 4.5.3 (\widehat{ds})

$$\begin{aligned}
 \widehat{ds} &: [\widehat{\mathbb{S}} \rightarrow \{\text{true}, \text{false}\}] \\
 \widehat{ds}(\widehat{s}) &= \text{false} \\
 &\Downarrow \\
 \exists \text{var} \in \text{dom}(\widehat{s}) &: |\text{dom}(\widehat{s}(\text{var}))| > 1
 \end{aligned}$$

DEFINITION 4.5.4 (\widehat{d})

$$\begin{aligned}
 \widehat{d} &: [\widehat{\Psi} \rightarrow \{\text{true}, \text{false}\}] \\
 \widehat{d}(\widehat{f}) &= \text{false} \\
 &\Downarrow \\
 \exists t \in \text{dom}(\widehat{f}) &: \widehat{\tau} = \widehat{f}(t), \exists i \in [0.. \text{len}(\widehat{\tau})] : \widehat{ds}(\widehat{\tau}(i)) = \text{false}
 \end{aligned}$$

LEMMA 4.5.5 (SOUNDNESS OF \widehat{d}) .

$$\forall \theta \in \wp(\Psi) : d(\theta) = \text{false} \Rightarrow \widehat{d}(\alpha_\Psi(\theta)) = \text{false}$$

PROOF. We have to prove that $\widehat{d}(\alpha_\Psi(\theta)) = \text{false}$. By definition of \widehat{d} , this implies that:

$$\begin{aligned} & \exists t \in \text{dom}(\alpha_\Psi(\theta)) : \widehat{\tau} = \alpha_\Psi(\theta)(t), \\ & \exists i \in [0..len(\widehat{\tau})] : \widehat{\mathbf{s}} = \widehat{\tau}(i), \exists \text{var} \in \text{dom}(\widehat{\mathbf{s}}) : |\text{dom}(\widehat{\mathbf{s}}(\text{var}))| > 1 \end{aligned}$$

By definition of α_Ψ , we have that

$$\begin{aligned} & \exists f_1, f_2 \in \theta, \exists t \in \text{dom}(f_1) \cap \text{dom}(f_2) : \widehat{\tau} = \alpha_{\mathcal{S}^\neq}(\{f_1(t), f_2(t)\}), \\ & \exists i \in [0..len(\widehat{\tau})] : \widehat{\mathbf{s}} = \widehat{\tau}(i), \exists \text{var} \in \text{dom}(\widehat{\mathbf{s}}) : |\text{dom}(\widehat{\mathbf{s}}(\text{var}))| > 1 \end{aligned}$$

By definition of $\alpha_{\mathcal{S}^\neq}$ it implies that

$$\begin{aligned} & \exists f_1, f_2 \in \theta, \exists t \in \text{dom}(f_1) \cap \text{dom}(f_2) : \tau_1 = f_1(t), \tau_2 = f_2(t), \\ & \exists i \in [0.. \min(len(\tau_1), len(\tau_2))] : \widehat{\mathbf{s}} = \alpha_{\mathcal{S}}(\{\tau_2(i), \tau_2(i)\}), \\ & \exists \text{var} \in \text{dom}(\widehat{\mathbf{s}}) : |\text{dom}(\widehat{\mathbf{s}}(\text{var}))| > 1 \end{aligned}$$

By definition of $\alpha_{\mathcal{S}}$ we obtain that

$$\begin{aligned} & \exists f_1, f_2 \in \theta, \exists t \in \text{dom}(f_1) \cap \text{dom}(f_2) : \tau_1 = f_1(t), \tau_2 = f_2(t), \\ & \exists i \in [0.. \min(len(\tau_1), len(\tau_2))] : \mathbf{s}_1 = \tau_1(i), \mathbf{s}_2 = \tau_2(i), \\ & \exists \text{var} \in \text{dom}(\mathbf{s}_1) \cap \text{dom}(\mathbf{s}_2) : \mathbf{s}_1(\text{var}) = (\text{val}_1, t_1), \mathbf{s}_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2 \end{aligned}$$

By definition of d it implies that $d(\theta) = \text{false}$. This condition is true by hypothesis. So we proved that

$$\forall \theta \in \wp(\Psi) : d(\theta) = \text{false} \Rightarrow \widehat{d}(\alpha_\Psi(\theta)) = \text{false}$$

■

4.5.4 Second Level of Abstraction

DEFINITION 4.5.6 (\overline{ds})

$$\begin{aligned} & \overline{ds} : [\overline{\mathcal{S}} \rightarrow \{\text{true}, \text{false}\}] \\ & \overline{ds}(\overline{\mathbf{s}}) = \text{false} \Leftrightarrow \exists \text{var} \in \text{dom}(\overline{\mathbf{s}}) : |\pi_2(\overline{\mathbf{s}}(\text{var}))| > 1 \end{aligned}$$

DEFINITION 4.5.7 (\overline{d})

$$\begin{aligned} & \overline{d} : [\overline{\Psi} \rightarrow \{\text{true}, \text{false}\}] \\ & \overline{d}(\overline{\mathbf{f}}) = \text{false} \\ & \Downarrow \\ & \exists t \in \text{dom}(\overline{\mathbf{f}}) : \overline{\tau} = \overline{\mathbf{f}}(t), \exists i \in [0..len(\overline{\tau})] : \overline{ds}(\overline{\tau}(i)) = \text{false} \end{aligned}$$

LEMMA 4.5.8 (SOUNDNESS OF \overline{d}) .

$$\forall \mathbf{f} \in \widehat{\Psi} : \widehat{d}(\mathbf{f}) = \text{false} \Rightarrow \overline{d}(\alpha_{\widehat{\Psi}}(\mathbf{f})) = \text{false}$$

PROOF. We want to prove that $\bar{d}(\alpha_{\widehat{\Psi}}(\widehat{f})) = \text{false}$. By definition of \bar{d} , this implies that:

$$\begin{aligned} \exists t \in \text{dom}(\alpha_{\widehat{\Psi}}(\widehat{f})) : \bar{\tau} = \alpha_{\widehat{\Psi}}(\widehat{f})(t), \exists i \in [0..\text{len}(\bar{\tau})] : \\ \bar{s} = \bar{\tau}(i), \exists \text{var} \in \text{dom}(\bar{s}) : |\pi_2(\bar{s}(\text{var}))| > 1 \end{aligned}$$

By definition of $\alpha_{\widehat{\Psi}}$, it implies that

$$\begin{aligned} \exists t \in \text{dom}(\widehat{f}) : \bar{\tau} = \alpha_{\widehat{\Psi}^+}(\widehat{f})(t), \exists i \in [0..\text{len}(\bar{\tau})] : \\ \bar{s} = \bar{\tau}(i), \exists \text{var} \in \text{dom}(\bar{s}) : |\pi_2(\bar{s}(\text{var}))| > 1 \end{aligned}$$

By definition of $\alpha_{\widehat{\Psi}^+}$, this implies that

$$\begin{aligned} \exists t \in \text{dom}(\widehat{f}) : \widehat{\tau} = \widehat{f}(t), \exists i \in [0..\text{len}(\widehat{\tau})] : \\ \bar{s} = \alpha_{\widehat{\Psi}}(\widehat{\tau}(i)), \exists \text{var} \in \text{dom}(\bar{s}) : |\pi_2(\bar{s}(\text{var}))| > 1 \end{aligned}$$

By definition of $\alpha_{\widehat{\Psi}}$, this implies that

$$\begin{aligned} \exists t \in \text{dom}(\widehat{f}) : \widehat{\tau} = \widehat{f}(t), \exists i \in [0..\text{len}(\widehat{\tau})] : \\ \widehat{s} = \widehat{\tau}(i), \exists \text{var} \in \text{dom}(\widehat{s}) : |\text{dom}(\widehat{s}(\text{var}))| > 1 \end{aligned}$$

By definition of \widehat{d} it implies that $\widehat{d}(\widehat{f}) = \text{false}$. This condition is true by hypothesis. So we proved that

$$\forall \widehat{f} \in \widehat{\Psi} : \widehat{d}(\widehat{f}) = \text{false} \Rightarrow \bar{d}(\alpha_{\widehat{\Psi}}(\widehat{f})) = \text{false}$$

■

4.5.5 The Example

Thread 1 : [a.amount \mapsto (1.000, <u>System</u>)] \mapsto [a.amount \mapsto (1.100, Thread 1)]
Thread 2 : [a.amount \mapsto (1.100, <u>Thread 1</u>)] \mapsto [a.amount \mapsto (1.000, Thread 2)]
Thread 1 : [a.amount \mapsto (900, <u>Thread 2</u>)] \mapsto [a.amount \mapsto (1.000, Thread 1)]
Thread 2 : [a.amount \mapsto (1.000, <u>System</u>)] \mapsto [a.amount \mapsto (900, Thread 2)]

Figure 4.5: The non-deterministic behaviors in the concrete semantics

The deterministic property on the concrete semantics discovers that the example introduced in Section 4.1.3 does not respect it. In particular, the two executions differ on the read values. In the first case Thread 1 reads a value written by System and Thread 2 the one written by Thread 1. In the second case Thread 1 reads the value of Thread 2 and Thread 2 the one of System. In Figure 4.5 we underlines the identifiers of threads in the cases in which a non-deterministic behavior arises.

The deterministic property is not validated in the two level of abstractions as they are sound.

4.6 Weak Determinism

In this section we introduce a new concept of determinism which is weaker than the previous definitions. It is defined on the first level of abstraction.

4.6.1 Approximating Numerical Values

The first level of abstraction is parameterized by a non-relational abstract domain that approximates the numerical values. Moreover, we collect an abstract value for each thread that approximates all the concrete values it may have written at that point on a given shared variable. We are in position to define a new idea through these observations: the weak determinism.

The idea is that two different concrete values do not produce different observable behaviors if their abstraction is the same. This concept has to be tuned on an abstract domain. For instance, with the Sign domain it would mean that at a given point all the values written in parallel on a given variable have the same sign.

4.6.2 Formal Definition

DEFINITION 4.6.1 (WEAK DETERMINISM ON SHARED MEMORY)

$$\begin{aligned} \widehat{wds} : [\widehat{\mathcal{S}} \rightarrow \{\text{true}, \text{false}\}] \\ \widehat{wds}(\widehat{\mathcal{S}}) = \text{false} \\ \Downarrow \\ \exists \text{var} \in \text{dom}(\widehat{\mathcal{S}}) : |\text{dom}(\widehat{\mathcal{S}}(\text{var}))| > 1 \wedge \exists t_1, t_2 \in \text{dom}(\widehat{\mathcal{S}}(\text{var})) : \widehat{\mathcal{S}}(\text{var})(t_1) \neq \widehat{\mathcal{S}}(\text{var})(t_2) \end{aligned}$$

DEFINITION 4.6.2 (WEAK DETERMINISM ON MULTITHREADED STATE)

$$\begin{aligned} \widehat{wd} : [\widehat{\Psi} \rightarrow \{\text{true}, \text{false}\}] \\ \widehat{wd}(\widehat{\mathbf{f}}) = \text{false} \\ \Downarrow \\ \exists t \in \text{dom}(\widehat{\mathbf{f}}) : \widehat{\tau} = \widehat{\mathbf{f}}(t), \exists i \in [0..len(\widehat{\tau})] : \widehat{wds}(\widehat{\tau}(i)) = \text{false} \end{aligned}$$

Note that even if a program is not deterministic the weak determinism may validate it. In fact the values written by different threads may produce the same abstract element. On the other hand, if the weak deterministic property is not respected also the deterministic one will not be.

LEMMA 4.6.3 ($\widehat{wd}(\widehat{\mathbf{f}}) = \text{false} \Rightarrow \widehat{d}(\widehat{\mathbf{f}}) = \text{false}$) *If $\widehat{wd}(\widehat{\mathbf{f}}) = \text{false}$ then $\widehat{d}(\widehat{\mathbf{f}}) = \text{false}$.*

PROOF. The proof follows immediately from the definitions of \widehat{wds} and \widehat{ds} as \widehat{wd} and \widehat{d} performs exactly the same checks, the first one relying on \widehat{wds} while the second one uses \widehat{ds} . ■

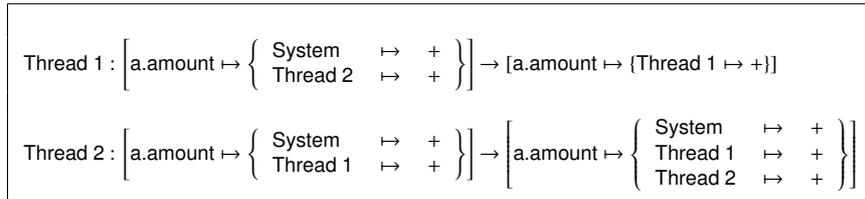


Figure 4.6: The abstract semantics

4.6.3 Example 2

Suppose to modify the multithreaded program introduced in Section 4.1.3 executing method `withdrawNoDebts` of class `Account` (defined in Section 2.3) instead of method `withdraw`. This method allows the withdrawing only if there is enough money in the bank account. We use the `Sign` domain to capture numerical information. In this way we capture for each numerical value if it is positive (+), negative (-) or equal to zero (0). The results of the analysis at the first level of abstraction are depicted by Figure 4.6.

Applying the weak deterministic property we obtain that it is validated. In fact the value stored in the field `amount` is always positive in all the possible executions. On the other hand, the full determinism is not guaranteed as the amount of money may have been written in parallel by different threads.

4.7 Tracing Nondeterminism

Until now, we studied the non-deterministic behaviors when accessing the shared memory. In this section we sketch some different approaches in order to

- trace how non-determinism may influence the execution of a thread,
- trace how it may flow starting from the read and write operations on the shared memory,
- abstract it.

4.7.1 Modifying a Value

At the first level of abstraction we collected the value that each thread may have written on a shared variable. When the information flows from the shared to the private memory, we can distinguish three approaches. For instance, consider the case in which

1. a value is read from the shared memory,
2. an arithmetic operation is performed on that,
3. and finally the result is stored on the shared memory.

In this context we may have:

- first approach: we extract the abstract value making the least upper bound between all the values written by different threads. Then we perform the arithmetic operation. Finally we relate the result to the thread that is writing it on the shared memory. In this way, the only complexity added by our approach is the computation of the least upper bound. Note that we trace only the non determinism induced on read and write actions on the shared heap, but not how it is propagated during the computation. This is the approach adopted until here.
- second approach: we perform on each value the arithmetical operation relating the result to the initial thread. In this way we obtain precise information about
 - which thread induces nondeterministic behavior,
 - on which variables,
 - and which are the different abstract values.

This approach is the most complex one.

- third approach: we perform the operation using the least upper bound of values, and we relate the result to the set of threads that may influence it. So the set of abstract values is $\overline{\text{Val}} \times \wp(\text{TId})$. In this way, we trace which thread induces nondeterministic behaviors and on which variables, but not which are the different abstract values.

4.7.2 An Example

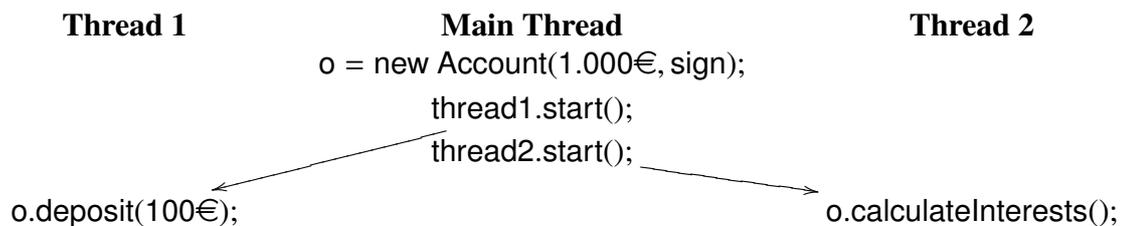


Figure 4.7: Using different approaches

Consider the example presented by Figure 4.7. The main thread instantiates an account with 1.000 €, and it launches in parallel two threads. The first one deposits 100 €, while the second one calculates the interests.

Using the three approaches just presented we obtain the following results:

- first approach: we are able to check that the value read by Thread 2 may be non-deterministic. In fact, it may have been written by the main thread or by Thread 1. Since this information does not propagate when a value is modified, we do not arrive to check that at the end of the execution of Thread 2 also the field `o.amount` may be influenced by it;

- third approach: in this case we propagate the information that the value read by Thread 2 may be nondeterministic. So we arrive to conclude that the value written in the field `o.amount` may be influenced by both the other threads at the end of the execution of Thread 2;
- second approach: applying the Sign domain we trace that both the written values are positive. We discover that the value written by Thread 2 is positive. For instance, we can conclude that they do not expose non-deterministic behaviors on the sign of the variables through the weak deterministic property. Instead, using the Interval domain we check that the two values are different. Using the value written by Thread 1 we obtain a bigger amount of money as the interests are calculated after that the deposit has been registered.

4.7.3 Writing on the Shared Memory

When a value is written on the shared memory we relate it to the identifier of the thread that performs this operation. In this way we trace the origin of the values. Also in this case, other approaches are possible.

During the analysis we may have to perform the least upper bound of many values originated by the same thread but at different points of the program. At abstract level, our approach approximates together all the values written at different points. We can obtain a more precise analysis tracing also the program point. In this way, we would collect for each variable an abstract value for each pair composed by

- a thread identifier,
- a program counter that points to the statement that writes in parallel.

Note that in this case the complexity of the analysis increases noticeably in particular when combining it with the second approach presented in Section 4.7.1.

4.7.4 Discussion

In this section we sketched some ideas on how tuning the analysis at different levels. These approaches allow to obtain a more precise and complex or a faster but less precise analysis. It is necessary to further investigate which approaches are more appropriate and in which contexts. We do not think that a unique final solution exists: it depends on what we are interested to analyze. Moreover it is necessary to perform some practical tests in order to understand how the computational times of the analysis change adopting different approaches.

4.8 Projecting Traces and States

Usually, we are interested in proving the determinism of a program only on a part of the memory or of the performed actions. In this section, we formalize this intuition. In this way we are able to build up a hierarchy of different levels of determinism. We will use these projections in order to formally link the deterministic property with data races and SQL phenomena in the next two sections.

4.8.1 Concrete States

The first way adopted to project the analysis of the determinism is to check it only on a part of the shared memory. We represent it as an abstraction parameterized by a selector that given a variable in the shared memory returns true or false.

DEFINITION 4.8.1 (α_S^{stPrj}) *Let $stPrj : [\text{Var} \rightarrow \{\text{true}, \text{false}\}]$ be a function that given a variable returns true or false. Through this function we define the projection of states as an abstraction:*

$$\begin{aligned} \alpha_S^{stPrj} &: [\mathbb{S} \rightarrow \mathbb{S}] \\ \alpha_S^{stPrj}(\mathbf{s}) &= \{\mathbf{s}' : \text{dom}(\mathbf{s}') \subseteq \text{dom}(\mathbf{s}), \forall \text{var} \in \text{dom}(\mathbf{s}') : \\ &\quad \mathbf{s}'(\text{var}) = \mathbf{s}(\text{var}) \wedge stPrj(\text{var}) = \text{true}\} \end{aligned}$$

We can build up a deterministic property parameterized by this abstraction in order to check it only on the selected variables.

DEFINITION 4.8.2 (ds_{stPrj}) *ds_{stPrj} checks the deterministic property only on the shared variables such that $stPrj = \text{true}$.*

$$\begin{aligned} ds_{stPrj} &: [\mathbb{S} \times \mathbb{S} \rightarrow \{\text{true}, \text{false}\}] \\ ds_{stPrj}(\mathbf{s}_1, \mathbf{s}_2) &= \text{false} \\ &\quad \Downarrow \\ \exists \text{var} \in \text{dom}(\alpha_S^{stPrj}(\mathbf{s}_1)) \cap \text{dom}(\alpha_S^{stPrj}(\mathbf{s}_2)) &: \mathbf{s}_1(\text{var}) = (\text{val}_1, t_1), \mathbf{s}_2(\text{var}) = (\text{val}_2, t_2), t_1 \neq t_2 \end{aligned}$$

LEMMA 4.8.3 ($ds_{stPrj}(\mathbf{s}_1, \mathbf{s}_2) = \text{false} \Rightarrow ds(\mathbf{s}_1, \mathbf{s}_2) = \text{false}$) *If ds_{stPrj} detects a non-deterministic behavior, then also ds will detect it.*

PROOF. The proof follows immediately from the fact that ds_{stPrj} checks the determinism on a subset of the shared variables with respect to ds . ■

LEMMA 4.8.4 ($d_{stPrj}(\theta) = \text{false} \Rightarrow d(\theta) = \text{false}$) *Let d_{stPrj} be the deterministic property as defined by Definition 4.5.2, in which ds is replaced by ds_{stPrj} . If $d_{stPrj}(\theta) = \text{false}$ then $d(\theta) = \text{false}$.*

PROOF. The proof follows immediately from Lemma 4.8.3. ■

LEMMA 4.8.5 ($d_{stPrj}(\theta) = \text{false} \Rightarrow d(\theta) = \text{false}$) *Let d_{stPrj} be the deterministic property as defined by Definition 4.5.2, in which ds is replaced by ds_{stPrj} .*

If $d_{stPrj}(\theta) = \text{false}$ then $d(\theta) = \text{false}$.

PROOF. The proof follows immediately from Lemma 4.8.3. ■

LEMMA 4.8.6 (HIERARCHY OF SHARED MEMORY'S PROJECTIONS) *Given two functions $stPrj_1$ and $stPrj_2$ such that $\{\text{var} : stPrj_1(\text{var}) = \text{true}\} \supseteq \{\text{var} : stPrj_2(\text{var}) = \text{true}\}$, then $d_{stPrj_1}(\theta) = \text{false} \Rightarrow d_{stPrj_2}(\theta) = \text{false}$.*

PROOF. Also this proof follows immediately from Lemma 4.8.3. ■

In this context, Lemma 4.8.5 can be seen as a particular case of Lemma 4.8.6 in which $d = d^{stPrj'} : \forall \text{var} \in \text{Var} : stPrj'(\text{var}) = \text{true}$.

4.8.2 Abstract States

All these definitions can be canonically extended to the deterministic properties of the first and second level of abstraction, and they can be applied also on the weak determinism. In particular, we denote by:

- $\alpha_{\overline{S}}^{stPrj}$ the abstraction function that projects the shared memories of the first level of abstraction only on the variables such that $stPrj = \text{true}$;
- $\alpha_{\overline{S}}^{stPrj}$ this function on the second level of abstraction;
- \widehat{ds}_{stPrj} the deterministic property at the first level of abstraction projecting shared memories following $stPrj$;
- \overline{ds}_{stPrj} this function on the second level of abstraction;
- \widehat{d}_{stPrj} the deterministic property on the traces of the first level of abstraction projecting shared memories following $stPrj$;
- \overline{d}_{stPrj} this function at the second level of abstraction;
- \widehat{wds}_{stPrj} the weak deterministic property on states projecting shared memory following $stPrj$;
- \widehat{wd}_{stPrj} this function applied to traces.

As in the previous subsection, some lemmas define links between deterministic properties on full memories or on their projection. Without entering in formal details, the most interesting relations are the following ones.

LEMMA 4.8.7 $\widehat{d}_{stPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}(\widehat{f}) = \text{false}$

LEMMA 4.8.8 $\bar{d}_{stPrj}(\bar{f}) = \text{false} \Rightarrow \bar{d}(\bar{f}) = \text{false}$

LEMMA 4.8.9 $\widehat{wd}_{stPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{wd}(\widehat{f}) = \text{false}$

Following Lemma 4.8.6 we obtain the following results (where $stPrj_1$ and $stPrj_2$ are such that $\{\text{var} : stPrj_1(\text{var}) = \text{true}\} \supseteq \{\text{var} : stPrj_2(\text{var}) = \text{true}\}$).

LEMMA 4.8.10 $\widehat{d}_{stPrj_1}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{stPrj_2}(\widehat{f}) = \text{false}$

LEMMA 4.8.11 $\bar{d}_{stPrj_1}(\bar{f}) = \text{false} \Rightarrow \bar{d}_{stPrj_2}(\bar{f}) = \text{false}$

LEMMA 4.8.12 $\widehat{wd}_{stPrj_1}(\widehat{f}) = \text{false} \Rightarrow \widehat{wd}_{stPrj_2}(\widehat{f}) = \text{false}$

Finally, following the results of Lemma 4.6.3 we obtain that if the projected weak determinism is not validated then the same will be done by the projected determinism.

LEMMA 4.8.13 $\widehat{wd}_{stPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{stPrj}(\widehat{f}) = \text{false}$

Thanks to all these lemmas, we will be in position to provide a global hierarchy of deterministic properties using different projections.

4.8.3 Concrete Traces

We follow an approach quite similar to the one adopted for concrete states. In particular, we define the deterministic property projected following a selector. Given an index it returns true iff the deterministic property has to be checked on the states at that index in the traces. Then we prove some relations between the deterministic property and its projected versions.

DEFINITION 4.8.14 (d_{trPrj}) *Let $trPrj : [\mathbb{N} \rightarrow \{\text{true}, \text{false}\}]$ be a function that, given an natural number, returns true or false. d_{trPrj} checks the deterministic property only on the indexes of traces such that $trPrj = \text{true}$.*

$$\begin{aligned} d_{trPrj} &: [\wp(\Psi) \rightarrow \{\text{true}, \text{false}\}] \\ d_{trPrj}(\theta) &= \text{false} \\ &\Downarrow \\ &\exists f_1, f_2 \in \theta : \exists t \in \text{dom}(f_1) \cap \text{dom}(f_2) : \tau_1 = f_1(t), \tau_2 = f_2(t), \\ &\exists i \in [0.. \min(\text{len}(\tau_1), \text{len}(\tau_2))] : trPrj(i) = \text{true} \wedge ds(\tau_1(i), \tau_2(i)) = \text{false} \end{aligned}$$

LEMMA 4.8.15 ($d_{trPrj}(\theta) = \text{false} \Rightarrow d(\theta) = \text{false}$) *If $d_{stPrj}(\theta) = \text{false}$ then $d(\theta) = \text{false}$.*

PROOF. The proof follows immediately from the fact that d_{trPrj} controls the determinism on a subset of the states checked by d . ■

LEMMA 4.8.16 (HIERARCHY OF TRACES' PROJECTIONS) *Given two functions $trPrj_1$ and $trPrj_2$ such that $\{i : trPrj_1(i) = \text{true}\} \supseteq \{i : trPrj_2(i) = \text{true}\}$, then $d_{trPrj_1}(\theta) = \text{false} \Rightarrow d_{trPrj_2}(\theta) = \text{false}$.*

PROOF. The proof follows immediately from the fact that by hypothesis d_{trPrj_1} controls the determinism on a subset of the states checked by d_{trPrj_2} . ■

4.8.4 Abstract States

In parallel with the approach adopted in previous subsections we briefly introduce the same concepts in the first and second level of abstraction. So we denote by:

- \widehat{d}_{trPrj} the deterministic property on the traces of the first level of abstraction projecting traces following $trPrj$;
- \bar{d}_{trPrj} this function at the second level of abstraction;
- \widehat{wds}_{trPrj} the weak deterministic property on states projecting traces following $trPrj$;
- \widehat{wd}_{trPrj} this function applied to traces.

LEMMA 4.8.17 $\widehat{d}_{trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}(\widehat{f}) = \text{false}$

LEMMA 4.8.18 $\bar{d}_{trPrj}(\bar{f}) = \text{false} \Rightarrow \bar{d}(\bar{f}) = \text{false}$

LEMMA 4.8.19 $\widehat{wd}_{trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{wd}(\widehat{f}) = \text{false}$

Following Lemma 4.8.16 we obtain the following results (where $trPrj_1$ and $trPrj_2$ are such that $\{i : trPrj_1(i) = \text{true}\} \supseteq \{i : trPrj_2(i) = \text{true}\}$).

LEMMA 4.8.20 $\widehat{d}_{trPrj_1}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{trPrj_2}(\widehat{f}) = \text{false}$

LEMMA 4.8.21 $\bar{d}_{trPrj_1}(\bar{f}) = \text{false} \Rightarrow \bar{d}_{trPrj_2}(\bar{f}) = \text{false}$

LEMMA 4.8.22 $\widehat{wd}_{trPrj_1}(\widehat{f}) = \text{false} \Rightarrow \widehat{wd}_{trPrj_2}(\widehat{f}) = \text{false}$

LEMMA 4.8.23 $\widehat{wd}_{trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{trPrj}(\widehat{f}) = \text{false}$

4.8.5 Projecting both States and Traces

We can combine together the projections on states and traces. We denote by $stPrj \odot trPrj$ the composition of these two projections, and so by $d_{stPrj \odot trPrj}$, $\widehat{d}_{stPrj \odot trPrj}$, $\bar{d}_{stPrj \odot trPrj}$, and $\widehat{wd}_{stPrj \odot trPrj}$ the determinism projected both on traces and state applied respectively to the concrete domain, to the first and second level of abstraction, and the projected weak determinism.

Intuitively, all the lemmas defined on the projections on states and traces hold also on these composed projections. So the following lemmas relates this combination to the projections on states and traces.

LEMMA 4.8.24 $d_{stPrj \odot trPrj}(\theta) = \text{false} \Rightarrow d_{trPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.25 $d_{stPrj \odot trPrj}(\theta) = \text{false} \Rightarrow d_{stPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.26 $\widehat{d}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{trPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.27 $\widehat{d}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{stPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.28 $\bar{d}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \bar{d}_{trPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.29 $\bar{d}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \bar{d}_{stPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.30 $\widehat{wd}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{wd}_{trPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.31 $\widehat{wd}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{wd}_{stPrj}(\widehat{f}) = \text{false}$

LEMMA 4.8.32 $\widehat{wd}_{stPrj \odot trPrj}(\widehat{f}) = \text{false} \Rightarrow \widehat{d}_{stPrj \odot trPrj}(\widehat{f}) = \text{false}$

4.8.6 Hierarchy

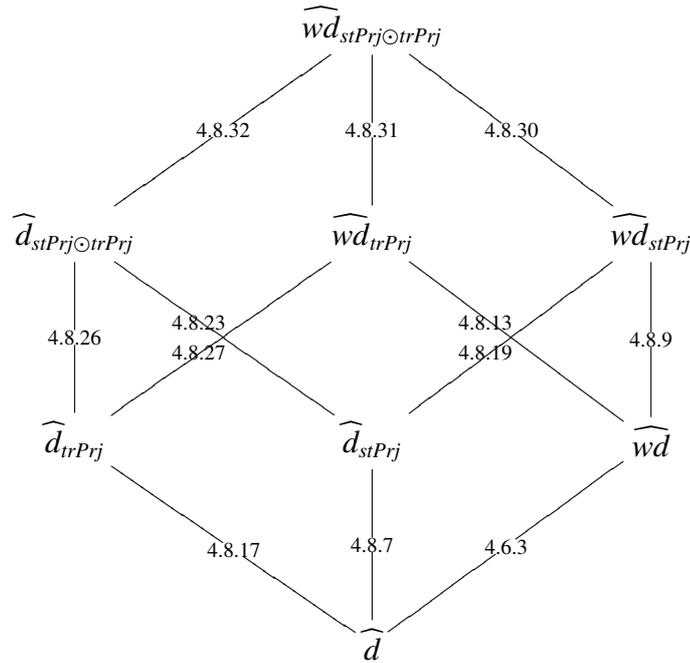


Figure 4.8: A global hierarchy of deterministic properties

Figure 4.8 depicts a global hierarchy on the first level of abstraction of the deterministic and the weak deterministic property. Similar hierarchies can be defined at concrete level and on the second level of abstraction. We focused on the first level as weak determinism is defined at this level. The upper a property is in the diagram, more relaxed it is. This means that it may validate programs that are not validated at lower levels. The number depicted on the links between different deterministic properties reports the Lemma that proves the connection.

Thread 1	Thread 2
a.deposit(100)	a.printAmount()

Figure 4.9: Depositing and printing the amount in parallel

In addition each projection can be expanded into a similar hierarchy following for the first level of abstraction Lemmas 4.8.10 and 4.8.20.

In particular, we can build up a lattice on the functions used to project states and traces. The ordering operator on $stPrj$ is defined as follows:

$$\begin{aligned} \leq_{stPrj}: & [([Var \rightarrow \{true, false\}] \times [Var \rightarrow \{true, false\}]) \rightarrow \{true, false\}] \\ \leq_{stPrj} (stPrj_1, stPrj_2) = & true \text{ iff} \\ & \{var : stPrj_1(var) = true\} \supseteq \{var : stPrj_2(var) = true\} \end{aligned}$$

As the ordering operator relies on the superset operator $\langle stPrj, \leq_{stPrj} \rangle$ is a poset. Following this ordering we can define a hierarchy using different $stPrj$ functions.

The same result can be obtained on $trPrj$, defining the partial ordering as follows:

$$\begin{aligned} \leq_{trPrj}: & [([\mathbb{N} \rightarrow \{true, false\}] \times [\mathbb{N} \rightarrow \{true, false\}]) \rightarrow \{true, false\}] \\ \leq_{trPrj} (trPrj_1, trPrj_2) = & true \text{ iff} \\ & \{i : stPrj_1(i) = true\} \supseteq \{i : stPrj_2(i) = true\} \end{aligned}$$

Finally we obtain a similar result on $stPrj \odot trPrj$ simply defining the partially ordered set as the Cartesian product between $\langle stPrj, \leq_{stPrj} \rangle$ and $\langle trPrj, \leq_{trPrj} \rangle$.

Each of these results allows us to define a local hierarchy on different types of projection.

4.8.7 An example

Consider the example depicted by Figure 4.9. Two threads are executed in parallel and they work on the same bank account. The first thread withdraws 100. The second one prints the amount of money in the account. A non-deterministic behavior may arise when executing Thread 2 on the screen of the ATM, and so the full deterministic property is not validated. Instead, we may be not interested to detect this behavior. For instance, it may be the case that:

- `printAmount()` is not a critical operation, and so we are not interested to prove its determinism. We can project the trace not to check the deterministic property when analyzing the traces produced by this method;
- the screen of the ATM can tolerate non-deterministic behaviors as it is not a critical area of memory, and so we can project the states of execution ignoring this area.

4.8.8 Discussion

In this section we formalized some ways of relaxing the deterministic property projecting traces and states. Other solutions can be adopted, as for instance the weak determinism introduced in Section 4.6. All these solutions can be combined together in order to obtain multiple levels of determinism.

These results make evidence of the flexibility of the deterministic property. Focusing on particular properties like data and general races obliges to develop programs respecting some rigid rules. Instead, we can tune the property to the program we want to analyze with our approach, and in particular to the level of determinism we want to achieve.

In this context, we think that the deterministic property we proposed can get over the actual limits of static analysis applied to multithreaded programs. In the next sections we will sketch which level of determinism two well-known properties, i.e. SQL phenomena and data races, correspond to.

4.9 SQL Phenomena

In this section we link the phenomena defined on SQL language by the ANSI SQL standard [4] to our deterministic property.

4.9.1 The SQL Approach

In the SQL approach a program is composed by a set of transactions that may be executed in parallel. A transaction is composed by a sequence of SQL queries, and it is seen as a sequence of actions executed by the same process. The execution of a program is represented by a sequence of actions $a_1[x_1]a_2[x_2]...end_1$. An action $a_i[x]$ is performed by process i on the memory location x . A read action is denoted by $r_i[x]$, while a write one by $w_i[x]$. end_i denotes the end of the execution of process i . The three dots ... means that there may be an undefined number of actions. Since all the executions end we consider only finite traces.

The DBMS is aimed at guaranteeing that the execution of a program is serializable (i.e. its result can be obtained with a serial execution) or it respects one of the relaxed versions of this property. As the DBMS works checking at run-time the actions, this goal is obtained through the definition of 4 phenomena that must be avoided during the execution. The relaxed versions are obtained allowing one or more of them.

Berenson et al. [15] defined 4 types of phenomena:

- dirty read - $...w_1[x]...r_2[x]...end_1$
- non-repeatable read - $...r_1[x]...w_2[x]...end_1$
- phantom - $...r_1[P]...w_2[y \in P]...end_1$, where $w_2[y \in P]$ means that transaction two writes a value that would have been read by $r_1[P]$ if it had been executed before it

- lost update - ... $w_1[x]$... $w_2[x]$... end_1 .

4.9.2 SQL Phenomena in our Framework

Since in our framework we can read and write on a memory location and not on a set, the phantom phenomenon collapses in the non-repeatable read one.

The model of execution in SQL is to partition the application into many tasks (i.e. transactions) to be executed in parallel. It is exactly the same approach of multithreaded programs that define an application as composed by many threads. Each transaction is an ordered sequence of actions. This corresponds exactly to representing an execution of a multithreaded program as a trace. In this way our thread-partitioning domain is well-fitted in order to represent also transactions. A transaction can be seen as a trace related to a thread in this domain. In addition SQL phenomena read and write in the database that is shared among all the transactions: this corresponds to the concept of shared memory.

In our thread-partitioning domain $w_i[x]$ is equivalent to a state σ belonging to a trace related to thread i . This chapter studies the determinism and so we represent states collecting only the state of the shared memory. On the other hand, in Chapter 3 we have defined the same domain, and we can obtain some information on that through few functions. In this way, if we collect states instead of shared memories only, we can check through $action(\sigma)$ (this function was defined in Section 3.3.1) if the performed action is a write one.

In the same way, $r_i[x]$ is equivalent to a state σ related to the thread i in our thread-partitioning domain and such that $\pi_1(action(\sigma)) = r$.

Phenomena work analyzing the arbitrary interleaving during the execution of different transactions. In the SQL environment we have that actions are totally ordered during the execution, as the DBMS does not execute actions in parallel. We can see it similar to a mono-core processor. The order of execution is abstracted away by our thread-partitioning domain. On the other hand, we may analyze the effects of the execution in different order of the actions on the shared memory. Note that the actions interesting for SQL phenomena are only reads and writes on the database.

4.9.3 Effects of Phenomena on the Determinism

DBMS does not provide any synchronization primitive. Then if a phenomenon happens there would be another execution (in which the two transactions are serially executed) without it. The following proposition explains this concept formally:

PROPOSITION 4.9.1 *Let \mathbb{T} be the set of all the finite executions of a SQL program. Let $t = \dots a_1[x] \dots a'_2[x] \dots end_1 \in \mathbb{T}$ be a trace representing an execution, where a and a' are two actions, performed respectively by thread 1 and 2, that work on a same area of memory x , and such that at least one is a write action. Then $\exists t' = \dots a_1[x'] \dots end_1 \in \mathbb{T} : a'_2[x'] \notin a_1[x'] \dots end_1$.*

In our framework, if two actions represented by states σ_i and σ_j are executed by two different threads, then our thread-partitioning domain relates it to different threads. Formally, it means that a given concrete states f of our multithreaded domain is such that $\sigma_i \in \tau_1, \sigma_j \in \tau_2 : \exists t_1, t_2 \in \text{Tld} : f(t_1) = \tau_1 \wedge f(t_2) = \tau_2 \wedge t_1 \neq t_2$. The type of action and the location on which it works are obtained through the function *action*.

Applying these considerations we define the *phenomena* function. Given two types of action and the set of all the executions, it returns true if the phenomenon represented by the given two types happens at least in one execution.

DEFINITION 4.9.2 (*phenomena* FUNCTION)

$$\begin{aligned} & \textit{phenomena} : [\{\textit{read}, \textit{write}\} \times \{\textit{read}, \textit{write}\} \times \wp(\Psi) \rightarrow \{\textit{true}, \textit{false}\}] \\ & \textit{phenomena}(\mathbf{a}_1, \mathbf{a}_2, \Phi) = \textit{true} \\ & \quad \Updownarrow \\ & \exists \mathbf{e} \in \Phi : \exists t \in \textit{dom}(f) : \exists \sigma_k \in f(t) : \textit{action}(\sigma_k) = (\mathbf{a}_1, \mathbf{x}), \\ & \quad \exists t_1 \in \textit{dom}(f) : t_1 \neq t, \exists \sigma_w \in f(t_1) : \textit{action}(\sigma_w) = (\mathbf{a}_2, \mathbf{x}) \end{aligned}$$

Let $\Phi \in \Psi$ be the set of all possible executions of a transaction. As our multithreaded domain abstracts away the order of execution of different threads, the dirty read and the non-repeatable read phenomena correspond to the same case of *phenomena* function. So a dirty or non-repeatable read phenomenon happens if and only if $\textit{phenomena}(\textit{write}, \textit{read}, \Phi) = \textit{true}$. A lost update phenomenon happens if and only if $\textit{phenomena}(\textit{write}, \textit{write}, \Phi) = \textit{true}$.

4.9.4 Phenomena and Deterministic Property

As SQL transitions have no synchronization primitives, all the transitions are executed in parallel. So if a phenomenon happens, our concrete semantics will contain two executions in which the two actions are executed in the opposite order. In this context, they expose a non-deterministic behavior and we can relate them to our deterministic property.

Since all the phenomena deal only with read and write actions, we can project the traces on the read and write transitions following the approach described in Section 4.8.3. For each phenomenon we check which version of the deterministic property checks it surely, i.e. such that if the phenomenon happens it discovers that the program is not deterministic.

Dirty read or non-repeatable read: By definition of dirty read phenomenon on traces the difference between the two traces is on a read action. Since a read action modifies the private memory of the thread that executes the action we can project also the states on the private memory of thread.

Lost update: In the same way, by definition of dirty read phenomenon on traces the difference between the two traces is on a write action. Thus we can project the states on the shared memory.

Absence of phenomena: If phenomena do not happen during the execution of a program, this means that there will not be two parallel actions (and at least a write) on

the shared memory executed in parallel by different threads. Supposing that the only way that threads have to communicate is through the shared memory, this guarantees that the program is deterministic as

- two reads do not expose non-deterministic behavior since they do not modify the state of the shared memory
- there is no other way to produce a non-deterministic behavior through the arbitrary interleaving of threads' execution.

4.9.5 In the Abstract

Thanks to the soundness of our approach, all the results obtained on the concrete executions can be applied to the abstract semantics. In this way, we can check at compile time if a phenomenon may happen in any execution of a program. In addition, if our deterministic property is validated we are sure that phenomena do not happen.

4.10 Data Race Condition

A data race occurs when “multiple threads access the same data without any intervening synchronization operation, and one of the accesses is a write”[121]. Data races have been widely studied in the field of static analysis. The intuition is that a data race may be the symptom of a bug. The absence of data races does not guarantee the determinism of the program. In fact, even if two accesses to the same shared location are synchronized, we may not be sure that they will be executed always in the same order.

4.10.1 Synchronization

A new concept is introduced by the definition of data race with respect to the database approach: the synchronized accesses. While the SQL language does not provide any primitive to synchronize different transactions, usually a multithreaded programming language does it.

In Chapter 3 we suppose that a function *synchronized* is provided. Given a state, it returns the set of synchronizable elements owned at that point of the execution. These synchronizable elements may be monitors, objects, etc.. In this way, two states $\sigma_1, \sigma_2 \in \Sigma$ are respectively synchronized if $synchronized(\sigma_1) \cap synchronized(\sigma_2) \neq \emptyset$.

DEFINITION 4.10.1 (DATA RACES) *Given a set of trace-partitioning domain elements representing all the executions of a program, the datarace function returns true if and only if at least one of these executions contains a data race.*

$$\begin{aligned} \text{datarace} &: [\wp(\Psi) \rightarrow \{\text{true}, \text{false}\}] \\ \text{datarace}(\Phi) &= \text{true} \Leftrightarrow \exists f \in \Phi : \text{dataracesingleexecution}(f) = \text{true} \end{aligned}$$

where

$$\begin{aligned}
 & \text{dataracesingleexecution} : [\Psi \rightarrow \{\text{true}, \text{false}\}] \\
 & \text{dataracesingleexecution}(f) = \text{true} \Leftrightarrow \exists t_1, t_2 \in \text{dom}(f) : t_1 \neq t_2, \\
 & \quad \exists \sigma_1 \in f(t_1) : \text{action}(\sigma_1) = (\mathbf{a}_1, l_1), \\
 & \quad \exists \sigma_2 \in f(t_2) : \text{action}(\sigma_2) = (\mathbf{a}_2, l_2) : l_1 = l_2 \wedge \\
 & \quad (\mathbf{a}_1 = \text{write} \vee \mathbf{a}_2 = \text{write}) \wedge \text{synchronized}(\sigma_1) \cap \text{synchronized}(\sigma_2) = \emptyset
 \end{aligned}$$

This function will be used to formally compare the data race condition with the phenomena.

4.10.2 Data Races and SQL Phenomena

The last definition is quite similar to the SQL phenomena one presented in Section 4.9. Since the SQL language does not provide any synchronization primitive, we analyze the definition of data races without any synchronization constraint. Formally, it means that $\forall \sigma \in \Sigma : \text{synchronized}(\sigma) = \emptyset$. In this way the difference between the definitions of *datarace* and *phenomena* functions is only on the actions checked. While the *phenomena* function is generic on them, the *datarace* one checks that at least one of the two actions is a write one.

The *phenomena* function is a generalization of the SQL phenomena, and we instantiated it in two ways. Through the instances that check the dirty or non-repeatable read phenomenon and the lost update one, we obtain exactly the results of *datarace* function. This consideration leads to the following proposition.

PROPOSITION 4.10.2 *Let $\Phi \subseteq \Psi$ be the set of all the executions of a program. If $\forall \sigma \in \Sigma : \text{synchronized}(\sigma) = \emptyset$ (where Σ is the set of all the states that can compose traces of elements in Ψ), then $\text{datarace}(\Phi) = \text{true} \Leftrightarrow \text{phenomena}(\text{write}, \text{read}, \Phi) = \text{true} \vee \text{phenomena}(\text{write}, \text{write}, \Phi) = \text{true}$.*

4.10.3 Deterministic Property

In the previous section we proved that the absence of phenomena implies a well-defined level of determinism. In this section we proved that the absence of data races corresponds to the absence of SQL phenomena if we do not consider the synchronization actions.

In this way we also prove a relation between the absence of data races and the deterministic property. Note that it does not mean that the data race condition assures the full determinism of a program: if two parallel accesses are synchronized they still may cause non-deterministic behaviors, but they do not form a data race. This idea is quite similar to the one introduced by weak determinism. While this property relaxes determinism checking only if at abstract level the values written are different, data race condition restricts it only on values accessed by threads while they were not reciprocally synchronized.

4.10.4 Abstract States

As our abstract semantics are sound all the results obtained on the concrete executions can be applied in the abstract. In this way, we can check at compile time if a data race may happen. In particular, if our deterministic property is validated on a given program, we are sure that it does not contain any data race.

4.11 From Determinism to Semi-Automatic Parallelization

In this section we sketch how the determinism may be useful in order to semi-automatically parallelize sequential programs.

4.11.1 Motivation

As already pointed out, multi-core architectures appeared recently in a broad market. The only way for a single application to take advantage from this technology is to be partitioned in many subtasks that may be run in parallel. On the other hand reasoning on parallel applications is strictly more difficult than on sequential programs. A consequence is that (semi-) automatic tools in order to find and detect possible parallelizable fragments of sequential code are particularly useful.

4.11.2 Determinism and Parallelism

A sequential program performs an ordered set of operations. An operation may read some data written or modified by previous statements, and it may write values that will be used during the rest of the computation. When each sequential operation deals with a disjointed set of variables the program is trivially parallelizable. However this condition does not apply to the majority of sequential programs.

Given two parts of a sequential program we can analyze them as if they were executed in parallel, and check if there are some non-deterministic behaviors due to the parallel execution. If it is not the case, the two parts of the sequential program can run in parallel without inducing any new behavior because of the parallel execution.

4.11.3 Relaxing the Deterministic Property

In order to find parallelizable blocks in a program, we may relax the deterministic property in order to focus only on the critical parts of the program. Section 4.6 presented the weak determinism that is a relaxation of the deterministic property. Section 4.8 sketched some different ways in order to project traces and states with the final goal of relaxing the deterministic property.

All these approaches require the developer to give an input to the analysis. In particular,

he has to specify which type of abstract information we have to infer and how to project states and traces. Given this input we can check if two sequential blocks cause non-deterministic behaviors if executed in parallel. If it is not the case we can automatically parallelize a sequential program.

4.11.4 An example

Consider the following sequential program:

```
Account account=new Account(1000, mysignature);
account.deposit(100);
account.printAmount();
```

We might be not interested that the print of the amount reports exactly the last updated value. So we may accept that the deposit of money and this action are executed in parallel. Eventually they may produce a non-deterministic behavior. We can relax our deterministic property projecting traces or states as described in Section 4.8.7. Then we can prove that this program respect it, and so we can parallelize this sequential program.

4.12 Related Work

Data race: We introduced the effects of data races on determinism in Section 4.10. Data race condition allows some nondeterministic behaviors, e.g. when the communications through shared memory are synchronized on a monitor. This may be represented in our framework as a way of relaxing our deterministic property. In addition, if a program respects the full deterministic property, it does not contain data races. Our framework appears to be more expressive and flexible than data race condition.

Software transactional memory: Using the Software Transactional Memory (STM) [127] a developer can tag a piece of code called transaction as **atomic**. The execution of a transaction will be seen as performed completely in a unique step by other transactions. This idea is the extension of database transactions [117] to programming languages. The DBMS checks that some interactions, i.e. SQL phenomena [4], do not happen during the execution of a transaction, and in this way it enforces the atomicity (i.e. that the execution of a transaction can be seen as executed in one unique step) at runtime. SQL phenomena have been introduced in our framework in Section 4.9, and they have been related to data race condition in Section 4.10. In this context, atomicity and the data race condition seem to be two close ideas with respect to the determinism of multithreaded programs. Even if we did not investigate deeply this relation, we think that

- the deterministic property may be used to check if two transactions are atomic in all the possible executions, and so we have not to perform runtime checks on them. The intuition is that if a multithreaded program is deterministic, then the execution of threads can be seen as atomic;

- as for the data race condition, our approach seems to be more flexible and expressive than atomic transactions. In particular, the relaxed versions of the deterministic property have no counterpart in STM.

Automatic parallelization: The automatic parallelization of programs has been studied by optimizing compilers [6]. These techniques have been applied to many languages, and in particular they obtain significant results for logic and constraint programming [62]. When applied to imperative languages, the efforts are focused on the parallelization of computations involving arrays and loops.

The basic idea of optimizing compilers is to perform a program transformation if the transformed program produces the same output of the original one when applied to the same input. Usually these techniques relies on an independence analysis [119, 58, 70, 67]. The idea is that if two sets of sequential statements are independent, then they may be executed in parallel as they access disjointed areas of memory. Intuitively, this corresponds to using full determinism in order to parallelize sequential programs. In fact, if two partitions of a sequential program respect the full deterministic property when executed in parallel, they will produce the same output when applied to the same input. The idea sketched in Section 4.11 is a little bit different, as we talked about semi-automatic parallelization of programs. This process requires the developer to provide an input in order to know which nondeterministic behaviors may be tolerated. Finally, our approach is slightly different, even if a comparison is possible.

We can apply the full deterministic property in order to automatically parallelize programs. In this context, the resulted multithreaded program would usually be less optimized than the ones obtained applying specific techniques of optimizing compilers on arrays and loops. On the other hand, the semi-automatic parallelization using relaxed versions of the deterministic property may gain more parallelism than optimizing compilers, but it requires an input from developers.

4.13 Discussion

In this chapter we presented a generic approach to the study of the determinism of multithreaded programs. In order to apply it to a real programming language and to a more generic context, we need to analyze deeply some details of our approach.

4.13.1 Relational Domains

In the definition of the abstract domain, we supposed that our analysis is parameterized by a non-relational numerical domain. We built up a boxed representation (i.e. a domain that is a function that relates each variable to its abstract value) on this, we defined the semantics of read and write statements on the shared memory and the determinism, and we proved the soundness of our approach. On the other hand, our approach has to be extended in order to support also relational domains.

In order to achieve this goal, we have to redefine only the part of the analysis that concerns the read and write operations on the shared memory. These have to be expressed in terms of evaluation of expressions (read) and value assignments (value). The soundness of this approach would follow directly from the soundness of these primitives of the relational abstract domain.

On the first level of abstraction for each variable the relational domain is required to trace an abstract value for each thread. This means that the set of variables on which relations are inferred will be the Cartesian product of the set of the program's variables and the one of threads' identifiers.

On the second level of abstraction, we gathered together all the values written by different threads. In this context, the relational domain infers relations on the set of program's variables. Then we make the Cartesian product of this domain with a domain that traces for each shared variable the set of threads that may have written concurrently on it.

4.13.2 States in Traces

In order to check the determinism of multithreaded programs we compared states of different executions that appear in the same position in the trace. This simplistic approach hides some important issues on how executions are represented. For instance, we may have that the number of iterations of a loop depends on an input of the user. So we obtain traces with different lengths and in which states at the same position are the results of the executions of different statements. Usually, when reasoning on the determinism we want to compare the results of the execution of the same statements. In this context, we have to represent the executions of a program with a control flow graph. This approach will be formalized by the next Chapter, and in particular in Section 5.6.

4.13.3 Thread Identifiers

When dealing with multithreading, we assumed that the sets of threads' identifiers in the concrete and in the abstract are the same. However, this condition does not apply to modern programming languages. For instance, in Java threads are objects and so they are identified by reference. In this context, the number of threads is potentially unbounded, as references are created at runtime. This issue will be considered in Chapter 5, where we will introduce an ad-hoc alias analysis in order to abstract the concrete references. This abstraction will allow us to keep the initial assumption on threads' identifier, approximating the concrete threads in a finite way.

5

Concrete and Abstract Domain and Semantics of Java Bytecode

In this chapter, we define and abstract a low-level domain and semantics of Java bytecode. First of all, we define the concrete domain and semantics and then we abstract it proving the correctness of our approach. These definitions formalize the specification of the Java Virtual Machine [89]. Finally, we instantiate all the functions (as stated in previous chapters) in order to apply the happens-before memory model and the deterministic property on the analysis of Java multithreaded programs. In this way, we apply the theoretical results introduced in Chapters 3 and 4 to a real-world programming language. This chapter is partly based on the published work [43].

5.1 Notation

In order to formalize the behaviors of the Java virtual machine we need to deal formally with arrays and stacks. An array is a partial function that relates natural numbers to elements. Formally, let A be a generic set. An array on this set is defined as $\mathbb{A}\mathbb{R}(A) : [\mathbb{N} \rightarrow A]$.

We define a stack $\mathbb{S}\mathbb{T}(A) : [\mathbb{N} \rightarrow A]$ as an array on which two functions are defined.

DEFINITION 5.1.1 *pop* : $[\mathbb{S}\mathbb{T}(A) \rightarrow \mathbb{S}\mathbb{T}(A) \times (A \cup \{\perp\})]$ returns the element at the top of the stack (i.e. the element related with the greatest integer value in the domain) and the stack without that element. It returns \perp if the stack is empty.

$$\text{pop}(\mathbf{s}) = \begin{cases} (\emptyset, \perp) & \text{if } \text{dom}(\mathbf{s}) = \emptyset \\ (\mathbf{s} \setminus [i \mapsto \mathbf{s}(i)], \mathbf{s}(i)) : i \in \text{dom}(\mathbf{s}) \wedge \forall j \in \text{dom}(\mathbf{s}) : i \geq j & \text{otherwise} \end{cases}$$

DEFINITION 5.1.2 *push* : $[(\mathbb{S}\mathbb{T}(A) \times A) \rightarrow \mathbb{S}\mathbb{T}(A)]$ pushes the given element on the top of the stack and returns the stack containing this element.

$$\text{push}(\mathbf{s}, v) = \mathbf{s}[i \mapsto v : i = \begin{cases} 0 & \text{if } \text{dom}(\mathbf{s}) = \emptyset \\ m + 1 : m \in \text{dom}(\mathbf{s}) \wedge \forall j \in \text{dom}(\mathbf{s}) : m \geq j & \text{otherwise} \end{cases}]$$

Finally, we denote by

- C the set of Java classes' names,
- M the set of methods' names,
- F the set of fields' identifiers,
- MSig the set of methods' signatures, i.e. the name of the method and the list of parameters.

5.2 Supported Language

Our analysis supports all the Java bytecode language. Most of the bytecode statements are specific for a given type (e.g. `aload`, `iload`, ...), a particular context (e.g. `invokevirtual`, `invokespecial`, `invokestatic`, ...), etc.. So many of them have almost the same semantics. This is why we formalize the semantics on a representation of Java bytecode language. This approach is common when dealing with bytecode [131]. The language on which we formalize our semantics is as follows:

- `putfield < class > < id >` and `getfield < class > < id >` (read and write on object's fields), where `< class > < id >` identifies a field
- `load i` and `store i` (load and store of local variables), where `i` is an index in the local variables array
- `const < val >` (creation of numerical or string constants), where `< val >` may be null, a numerical (both integer or float) value, or a constant string
- `arith < op >` (arithmetic operations), where `< op >` is an arithmetical binary operator (e.g. `+`)
- `new < class >`, and `newarray` (instantiation of objects and arrays), where `< class >` is a class of the current Java program
- `aload` and `astore` (loads and stores on arrays)
- `goto #i`
- `if < op > #i` (conditional jumps), where `< op >` is a conditional operator (e.g. `<`);
- `monitorenter` and `monitorexit` (lock and release of monitors)
- `invoke < method >`, where `< method >` is the signature of a method in MSig
- `return` (end the execution of a method) and `vreturn` (return a value ending the execution of a method)

```
public void withdraw(int money) {
    synchronized(this) {
        account.amount -= money;
    }
}
```

Figure 5.1: withdraw method

5.3 An Example

In order to show how our concrete and abstract domains and semantics work, we will apply them to the method `withdraw` of class `Account` introduced in Section 2.3. The code is reported by Figure 5.1.

Once compiled with `javac`, the resulting Java bytecode is represented in our syntax by following program

```
1 load 0
2 monitorenter
3 load 0
4 load 0
5 getfield <Account> <amount>
6 load 1
7 arith <sub>
8 putfield <Account> <amount>
9 load 0
10 monitorexit
11 return
```

In the following of this chapter, we will suppose that, when this method is called, `this.amount` is equal to 1.000 and the parameter `amount` is equal to 100.

5.4 Concrete Domain

This section formalizes the structure of the computation of the Java Virtual Machine presented in Chapter 3 of its specification[89].

For the sake of simplicity we consider as values only integers and references. Other types can be added to our formalization.

DEFINITION 5.4.1 (VALUES) *Let Ref be the set of addresses. A value may be an address or a numerical value: $\text{Val} = \text{Ref} \cup \mathbb{Z}$*

The operand stack is used to pass values and to perform arithmetical operations on them.

DEFINITION 5.4.2 (OPERAND STACK) *The operand stack is a stack of values: $Op = \mathcal{ST}(\text{Val})$*

Local variables store the values of variables that are accessible only from the current method.

DEFINITION 5.4.3 (LOCAL VARIABLES) *The local variables are represented as an array of values: $LV = \mathcal{AR}(\text{Val})$*

A monitor is defined on each object. Objects are identified by reference and it is possible to lock more than once on the same monitor.

DEFINITION 5.4.4 (MONITORS) *The owned monitors are represented by functions that relate references to integers. The integers represent the number of times which the current thread has locked (and not yet released) on the given monitor: $L : [\text{Ref} \rightarrow \mathbb{N}]$*

The state of an object relates its fields to values. A field is identified by a name and a class. Notice that is not sufficient considering the name only, as through polymorphism a class may contain a field with the same name of one of its superclass.

DEFINITION 5.4.5 (OBJECTS) *We represent the state of objects as functions that relate fields to numerical values or references: $Obj : [(\text{C} \times \text{F}) \rightarrow \text{Val}]$*

DEFINITION 5.4.6 (ARRAYS) *An array is a function that relates integer indexes to concrete values, and an integer value that contains the length of the array: $Arr = \mathcal{AR}(\text{Val}) \times \mathbb{N}$*

DEFINITION 5.4.7 (STRINGS) *Strings are a particular case of an array, i.e. an array of characters. At bytecode level characters are represented by integer values: $\text{Str} = \mathcal{AR}(\mathbb{N}) \times \mathbb{N}$*

DEFINITION 5.4.8 (HEAP) *The heap relates each address to an object, an array, or a string: $H : [\text{Ref} \rightarrow (\text{Obj} \cup \text{Arr} \cup \text{Str})]$*

The single thread state contains also the control of the program.

DEFINITION 5.4.9 (PROGRAM COUNTERS) *A program counter identifies a statement. It is represented as a triple composed by a Java class, a method in it, and an integer value representing the index of the statement inside the method: $\text{PC} = \text{C} \times \text{M} \times \mathbb{N} \cup \{-1\}$*

We suppose to have a function $next : [\text{PC} \rightarrow \text{PC}]$ that given a program counter returns the next one following the sequential order. Given a class c and a method m , the program counter $(c, m, -1)$ means that we are at an exit point of the method.

DEFINITION 5.4.10 (SINGLE-THREAD STATE) *A single-thread state is a tuple composed by*

- *the operand stack,*
- *the array of local variables,*
- *the heap,*
- *the owned monitors,*
- *the program counter pointing to the next statement to be executed*

Formally, $\Sigma = Op \times LV \times H \times L \times PC$

5.5 Concrete Operational Semantics

This section is based on the specification of the Java Virtual Machine instruction set presented in [89], namely on Chapter 6.

5.5.1 Load and Store

Statement `load #i` pushes on the top of the operand stack the value at index `i` in the local variables. Statement `store #i` pops the operand stack, and it stores this value at index `i` in the local variables. These descriptions can be formalized by the following rules.

$$\frac{\text{os}' = \text{push}(\text{os}, \text{lv}(i))}{\langle \text{load } \#i, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, \text{h}, \text{l}, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}', v) = \text{pop}(\text{os}), \text{lv}' = \text{lv}[i \mapsto v]}{\langle \text{store } \#i, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}', \text{h}, \text{l}, \text{next}(\text{pc}) \rangle}$$

5.5.2 Monitors

Statement `monitorenter` pops a reference from the operand stack and it acquires this monitor. Statement `monitorexit` pops a reference from the operand stack and it releases this monitor. As monitor re-entrance is allowed, we count the number of times a monitor has been already locked without being released yet.

$$\frac{(\text{os}', r) = \text{pop}(\text{os}), \text{l}' = \text{l}[r \mapsto \text{l}(r) + 1] \text{ if } r \in \text{dom}(\text{l})}{\langle \text{monitorenter}, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, \text{h}, \text{l}', \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}', r) = \text{pop}(\text{os}), \text{l}' = \text{l}[r \mapsto 1] \text{ if } r \notin \text{dom}(\text{l})}{\langle \text{monitorenter}, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, \text{h}, \text{l}', \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}', r) = \text{pop}(\text{os}), \text{l}(r) = n, \text{l}' = \text{l}[r \mapsto n - 1] \text{ if } n > 1}{\langle \text{monitorexit}, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, \text{h}, \text{l}', \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}', r) = \text{pop}(\text{os}), \text{l}(r) = n, \text{l}' = \text{l} \setminus \{r \mapsto 1\} \text{ if } n = 1}{\langle \text{monitorexit}, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, \text{h}, \text{l}', \text{next}(\text{pc}) \rangle}$$

5.5.3 Objects

Statement `new < class >` creates a new object of type `< class >` in the heap, and it pushes the allocated reference on the operand stack. Statement `getfield < class > < id >` pops a reference from the operand stack, and it pushes the value contained by field `(< class >, < id >)` of the objects pointed by this reference. Finally, statement `putfield < class > < id >` pops a value and a reference from the operand stack, and it assigns the given value to the field `(< class >, < id >)` of the object pointed by this reference.

$$\frac{(r, h') = \text{alloc}(\text{< class >}, \text{h}), \text{os}' = \text{push}(\text{os}, r)}{\langle \text{new } \text{< class >}, (\text{os}, \text{lv}, \text{h}, \text{l}, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, \text{h}', \text{l}, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}_1, r) = \text{pop}(\text{os}), v = h(r)(\langle \text{class} \rangle, \langle \text{id} \rangle), \text{os}' = \text{push}(\text{os}_1, v)}{\langle \text{getfield } \langle \text{class} \rangle \langle \text{id} \rangle, (\text{os}, \text{lv}, h, l, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, h, l, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}_1, v) = \text{pop}(\text{os}), (\text{os}', r) = \text{pop}(\text{os}_1), \text{obj} = h(r), \text{obj}' = \text{obj}[(\langle \text{class} \rangle, \langle \text{id} \rangle) \mapsto v], h' = h[r \mapsto \text{obj}']}{\langle \text{putfield } \langle \text{class} \rangle \langle \text{id} \rangle, (\text{os}, \text{lv}, h, l, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, h', l, \text{next}(\text{pc}) \rangle}$$

alloc : $[(\mathbb{C} \times \mathbb{H}) \rightarrow (\text{Ref} \times \mathbb{H})]$ creates a location containing the default values (i.e. 0 or null) for all the fields of the given class. It returns a fresh reference that points to this location.

5.5.4 Arrays

Statement *newarray* creates a new array in the heap, and it pushes the allocated address on the operand stack. Statement *aload* pops an index and a reference, and it pushes the value contained at the given index by the array pointed by the given reference. Statement *astore* pops a value, an index, and a reference, and it assigns the value to the array pointed by the given reference at the given index.

$$\frac{(\text{os}_1, i) = \text{pop}(\text{os}), (r, h') = \text{allocArray}(i, h), \text{os}' = \text{push}(\text{os}_1, r)}{\langle \text{newarray}, (\text{os}, \text{lv}, h, l, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, h', l, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}_1, i) = \text{pop}(\text{os}), (\text{os}_2, r) = \text{pop}(\text{os}_1), v = h(r)(i), \text{os}' = \text{push}(\text{os}_1, v)}{\langle \text{aload}, (\text{op}, \text{lv}, h, l, \text{pc}) \rangle \rightarrow \langle \text{op}', \text{lv}, h, l, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\text{os}_1, v) = \text{pop}(\text{os}), (\text{os}_2, i) = \text{pop}(\text{os}_1), (\text{os}', r) = \text{pop}(\text{os}_1), \text{ar} = h(r), \text{ar}' = \text{ar}[i \mapsto v], h' = h[r \mapsto \text{ar}']}{\langle \text{astore}, (\text{os}, \text{lv}, h, l, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, h', l, \text{next}(\text{pc}) \rangle}$$

allocArray : $[(\mathbb{N} \times \mathbb{H}) \rightarrow (\text{Ref} \times \mathbb{H})]$ receives a numerical value representing a length and a heap. It returns a fresh reference and a heap. This relates the returned reference to an array of the given length in which all the elements have been set to default value, and the updated state of the heap.

5.5.5 Arithmetic Expressions

Statement *arith* $\langle \text{op} \rangle$ pops two numerical values from the operand stack, and it pushes the result of the arithmetical operation $\langle \text{op} \rangle$ applied on them.

$$\frac{(\text{os}_1, (i_1)) = \text{pop}(\text{os}), (\text{os}_2, i_2) = \text{pop}(\text{os}_1), \text{os}' = \text{push}(\text{os}_2, i_2 \langle \text{op} \rangle i_1)}{\langle \text{arith } \langle \text{op} \rangle, (\text{os}, \text{lv}, h, l, \text{pc}) \rangle \rightarrow \langle \text{os}', \text{lv}, h, l, \text{next}(\text{pc}) \rangle}$$

5.5.6 Constants

Statement `const < val >` pushes on the operand stack the value `< val >`. This can be an integer or a string.

$$\frac{os' = push(os, < val >), \text{ if } < val > \text{ is a numerical value, or it is equal to null}}{\langle const \ < val \ >, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv, h, l, next(pc) \rangle}$$

$$\frac{(r, h') = allocString(< val >, h), os' = push(os, r), \text{ if } < val > \text{ is a string}}{\langle const \ < val \ >, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv, h', l, next(pc) \rangle}$$

allocString : [(Str × H) → (Ref × H)] receives as parameters a string and a heap. It returns a fresh reference and the heap that relates it to the given string.

5.5.7 Jumps

Statement `goto #i` jumps the control to the instruction at index `#i` of the current method. Statement `if < op > #i` jumps to the given index if the boolean condition `< op >` applied to the two elements at the top of the operand stack is evaluated to true. Otherwise it goes on with the next instruction.

$$\frac{pc = (c, m, n), pc' = (c, m, \#i)}{\langle goto \ \#i, (os, lv, h, l, pc) \rangle \rightarrow \langle os, lv, h, l, pc' \rangle}$$

$$\frac{\text{if } evalCondition(< cond \ >, os) = (false, os')}{\langle if \ < op \ > \ \#i, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv, h, l, next(pc) \rangle}$$

$$\frac{pc = (c, m, n), pc' = (c, m, \#i) \text{ if } evalCondition(< cond \ >, os) = (true, os')}{\langle if \ < op \ > \ \#i, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv, h, l, pc' \rangle}$$

evalCondition receives as parameters a boolean condition and an operand stack. It returns the result of the condition's evaluation and the resulting operand stack.

5.5.8 Method Invocation

We suppose to have a global stack `callStack`. It contains the stack of the `invoke` statements and the local states. These states are represented as triples composed by

- the program counter that invokes the method,
- the local state of the operand stack,
- the local state of the local variables.

Statement `return` ends the execution of the current method, getting the control back to the caller or ending the computation possibly. Statement `vreturn` passes the value at the top of the operand stack to the caller, pushing it on the operand stack of the caller. Statement `invoke < method >` invokes the method identified by a given method signature, according to the type of the reference on which it is invoked.

$$\begin{array}{c}
(r, lv', os_1) = \text{extractLV}(os, \langle \text{method} \rangle), \text{callStack} = \text{push}(\text{callStack}, (pc, os_1, lv)), \\
(c, m) = \text{solveDynamicClass}(r, \langle \text{method} \rangle), pc' = (c, m, 0) \\
\hline
\langle \text{invoke } \langle \text{method} \rangle, (os, lv, h, l, pc) \rangle \rightarrow \langle \emptyset, lv', h, l, pc' \rangle \\
\\
(\text{callStack}, (pc_1, os', lv')) = \text{pop}(\text{callStack}) \text{ if } \text{callStack} \neq \emptyset \\
\hline
\langle \text{return}, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv', h, l, \text{next}(pc_1) \rangle \\
\\
(\text{callStack}, (pc_1, os_1, lv')) = \text{pop}(\text{callStack}), \\
v = \text{pop}(os), os' = \text{push}(os_1, v) \text{ if } \text{callStack} \neq \emptyset \\
\hline
\langle \text{vreturn}, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv', h, l, \text{next}(pc_1) \rangle \\
\\
pc = (c, m, i), pc' = (c, m, -1) \text{ if } \text{callStack} = \emptyset \\
\hline
\langle \text{return}, (os, lv, h, l, pc) \rangle \rightarrow \langle os, lv, h, l, pc' \rangle \\
\\
pc = (c, m, i), pc' = (c, m, -1), (os', v) = \text{pop}(os) \text{ if } \text{callStack} = \emptyset \\
\hline
\langle \text{vreturn}, (os, lv, h, l, pc) \rangle \rightarrow \langle os', lv, h, l, pc' \rangle
\end{array}$$

extractLV : [(Op × MSig) → (Ref × LV × Op)] receives as parameters an operand stack and the signature of a method. It returns

- the reference on which the method is called,
- the local variables at the beginning of the called method, i.e. with the reference to this at index 0, followed by the arguments of the method,
- the operand stack obtained after the extraction of the arguments.

solveDynamicClass : [(Ref × MSig) → (C × M)] receives as parameters a reference and its signature. It returns the called method and the class to which it belongs.

5.5.9 Applying it to the Example

Table 5.1 depicts the complete execution of the example introduced in Section 5.3 graphically. For each state, we show its four components: on the left there is the operand stack, then the local variables, the heap, and finally the function that traces the owned monitors. We do not trace the program counters, as it is a sequential piece of code without nor jumps neither if statements. In this representation, we suppose that the address of this, which is stored at index 0 of local variables according to the Java Virtual Machine Specification, is #0. Note that the value of the parameter amount is passed through index 1 of local variables.

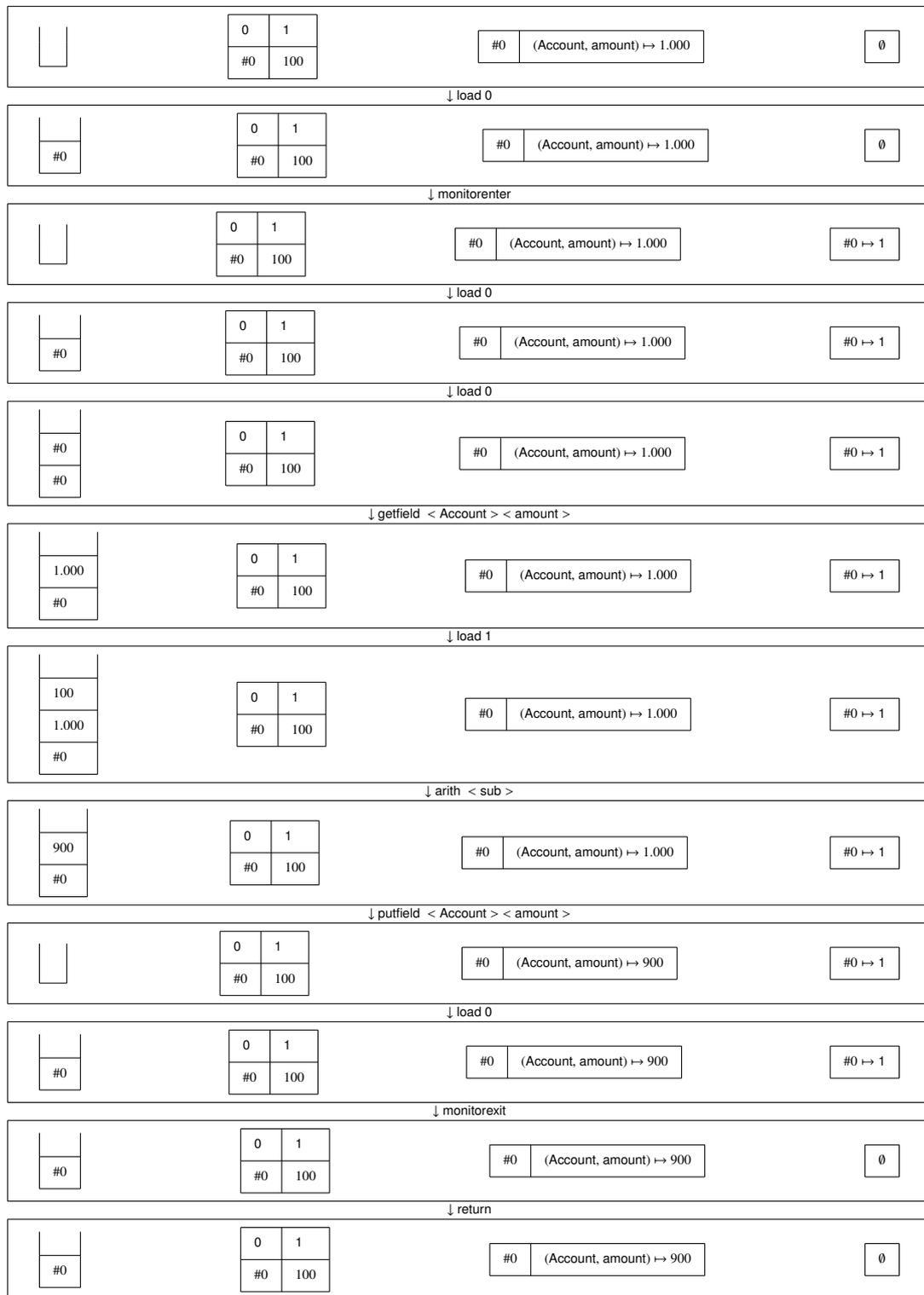


Table 5.1: The concrete execution of the example

5.6 Control Flow Graph

So far, we described executions as traces of states. Since bytecode programs are not structured, we have to build up a structured representation of their execution. The aim of this section is to formalize the construction of executions on the control flow graph as an abstraction of the trace of execution. Note that all the targets of branch instructions can be statically determined. So we do not need to take into account indirect branches. Otherwise, if the target of jump and conditionals statements were not statically determined, we would require an ad-hoc static analysis in order to reconstruct the control flow graph [78]. This is not necessary on the language defined in Section 5.2.

5.6.1 Formal Definition

Let St be the set of bytecode statements. A program is represented as a trace of statements, i.e. $P = St^{\dagger}$. The control flow graph is composed by an array of blocks (i.e. traces of statements) and by a set of edges that relate blocks.

DEFINITION 5.6.1 (CONTROL FLOW GRAPH) *A control flow graph is composed by*

- *an array of sequential blocks ($V = \mathbb{A}\mathbb{R}(St^{\dagger})$),*
- *a set of edges linking different blocks ($E = \mathbb{N} \times \mathbb{N} \times \langle \text{cond} \rangle$).*

Each block is identified by its position in the array. The third component of edges stores the condition required in order to cross this edge.

$$CFG = V \times E$$

Since the outgoing edges can be defined only by jump statements, we need to consider three cases:

- the last statement of the block is `return` or `vreturn`. In this case there is no outgoing edge;
- the last statement of the block is `goto`. In this case we have only one outgoing edge in which the condition is true;
- the last statement of the block is `if < cond >`. In this case we have two outgoing edges. One of them points to the next block (in the case the condition is evaluated to false), while the other one points to a given index (if the condition is evaluated to true).

As the indexes of `if` and `goto` instructions are statically known, we can build up this graph before analyzing the program. This information is provided by function *extractCFG* : $St^{\dagger} \rightarrow CFG$. Then, we may disregard `if` and `goto` statements, as they are represented by edges in the control flow graph.

An execution on a control flow graph is represented as an array of set of traces. Since the program may contain loops, the same block may be executed more than once. This is why each block is related to a set of its executions.

DEFINITION 5.6.2 (CONTROL FLOW GRAPH EXECUTION) *An execution on a control flow graph is represented by the control flow graph itself and an array of set of traces of executions.*

$$\text{exCFG} = \text{CFG} \times \text{AR}(\wp(\Sigma^{\mp}))$$

5.6.2 Soundness with respect to $\langle \wp(\Sigma^{\mp}), \subseteq \rangle$

The partial order operator simply applies the subset ordering on all the elements of the two executions of the control flow graph.

DEFINITION 5.6.3 (ORDERING OPERATOR \sqsubseteq_{CFG})

$$\begin{aligned} (\text{cfg}_1, \text{ex}_1) \sqsubseteq_{\text{CFG}} (\text{cfg}_2, \text{ex}_2) = \text{true iff} \\ \forall i \in \text{dom}(\text{ex}_1) : i \in \text{dom}(\text{ex}_2) \wedge \text{ex}_1(i) \subseteq \text{ex}_2(i) \end{aligned}$$

LEMMA 5.6.4 $\langle \text{exCFG}, \sqsubseteq_{\text{CFG}} \rangle$ forms a complete lattice

PROOF. The proof follows trivially from the fact that \sqsubseteq_{CFG} is the pointwise application of \subseteq . ■

LEMMA 5.6.5 $\langle \wp(\Sigma^{\mp}), \subseteq \rangle$ forms a complete lattice

Starting from an execution of a control flow graph we can build up the set of all the possible executions represented by it.

DEFINITION 5.6.6 (CONCRETIZATION γ_{CFG})

$$\begin{aligned} \gamma_{\text{CFG}} : [\text{exCFG} \rightarrow \wp(\Sigma^{\mp})] \\ \gamma_{\text{CFG}}((b, v), \text{ex}) = \{ \tau_0 \rightarrow \dots \rightarrow \tau_n : \\ (1) \quad \tau_0 \in \text{ex}(0), \\ (2) \quad \tau_n \in \text{ex}(k) : k \in \text{dom}(\text{ex}), \nexists (i_1, i_2) \in v : i_1 = k, \\ (3) \quad \forall i \in [1..n-1] : \exists j \in \text{dom}(\text{ex}) : \tau_i \in \text{ex}(j), \exists j_1, j_2 \in \text{dom}(\text{ex}) : \\ \quad \tau_{i-1} \in \text{ex}(j_1), \tau_{i+1} \in \text{ex}(j_2), (j_1, j) \in v, (j, j_2) \in v \\ \} \end{aligned}$$

- (1) formalizes that the block 0 is the entry point of the control flow graph.
- (2) means that the last part of the trace is the execution of an exit-point block.
- (3) formalizes that all the intermediate traces must be obtained as the concretization of a block such that there is

- an edge from the previous block to the current one,
- an edge from the current one to the next one.

LEMMA 5.6.7 γ_{CFG} is a complete \sqcap_{CFG} -morphism, where \sqcap_{CFG} is the lower bound operator induced by \sqsubseteq_{CFG}

THEOREM 5.6.8 $\langle \wp(\Sigma^\ddagger), \sqsubseteq \rangle \xleftrightarrow[\alpha_{\text{CFG}}]{\gamma_{\text{CFG}}} \langle \text{exCFG}, \sqsubseteq_{\text{CFG}} \rangle$ where

$$\begin{aligned} \alpha_{\text{CFG}} &: [\wp(\Sigma^\ddagger) \rightarrow \text{exCFG}] \\ \alpha_{\text{CFG}} &= \lambda T. \sqcap_{\text{CFG}} \{ \text{ex} : T \sqsubseteq \gamma(\text{ex}) \} \end{aligned}$$

PROOF. By Lemma 5.6.7 γ_{CFG} is a complete \sqcap_{CFG} -morphism. Therefore, by applying Theorem 2.2.3 we get that $\langle \wp(\Sigma^\ddagger), \sqsubseteq \rangle \xleftrightarrow[\alpha_{\text{CFG}}]{\gamma_{\text{CFG}}} \langle \text{exCFG}, \sqsubseteq_{\text{CFG}} \rangle$. ■

In this way, we provide a sound abstraction of the possible executions of a program as executions on a control flow graph.

5.7 Method Calls

Another level of abstraction is the representation of method calls as control flow graph executions. When we represent the executions as traces, method calls are represented in this way too. For instance, suppose to have executed a trace $\sigma_0 \rightarrow \dots \rightarrow \sigma_i$. Then the analysis of a method call produces a trace $\sigma'_0 \rightarrow \dots \rightarrow \sigma'_i$. Finally, we go on with the execution with $\sigma_k \rightarrow \dots$. This execution is represented by the trace $\sigma_0 \rightarrow \dots \rightarrow \sigma_i \rightarrow \sigma'_0 \rightarrow \dots \rightarrow \sigma'_i \rightarrow \sigma_k \rightarrow \dots$.

When we deal with a control flow graph of a program (and so also the body of a method) we represent the invocation of a method as a control flow graph and an array of set of traces, i.e. with an element in exCFG .

Given a trace representing an execution, we can build up a function *translateMethodCall* : $[\wp(\Sigma^\ddagger) \rightarrow \wp(\text{exCFG} \cup \Sigma^\ddagger)]$. This represents each method call as an element in exCFG . Intuitively, once this function detects one method call, it can extract the trace that represents its execution, and it can abstract the execution through α_{CFG} .

5.8 Abstract Domain

While abstracting the concrete domain presented in Section 5.4, the most important issue is the analysis of addresses through an ad-hoc alias analysis. For other components, the abstract domain is the simple approximation of its concrete counterpart.

5.8.1 Alias Analysis

In order to obtain an effective analysis of Java multithreaded programs we need to precisely trace

- when two accesses on the shared memory may be on the same location,
- when two threads are always synchronized on the same monitors.

In Java the shared memory is the heap. It relates references to objects, arrays, or strings. Monitors are defined on objects, and threads are objects. So they are both identified by reference.

In this context alias analysis (i.e. the way in which we abstract references) is the critical point of our analysis. We need to precisely check

- when two references always point to the same location (must-aliasing),
- when two references may point to the same location (may-aliasing).

Must-Aliasing

In order to check when two references point to the same location, we link each abstract reference to an equivalence class. Two values related to the same equivalence class contain the same value at that point in all possible executions of the program. Each time we analyze a new statement we create a new equivalence class. When we make the join, if a variable points to two different equivalence classes in the two branches we instantiate a new equivalence class. Note that in this way we have a loss of precision: two variables may be equal in both branches but through different equivalence classes. When we join them, we lose this relation. We may refine our operator but this would increase its complexity. In fact, for each variable we would have to check all the variables that point to the same equivalence class. On the other hand this operator may be easily optimized. On the examples on which we tested our analysis we did not need to refine and optimize the join operator.

Formally, we denote by \bar{E} the set of the equivalence classes on references. The function $\overline{fresh}()$ returns a new equivalence class. The join operator $\sqcup_{\bar{E}}$ is defined as:

$$\bar{e}_1 \sqcup_{\bar{E}} \bar{e}_2 = \begin{cases} \bar{e}_1 & \text{iff } \bar{e}_1 = \bar{e}_2 \\ \overline{fresh}() & \text{otherwise} \end{cases}$$

In order to define the ordering operator, we need to work on a boxed domain in which each variable is related to an equivalence class. Let be $\bar{f}_1, \bar{f}_2 \in [\text{Var} \rightarrow \bar{E}]$, where Var is the set of the variables of the program. Then $\bar{f}_1 \leq_{\bar{E}} \bar{f}_2$ iff $\forall v_1, v_2 \in \text{dom}(\bar{f}_2) : \bar{f}_2(v_1) = \bar{f}_2(v_2) \Rightarrow v_1, v_2 \in \text{dom}(\bar{f}_1) \wedge \bar{f}_1(v_1) = \bar{f}_1(v_2)$.

The concretization function must be defined on $[\text{Var} \rightarrow \bar{E}]$ too.

$$\gamma_{\bar{E}}(\bar{f}) = \{f : \forall v_1, v_2 \in \text{dom}(\bar{f}) : \bar{f}(v_1) = \bar{f}(v_2) \Rightarrow f(v_1) = f(v_2)\}$$

A special equivalence class is reserved to null pointers.

May-Aliasing

The basic idea of our may-alias analysis is to represent all the possible concrete references with a finite set of abstract references. Then we link them to the point of the program that instantiated them. This approach is similar to the one adopted in [42].

DEFINITION 5.8.1 (MAY-ALIASING) *The may-aliasing represents references as triples composed by*

- *the program counter that created the reference;*
- *the stack of the called methods. Each method call is represented by the program counter of the invoke statement. The stack is reduced in order to approximate recursive calls with the same abstract reference. Intuitively, each time we found twice the same program counter in the stack, this means that we are analyzing a recursive method. So we project the stack to the first method call, tracing that we are analyzing a recursive method;*
- *the abstract reference of the thread that instantiated the reference.*

Formally, $\bar{D} = (\text{PC} \times \text{ST}(\text{PC}) \times \bar{D}) \cup \{\#\text{mainthread}\}$.

This definition is recursive as each abstract reference contains another abstract reference representing the thread that instantiated it. The set of abstract references contains also the special element $\#\text{mainthread}$ that represents the main thread. We need this element as the main thread is instantiated by the system and not by another thread. In addition all the threads are started directly or indirectly from it, and this guarantees that recursion is not possible. Intuitively, the creation of threads can be represented as a tree. In fact, each thread is created by another one (except the main thread, that is the root of the tree), and it can create a set of threads. In addition, loops are not possible, as you cannot have that a thread t_1 creates a thread t_2 which creates t_1 . So recursion is not possible.

A value may contain references created at different program points, e.g. because of a non deterministic if statement. Then the may-alias analysis represents an abstract reference as a set of these triples. In this way the ordering, join and meet operators are the ones of sets. The set of program counters is finite and the stack of the called methods is reduced in order to avoid recursive calls. So the set of abstract references is finite. Finally the lattice satisfies the ascending chain condition, and so we do not need to define a widening operator.

The concretization function returns all the possible concrete addresses that may be created by the given program counter following the given call stack (and all its possible recursive extensions).

Static references: This aliasing domain has to be extended in order to support static references. Since static fields are initialized by the system before the launch of the application calling the static constructors of the classes, we need to define specific abstract

references for them. In this context, we augment our may aliasing domain adding an abstract address for each class identified by the class itself. This address is used to store the information on static fields and when static methods are invoked.

DEFINITION 5.8.2 (MAY-ALIASING OF STATIC REFERENCES) *The set of abstract static references \overline{S} is represented by the set of classes: $\overline{S} = C$.*

The concretization function $\gamma_{\overline{S}}$ simply returns the concrete references pointing to the static object of the given class.

DEFINITION 5.8.3 (MAY-ALIASING) *The may-aliasing domain is the union between the may-aliasing of dynamic and static references: $\overline{P} = \overline{D} \cup \overline{S} \cup \text{null}$. null element is used in order to represent null pointers.*

DEFINITION 5.8.4 (MAY-ALIASING LATTICE) *The may-aliasing lattice is the one composed by a set of elements of the may-aliasing domain using the common set operators: $\langle \wp(\overline{P}), \subseteq, \emptyset, \overline{P}, \cup, \cap \rangle$.*

Abstract References

The abstract domain represents references as the Cartesian product of may- and must-aliasing domains.

DEFINITION 5.8.5 (ABSTRACT REFERENCES) $\overline{\text{Ref}} = \overline{E} \times \overline{P}$

5.8.2 Domain

Our analysis is parameterized by a non-relational numerical domain.

PROPOSITION 5.8.6 (ABSTRACT NUMERICAL DOMAIN) *We suppose that a non-relational numerical domain $\overline{\text{Num}}$ is given. It has to approximate concrete numerical values soundly.*

$$\langle \wp(\mathbb{N}), \subseteq \rangle \xleftrightarrow[\alpha_{\overline{\text{Num}}}]^{\gamma_{\overline{\text{Num}}}} \langle \overline{\text{Num}}, \subseteq_{\overline{\text{Num}}} \rangle$$

DEFINITION 5.8.7 (VALUES) $\overline{\text{Val}} = \overline{\text{Ref}} \cup \overline{\text{Num}}$

DEFINITION 5.8.8 (OPERAND STACK) $\overline{\text{Op}} = \mathcal{ST}(\overline{\text{Val}})$

DEFINITION 5.8.9 (LOCAL VARIABLES) $\overline{\text{LV}} = \text{AR}(\overline{\text{Val}})$

DEFINITION 5.8.10 (MONITORS) $\overline{\text{L}} : [\overline{\text{Ref}} \rightarrow \mathbb{N}]$

DEFINITION 5.8.11 (OBJECTS) $\overline{\text{Obj}} : [(C \times F) \rightarrow \overline{\text{Val}}]$

Analyzing precisely arrays is an orthogonal issue with respect to our goal. It deserves to be considered separately. Nevertheless, we implement a simple analysis of arrays in which all the cells are abstracted into a unique abstract value. In addition we trace also the abstract length of the array. This minimal approach allows us to deal with programs that contain arrays. For the properties on which we are interested, it seems not to afflict the precision of the analysis.

DEFINITION 5.8.12 (ARRAYS' STATE) *The abstract state of an array is a pair composed by an abstract value (approximating the values of all the cells of the array) and an abstract numerical value (approximating the length of the array) : $\overline{\text{Arr}} = \overline{\text{Val}} \times \overline{\text{Num}}$.*

DEFINITION 5.8.13 (STRINGS) $\overline{\text{Str}} = \text{A.R.}(\overline{\text{Num}}) \times \overline{\text{Num}}$

DEFINITION 5.8.14 (HEAP) *The abstract heap relates elements of may-alias domain to objects or arrays: $\overline{\text{H}} : [\overline{\text{P}} \rightarrow (\overline{\text{Obj}} \cup \overline{\text{Arr}} \cup \overline{\text{Str}})]$*

DEFINITION 5.8.15 (SINGLE-THREAD STATE) $\overline{\Sigma} = \overline{\text{Op}} \times \overline{\text{LV}} \times \overline{\text{H}} \times \overline{\text{L}} \times \text{PC}$

5.9 Abstract Operational Semantics

The abstract operational semantics is mostly the abstraction of the concrete definition presented in Section 5.5.

5.9.1 Load and Store

$$\frac{\overline{\text{os}}' = \text{push}(\overline{\text{os}}, \overline{\text{lv}}(i))}{\langle \text{load } \#i, (\overline{\text{os}}, \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{l}}, \text{pc}) \rangle \rightarrow \langle \overline{\text{os}}', \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{l}}, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\overline{\text{os}}', \overline{\text{v}}) = \text{pop}(\overline{\text{os}}), \overline{\text{lv}}' = \overline{\text{lv}}[i \mapsto \overline{\text{v}}]}{\langle \text{store } \#i, (\overline{\text{os}}, \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{l}}, \text{pc}) \rangle \rightarrow \langle \overline{\text{os}}', \overline{\text{lv}}', \overline{\text{h}}, \overline{\text{l}}, \text{next}(\text{pc}) \rangle}$$

5.9.2 Monitors

$$\frac{(\overline{\text{os}}', \overline{\text{r}}) = \text{pop}(\overline{\text{os}}), \overline{\text{lv}}' = \overline{\text{lv}}[\overline{\text{r}} \mapsto \overline{\text{l}}(\overline{\text{r}}) + 1] \text{ if } \overline{\text{r}} \in \text{dom}(\overline{\text{l}})}{\langle \text{monitorenter}, (\overline{\text{os}}, \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{l}}, \text{pc}) \rangle \rightarrow \langle \overline{\text{os}}', \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{lv}}', \text{next}(\text{pc}) \rangle}$$

$$\frac{(\overline{\text{os}}', \overline{\text{r}}) = \text{pop}(\overline{\text{os}}), \overline{\text{lv}}' = \overline{\text{lv}}[\overline{\text{r}} \mapsto 1] \text{ if } \overline{\text{r}} \notin \text{dom}(\overline{\text{l}})}{\langle \text{monitorenter}, (\overline{\text{os}}, \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{l}}, \text{pc}) \rangle \rightarrow \langle \overline{\text{os}}', \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{lv}}', \text{next}(\text{pc}) \rangle}$$

$$\frac{(\overline{\text{os}}', \overline{\text{r}}) = \text{pop}(\overline{\text{os}}), \overline{\text{l}}(\overline{\text{r}}) = n, \overline{\text{lv}}' = \overline{\text{lv}}[\overline{\text{r}} \mapsto n - 1] \text{ if } n > 1}{\langle \text{monitorexit}, (\overline{\text{os}}, \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{l}}, \text{pc}) \rangle \rightarrow \langle \overline{\text{os}}', \overline{\text{lv}}, \overline{\text{h}}, \overline{\text{lv}}', \text{next}(\text{pc}) \rangle}$$

$$\frac{(\overline{os'}, \bar{r}) = pop(\overline{os}), \bar{l}(\bar{r}) = n, \bar{l}' = \bar{l} \setminus \{\bar{r} \mapsto n\} \text{ if } n = 1}{\langle \text{monitorexit}, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, pc) \rangle \mapsto \langle \overline{os'}, \bar{lv}, \bar{h}, \bar{l}', next(pc) \rangle}$$

Note that since our abstract references are an approximation of the concrete ones, we may be not able to unlock monitors precisely. For instance, consider the following piece of code:

```

if (i>0)
    lock(a)
else lock(b);
//do something without modifying i, a, and b
if (i>0)
    unlock(a)
else unlock(b);

```

If at the abstract level we would not be able to precisely check the condition $i > 0$, we may be not able to check that we always release a monitor previously locked. In this case, we should release all the owned monitors in order to preserve the soundness of the approach. On the other hand, we never found such a case when analyzing a bytecode obtained by compiling Java code through javacc. At bytecode level `monitorexit` always deals with monitors that can be trivially proved to be acquired by a previous `monitorenter` statement.

5.9.3 Objects

$$\frac{(\bar{r}, \bar{h}') = \overline{alloc}(\langle \text{class } \rangle, \bar{h}, pc), \overline{os}' = push(\overline{os}, \bar{r})}{\langle \text{new } \langle \text{class } \rangle, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, pc) \rangle \mapsto \langle \overline{os}', \bar{lv}, \bar{h}', \bar{l}, next(pc) \rangle}$$

$$\frac{(\overline{os}_1, (\bar{R}, \bar{e})) = pop(\overline{os}), \bar{v} = \bigsqcup_{\bar{r} \in \bar{R}} \bar{h}(\bar{r})(\langle \text{class } \rangle, \langle \text{id } \rangle), \overline{os}' = push(\overline{os}_1, \bar{v})}{\langle \text{getfield } \langle \text{class } \rangle \langle \text{id } \rangle, (\overline{op}, \bar{lv}, \bar{h}, \bar{l}, pc) \rangle \mapsto \langle \overline{op}', \bar{lv}, \bar{h}, \bar{l}, next(pc) \rangle}$$

$$\frac{(\overline{os}_1, \bar{v}) = pop(\overline{os}), (\overline{os}', (\bar{e}, \bar{R})) = pop(\overline{os}_1), \bar{obj} = \bar{h}(\bar{r}), \bar{h}' = \bar{h}[\{\bar{r} \mapsto \bar{obj}' : \bar{r} \in \bar{R}, \bar{obj}' = \bar{h}(\bar{r})(\langle \text{class } \rangle, \langle \text{id } \rangle) \mapsto \bar{v}\}], \text{ if } \bar{R} = \{\bar{r}\} \wedge \overline{isSingle}(\bar{r}) = \text{true}}{\langle \text{putfield } \langle \text{class } \rangle \langle \text{id } \rangle, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, pc) \rangle \mapsto \langle \overline{os}', \bar{lv}, \bar{h}', \bar{l}, next(pc) \rangle}$$

$$\frac{(\overline{os}_1, \bar{v}) = pop(\overline{os}), (\overline{os}', (\bar{e}, \bar{R})) = pop(\overline{os}_1), \bar{obj} = \bar{h}(\bar{r}), \bar{h}' = \bar{h}[\{\bar{r} \mapsto \bar{obj}' : \bar{r} \in \bar{R}, \bar{obj}' = \bar{h}(\bar{r})(\langle \text{class } \rangle, \langle \text{id } \rangle) \mapsto \bar{h}(\bar{r})(\langle \text{class } \rangle, \langle \text{id } \rangle) \sqcup_{\text{Val}} \bar{v}\}], \text{ if } \bar{R} \neq \{\bar{r}\} \vee \overline{isSingle}(\bar{r}) = \text{false}}{\langle \text{putfield } \langle \text{class } \rangle \langle \text{id } \rangle, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, pc) \rangle \mapsto \langle \overline{os}', \bar{lv}, \bar{h}', \bar{l}, next(pc) \rangle}$$

$\overline{alloc} : [(C \times \bar{H} \times PC) \rightarrow (\bar{Ref} \times \bar{H})]$ receives as parameters a class, a state of the heap, and a program counter. It allocates an abstract location containing the abstraction of the default values for all the fields of the given class. Finally it returns a fresh reference that

points to this location in the returned heap.

$\overline{isSingle} : [\overline{P} \rightarrow \{\text{true}, \text{false}\}]$ is a function that, given an abstract element of our may aliasing domain, returns true iff it represents exactly one concrete reference. Since each abstract element is related to a specific program counter and to the stack of method called, we can check if this allocation may be inside a loop or if it is inside a recursive context. If both these condition are false, we prove statically that the abstract reference approximates exactly one concrete address, and so $\overline{isSingle}$ returns true.

5.9.4 Arrays

$$\frac{\begin{array}{l} (\overline{os}_1, \bar{i}) = \text{pop}(\overline{os}), (\bar{r}, \bar{h}') = \overline{allocArray}(\bar{i}, \bar{h}), \overline{os}' = \text{push}(\overline{os}_1, \bar{r}) \\ \langle \text{newarray}, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, \text{pc}) \rangle \rightarrow \langle \overline{os}', \bar{lv}, \bar{h}', \bar{l}, \text{next}(\text{pc}) \rangle \end{array}}{\begin{array}{l} (\overline{os}_1, \bar{i}) = \text{pop}(\overline{os}), (\overline{os}_2, (\bar{R}, \bar{e})) = \text{pop}(\overline{os}_1), \bar{v} = \bigsqcup_{\bar{r} \in \bar{R}} (\pi_1(\bar{h}(\bar{r}))), \overline{os}' = \text{push}(\overline{os}_1, \bar{v}) \\ \langle \text{aload}, (\overline{op}, \bar{lv}, \bar{h}, \bar{l}, \text{pc}) \rangle \rightarrow \langle \overline{op}', \bar{lv}, \bar{h}, \bar{l}, \text{next}(\text{pc}) \rangle \end{array}} \\ \frac{\begin{array}{l} (\overline{os}_1, \bar{v}) = \text{pop}(\overline{os}), (\overline{os}_2, \bar{i}) = \text{pop}(\overline{os}_1), (\overline{os}', (\bar{R}, \bar{e})) = \text{pop}(\overline{os}_1), \\ \bar{h}' = \bar{h}[\{\bar{r} \mapsto \bar{ar}' : \bar{r} \in \bar{R}, \bar{ar}' = (\pi_1(\bar{ar}'(\bar{r})) \sqcup \bar{v}, \pi_2(\bar{ar}'(\bar{r})))\}] \end{array}}{\langle \text{astore}, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, \text{pc}) \rangle \rightarrow \langle \overline{os}', \bar{lv}, \bar{h}', \bar{l}, \text{next}(\text{pc}) \rangle}$$

$\overline{allocArray} : [(\overline{\text{Num}} \times \overline{\text{H}}) \rightarrow (\overline{\text{Ref}} \times \overline{\text{H}})]$ receives as parameters an abstract numerical value representing a length and a heap. It returns an abstract reference and the heap that relates this reference to an array of the given length.

5.9.5 Arithmetic Expressions

$$\frac{\begin{array}{l} (\overline{os}_1, (\bar{i}_1)) = \text{pop}(\overline{os}), (\overline{os}_2, \bar{i}_2) = \text{pop}(\overline{os}_1), \\ \overline{os}' = \text{push}(\overline{os}_2, \overline{evalExprAr}(\bar{i}_2, \bar{i}_1, < \text{op} >)) \end{array}}{\langle \text{arith } < \text{op} >, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, \text{pc}) \rangle \rightarrow \langle \overline{os}', \bar{lv}, \bar{h}, \bar{l}, \text{next}(\text{pc}) \rangle}$$

$\overline{evalExprAr}$ receives as parameters two abstract numerical values and a binary arithmetic operator. It returns the result of the abstract execution of this arithmetic operation. We suppose that this evaluation is sound, i.e.

$$\alpha_{\overline{\text{Num}}}(\bar{i}_1 < \text{op} > \bar{i}_2) \sqsubseteq_{\overline{\text{Num}}} \overline{evalExprAr}(\alpha_{\overline{\text{Num}}}(\bar{i}_1), \alpha_{\overline{\text{Num}}}(\bar{i}_2), < \text{op} >)$$

5.9.6 Constants

$$\frac{\overline{os}' = \text{push}(\overline{os}, \overline{evalConst}(< \text{val} >)), \text{ if } < \text{val} > \text{ is a numerical value}}{\langle \text{const } < \text{val} >, (\overline{os}, \bar{lv}, \bar{h}, \bar{l}, \text{pc}) \rangle \rightarrow \langle \overline{os}', \bar{lv}, \bar{h}, \bar{l}, \text{next}(\text{pc}) \rangle}$$

$$\frac{\overline{os'} = \text{push}(\overline{os}, \text{null}), \text{ if } \langle \text{val} \rangle = \text{null}}{\langle \text{const } \langle \text{val} \rangle, (\overline{os}, \overline{lv}, \overline{h}, \overline{l}, \text{pc}) \rangle \rightarrow \langle \overline{os'}, \overline{lv}, \overline{h}, \overline{l}, \text{next}(\text{pc}) \rangle}$$

$$\frac{(\overline{r}, \overline{h'}) = \overline{\text{allocString}}(\langle \text{val} \rangle, \overline{h}), \overline{os'} = \text{push}(\overline{os}, \overline{r}), \text{ if } \langle \text{val} \rangle \text{ is a string}}{\langle \text{const } \langle \text{val} \rangle, (\overline{os}, \overline{lv}, \overline{h}, \overline{l}, \text{pc}) \rangle \rightarrow \langle \overline{os'}, \overline{lv}, \overline{h'}, \overline{l}, \text{next}(\text{pc}) \rangle}$$

$\overline{\text{allocString}} : [(\text{Str} \times \overline{\text{H}}) \rightarrow (\overline{\text{Ref}} \times \overline{\text{H}})]$ receives as parameters a string and an abstract heap. It returns an abstract reference and the heap that relates it to the abstract representation of the given string.

$\overline{\text{evalCondition}} : [\mathbb{N} \rightarrow \overline{\text{Num}}]$ is a function that given a numerical value returns its abstract representation.

5.9.7 Jumps, If and Method Calls

Our abstract semantics is defined on the control flow graph depicted in Sections 5.6 and 5.7. In this way we do not need to deal with goto, if statements, and method calls.

5.9.8 Applying it to the Example

Table 5.2 depicts the results of the abstract analysis of the example presented in Section 5.3 using the Interval domain. Since we are analyzing this method when the current object has been already allocated, we represent its abstract reference by $(\{\overline{r}\}, \overline{0})$. We suppose that $\overline{\text{isSingle}}(\overline{r}) = \text{false}$, e.g. because the program counter of the new statement that allocated \overline{r} is inside a loop. So when we analyze putfield we perform a weak assignment. At the end of the computation in the abstract the field (Account, amount) is related to the interval [900..1.000]. So we obtain a final result approximated with respect to the concrete result because of this weak assignment.

5.10 Soundness

5.10.1 Domain

Thanks to our low-level approach, the abstraction function can be defined as the pointwise applications of the abstraction of numerical values and of references. In general we define by $\alpha_{\mathcal{S}}$ this function when applied to elements in $\overline{\mathcal{S}}$.

THEOREM 5.10.1 (SOUNDNESS OF $\langle \overline{\Sigma}, \overline{\sqsubseteq}_{\Sigma} \rangle$) $\langle \overline{\Sigma}, \overline{\sqsubseteq}_{\Sigma} \rangle$ is a sound approximation of $\langle \wp(\Sigma), \sqsubseteq \rangle$, i.e.

$$\langle \wp(\Sigma), \sqsubseteq \rangle \xrightleftharpoons[\alpha_{\Sigma}]{\gamma_{\Sigma}} \langle \overline{\Sigma}, \overline{\sqsubseteq}_{\Sigma} \rangle$$

PROOF. For numerical values, Proposition 5.8.6 guarantees that our non-relational numerical domain is sound. For references, the abstraction of may-aliasing domain can be easily built up by tracing for each concrete address, which threads and statements create it, and

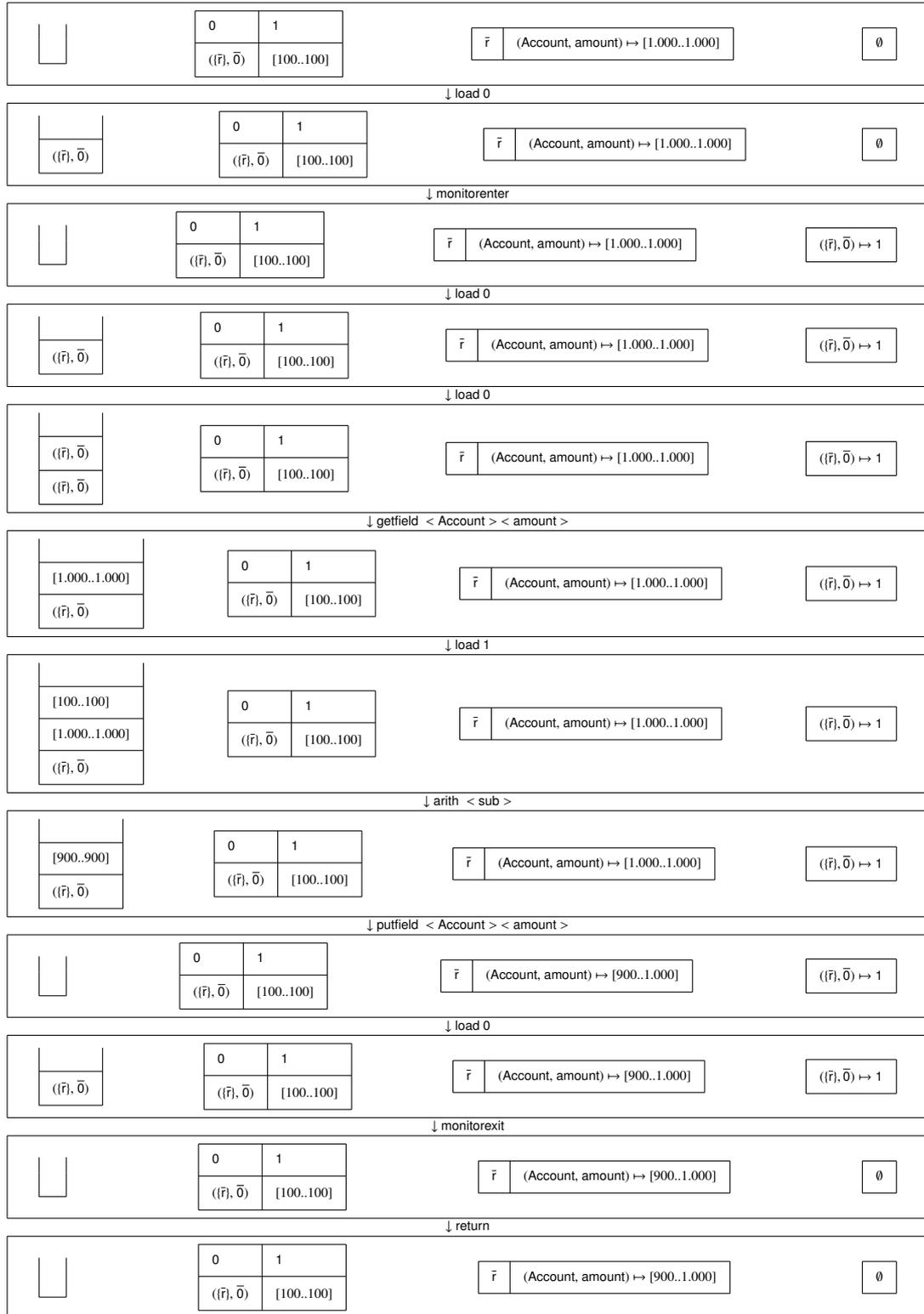


Table 5.2: The abstract analysis of the example

which was the call stack at that point of the computation (possibly projecting it in order to avoid recursion). $\alpha_{\bar{p}}$ represents this abstraction function. Equivalence classes of the must-aliasing domain can be abstracted checking which concrete references are equal. The abstraction function on operand stacks, local variables, heaps, etc.. are the pointwise application of these two abstraction functions.

The same situation occurs for the upper bound operator, that can be obtained as the pointwise application of upper bound operators of numerical values and references. As for abstraction functions, \sqsubseteq_S and \sqcup_S represents respectively the partial order and the upper bound operators on elements of the set \bar{S} .

As abstraction functions and upper bound operators rely on sound operators on references and numerical values, we get that our abstract domain forms a Galois connection with respect to its concrete counterpart. ■

5.10.2 Semantics

We need to prove the soundness of our abstract semantics when reading from and writing on objects. The other cases can be trivially proved as they are obtained as the abstraction of the concrete definition, or rely on the soundness of the numerical domain.

5.10.3 Objects

LEMMA 5.10.2 (SOUNDNESS OF `getfield`) $\overrightarrow{}$ is sound with respect to \rightarrow when applied to `getfield`, i.e.

$$\forall \sigma \in \Sigma : \alpha_{\Sigma}(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_{\Sigma} \overline{\sigma'} : \alpha_{\Sigma}(\{\sigma\}) \overrightarrow{} \overline{\sigma'}$$

PROOF. By definition of \rightarrow when applied to `getfield` (Section 5.5.3) the local variables, the heap, and the locked monitors are not modified. So we need to focus only on the operand stack.

\rightarrow pops from the operand stack a reference r , and it pushes the value $h(r)(\langle \text{class} \rangle, \langle \text{id} \rangle)$. Finally, the abstraction of this state will approximate the top of the stack with $\alpha_{\text{Val}}(h(r)(\langle \text{class} \rangle, \langle \text{id} \rangle))$

When applying $\overrightarrow{}$ to the abstraction of the initial state (Section 5.9.3), we would have at the top of the operand stack the value $\bigsqcup_{\bar{r} \in \alpha_{\bar{p}}(r)} (\alpha_{\text{H}}(h)(\bar{r})(\langle \text{class} \rangle, \langle \text{id} \rangle))$, where $\alpha_{\bar{p}}$ is the abstraction function of our may-aliasing domain. $\alpha_{\bar{p}}$ and α_{H} are the pointwise application of sound abstractions of numerical values and references. As we take the upper bound of all the possible abstract references, we have that

$$\alpha_{\text{Val}}(h(r)(\langle \text{class} \rangle, \langle \text{id} \rangle)) \sqsubseteq_{\text{Val}} \bigsqcup_{\bar{r} \in \alpha_{\bar{p}}(r)} (\alpha_{\text{H}}(h)(\bar{r})(\langle \text{class} \rangle, \langle \text{id} \rangle))$$

As \sqsubseteq_{Σ} is the pointwise application of the ordering of numerical values and references, and the other components of the state are not modified, we proved that

$$\forall \sigma \in \Sigma : \alpha_{\Sigma}(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_{\Sigma} \overline{\sigma'} : \alpha_{\Sigma}(\{\sigma\}) \overrightarrow{} \overline{\sigma'}$$

when working on getfield. ■

LEMMA 5.10.3 (SOUNDNESS OF putfield) $\overrightarrow{\quad}$ is sound with respect to \rightarrow when applied to putfield, i.e.

$$\forall \sigma \in \Sigma : \alpha_{\Sigma}(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_{\Sigma} \overline{\sigma'} : \alpha_{\Sigma}(\{\sigma\}) \overrightarrow{\quad} \overline{\sigma'}$$

PROOF. By definition of \rightarrow when applied to putfield (Section 5.5.3) the local variables, and the locked monitors are not modified. So we need to focus only on the operand stack and the heap.

\rightarrow pops from the operand stack a value v and a reference r , and it assigns the value to the given field of the given reference, i.e. $\text{obj}' = \text{obj}[(< \text{class} >, < \text{id} >) \mapsto v]$, $h' = h[r \mapsto \text{obj}']$. Finally, the abstraction of this state will contain the abstraction of the popped value assigned the given field of the object pointed by the abstraction of the given reference.

When applying $\overrightarrow{\quad}$ to the abstraction of the initial state (Section 5.9.3), we need to distinguish two cases:

- $\alpha_{\overline{\text{p}}}(r) = \{\overline{r}\} \wedge \overline{\text{isSingle}}(\overline{r}) = \text{true}$: in this case the may-aliasing abstraction of the concrete reference represents exactly one reference. In this way, assigning to $\overline{h}(\overline{r})$ the object that relates the given field to the abstraction of the concrete value is sound.
- $\alpha_{\overline{\text{p}}}(r) \neq \{\overline{r}\} \vee \overline{\text{isSingle}}(\overline{r}) = \text{false}$: in this case the may-aliasing abstraction of the concrete reference may represent more than one concrete references. Then we are not sure on which concrete reference we are assigning. In order to be sound we need to assign to each possible abstract reference the upper bound between the old value and the abstraction of the assigned one, and this is performed by $\overrightarrow{\quad}$.

As \sqsubseteq_{Σ} is the pointwise application of the ordering of numerical values and references, and the other components of the state are not modified, we proved that

$$\forall \sigma \in \Sigma : \alpha_{\Sigma}(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_{\Sigma} \overline{\sigma'} : \alpha_{\Sigma}(\{\sigma\}) \overrightarrow{\quad} \overline{\sigma'}$$

when working on putfield. ■

THEOREM 5.10.4 (SOUNDNESS OF $\overrightarrow{\quad}$ WITH RESPECT TO \rightarrow) $\overrightarrow{\quad}$ is sound with respect to \rightarrow , i.e.

$$\forall \sigma \in \Sigma : \alpha_{\Sigma}(\{\sigma' : \sigma \rightarrow \sigma'\}) \sqsubseteq_{\Sigma} \overline{\sigma'} : \alpha_{\Sigma}(\{\sigma\}) \overrightarrow{\quad} \overline{\sigma'}$$

PROOF. Lemma 5.10.2 and 5.10.3 prove the soundness of $\overrightarrow{\quad}$ respectively on getfield and putfield. When it is applied to load and store statements (Section 5.9.1), it is defined as the application of the abstraction function on its concrete counterpart (Section 5.5.1 and 5.5.2). The soundness of lock and release of monitors (Section 5.9.2) relies on the soundness of $\overline{\text{E}}$ domain. About arithmetic expressions (Section 5.9.5) and constants' evaluation (Section 5.9.6), the soundness is guaranteed as we suppose that the abstract numerical domain correctly approximates the concrete one. Finally, when it is applied to arrays (Section 5.9.4) it is sound as it approximates all the concrete cells with one abstract value, and we consider the upper bound between old and new values when assigning.

So in all the possible cases we have that $\overrightarrow{\quad}$ is sound with respect to \rightarrow . ■

5.11 Related Work

Many generic analyses of Java programs have been proposed recently.

Some of them have been applied at source code level. It is the case of Cibai [91]. Its aliasing analysis is quite similar to our: each `new` statement can allocate at most k abstract references (where k is a parameter of the analysis). This alias domain is sometimes more precise than our may-aliasing, e.g. inside a loop it tracks k abstract addresses for the same `new` statement, while our analysis tracks just one reference. In other cases our may aliasing domain is more precise, e.g. in case of several method calls. In addition, there is no must alias analysis. Cibai is parameterized on a numerical domain, it supports also relational domains, and octagons were implemented in it. Its domain is composed by an environment and a store. Intuitively, these concepts are translated at bytecode level into local variables and heaps. Cibai is applied to the modular analysis of verification of Java classes.

Another analysis at source code level is the one presented by Pollet [111, 112]. It can be tuned with three different alias analyses. It performs an inter-procedural analysis. It abstracts away numerical values. The structure of the domain is similar to our one and to the one of Cibai, but it is more generic as it can be plugged with different abstractions of the store. On the other hand, in our case it is not possible to be generic on it, as we need to deal with the structure of the store and its abstraction in order to develop a static analysis of multithreaded programs.

Clousot is a generic static analyzer that works at MSIL bytecode level. It is parameterized on an abstract numerical domain and on a property, and it has been successfully applied to the analysis of array out-of-bounds accesses [92], of buffer overrun [47], and to a new relational domain [84]. It performs three transformations of the bytecode (i.e. stack elimination, heap abstraction, and expression recovery) [93] in order to obtain a precise analysis. In this way, it results to be more precise than our approach. On the other hand, we cannot abstract the heap, as in this way we would not be able to check when two threads communicate through the shared memory. In addition, it performs an intra-method analysis, relying on the Design by Contract methodology [100] in order to gain precision on method calls, while our analysis is whole-program.

Julia [131] is another generic analyzer at Java bytecode level. It can be plugged with different abstract domains. In this way, the level of parameterization is higher than in our approach: the user can define his domain freely. On the other hand, this requires the user to define it fully, and he cannot focus only on the numerical domain and on the property of interest. In particular, no alias analysis is implemented in Julia, and the plugged domain has to care about references. Julia has been applied to a wide set of properties: information flow analysis [53], escape analysis [64], magic-sets transformation [110], constancy analysis [54], and nullness analysis [132].

Navas, Mendez-Lojo and Hermenegildo propose a language independent analysis [105]. They represent a program through a control flow graph, and they are parameterized both on the semantics of statements and on the abstract domain.

5.12 Application to the Happens-Before Memory Model

In order to apply the static analysis of the happens before memory model introduced in Chapter 3 we need to define on our domain some elements and functions as required by Sections 3.3.1 and 3.4.1.

5.12.1 Concrete Domain

In Java threads are objects and so identified by reference, i.e. $TId = Ref$. In the same way, the shared memory is the heap, i.e. $Sh = H$, and locations are identified by reference, i.e. $Loc = Ref$. Threads can be synchronized on monitors that are defined on objects and so identified by reference, i.e. $Sync = Ref$. The set of states of the concrete domain is Σ , i.e. $St = \Sigma$.

The transition function $\xrightarrow{\circ} : [St \times St \rightarrow \{true, false\}]$ is \rightarrow as defined in Section 5.5. About the functions, they are defined as follows.

DEFINITION 5.12.1 (*shared*)

$$shared((op, lv, h, l, pc)) = h$$

DEFINITION 5.12.2 (*action*)

$$\begin{aligned} action((op, lv, h, l, pc)) &= \perp_a \text{ iff } ns(pc) \notin \{putfield, getfield\} \\ action((op, lv, h, l, pc)) &= (r, l, \perp_v) \text{ iff } ns(pc) = getfield \text{ where } (op', l) = pop(op) \\ action((op, lv, h, l, pc)) &= (w, l, v) \text{ iff } ns(pc) = putfield \text{ where } (op', v) = pop(op), \\ &\quad (op'', l) = pop(op') \end{aligned}$$

where $ns : [PC \rightarrow S]$ is the function that returns the statement pointed by the given program counter.

DEFINITION 5.12.3 (*synchronized*)

$$synchronized((op, lv, h, l, pc)) = dom(l)$$

DEFINITION 5.12.4 (*assign*)

$$assign(h, r, v) = h[r \mapsto v]$$

DEFINITION 5.12.5 (*setshared*)

$$setshared((op, lv, h, l, pc), h') = (op, lv, h', l, pc)$$

5.12.2 Abstract Thread Identifiers

At abstract level we identify threads through the may-alias domain. In Chapter 3 we supposed that the set of threads is exactly the same both in the abstract and in the concrete domains. This is not true in Java, as threads are objects, and they can be dynamically created and launched. An abstract element of our may-aliasing can represent many concrete references, i.e. when we have a `new` statement inside a loop or a recursive method.

In particular, \overline{vis} function is defined as follows:

$$\begin{aligned} \overline{vis} &: [\overline{Tid} \times \overline{Loc} \times \wp(\overline{Sync}) \times \overline{\Psi} \times (\overline{Tid} \times \mathbb{N}) \rightarrow \wp(\overline{Val})] \\ \overline{vis}(t, \bar{l}, \overline{S}, \bar{f}, (t', i')) &= \\ &= \overline{project}(\bar{l}, \overline{suffix}(\bar{f}(t'), i'), \overline{S}) \cup \\ &\{\bar{v} : \bar{v} \in \overline{project}(\bar{l}, \bar{f}(t''), \overline{S}) : t'' \in dom(\bar{f}) \setminus \{t, t'\}\} \end{aligned}$$

Intuitively,

- in the first part of the definition of \overline{vis} ($\overline{project}(\bar{l}, \overline{suffix}(\bar{f}(t'), i'), \overline{S})$) if the abstract identifier (i.e. an element of our may-alias domain) that corresponds to t' approximates many concrete references, we cannot discard the values written in parallel by this thread before the launch of the current thread. In fact, these values may be written in parallel by another thread represented by the same abstract identifier, but that does not launch the current thread;
- in the second part of the definition of \overline{vis} ($\{\bar{v} : \bar{v} \in \overline{project}(\bar{l}, \bar{f}(t''), \overline{S}) : t'' \in dom(\bar{f}) \setminus \{t, t'\}\}$) if the abstract identifier of the current thread t approximates many concrete references, we cannot discard the values written in parallel by t . In fact, these values may be written in parallel by another concrete thread represented by the same abstract identifier, but that is different from the current one.

Applying these considerations, we preserve the soundness of the analysis.

The \overline{vis} can be redefined as follows when applied to this context and using the may-alias abstract domain to identify threads:

$$\begin{aligned} \overline{vis} &: [\overline{P} \times \overline{Loc} \times \wp(\overline{Sync}) \times \overline{\Psi} \times (\overline{P} \times \mathbb{N}) \rightarrow \wp(\overline{Val})] \\ \overline{vis}(\bar{t}, \bar{l}, \overline{S}, \bar{f}, (\bar{t}', i')) &= \\ &= \overline{project}(\bar{l}, \overline{suffix}(\bar{f}(\bar{t}'), i'), \overline{S}) \cup \\ &\{\bar{v} : \bar{v} \in \overline{project}(\bar{l}, \bar{f}(\bar{t}''), \overline{S}) : \bar{t}'' \in dom(\bar{f}) \setminus \{\bar{t}, \bar{t}'\} \wedge |\gamma_{\overline{P}}(\bar{t}'')| = 1\} \end{aligned}$$

where $\gamma_{\overline{P}}$ is the concretization function of the may-alias domain.

5.12.3 Abstract Domain

We use the may-alias domain in order to trace on which locations of the heap threads access, i.e. $\overline{Loc} = \overline{P}$. We use the must-alias domain in order to infer on which monitors

threads synchronize, i.e. $\overline{\text{Sync}} = \overline{E}$. The other sets are obtained as the pointwise abstraction, i.e. $\overline{\text{Sh}} = \overline{H}$ and $\overline{\text{St}} = \overline{\Sigma}$.

The abstract transfer function $\overline{\circ} : [\overline{\text{St}} \times \overline{\text{St}} \rightarrow \{\text{true}, \text{false}\}]$ is $\overline{\circ}$ as defined in Section 5.9. The other functions can be obtained as the pointwise application of the abstraction function on the concrete functions just defined.

5.13 Application to the Deterministic Property

In Chapter 4 we stated a few assumptions on how concrete and abstract shared memories are defined. We need to discuss these issues in order to apply the analysis of the determinism of multithreaded programs to the analysis presented right now.

5.13.1 Concrete Domain

In the concrete domain we defined the shared memory as a function that given a shared variable returns the value contained in it and the thread that wrote it, i.e. $S : [\text{Var} \rightarrow (V \times \text{Tid})]$. In this chapter the shared memory, i.e. the heap, has been defined as $H : [\text{Ref} \rightarrow (\text{Obj} \cup \text{Arr} \cup \text{Str})]$.

The set of shared variables is

- a reference when we are accessing an array or a string,
- a triple composed by a reference, a class, and a field's identifier when we are accessing an object, as object are defined as $\text{Obj} : [(C \times F) \rightarrow \text{Val}]$.

The codomain of the shared memory relates values to threads. The heap traces only values. The thread that has written it can be easily inferred. In particular we have simply to augment the *vis* function when reading through the shared memory values written in parallel by other threads following the approach defined in Chapter 3.

5.13.2 Abstract Domain

The first level of abstraction relates each shared variable to a function. This infers for each thread the abstract value it may have written in parallel, i.e. $\widehat{S} : [\text{Var} \rightarrow [\text{Tid} \rightarrow \widehat{V}]]$. The domain of the heap introduced by this Chapter is composed by elements of the may-aliasing domain. So the abstract domain soundly approximates the concrete heap with these elements. We combine with class and fields' identifiers that are statically defined in the Java bytecode each time heaps are accessed.

When reading a value through the shared memory, the abstract value is obtained making the upper bound of all the values written in parallel. In order to infer the information required by the first level of abstraction, we need to track one abstract value for each thread, that can be easily inferred thanks to the structure of the thread-partitioning domain (as it relates each thread identifier to the abstract trace approximation its executions).

5.13.3 Second Level of Abstraction

At the second level of abstraction we relate each shared variable to a pair composed by an abstract value and a set containing threads' identifiers, i.e. $\bar{S} : [\text{Var} \rightarrow (\widehat{V} \times \wp(\text{TId}))]$.

This level of abstraction can be directly obtained from the previous one. In it we traced a value for each thread. So we need only to collapse all these values considering their upper bound, and the set of threads' identifiers as the second component of the pair.

5.14 Discussion

In this chapter we presented the concrete and abstract domains and semantics of Java bytecode. These are aimed at applying the theoretical frameworks defined in Chapters 3 and 4 to the analysis of Java multithreaded programs. Our domains and semantics are tuned at low-level in order to soundly and precisely infer how threads access the shared memory and synchronize. We proved the soundness of our approach. This chapter represents the bridge between the theoretical approaches developed until here and their application to a real language.

In this way, we are now in position to

- implement a static analyzer generic with respect to an abstract numerical domain, a memory model, and a property of interest,
- extend it in order to soundly analyze a program with respect to the happens before memory model,
- analyze the determinism of multithreaded programs.

In addition we may instantiate this analyzer to other properties (like the data race condition, presence of deadlocks, access to null pointers). We can also apply it to some well-known non-relational domains.

This analyzer will be presented in the next Chapter.

6

Checkmate: a Generic Static Analyzer of Java Multithreaded Programs

In this chapter we present the implementation of Checkmate, a generic static analyzer of Java multithreaded programs at bytecode level. This analyzer is generic with respect to a non-relational numerical domain, the property of interest, and a memory model. In this context, we implemented the happens-before memory model, the deterministic and weak deterministic properties. We adopt the domain and semantics introduced in Chapter 5. After an overview of the structure of Checkmate, we will study the experimental results deeply in order to investigate both the precision and the complexity of our approach. Checkmate can be freely downloaded at URL [http : //www.pietro.ferrara.name/checkmate](http://www.pietro.ferrara.name/checkmate).

This chapter is based on the published work [46].

6.1 Generic Analyzers

Many authors proposed and developed generic analyzers relying on the abstract interpretation theory. We have already introduced some of them in Section 5.11. There we studied the theoretical approach of such analyzers mostly. We recall them here fusing on the practical and implementation details.

JULIA [131, 130] is a free tool that implements a generic static analysis of Java bytecode based on the abstract interpretation theory. It analyzes sequential program. In addition, when some particular conditions are satisfied it may obtain sound results for multithreaded programs. Moreover the author argues that there was a theoretical lack about the static analysis of multithreaded, imperative, and object-oriented programs. Our work is aimed exactly to fill this lack.

Clousot [93, 47, 92, 84] analyzes MSIL bytecode. It is parameterized by the property and the numerical domain. It is sound only with respect to single thread executions. It has been already successfully applied to the verification of some properties on industrial software.

Cibai [91] is a generic static analyzer for modular analysis and verification of Java classes. It is not sound with respect to multithreaded executions.

Pollet et al. [112] developed a generic static analyzer for Java code. It does not support concurrency as *Cibai*, and so it is not sound with respect to multithreaded executions.

Méndez-Lojo et al.[105] presents a generic framework based on a representation of a program as a control flow graph. The overall framework is generic also with respect to the programming language, and it has been implemented in order to analyze Java programs at bytecode level. In particular, a great effort has been made in order to optimize the fix-point computation [99]. It is sound only on single-thread executions.

JAIL [42] is a generic static analyzer of JavaCard programs at source code level. It has been successfully applied to the firewall analysis. It can be plugged with different numerical domains and in order to analyze different properties. As JavaCard supports only mono-thread programs, it does not deal with concurrency.

Contribution: In this context, *Checkmate* is the first generic static analyzer of Java multithreaded programs. In particular, it is generic with respect to the numerical domain, the analyzed property, and the memory model. *Checkmate* comprehends some well-known numerical domains (e.g. Intervals), some properties (e.g. null pointer accesses and deadlocks on monitors), and some memory models. It supports the main features of Java multithreading system, and in particular

- dynamic unbounded threads creation,
- runtime creation and management of monitors,
- method-calls in presence of overloading, overriding and recursion,
- dynamic allocation of shared memory.

In addition, it supports all the common features of Java as strings, arrays, static fields and methods, etc..

Checkmate supports only Java bytecode language, and so it does not support all the features implemented through native methods, e.g. reflection and *wait/notify* on objects.

6.2 On Native Methods

Native methods are piece of code written in languages different from Java. They are interfaced with Java programs through the “Java Native Interface”[88]. They are used when developers need to go over the limits imposed by Java language, e.g. to use direct pointers to the memory. These functionalities are required in order to develop some specific applications, e.g. to interface with the operating systems or with memory-mapped devices.

Native methods cannot be automatically analyzed by *Checkmate* as we defined the semantics of Java bytecode statements only. In addition, they may depend on the operating system on which we are executing the applications or on its hardware architecture. Then it is not sufficient to analyze one implementation, but we should take into account all the versions of the Java virtual machine.

System	MyThread
public static Account a = new Account();	private Account a;
public static int main(String[] args){	
MyThread th = new MyThread();	
System.a.amount = 1.000;	
th.start();	public void run(){
synchronized(System.a) {	synchronized(System.a) {
System.a.printAmount();	int temp = System.a.getAmount();
}	if(temp < 100)
}	System.a = null;
}	else System.a.withdraw(100);
	}
	}
	}

Figure 6.1: A multithreaded application

We adopt a minimal approach to the analysis of native methods. We define and implement by hand the semantics of a restricted number of native methods, e.g. the ones of `java.lang.StrictMath`.

6.3 An Example

Figure 6.1 depicts a program composed by 2 threads. `System` creates an `Account` object that instances the class presented in Section 2.3. The initial value is set to 1.000. The object is stored on a public static field, and then `MyThread` is launched. The two threads are both synchronized on the same monitor. `System` prints the amount contained by the account. `MyThread` sets to null the amount if there is less than 100. Otherwise it withdraws 100 from that.

On this example we may be interested to analyze different properties and in particular the presence of data races and of accesses to a null pointer. In order to check them precisely, we would need specific numerical domains and memory models. We will show in Section 6.5 how using different parameters in `Checkmate` to analyze it successfully. Note that the example is written in Java style code, but the analysis works on the bytecode obtained compiling it with `javac`.

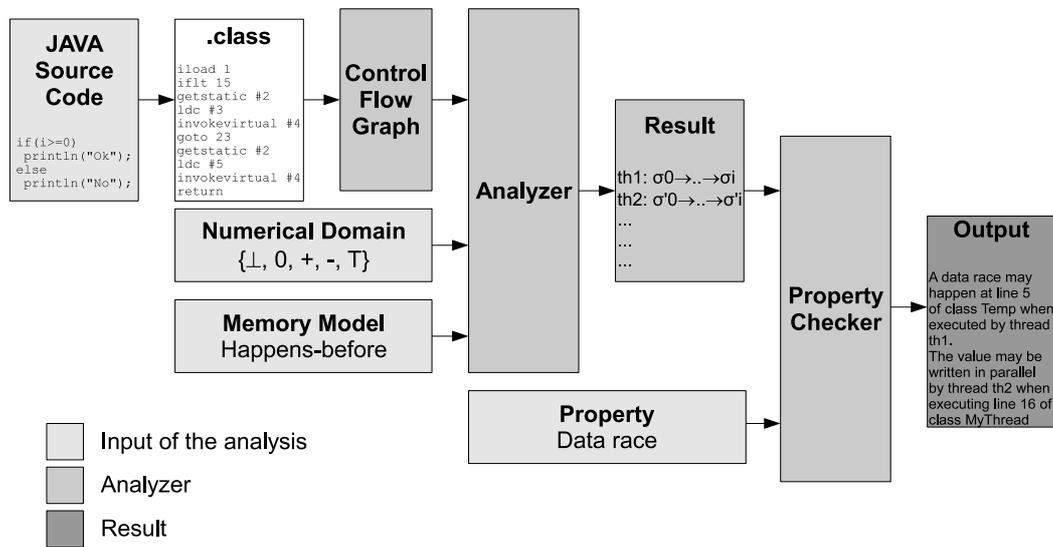


Figure 6.2: Overall structure of Checkmate

6.4 Structure

Figure 6.2 depicts the overall structure of Checkmate. The first step of the analysis is to receive the source code of a Java program, compile it with `javac`, and build up the control flow graph. Then Checkmate builds up an approximation of the program's semantics, i.e. an element of the thread-partitioning trace semantics. The inputs are a memory model and an abstract numerical domain. Finally it checks if a given property received as input is respected by the given abstraction. If it is not the case, a list of warnings is displayed following one of the two user interfaces introduced in Section 6.6.

In the following of this section we present the interfaces of the three inputs of the analysis, i.e. memory models, numerical domains, and properties.

6.4.1 Property

Figure 6.3 depicts the UML object diagram of the implementation of properties. Property Interface requires to define a method `check` that given a state of thread-partitioning trace domain returns an object of type `Alert`. This contains all the warnings produced checking the property.

Property interface is implemented by two classes:

- **DeadlockProperty**: it checks if a program may contain a deadlock on monitors. When a monitor is locked, the reference representing this monitor is passed to `monitorenter`. Through our may-aliasing domain we have an over-approximation of all the possible concrete references that may be locked by each `monitorenter` statement. So we can build up an abstract waiting graph that over-approximates all

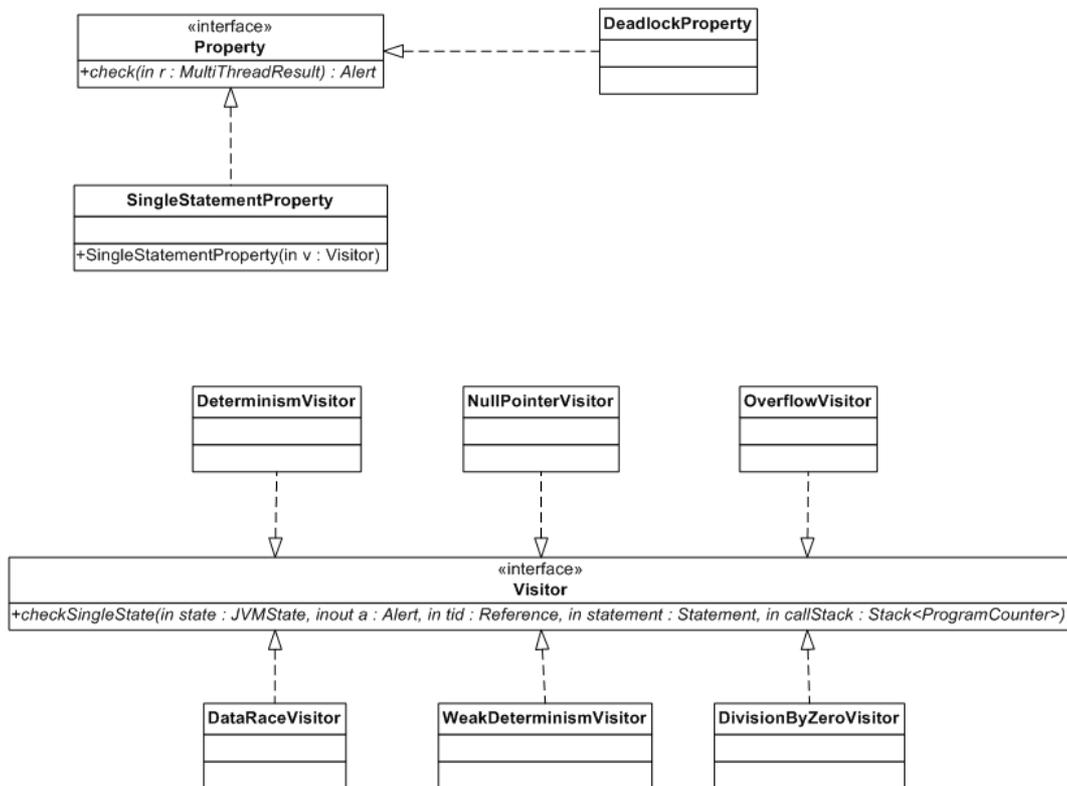


Figure 6.3: The UML object diagram of Property class

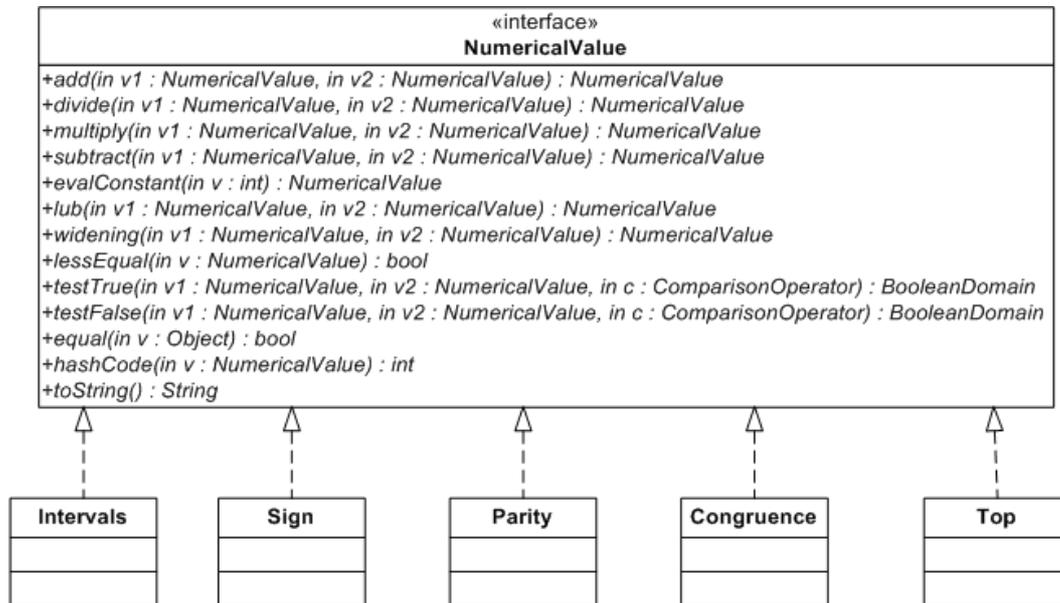


Figure 6.4: The UML object diagram of NumericalDomain class

the concrete waiting graphs. We check on it if a loop may appear. If it is the case, it means that the program may contain a deadlock on monitors;

- **SingleStatementProperty**: it is a generic class that allows to instantiate different properties. These have to work checking each state of computation separately. In particular, the constructor of this class receives an object of type **Visitor**. This interface requires to implement a method `checkSingleStatement`. It receives as parameters a state, an object **Alert**, a thread identifier, the analyzed statement, and the call stack. It checks if the property is validated, and eventually it adds warnings on the given **Alert** object. All the properties except deadlock are implemented through these visitors.

6.4.2 Numerical Domain

Figure 6.4 depicts the UML object diagram of the implementation of numerical domains. Interface **NumericalValue** requires to implement all the abstract arithmetical operators (`add`, `multiply`, ...), the evaluation of conditions (`testTrue` and `testFalse`), and the common operators on lattices (`lessEqual`, `lub`, and `widening`). This interface is directly implemented by all our numerical domains.

6.4.3 Memory Model

Figure 6.5 depicts the object diagram of the interface of the memory model following the standard UML. The core of the diagram is the description of the interface **MemoryModel**.

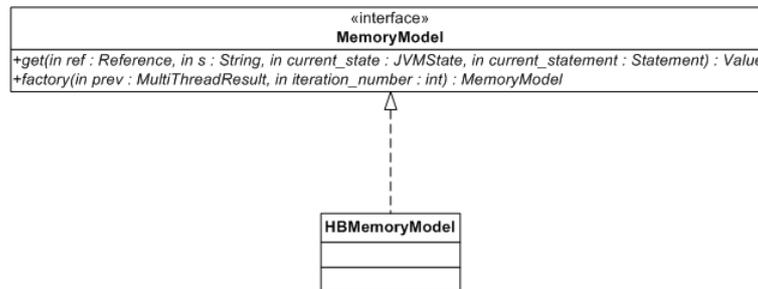


Figure 6.5: The UML object diagram of MemoryModel class

Two methods are defined on it:

- **get**: it receives as parameters a reference, a string identifying the field to be read, the current state (containing also the call stack and the thread that executes the current read), and the statement that is used to read the value. It returns the value read on that locations. It takes the upper bound of all the values written in parallel by other threads and that can be seen following a memory model.
- **factory**: it receives as parameters an object of type `MultiThreadResult` (that is the class that represents elements of our abstract thread-partitioning trace semantics) and the number of iteration of the multithreaded fixpoint semantics. It returns an object of type `MemoryModel`. This has to provide the values written in parallel following the given abstract element. It chooses to apply the lub or the widening operators relying on the number of iterations already performed.

In `Checkmate` this interface is implemented only by the class `HMemoryModel`. The three memory models implemented in `Checkmate` are obtained instantiating this class passing different parameters to the constructor. The implementation is based on the theoretical approach developed in Chapter 3.

6.4.4 An Example of Interaction

Figure 6.6 depicts an UML sequence diagram that represents one possible execution of `Checkmate`. The analyzer requires a memory model and a numerical domain when the analysis is launched. During the analysis, `Checkmate` uses the memory model in order to know which values written in parallel are visible at a given point of execution, and the numerical domain in order to approximate numerical values. Once a fixpoint is reached, the analysis obtains an object representing the abstraction of all the possible executions of the program. Then this abstract result is passed to a `Property` object that checks if the property is respected. Also in this context, the memory model and the numerical domain are used in order to have information on the execution as during the analysis. Finally, `Checkmate` obtains an object of type `Alert`: it contains all the warnings produced while checking the property. So it shows the warnings and it ends the analysis.

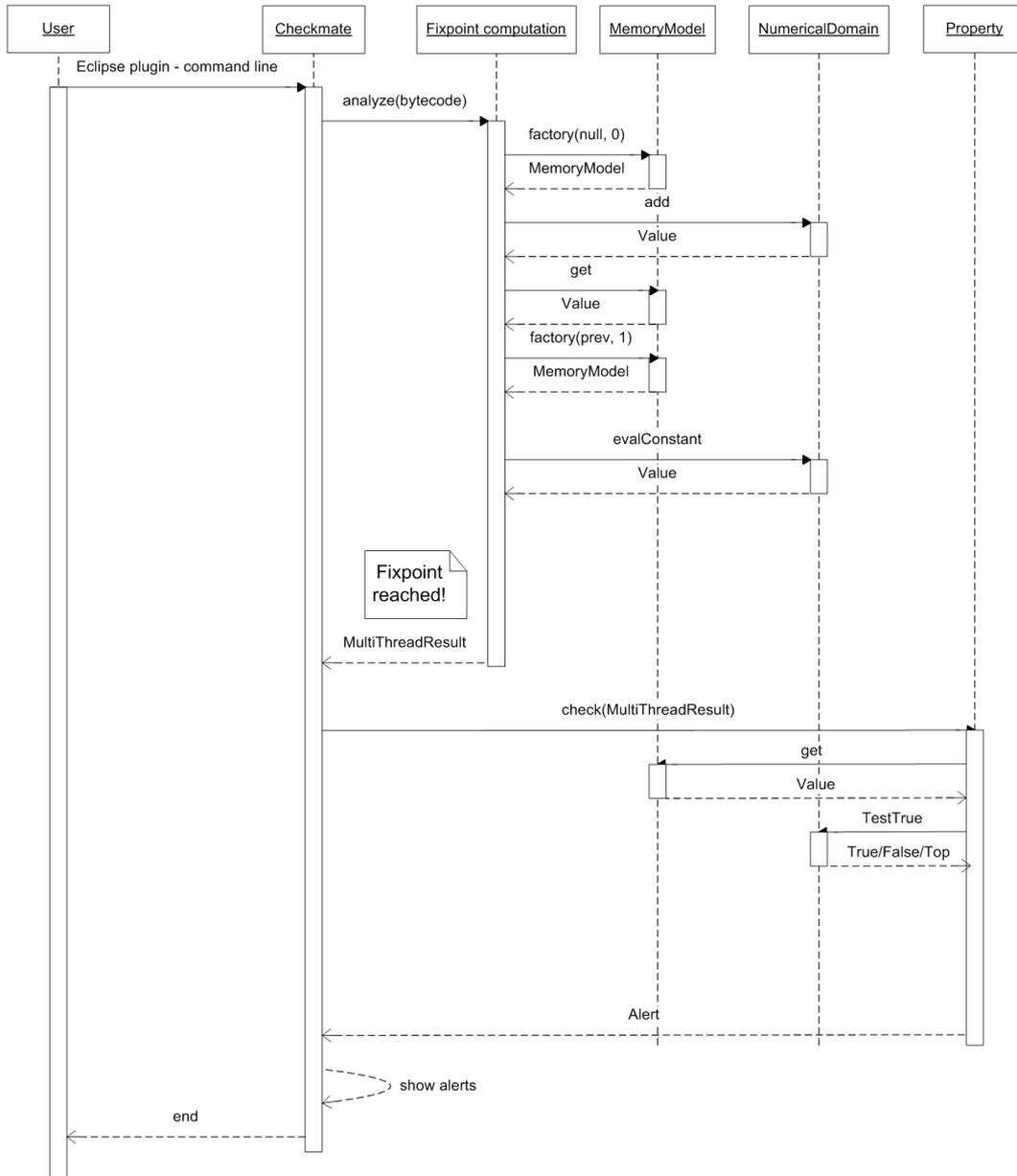


Figure 6.6: An example of interaction during the analysis

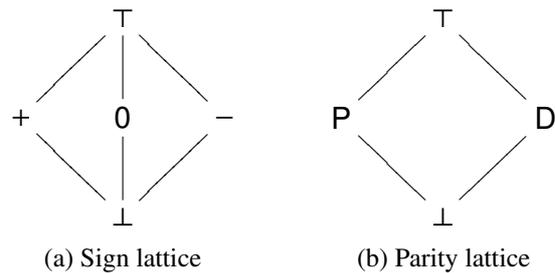


Figure 6.7: Numerical domains

6.5 Parameters

In this section we introduce the main parameters of **Checkmate**.

6.5.1 Properties

Many different properties of multithreaded programs may be interesting. A first set is composed by the ones interesting also at single-thread level, e.g. division by zero. In addition there are other properties specific of parallel programs, e.g. data race condition. In **Checkmate** we implemented a representative set of properties of both groups:

- division by zero,
- null pointer accesses,
- overflow,
- data races,
- deadlock on monitors,
- determinism and weak determinism, as defined in Chapter 4.

Other properties may be easily added to **Checkmate**.

Example: On the example presented in Section 6.3 we are interested in checking if a data race may happen, and if a `NullPointerException` may be thrown. As **Checkmate** is parameterized by the property, we can build up an abstraction of its multithreaded executions, and we can check on that both the properties.

6.5.2 Numerical Domain

We implemented some well-known non-relational abstract domains: Sign [25] (Figure 6.7a), Interval [25] (Figure 6.8), Parity [27] (Figure 6.7b), and Congruence [56] (Figure 6.9).

Example: Suppose to analyze the example introduced in Section 6.3 using Sign domain.

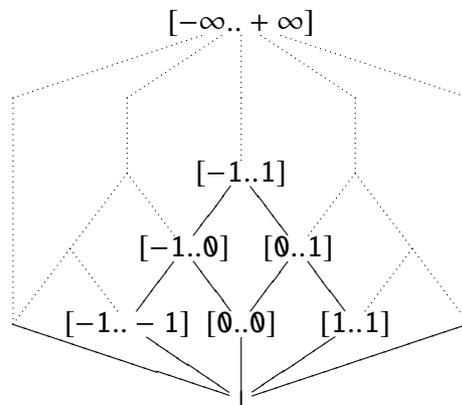


Figure 6.8: Interval lattice

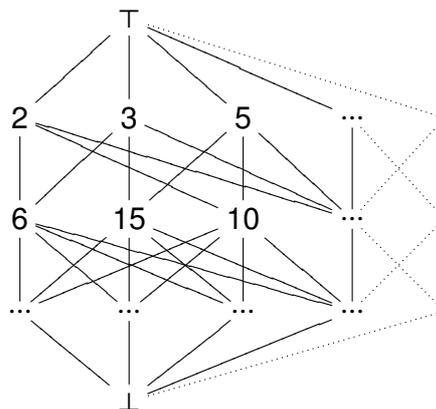


Figure 6.9: Congruence lattice

We check that the amount of the bank account is positive (+), but we cannot precisely analyze the condition `if(temp < 100)` of `MyThread`. In fact, + may be `< 100`. So we conclude that `null` may be assigned to `System.a`. After that this write action is propagated, `System.a.printAmount()` in `System` may cause a `NullPointerException` as `System.a` may be `null`. This happens because the numerical domain is too approximated. If we use the Interval domain, we check that the value written by `System` is `[1000..1000]`. So the condition `if(temp < 100)` cannot be evaluated to true, `null` cannot be assigned to field `System.a`, and finally the `NullPointerException` cannot be thrown.

6.5.3 Memory Models

Memory models specify which values written in parallel may be seen by a read action. Intuitively, it may be implemented as a method that given a point of the computation, a shared variable, and a state of the multithreaded execution, returns a set of visible values,

i.e. values written in parallel by other threads. This approach has been formalized in Chapter 3. In this way, we are in position to parameterized **Checkmate** on it, and we may develop many memory models.

First of all, we implement the happens-before memory model. In addition, **Checkmate** contains other two memory models that are more approximated. The goal of their implementation is to compare the computational overhead induced by more precise memory models. A first abstraction ignores the synchronize-with relation on monitors. The second one abstracts away also the relation that traces when and by whom a thread is launched.

Example: Suppose now to analyze the data race condition on the example introduced in Section 6.3. If we use the most approximated memory model, we suppose that all the values might be written in parallel. This means that the synchronize-with relation is not traced when a thread is launched. So the values written before this action would be seen as written in parallel with the statements executed by the started thread. Then `System.a.amount = 1.000` of `System` would be seen as written in parallel with all the statements of `MyThread`. As the first action is not synchronized on any monitor, **Checkmate** produces a false alarm signaling that there may be a data race. Using a more refined memory model (both the happens-before one and the intermediate version, that traces the synchronize-with relation when a thread is launched) we check that `System.a.amount = 1.000` of `System` cannot be executed in parallel with statements of `MyThread`. So that they do not form a data race.

Thanks to our must-aliasing domain we precisely discover that the accesses performed inside the two synchronized blocks are synchronized on the same monitor, and so that they cannot produce a data race.

6.6 User Interfaces

We implemented two user interfaces:

- a command line tool,
- an Eclipse plugin, that can be installed in this development tool and used through a graphic interface.

6.6.1 Command Line

The command line tool is composed by one file. It can be executed launching `java -jar checkmate.jar [...]`.

[...] contains the parameters of the analysis as follows:

`<mainclass> -p:<prop> [-d:<dir>] [-n:<num>] [-m:<mm>]` where:

- `<mainclass>` is the name of the class to be analyzed. It has to contain the `main` method of the multithreaded application.

- `-p:<pro>` sets the property to be analyzed.
`<pro>`:
 - `d` : Data race
 - `l` : Deadlock
 - `n` : Null pointer access
 - `o` : Overflow
 - `z` : Division by zero
 - `p` : Determinism
 - `w` : Weak determinism

- `-d:<dir>` sets the directory containing the `.class` files to be analyzed.

- `-n:<num>` sets the numerical domain, where
`<num>`:
 - `t` : Top (don't collect numerical information)
 - `s` : Signs
 - `p` : Parity
 - `i` : Intervals (default)
 - `c` : Congruence

- `-m:<mm>` sets the memory model, where
`<mm>`:
 - `a` : All values in parallel
 - `t` : Launch of threads
 - `h` : Happens-before (default)

The name of the class and the property must be specified. The other parameters are optional.

Typically during the analysis some informations are printed on the standard output.

- “Iteration `n`”: it reports which iteration of fixpoint computation is going to be computed. A priori we do not know after how many iterations the computations will end. Usually it requires no more than 5 iterations, but this value may change considerably from a program to another, and it also relies on the numerical domain.

- “Class `<class>` not found in the provided directory. Read from the local repository of JVM. Static variables not initialized”: it means that a class has not been found in the directory passed to the analyzer, and it was read by the repository of the current virtual machine. In this way we cannot

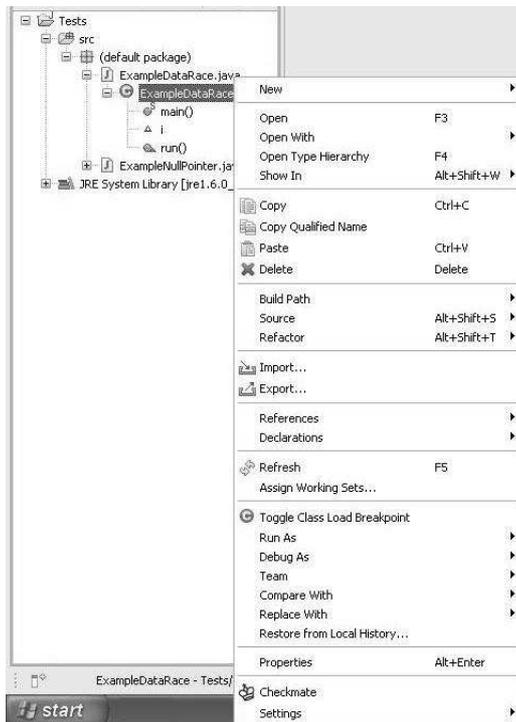


Figure 6.10: Launching the analysis

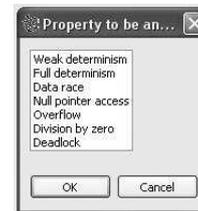


Figure 6.11: Choosing the property



Figure 6.12: Output

analyze the static fields at the beginning of the computation. Accessing one of these fields would produce an error. Usually we need to read classes from the current repository because all the Java applications refer to libraries. For instance all the classes when are instantiated call the constructor of class `java.lang.Object`.

The warnings will be display on the standard output at the end of the analysis. If the property is validated a specific message will confirm it.

6.6.2 Eclipse Plugin

Installation

As common for Eclipse plugins, the application is composed by a `.jar` file. This file has to be copied inside `plugins` directory of Eclipse main directory. Then it is necessary to launch Eclipse with `-clean` option.

Running the Analysis

In order to start the analysis, the class to be analyzed has to be opened in the package explorer window. Then the user has to click on “Checkmate”. Figure 6.10 shows it. Once he clicks on it, a dialog will appear (Figure 6.11). This requires to select which property we want to analyze. Finally, the results of the analysis will be displayed in a

view (Figure 6.12).

In addition, the user can set which memory model and numerical domain apply during the analysis. The default values are the happens-before memory model and the Interval domain.

6.7 Experimental Results

We test Checkmate on some multithreaded programs presented in [138, 94, 85, 73]. We investigate both the precision and the performances of the analysis.

We execute it on an Intel Pentium D 3.0 Ghz with 2 GB of RAM running a Windows Server 2003 with Java virtual machine version 1.6.0_06.

6.7.1 Common Patterns of Multithreaded Programs

Lea [85] presented an overview of common patterns when developing concurrent programs in Java. In particular, he introduced some representative examples in order to explain in practice the concepts presented throughout the book. He showed which errors may arise on these examples and how these can be fixed. We apply Checkmate to some examples in order to discover such errors. Usually Lea presents some classes, and then explains by words why this class has to be considered correct, or which undesirable behaviors may expose. Since Checkmate performs a whole-program analysis, for each example we develop a main method that exposes the behavior of interest.

ExpandableArray (Appendix A.1): This class implements an array that is automatically expanded if the user want to append an object when the array is full. All the methods are synchronized. If an user performs in parallel two writes or a read and a write, a conflict arises. In fact, even if all the methods are synchronized, the position of the elements in the array and the read element may be non deterministic because of different interleavings of threads' executions.

This program does not contain data races, and Checkmate precisely discovers it. Instead the conflicts are exposed by the deterministic property, that precisely signals the non deterministic behaviors of the two accesses.

LinkedCell (Appendix A.2): This class implements a list of double values. The methods that read the value contained in the current cell and write a new value are both synchronized. The method that returns the sum of all the cells is not synchronized but it relies on synchronized methods, and so it does not expose any data race. Finally, a method perform an incorrect sum, reading without synchronized methods the value contained by the first element of the list, thus potentially causing a data race.

Checkmate precisely discovers that the well-synchronized sum method does not expose any data race if executed in parallel with writes on the list. In the same way, it discovers that the ineffectively synchronized sum causes a data race. If we apply the deterministic property we discover that a non deterministic behavior happens even if the program is well-synchronized.

Document (Appendix A.3): This class implements a document that contains an enclosure. A synchronized `print` method that prints the content of the document is provided. Another synchronized `printAll` method prints all the content using the synchronized `print` method of the current object, and then invokes the same method on the enclosure. Suppose now to have two documents `d1` and `d2` whose enclosure is the other document, e.g. the enclosure of document `d1` is `d2`. If we print concurrently these two documents, this may cause a deadlock. For instance the first thread may start the execution of `printAll` and acquire the monitor of `d1` starting the execution of `printAll`. Then the control may switch to the second thread, that acquires the monitor of `d2` and it yields on the monitor of `d1`. Finally, the control switches to the first thread, that start yielding on the monitor of `d2`, causing a deadlock.

`Checkmate` precisely discovers that this program may contain a deadlock.

Dot (Appendix A.4): This class implements a dot in a Cartesian plane. Its coordinates are stored in a `Point` object. The methods provided by `Point` class in order to access the information are not synchronized, but all the methods of class `Dot` are synchronized. On the other hand, if we move a point and shift its x axis value concurrently we may obtain nondeterministic executions.

`Checkmate` validates this program applying the data race condition, as in fact this program is data race free. In addition, it discovers the nondeterministic behaviors precisely applying the deterministic property.

Cell (Appendix A.5): This class implements a cell containing an integer value. The `get` and `set` methods are both synchronized. In addition, another synchronized method allows to swap the content of the current object with the one of the object passed to the method as parameter, using the getter and setter methods. If we swap the content of two cells twice in parallel, we may obtain a deadlock.

`Checkmate` detects this behavior precisely applying the deadlock property.

TwoLockQueue (Appendix A.6): This class implements a queue on which we can take and put objects. If we execute a take and a put action in parallel when the queue is empty, the take action may return a null value, as it may be executed before the put action.

`Checkmate` precisely discovers it. In particular, if the queue is empty when the two threads are executed in parallel, it signals that the value returned by the take action may be null. In order to obtain this result, we add an access to a field of the object returned by the take action, and then we analyze this program with the `NullPointerException` property, that discovers that a `NullPointerException` may happen. If we add an element before launching the two actions in parallel, `Checkmate` precisely discovers that the value returned by the take action cannot be null.

Account (Appendix A.7): This example is quite complex and involves many classes. In particular, it implements an immutable and an updatable account, an account holder, and two account recorders, one correct and the other one evil. We refer the interested reader to [85] for more details about the implementation of this classes. The potential problem is that if the account holder accepts money without using an immutable instance of the recorder, an evil recorder may cause a non-deterministic behavior.

`Checkmate` precisely signals it. In particular, if the account recorder is not evil or the

account holder use an immutable instance of the recorder, it proves that the program is deterministic. On the other hand, if the account recorder is evil and the account holder does not oblige the use of an immutable instance of the recorder, it signals that a non-deterministic behavior may happen.

Discussion

Checkmate performs a precise and correct analysis of the representative set of examples we chose from [85]. In particular, in each case it discovers the bug or proves that the program is correct with respect to the behavior of interest. This result is achieved through the high level of flexibility of Checkmate. Using different properties allows us to tune the analysis in order to catch all the bugs. In addition, we found out that the deterministic property is often the only way to discover the behavior of interest. This confirms our impression that this property is in position to break the limits of existing properties applied to multithreaded programs, e.g. the data race condition.

6.7.2 Weak Memory Model

We take some challenging examples presented in [94] (journal version of the paper that introduced the Java memory model [95]) in order to test the precision of Checkmate. We write them in Java style (i.e. adding a method main that instantiates and launches the threads), we compile them with javacc, and we analyze the bytecode with Checkmate using the happens-before memory model and the Interval domain.

Figure 6.13a: This example is quite similar to the one presented in Section 3.1.1. A compiler may switch the statements of each thread. In fact they work on disjoint sets of variables, and so they are independent. Our analysis correctly traces this behavior, and it checks that $r1$, $r2$, and $r3$ may be equal to zero at the end of the execution.

Figure 6.13b: In order to obtain the required behavior, it seems that a thread may write a variable before it reads it. Instead, this behavior may be exposed by some compiler optimizations as pointed out in Section 2.2.2 of [94]. Our analysis soundly approximates it. This behavior is exposed after the third iteration of the multithread semantics as

- at the first iteration value 1 is written on x ,
- at the second iteration this value is written by Thread1 on $r1$ and then on y ,
- at the this iteration 1 is read by Thread2 through y and written in $r2$,
- during the fourth and last iteration the analysis does not expose any new behavior and so it converges.

Figure 6.13c: Thanks to the Interval domain, Checkmate precisely traces that only $[0..0]$ can be assigned to $r1$ and $r2$. As our analysis is context-sensitive, it checks that the conditions of both threads cannot be evaluated to true, and so that value 42 will never be assigned.

Thread1	Thread2
r1 = x; y = 1; r2 = x;	x = 1; r3 = y;

(a) Figure 1. Initially, $x == y == 0$. $r1 == r2 == r3 == 0$ is legal behavior

Thread1	Thread2
r1 = x; y = r1;	r2 = y; r3 = r2 1; x = r3;

(b) Figure 3. Initially, $x == y == 0$. $r1 == r2 == r3 == 1$ is legal behavior.

Thread1	Thread2
r1 = x; if(r1! = 0) y = 42;	r2 = y; if(r2! = 0) x = 42;

(c) Figure 4. Initially, $x == y == 0$. Correctly synchronized, so $r1 == r2 == 0$ is the only legal behavior.

Thread1	Thread2
r1 = x; if(r1 == 1) y = 1;	r2 = y; if(r2 == 1) x = 1; if(r2 == 0) x = 1;

(d) Figure 7. Initially, $x == y == 0$. $r1 == r2 == 1$ is legal behavior.

Thread1	Thread2
r3 = x; if(r3 == 0) x = 42; r1 = x; y = r1;	r2 = y; x = r2;

(e) Figure 11. Initially, $x == y == z == 0$. $r1 == r2 == r3 == 42$ is a legal behavior.

Thread1	Thread2	Thread3	Thread4
r1 = x; y = r1;	r2 = y; x = r2;	z = 42;	r0 = z; x = r0;

(f) Figure 12. Initially, $x == y == z == 0$. $r0 == 0, r1 == r2 == 42$ is legal behavior.

Thread1	Thread2
do{ r1 = x; }while(r1 == 0); y = 42;	do{ r2 = y; }while(r2 == 0); x = 42;

(g) Figure 25. Initially, $x == y == 0$. Correctly synchronized, so non-termination is the only legal behavior

Thread1	Thread2
r1 = x; if(r1 == 0) x = 1; r2 = x; y = r2;	r3 = y; x = r3;

(h) Figure 27. Initially, $x == y == 0$. Compiler transformations can result in $r1 == r2 == r3 == 1$.

Figure 6.13: Some examples taken from [94]

Figure 6.13d: We need three iterations in order to propagate the value 1. The first iteration writes it on variable `x`, the second propagates it on `r1` and `y`, and finally the third iteration assigns it to `r2`. In this way we obtain the result required by the example.

Figure 6.13e: Value 42 is assigned to `x` and `y` by `Thread1`. Then it is assigned by `Thread2`. Finally it comes back to the first statement of `Thread1` that assigns to `r3` the value contained by `x`. In this way we capture the behavior of interest.

Figure 6.13f: As this example involves 4 threads, it requires some more iterations of multithread semantics in order to reach a fixpoint. Checkmate soundly discovers that a possible behavior is `r1 == 0, r1 == r2 == 42`.

Figure 6.13g: This example is similar to the one contained by Figure 6.13c. Interval domain precisely finds out that the condition of while loops cannot be evaluated to `false`. Then value 42 is never assigned neither to `x` nor to `y`, and so the threads never exit the loops. Checkmate discovers it.

Figure 6.13h: The situation is quite similar to the one depicted by Figure 6.13e. Checkmate is precise with respect to the expected behavior.

Discussion

In all the examples Checkmate analyzes them successfully producing a sound and precise abstraction of the behavior of interest. As the figures depict toy examples (usually no more than 200 bytecode statements and 4 threads), Checkmate requires always less than one second in order to execute the analysis.

These results are quite encouraging. We deal with examples aimed at explaining the main features of the Java memory model, and this is more refined than the happens-before one. Nevertheless, we precisely analyze them. In general, our analysis is able to catch the behaviors presented by the examples in [94] in all the cases in which they do not involve volatile variables. In other cases, our analysis does not take into account the fact that a variable is volatile. So we obtain results that are still sound but too much approximated. On the other hand, we think that our framework is extensible and flexible enough in order to take into account also volatile variables.

In addition, note that our analysis provides an approximation of all the possible behaviors of a multithreaded programs, and not only on a subset (e.g. with at most n interleavings) of them. Nevertheless, the analysis is really fast, as it is able to obtain an immediate output when analyzing small programs, i.e. with a couple of threads and less than 200 bytecode statements.

6.7.3 Incremental Example

We apply Checkmate to an incremental example that simulates the operations performed by a bank. The Java code used in these examples is reported by Appendix B. Checkmate analyzes the bytecode obtained compiling it with `javac`.

Table 6.1 reports the number of abstract threads and statements of each program. Table 6.2 reports the time of execution in milliseconds required to build up the abstraction of the

Program	# ab. th.	# st.
Test1	3	452
Test2	5	684
Test3	7	807
Test4	11	1049
Test5	13	1173
Test6	15	1405
Test7	17	1526
Test8	19	1758
Test9	20	1878
Test10	24	2294

Table 6.1: Number of abstract threads and statements

	Top	Sign	Intervals	Parity	Congruence
1	814	361	217	404	294
2	409	391	356	620	545
3	712	595	925	521	642
4	799	823	3806	703	642
5	1090	919	5887	779	616
6	1382	824	7161	900	986
7	1071	1647	9289	1340	863
8	1018	1269	10999	1263	1221
9	1421	2212	11691	1274	1623
10	1466	2432	17016	863	1906

Table 6.2: Times of analysis (msec)

	Weak det.	Det.	Data race	Null	Overflow	Div. by 0	Deadlock
1	31	32	47	31	15	16	16
2	78	78	125	47	47	47	15
3	125	125	172	187	63	62	16
4	250	250	359	125	94	94	62
5	359	360	484	172	125	125	78
6	547	562	828	266	219	203	125
7	719	734	1047	313	250	250	156
8	1000	1047	1609	438	359	360	250
9	1203	1234	1938	516	406	422	265
10	2031	2094	3609	828	688	687	500

Table 6.3: Times of properties' analysis (msec)

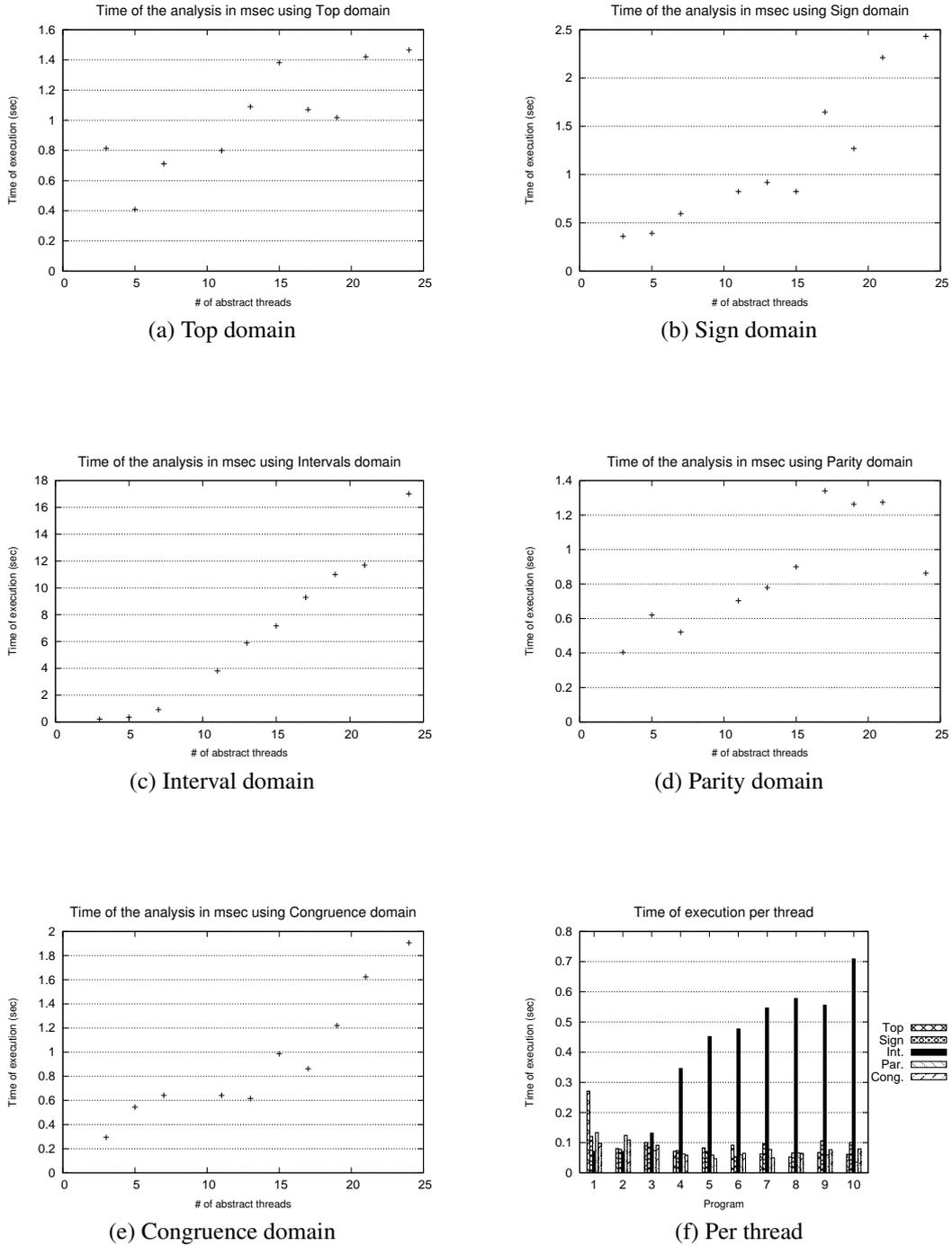


Figure 6.14: Times of execution

Program	# st.	# ab. th.
philo	213	2
forkjoin	170	2
barrier	363	3
sync	320	3
crypt	2636	3
sor	1121	2
elevator	1829	2
lufact	3732	2
montecarlo	3864	2
total	14248	21

Table 6.4: The analyzed programs

program using different numerical domains. The analyzer is quite fast: rarely it requires more than a couple of seconds in order to converge. Only Interval domain requires more time (about 17" in the worst case), as it is the more complex domain that we implemented. In particular we are interested in studying the complexity of the analysis with respect to the number of statements and abstract threads analyzed. So we draw a plot for each numerical domain with the number of abstract threads in x-axis, and the time of execution in y-axis.

The behavior of the Top domain (Figure 6.14a) is not regular. The time of executions increases but, as the analysis is quite fast, it is hard to conclude which is exactly the behavior of the analysis. More regular results are obtained with Parity (Figure 6.14d), Sign (Figure 6.14b), and Congruence (Figure 6.14e). The complexity is linear with respect to the number of abstract threads in all the cases. Finally, Interval domain (Figure 6.14c) seems to expose a quadratic complexity.

We study how the time of the analysis changes with respect to the number of abstract threads analyzed. Figure 6.14f plots the times of execution per thread. All the domains except Intervals require a constant time per thread. Instead, analyzing the application with Intervals the times per thread augment with respect to the number of abstract threads. This increase seems to be linear, and this confirms that the overall time required by the analysis is quadratic with respect to the number of abstract threads.

Starting from these experimental results, we can conclude that in practice the complexity of *Checkmate* is quadratic with respect to the number of threads and statements. This result is promising, but we think that the analysis must be optimized. In particular the fixpoint computation of single-thread semantics is sometimes slow as it works at really low level, i.e. simulating step by step the actions of the Java virtual machine.

6.7.4 Benchmarks

We apply Checkmate to the analysis of some well-known examples and benchmarks taken from the literature. Two applications (philo, and elevator) are taken from [138], while the others (barrier, forkjoin, sync, sor, crypt, lufact, and montecarlo) are taken from the Java Grande Forum Benchmark Suite [73]. We remove from the original programs only the calls to system functions (e.g. `System.out.println`) as sometimes they deal with native methods or reflection that are not supported by Checkmate. Table 6.4 reports the analyzed programs, the number of statements and the number of abstract threads. Note that in all the cases the abstract threads approximate a potentially unbounded number of concrete threads.

We apply the analysis to all the benchmarks with all the possible combinations of memory models and abstract numerical domains. Table 6.5 reports the computational times. For each numerical domains we report the times of execution

- using the more relaxed memory model (column **AP**),
- using the memory model that traces only when a thread is launched (column **TL**),
- using the happens-before memory model (column **HB**),
- required to compute the semantics of each thread in isolation (column **S.T.**).

For each numerical domain, we plot the times of the analysis using the three memory models with respect to the overall number of analyzed statements. Figure 6.15a reports the result obtained applying the top domain, Figure 6.15b with Sign domain, Figure 6.15c with Interval domain, Figure 6.15d with Parity domain, and Figure 6.15e with Congruence domain. In all the cases, for programs with less than 500 statements the analysis is quite fast. In addition, the computational times of the same program are comparable using different numerical domains. The analysis of `crypt` is always quite faster than the one of `elevator`, even if it is bigger. This happens because of the internal structure of the program. With the exception of Interval and in part of Sign domains, the time of the analysis does not grow too much with respect to the number of statements. The complexity seems to be almost linear with respect to it.

Intervals and in part Signs do not respect this rule. In particular, the analysis of `montecarlo` is quite slower. We want to check if this slowness is due to our approach or to the fixpoint computation of a single thread, i.e. to the structure of the program. Table 6.6 reports the overhead due to the multithread fixpoint computation with respect to the single-thread fixpoint semantics. In this way, we can check if the slowness depends only on the single-thread semantics, or on our global approach, i.e. the computation of two nested fixpoints.

Program	Top			Sign			Int.			Par.			Cong.			
	AP	TL	HB S.T.	AP	TL	HB S.T.	AP	TL	HB	S.T.	AP	TL	HB S.T.	AP	TL	HB S.T.
philo	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"
forkjoin	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"	<1"
barrier	<1"	<1"	<1"	<1"	1"	1"	1"	2"	2"	1"	1"	1"	1"	1"	1"	1"
sync	1"	1"	1"	1"	1"	1"	2"	2"	3"	1"	1"	1"	1"	1"	1"	1"
crypt	4"	4"	5"	6"	6"	6"	16"	17"	17"	13"	5"	5"	6"	4"	5"	2"
sor	4"	4"	4"	6"	6"	7"	16"	17"	17"	5"	5"	5"	6"	2"	5"	2"
elevator	27"	28"	31"	10"	11"	11"	18"	18"	19"	7"	29"	30"	30"	11"	29"	11"
lufact	25"	25"	27"	52"	52"	53"	5'56"	5'56"	5'59"	2'08"	28"	29"	29"	12"	27"	29"
montecarlo	54"	56"	1'02"	2'23"	2'26"	2'35"	1h00'33"	1h00'38"	1h00'56"	16'48"	1'38"	1'38"	1'43"	31"	54"	1'00"

Table 6.5: Times of analysis

Program	Top			Sign			Int.			Par.			Cong.			Total		
	AP	TL	HB	AP	TL	HB	AP	TL	HB									
philo	221%	263%	282%	245%	246%	259%	450%	480%	532%	145%	198%	198%	160%	297%	319%	233%	282%	300%
forkjoin	352%	403%	531%	282%	315%	332%	279%	314%	317%	422%	319%	559%	432%	454%	495%	348%	380%	434%
barrier	148%	193%	207%	173%	212%	255%	193%	205%	224%	182%	199%	218%	275%	278%	299%	193%	215%	236%
sync	233%	296%	310%	316%	323%	369%	398%	400%	435%	268%	300%	307%	228%	242%	282%	295%	309%	343%
crypt	295%	334%	349%	201%	208%	214%	120%	127%	128%	366%	341%	395%	258%	277%	281%	171%	181%	186%
sor	263%	261%	267%	252%	260%	301%	304%	324%	325%	225%	265%	247%	251%	256%	264%	269%	282%	292%
elevator	240%	249%	272%	234%	248%	252%	243%	252%	258%	256%	253%	266%	244%	251%	252%	245%	254%	262%
lufact	259%	262%	279%	252%	253%	258%	274%	277%	279%	230%	267%	238%	235%	249%	254%	265%	269%	272%
montecarlo	235%	245%	268%	320%	328%	347%	361%	361%	363%	311%	295%	328%	212%	235%	251%	352%	353%	357%
average	249%	279%	307%	253%	266%	287%	291%	304%	318%	267%	271%	306%	255%	282%	300%			

Table 6.6: Overhead of multithread fixpoint computation

Figure 6.15f depicts the overhead of the multithread fixpoint computation using Intervals and the happens-before memory model. It makes clear that this overhead does not depend on the number of threads or statements analyzed. Its values are between 250% and 450% (with the exception of `crypt`), they do not depend on the length of the program. In fact, we often obtain the biggest overhead for the smallest application. In addition, for bigger application (`sort`, `elevator`, `lufact`, and `montecarlo`) the overhead is almost stable (300% in average). This result is quite encouraging: it means that in average we need about 3 iterations of single-thread semantics in order to reach a fixpoint in our multithread semantics. In addition, we think that we can improve this result as our implementation is not optimized at all. For instance, we may parallelize the analysis of different threads during the same iteration of the multithread semantics.

Finally we compare the computational times using different memory models. For programs with less than 500 statements, the analysis is too fast to obtain significant comparisons. So we consider only the analysis of `crypt`, `sort`, `elevator`, `lufact`, and `montecarlo`. Figure 6.16a depicts the overhead of the analysis of happens-before memory model with respect to AP memory model. Figure 6.16b depicts it with respect to TL memory model. The overhead of HB with respect to AP is rarely more than 10%. In average it is about 5%. It is in average about 2% with respect to TL and rarely more than 5%. Also these results are quite encouraging: the overhead of more refined memory models is quite limited. This means tracing more and more relations between threads seems not to afflict dramatically the performances of the analysis.

6.8 Related Work

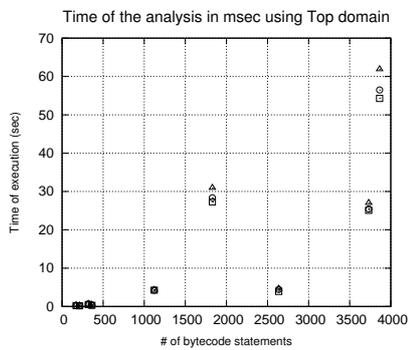
In the literature, some generic analyzers applied to sequential programs, and some analyses particular for a specific property have been proposed. We related our work with the existing generic analyzer in Section 6.1. Here we relate Checkmate with other analyses specific for a given property that is implemented in Checkmate.

6.8.1 Concurrency Properties

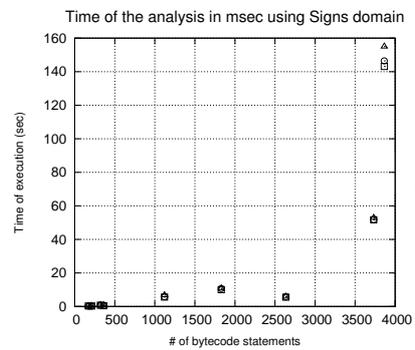
Many approaches have been developed in order to statically analyze multithreaded programs. Most of them deal with deadlock and data race detection [121]. First of all, usually these approaches suppose that the execution is sequentially consistent, but this assumption is not legal under, for instance, the Java Memory Model. In addition, these analyses are particular for this property, and they cannot be applied to other properties, while Checkmate has been already applied successfully to a wide set of properties.

Data Race Analysis

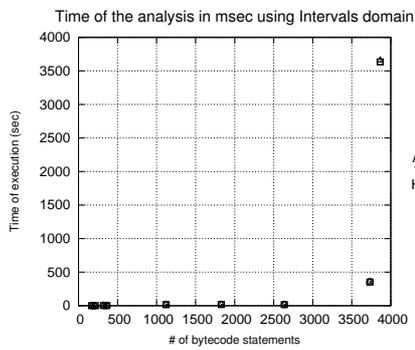
The (maybe) most known work of last years has been the type system developed by Abadi, Flanagan and Freund [1]. This work is modular, and it is proved to scale as it was suc-



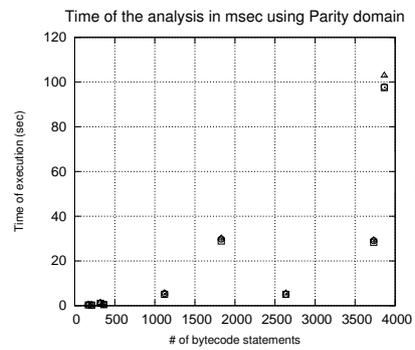
(a) Top domain



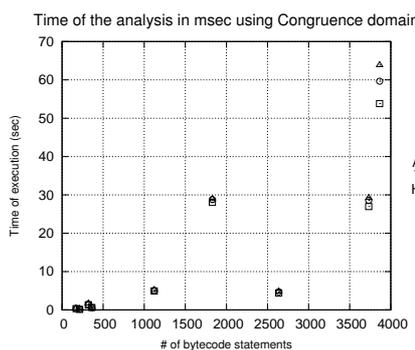
(b) Sign domain



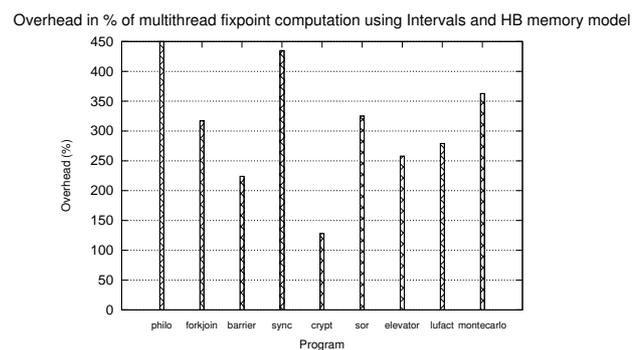
(c) Interval domain



(d) Parity domain



(e) Congruence domain



(f) Overhead of multithread fixpoint computation using Intervals

Figure 6.15: Times of execution using HB memory model

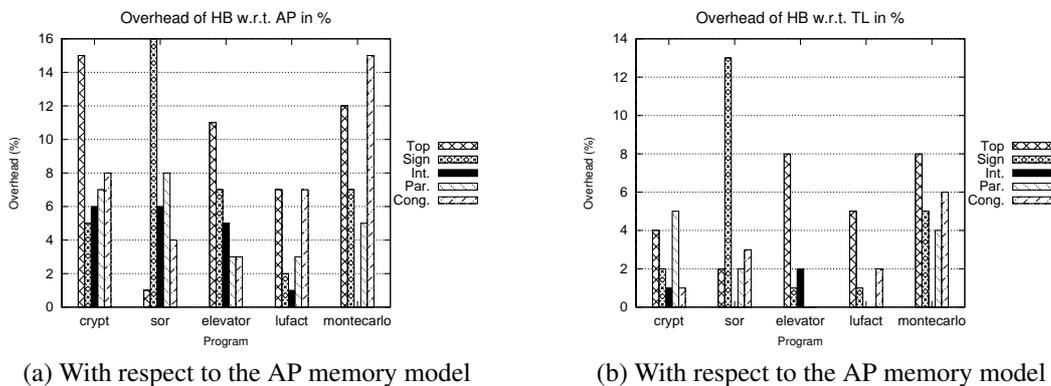


Figure 6.16: The overhead in % of the HB memory model

cessfully applied to programs of a couple of hundreds of thousands of code lines. On the other hand, it requires manual annotation, and it does not provide any information on possible missing locks when a data race is detected.

Naik and Aiken [103] apply a must not alias analysis through a specific type system in order to check the absence of data races. Race freedom is proved by checking that if two locks must not alias the same monitor, then the accesses to the shared memory must not be on the same location. The experimental results seem to underline that the approach can not scale up and they are worse than our results, as this analysis requires more than 3 minutes to analyze only 2 classes.

Kahlon et al. [75] presents a model-checking based analysis in order to statically detect data races. The work has been divided into three different steps: (i) discovering which variables share information (ii) checking through a must alias analysis the owned monitors when shared variables are accessed (iii) reducing the false warnings. The proposed must-alias analysis is quite similar to ours. This approach does not provide any information about possible missing locking actions.

Another data race detector based on model checking has been proposed by Henzinger et al. [61]. The analyzed programming language synchronizes through atomic sections, and so it is quite different from the lock-based synchronization of Java. Moreover the experimental results are afflicted by the well-known state explosion problem.

Deadlock Detection

Many works are focused on the dynamic detection of deadlocks [13, 14, 39]. These tools are able at runtime to detect if a deadlock happens at runtime. On the other hand, if a program may expose a deadlock but we test it only on deadlock-free executions, these tools do not discover the deadlock.

About static analyses, Williams et al. [140] propose an analysis that detects deadlock

on synchronized statements and wait invocations. This analysis was implemented and applied to many libraries, and it discovered 14 distinct deadlocks. This analysis makes some assumptions on how an user interacts with the libraries. In order to analyze a library it supposes that the client code “well-behaved”. In this way, even if a library is validated by this analysis, there may be a deadlock when using it without respecting these assumptions.

Awargal et al. [3] present a type system in order to detect at compile time potential deadlocks on synchronized statements. The information inferred by this static analysis is used in order to restrict the runtime checks of possible deadlocks only on locks that are not proved to be deadlock free at compile time. The analysis has not been implemented but only applied by hand. The authors studied in the details the speedup of the runtime that uses this information.

6.8.2 Other properties

Many static analyses and tools have been proposed in order to detect accesses to null pointers [132, 40, 68, 69], divisions by zero [30, 48], and overflows [74]. Without entering in the details, usually these approaches are sound only for sequential programs and they do not support concurrency. On the other hand, they are often more precise than the ones obtained applying **Checkmate** to these properties, as we applied our analyzer to this property without developing a specific analysis, e.g. a specific numerical domain.

6.9 Discussion

In this chapter we presented **Checkmate**, a generic static analyzer of Java multithreaded programs. We implemented some well-known non-relational numerical domains, some properties, and some memory models. We applied it to some interesting case studies, to an incremental application, and to a set of well-known benchmarks. These experimental results show both its precision and efficiency. In this way we applied the theoretical frameworks developed in Chapter 3 and 4 through the semantics of Java bytecode statements introduced in Chapter 5.

7

Static Analysis of Unsafe Code

The last step of our work will be to prove the industrial interest of generic analyzers, and to show the efforts required in order to instantiate such analyzers on a specific property. In this context, we extend an industrial product to the analysis of a specific property. Clousot is a generic static analyzer based on abstract interpretation developed at Microsoft Research. Even if it is sound only at single-thread level, it allows us to investigate generic techniques of static analysis and to show the efforts required in order to apply a generic analyzer to a property of interest. In the future, the same analysis might be introduced in Checkmate. This opens the way in order to apply our prototype to the analysis of industrial programs.

In particular, in this Chapter we apply Clousot to the analysis of unsafe code, i.e. .NET code containing direct pointers. We develop a new focused relational domain (Strp), and we combine it with some other well-known numerical domain improving its precision. We implement the analysis, and we study the experimental results. The analysis results to be both precise and scalable. It is in position to analyze more than 20.000 methods in a couple of minutes, and we find bugs on shipped code analyzing just a small case study. This chapter is based on the published work [47].

7.1 What is Unsafe Code

The .NET framework provides a multi-language execution environment which promotes the safe execution of code. For instance, in (safe) C# it is not possible to have uninitialized variables, unchecked out-of-bounds runtime accesses to arrays or dangling pointers. Memory safety is enforced by the type system and the runtime: it is not possible to access arbitrary memory locations. Object creation and references are allowed freely, but object life-time is managed by a garbage collector and it is not possible to get the address of an object; pointers' arithmetic is not allowed. Many other current programming languages as Java apply the same type of restrictions. As a consequence, safe C# provides a safer execution environment than C or C++.

Nevertheless, there are situations where direct pointer manipulations and direct memory accesses become a necessity. This is the case when interfacing with the underlying operating system, when implementing time-critical algorithms or when accessing memory-

mapped devices. For this purpose, C# provides the ability to write unsafe code (unsafe C#). In unsafe code, it is possible to declare and operate on pointers, to perform arbitrary casts, to take the address of variables or fields. C# provides syntactic sugar to denote blocks of unsafe code, which avoids the accidental use of unsafe features. Unsafe code cannot run in untrusted environments. This is the solution provided by .NET framework; instead Java does not allow at all the free management of pointers. Nevertheless, also Java programs need sometimes to use pointers; in order to do that, the solution provided by this language is to define native methods.

7.2 Design by Contracts

Our work can be seen also as a contribution in the context of the ongoing effort to improve the reliability of the .NET platform by systematic use of the Design by Contracts (DbC) methodology [100] supported by static checking tools.

The basic idea of DbC approach is to define some constraints on how software elements interact. Through these constraints the software reliability is improved, and developers can reason modularly when writing programs.

A huge application of this methodology has been in object-oriented programs. In this context, contracts are commonly used to specify:

- preconditions of methods, i.e. contracts that have to be satisfied each time the method is called;
- postconditions of methods, i.e. contracts that have to be satisfied at the end of the method;
- class invariants, i.e. contracts on class fields that have to be always satisfied.

For instance, consider the following method:

```
int div(int a, int b)
{
    int res=a;
    for(int i=0; i<=b; i++)
        res/=b;
    if (res<0) return -res;
    else return res;
}
```

A precondition of this method is that b must not be equal to zero; indeed, an implicit precondition is that b has to be positive. A valid postcondition is that the result is always ≥ 0 .

7.2.1 Foxtrot

Foxtrot is a language independent solution for contract specifications in .NET. It does not require any source language support or compiler modification. Preconditions and postconditions are expressed by invocations of static methods (`Contract.Requires` and `Contract.Ensures`) at the start of methods. Class invariants are contained in a method with an opportune name (`ObjectInvariant`) or tagged by a special attribute (`[ObjectInvariant]`). Dummy static methods are used to express meta-variables such as e. g. `Contract.Old(x)` for the value in the pre-state of `x` or `Contract.WritableBytes(p)` for the length of the memory region associated with `p`. These contracts are translated to MSIL using the standard source language compiler.

Contracts in the Foxtrot notation (using static method calls) can express arbitrary boolean expressions as pre-conditions and post-conditions. We expect the expressions to be side effect free (and only call side-effect free methods). We use a separate purity checker to optionally enforce this.

A binary rewriter tool enables dynamic checking. It extracts the specifications and instruments the binary with the appropriate runtime checks at the applicable program points, taking contract inheritance into account. Most Foxtrot contracts can be enforced at runtime.

For static checking, Foxtrot contracts are presented to Clousot as simple `assert` or `assume` statements. E.g., a pre-condition of a method appears as an assumption at the method entry, whereas it appears as an assertion at every call-site.

7.3 Our Contribution

Most of the checks commonly enforced by the runtime, such as bounds checking, are not present on pointer manipulating code. As a consequence the programmer is exposed when developing unsafe code to all the vagaries of C/C++ programming, such as buffer and array overflows, reading of un-initialized memory, type safety violations, etc.. Those errors are difficult to detect and track down, as no runtime exception is thrown at the error source. For instance, an application cannot immediately detect that some buffer overflow compromising its data consistency has occurred. Instead, it continues its execution in a bad state, only to fail (much) later due to a corrupted state. Tracing back the cause of such bugs to the original memory corruption is often very complicated and time consuming.

In this context, applying static analysis to check the respects of the bounds of allocated memory is particularly appealing. In order to gain this goal, we extend Clousot with a specific numerical domain in order to develop a precise and fast analysis of unsafe code.

7.3.1 Clousot

Clousot is a generic, language agnostic static analyzer based on abstract interpretation for .NET. It is generic in that it presents a pluggable architecture: analyses can be easily

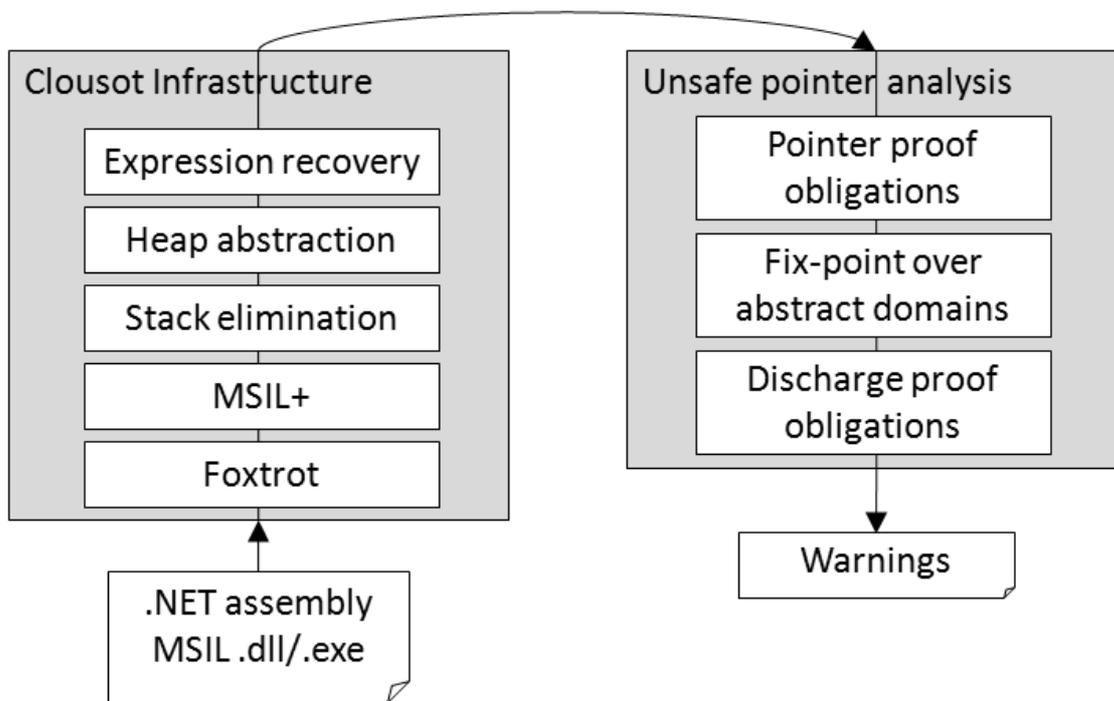


Figure 7.1: Clousot architecture

ldstack.i	duplicate i-th value on evaluation stack
ldresult	load the current result value
assert	assert top of stack is true
assume	assume top of stack is true
begin_old	evaluate next instructions in method pre-state
end_old	switch back to state at matching begin_old

Table 7.1: MSIL+ synthetic instructions

added by providing an implementation of a suitable abstract domain interface. In particular, it can be plugged with different numerical domains and in order to analyze different properties. It is language agnostic as it analyzes MSIL. All the programming languages in .NET emit MSIL: using the debug information we can trace back the results of the analysis to the source program.

Clousot has a layered structure as shown in Fig. 7.1. Each layer on the left presents an increasingly abstract view of the code. An MSIL reader sits at the lowest level, which presents a stack-based view of the code. Above that sits the Foxtrot extractor, which turns the dummy method calls expressing pre- and post-conditions into actual representations of these, separating them from the method body. The analysis performed by Clousot is intra-procedural. When a method is called, contracts written through Foxtrot are extracted; then Clousot checks if preconditions are respected, and assumes

postconditions in order to go on with the analysis after the method call.

The layer labeled MSIL+ represents an extension of MSIL with a number of synthetic instructions that allow us to express all contract code as simple stack instructions, similar to MSIL. The extensions used are listed in Table 7.1. Instruction `ldstack.i` is a generalization of a typical `dup` instruction that allows one to access values on the evaluation stack that are not at the top. This instruction is useful for example to access the parameters inside a pre-condition inserted at a call-site. The `ldresult` instruction is used in post-conditions to refer to the result of the method. The meaning of `assert` and `assume` is equivalent for run-time checking: they both result in failure if the condition is false. For static checking, they differ in that the checker tries to validate an `assert` condition and issues an error if it cannot be proven. However, the static checker simply adds the condition of an `assume` to its knowledge base without trying to validate it.

The next layers in the Clousot infrastructure (1) get rid of the stack by providing a view of the code in the 3-address form (the direct analysis of a stack-based language is hard and error-prone, [66]); (2) abstract away the heap by providing a view of the code as a scalar program, where aliasing has been resolved (a common approach to separate heap-analysis and value analysis, e. g. [18, 91]); and (3) reconstruct (most of the) expressions that have been lost during the compilation (large chunks of expressions are vital for a precise static analysis [92]).

On top of this infrastructure we build particular analyses, such as the one presented in this paper regarding unsafe memory accesses. Such analyses are built out of atomic abstract domains (e. g. `Intv`, `LinEq`, `Pntg`[93]), a set of generic domains (e. g. set of constraints), and a set of operators on abstract domains (e. g. the reduced cartesian product [27], the functional lifting). As a consequence Clousot allows building new and powerful abstract domains by refinement and composition of existing ones.

7.3.2 Applying Clousot to the Analysis of Unsafe Code

Our analysis infers and checks the memory regions accessed by read and write operations. A region of memory is denoted by a pair $\langle p, \text{WB}(p) \rangle$, where p is a pointer and $\text{WB}(p)$ stands for the WritableBytes of p , i.e., the size of the region in bytes accessible from p . We only allow positive offsets of pointers, thus $\text{WB}(p)$ is always non-negative.

Differently stated, the pair stands for the range of addresses $[p, p + \text{WB}(p) - 1]$. For instance, if x is an `Int32` and p is an `Int32*`, then the read operation $x = *(p + 2)$ is safe in the region $\langle p, 12 \rangle$: It reads 4 bytes (the size of an `Int32` in .NET) starting from the address $p + 8$ (as p is a pointer to `Int32`).

We use a combination of three domains to infer bounds on memory-accessing expressions. The core is the new abstract domain of Stripes (`Strp`) which captures properties of the form of $x - a * (y[+z]) \geq b$, where a and b are integer constants, x and y are variables and z is an optional variable ($[+z]$ means that z may be part of the constraint or not). Intuitively, a stripe constraint is used to validate the upper bound on memory accesses. Intervals (`Intv`) [25] are used to validate the lower bound of accesses. We use (a

modified version of) the Linear equalities domain (LinEq) [76] to track equalities between variables.

We implemented our analysis in Clousot, a generic, intraprocedural and language-agnostic static analyzer for .NET[9, 93]. It uses FoxTrot contracts to refine the analysis and to support assume/guarantee reasoning for method calls. FoxTrot allows specifying contracts in .NET without requiring any language support. Contracts are expressed directly in the language as method calls and are persisted to MSIL using the normal compilation process of the source language. We tried our analysis on all the assemblies of the .NET framework, validating on average about 57% of unsafe memory accesses automatically in a few minutes. In practice, the false alarms that we get are due to missing contracts: the use of contracts will allow us to improve the precision. The analysis is fast enough to be used in test builds.

The main contributions of the present work can be summarized as follows:

- We introduce the first static analysis to check memory safety in unsafe managed code. Our analysis handles the entire MSIL instruction set and is fully implemented in Clousot. This analyzer statically checks contracts, and can use them to refine the precision of the analysis, e. g. by exploiting preconditions. We tested it on all the assemblies of the .NET framework.
- We define the concrete and abstract semantics for an idealized MSIL-like bytecode. We prove soundness by using the abstract interpretation framework to relate the abstract semantics with the concrete semantics.
- We present a new abstract domain for the analysis of memory bounds. It is based on the co-operation of several specialized domains. We prove its soundness, and we show how it is effective in practice, by enabling a fast, yet precise analysis.
- We discuss some implementation issues necessary to avoid loss of precision, as e. g. the special handling that is required for the C# fixed statements (fixed is used to set a pointer to the address of an allocated area of memory, and to pin this area during the execution of the following sequence of instructions in order to prevent the garbage collector from moving it).

7.4 Examples

In order to develop a specific analysis for unsafe code, we need to look deeply into the peculiarities of this code. So our work starts analyzing the methods in .NET framework that deals with pointers. In this section, we report some representative examples taken from or inspired by this framework.

```
public virtual unsafe int GetByteCount(char* chars, int count, bool flush)
{
    // ... rest of the method omitted
    char[] chArray = new char[count];
    for (int i = 0; i < count; i++)
    {
        chArray[i] = chars[i];
    }
    // ... rest of the method omitted
}
```

Figure 7.2: A method that copy count chars starting from a given pointer

7.4.1 From Source Code to MSIL

When unsafe code is compiled into MSIL, some transformations are automatically done, following the type of pointers. When we access the i -th element starting from a pointer ptr of type $type$, we are accessing $sizeof(type)$ bytes starting from the $sizeof(type) * i$ -th byte of ptr . This operation is made explicit at MSIL level.

For instance consider the method depicted by Figure 7.2. The access $chars[i]$ performed inside the for loop is compiled into the following bytecode:

```
ldarg.1
ldloc.1
conv.i
ldc.i4.2
mul
add
ldind.u2
```

Without entering too much into details of bytecode semantics, we have that

- `ldarg.1` loads the argument `chars`,
- `ldloc.1` loads the local variable `i`,
- it is converted to `int`, it is multiplied with the integer constant 2 (i.e. `sizeof(char)`),
- finally, the value is read.

This piece of MSIL code can be decompiled into the arithmetical expression $*(chars + 2 * i)$. The unsafe access respects the bound of the allocated memory if and only if `chars` is defined at least on $*(chars + 2 * i + 1)$ bytes (as $*(chars + 2 * i)$ reads two bytes starting from index $2 * i$), i.e. $WB(chars) \geq 2 * i + 1$.

```

static unsafe void InitToZero(int* a, uint len)
{
    Contract.Requires(Contract.WritableBytes(a) >= len * sizeof(int ));

    for (int i = 0; i < len; i++)
    {
        *(a + i) = 0; // (1)
    }
}

```

Figure 7.3: A method that zeros a region of memory

7.4.2 Array Initialization

Consider the `InitToZero` method in Fig. 7.3. It initializes the memory region $[a, a+4*\text{len}-1]$ to zero. The precondition requires that at least $\text{len} * \text{sizeof}(\text{int})$ bytes starting from `a` are allocated. We express it using `FoxTrot` notation: contracts are specified by static method calls (e. g. `Contract.Requires(...)` for preconditions), and lengths of memory regions are denoted by `Contract.WritableBytes(...)`. Section 7.2.1 contains more information about contracts.

The write operation at (1) is correct provided that: (a) $i \geq 0$, and that (b) $\text{WB}(a) - 4 * i \geq 4$. We prove (a) using the `Intv` abstract domain, which infers the loop invariant $i \geq 0$. We prove (b) using the `Strp` abstract domain, which propagates the entry state $\text{WB}(a) - 4 * \text{len} \geq 0$ to the loop entry point, discovering the loop invariant $\text{WB}(a) - 4 * (i + 1) \geq 0$.

7.4.3 Callee Checking

Methods such as `InitToZero` that use unsafe pointers are typically internal to the .NET framework and accessible only through safe wrappers such as `FastInitToZero` shown in Fig. 7.4. This code casts the parameter array of `int` to a pointer to `int`, and then invokes `InitToZero`. This pattern of a safe wrapper around unsafe pointer manipulating code is pervasive in the .NET framework. Using our analysis together with method pre-conditions allows us to validate that callers into the framework cannot cause unintended memory access via the internal pointer operations.

In this example, `Clousot` figures out that at line 4 of Figure 7.4 the invariant $\text{WB}(a) = 4 * \text{arr.Length}$ holds, which is enough to prove the pre-condition of `InitToZero`. In order to track affine linear equalities as above, we use the abstract domain of `LinEq`. The combination of `Strp`, `Intv` and `LinEq` allows us to precisely analyze memory accesses in unsafe code without turning to expensive (exponential) abstract domains.

```
1 static public unsafe void FastInitToZero(int[] arr)
2 {
3   fixed (int* a = arr)
4   {
5     InitToZero(a, (uint) arr.Length);
6   }
7 }
```

Figure 7.4: Passing to functions pointers to arrays

7.4.4 Interaction with the Operating System

Unsafe code is also necessary for interfacing with the underlying operating system. Consider the code in Fig. 7.5. `FastCopy` uses the `CopyMemory` method from the Win32 API to copy the values of the array `s` into the array `p`. `FoxTrot` allows attaching external contracts to assemblies, and in particular to annotate external calls. For the sake of presentation, we made these contracts explicit in a proxy method.

The precondition for `CopyMemory`, informally stated in the Win32 documentation, is formalized in `CopyMemoryProxy`. It requires that (a) the destination buffer is large enough to hold `szSrc` bytes; (b) the two buffers are defined at least on the memory regions accessed by `CopyMemory`.

`Clousot` can then statically check the right usage of the API. For instance, it checks that `FastCopy` satisfies the precondition, provided that the length of the destination array is not strictly smaller than the source.

Discussion: Application to security. The example shows the relevance of our analysis to enforce security. Unsafe code in the .NET framework is a potential security risk if it is exploitable from safe managed code. Analyses such as `Clousot` provide more confidence that the managed to unmanaged transition does not expose the framework to such attacks. The same technique could be applied at the Java to native boundary which exhibits the same problems.

7.5 Syntax and Concrete Semantics

We present an idealized and simplified subset of MSIL, `uMSIL`. We define its transition semantics. The concrete semantics is instrumented to trace the region of allocated memory associated with a pointer. We treat out-of-region memory accesses as errors.

In `Checkmate`, this part corresponds to

- Section 5.2, in which we defined a representation of the Java bytecode language,
- Section 5.4, in which we defined the concrete domain,

```
[DllImport("kernel32.dll")]
unsafe static extern void CopyMemory(char* pdst, char* psrc, int size);

static unsafe private void CopyMemoryProxy(char* pdst, char* psrc, int szdst, int szsrc)
{
    Contract.Requires(szdst >= 0 && szsrc >= 0);
    Contract.Requires(szdst >= szsrc);
    Contract.Requires(Contract.WritableBytes(pdst) >= szdst*sizeof(char));
    Contract.Requires(Contract.WritableBytes(psrc) >= szsrc*sizeof(char));

    CopyMemory(pdst, psrc, szsrc);
}

public unsafe static void FastCopy(char[] d, char[] s)
{
    Contract.Requires(d.Length >= s.Length);

    fixed (char* pdst = d, psrc = s)
    {
        CopyMemoryProxy(pdst, psrc, d.Length, s.Length);
    }
}
```

Figure 7.5: An example illustrating the invocation of the Win32 API

- Section 5.5, in which we defined the transition semantics.

7.5.1 Syntax

We focus our attention on the MSIL instructions that are particular to our unsafe analysis. Thus, we do not discuss: (a) instructions that are “standard” such as jumps, assignments, method invocations, etc. (b) issues that are orthogonal to the unsafe code analysis, such as the precise handling of tests, expressions refinement, etc. We refer the interested reader to [92] that debates these topics and depicts the solution implemented in Clousot.

The instruction set we consider, uMSIL, is shown in Tab. 7.2. $T * p = \text{stackalloc } T[\text{exp}]$ allocates exp elements of type T on the stack. In .NET, memory can be allocated in the heap in two ways : (a) use the `new` keyword to allocate an object or (b) directly call the underlying operating system (e. g. by using the `HeapAlloc` Win32 API). In general, the garbage collector is free to move heap allocated objects. However, the construct `fixed(T * p = &x + exp){istr}` (a) sets a pointer p to the address $\&x + \text{exp}$; and (b) pins the variable p during the execution of the sequence of instructions istr , to prevent the garbage collector from moving it. The instruction $x = *(p + \text{exp})$ reads the value at address $p + \text{exp}$ and stores its value in x whereas $*(p + \text{exp}) = x$ stores at the address $p + \text{exp}$

```

istr ::= T * p = stackalloc T[exp]
      | fixed(T * p = &x + exp) { istr }
      | x = *(p + exp)
      | *(p + exp) = x
      | istr; istr

```

Table 7.2: uMSIL: an idealized version of the MSIL instructions

the value of x . Finally, we have instruction sequencing.

7.5.2 Concrete Domain

Let Var be a set of variables, let Add be a set of addresses, \mathbb{Z} be the set of numerical values (note that $\text{Add} \subseteq \mathbb{Z}$) and Ω a special state standing for a program error. For each variable $v \in \text{Var}$ we express by $\text{WB}(v)$ the number of bytes on which it is defined (if it is not a pointer, the domain would not trace information about it). We let $\text{WB}(\text{Var}) = \{\text{WB}(v) \mid v \in \text{Var}\}$ and $\text{Var}_{\text{WB}} = \text{Var} \cup \text{WB}(\text{Var})$.

The domain of concrete execution states is

$$C = ([\text{Var}_{\text{WB}} \rightarrow \mathbb{Z}] \times [\text{Add} \rightarrow \text{Byte}] \times \text{Add}) \cup \{\Omega\}$$

A concrete state is either: (a) a tuple consisting of an environment f mapping variables to values, a memory g mapping addresses to bytes, and a set t of addresses of objects pinned for the garbage collector, or (b) the special value Ω denoting that an error has occurred.

7.5.3 Concrete Transition Semantics

Figure 7.6 formally defines the concrete transition semantics. We use some auxiliary functions:

- $eval(\text{exp}, (f, g))$ evaluates a side-effect free expression exp in state (f, g) ;
- $alloc(T, n, g)$ returns a pair $\langle a, g' \rangle$ where a is the starting address of a freshly allocated region of g containing n elements of type T , and g' is the modified memory;
- $write(g, a, n, v)$ returns the updated memory $g[a + i \mapsto v_{[i]} \mid i \in [0, n]]$, $v_{[k]}$ denotes the k -th significant byte of v ;
- $read(g, a, n)$ reads n bytes from memory g and returns them packed as an integer;
- $sizeof(T)$ and $sizeof(x)$ return the length, expressed in bytes, respectively of an element of type T and of the variable x .

$$\frac{eval(exp, (f, g)) < 0}{\mathbb{C}[\mathbb{T} * p = stackalloc \mathbb{T}[exp]](f, g, t) \rightarrow \Omega}$$

$$\frac{n = eval(exp, (f, g)), n \geq 0, \langle a, g' \rangle = alloc(\mathbb{T}, n, g), f' = f[p \rightarrow a][WB(p) \rightarrow n * sizeof(\mathbb{T})]}{\mathbb{C}[\mathbb{T} * p = stackalloc \mathbb{T}[exp]](f, g, t) \rightarrow (f', g', t)}$$

$$\frac{WB(p) \notin dom(f) \vee eval(exp, (f, g)) < 0 \vee f(WB(p)) < sizeof(x) + eval(exp, (f, g)) * sizeof(*p)}{\mathbb{C}[*p + exp = x](f, g, t) \rightarrow \Omega}$$

$$\frac{WB(p) \in dom(f), n = eval(exp, (f, g)), n \geq 0, f(WB(p)) \geq sizeof(x) + n * sizeof(*p), g' = write(g, f(p) + n * sizeof(*p), sizeof(*p), f(x))}{\mathbb{C}[*p + exp = x](f, g, t) \rightarrow (f, g', t)}$$

$$\frac{WB(p) \notin dom(f) \vee f(eval(exp, (f, g))) < 0 \vee f(WB(p)) < sizeof(x) + eval(exp, (f, g)) * sizeof(*p)}{\mathbb{C}[x = *(p + exp)](f, g, t) \rightarrow \Omega}$$

$$\frac{WB(p) \in dom(f), n = eval(exp, (f, g)), n \geq 0, f(WB(p)) \geq sizeof(x) + n * sizeof(*p), v = read(g, f(p) + n * sizeof(*p), sizeof(x)), f' = f[x \rightarrow v]}{\mathbb{C}[x = *(p + exp)](f, g, t) \rightarrow (f', g, t)}$$

$$\frac{\text{var is a } T \text{ array, } t' = t \cup \{f(\text{var})\} \quad \mathbb{C}[\text{istr}](f', g, t') \rightarrow (f'', g'', t'')}{f' = f \quad [p \rightarrow f(\text{var}) + (eval(exp, (f, g))) * sizeof(T)] \quad [WB(p) \rightarrow (eval(\text{var.length}, -)exp(f, g)) * sizeof(T)]}{\mathbb{C}[\text{fixed}(T * p = \&\text{var} + exp)\{\text{istr}\}](f, g, t) \rightarrow (f'', g'', t)}$$

$$\frac{\mathbb{C}[\text{istr}_1](f, g, t) \rightarrow \Omega}{\mathbb{C}[\text{istr}_1; \text{istr}_2](f, g, t) \rightarrow \Omega}$$

$$\frac{\text{var is a string, } t' = t \cup \{f(\text{var})\} \quad \mathbb{C}[\text{istr}](f', g, t') \rightarrow (f'', g'', t'')}{f' = f \quad [p \rightarrow f(\text{var}) + (eval(exp, (f, g))) * 2] \quad [WB(p) \rightarrow (eval(\text{var.Length} - exp(f, g)), *)2]}{\mathbb{C}[\text{fixed}(T * p = \&\text{var} + exp)\{\text{istr}\}](f, g, t) \rightarrow (f'', g'', t)}$$

$$\frac{\mathbb{C}[\text{istr}_1](f, g, t) \rightarrow (f', g', t')}{\mathbb{C}[\text{istr}_1; \text{istrs}_2](f, g, t) \rightarrow \mathbb{C}[\text{istr}_2](f', g', t')}$$

Figure 7.6: The concrete transition semantics

The description of the transitions in Fig. 7.6 follows. The semantics for `stackalloc` first evaluates `exp`. If it is negative, it fails. Otherwise, it allocates a new region, sets a pointer for it to `p` and records the length of the region, expressed in bytes, in `WB(p)`.

A write operation `*(p + exp) = x` stores a number of bytes equal to the size of the type of `x` in the memory location `p + exp * sizeof(*p)`. If the region for `p` does not contain at least `sizeof(x) + exp * sizeof(*p)` bytes, a buffer overrun occurs, denoted by the error state Ω . The read operation is analogous.

The semantics for `fixed` is defined according to the type of `var`. In the two cases, (a) `p` will point to a memory address that is obtained by combining the address value `f(var)` and the offset `exp * s`, where `s` is the size of the elements; (b) the address of the pinned object `f(var)` is added to the set of pinned objects during the execution of `st`. As for the length of the memory regions associated with `p`: when `var` is (a) an array, then the size of the memory region associated with `p` is given by the length of the array minus the offset of the first element times the size of an element; (b) a string, then `p` will point to an element to the internal representation of the string as an array of `char`, and the length of the memory regions is computed accordingly.

The semantics of a sequence of instructions is the compositions of the semantics, unless the result is Ω . In this case, the error state is propagated.

7.6 Abstract Semantics

We derive our analysis by stepwise abstraction, [24]. First, we abstract away the values read and written through pointers and the aliasing introduced by the `fixed` instruction. Then, we derive a generic analysis for checking buffer overruns. The analysis is parameterized by the numerical abstract domain used to evaluate region indices.

In Checkmate, this part corresponds to

- Section 5.8, in which we defined the abstract domain,
- Section 5.9, in which we defined the abstract transition semantics,
- Section 5.10, in which we proved the soundness of our approach.

7.6.1 Abstracting Away the Values

The Domain

We preserve just the information on memory regions. We abstract away the second and the third component of \mathcal{C} , and we project the first component onto the memory regions, i.e. `WB(Var)`. The abstract domain is $\overline{\mathcal{C}} = ([WB(Var) \rightarrow \mathbb{Z} \cup \{\top\}]) \cup \{\Omega_?\}$. We add (a) \top to model values that are abstracted away, (b) $\Omega_?$ to model a set of concrete states that may contain the error state Ω .

$$\begin{array}{c}
\frac{!(\overline{eval}(\text{exp}, \bar{f}) \geq 0)}{\mathbb{A}[\mathbb{T} * \text{p} = \text{stackalloc } \mathbb{T}[\text{exp}]](\bar{f}) \rightarrow \Omega_?} \\
\\
\frac{\overline{eval}(\text{exp}, \bar{f}) \geq 0, \bar{f}' = \bar{f}[\text{WB}(\text{p}) \rightarrow \overline{eval}(\text{exp}, \bar{f}) * \text{sizeof}(\mathbb{T})]}{\mathbb{A}[\mathbb{T} * \text{p} = \text{stackalloc } \mathbb{T}[\text{exp}]](\bar{f}) \rightarrow \bar{f}'} \\
\\
\frac{!(\overline{eval}(\text{exp}, \bar{f}) \geq 0) \vee \text{WB}(\text{p}) \notin \text{dom}(\bar{f}) \vee !(\bar{f}(\text{WB}(\text{p})) \geq \text{sizeof}(\text{x}) + \overline{eval}(\text{exp} * \text{sizeof}(*\text{p}), \bar{f}))}{\mathbb{A}[*\text{(p} + \text{exp)} = \text{x}]](\bar{f}) \rightarrow \Omega_?} \\
\\
\frac{\text{WB}(\text{p}) \in \text{dom}(\bar{f}), \overline{eval}(\text{exp}, \bar{f}) \geq 0, \bar{f}(\text{WB}(\text{p})) \geq \text{sizeof}(\text{x}) + \overline{eval}(\text{exp} * \text{sizeof}(*\text{p}), \bar{f})}{\mathbb{A}[*\text{(p} + \text{exp)} = \text{x}]](\bar{f}) \rightarrow \bar{f}} \\
\\
\frac{!(\overline{eval}(\text{exp}, \bar{f}) \geq 0) \vee \text{WB}(\text{p}) \notin \text{dom}(\bar{f}) \vee !(\bar{f}(\text{WB}(\text{p})) \geq \text{sizeof}(\text{x}) + \overline{eval}(\text{exp} * \text{sizeof}(*\text{p}), \bar{f}))}{\mathbb{A}[\text{x} = *\text{(p} + \text{exp)}]](\bar{f}) \rightarrow \Omega_?} \\
\\
\frac{\text{WB}(\text{p}) \in \text{dom}(\bar{f}), \overline{eval}(\text{exp}, \bar{f}) \geq 0, \bar{f}(\text{WB}(\text{p})) \geq \text{sizeof}(\text{x}) + \overline{eval}(\text{exp} * \text{sizeof}(*\text{p}), \bar{f})}{\mathbb{A}[\text{x} = *\text{(p} + \text{exp)}]](\bar{f}) \rightarrow \bar{f}} \\
\\
\text{var is } T \text{ array} \\
\frac{\bar{f}' = \bar{f}[\text{WB}(\text{p}) \rightarrow \overline{eval}(\text{var.length} - \text{exp}, \bar{f}) * \text{sizeof}(\mathbb{T})], \bar{f}'' = \mathbb{A}[\text{istr}]](\bar{f}')}{\mathbb{A}[\text{fixed}(\mathbb{T} * \text{p} = \&\text{var} + \text{exp})\{\text{istr}\}]](\bar{f}) \rightarrow \bar{f}''} \\
\\
\frac{\mathbb{A}[\text{istr}_1]](\bar{f}) \rightarrow \Omega_?}{\mathbb{A}[\text{istr}_1; \text{istr}_2]](\bar{f}) \rightarrow \Omega_?} \\
\\
\text{var is a string} \\
\frac{\bar{f}' = \bar{f}[\text{WB}(\text{p}) \rightarrow (\overline{eval}(\text{var.Length} - \text{exp}, \bar{f})) * 2], \bar{f}'' = \mathbb{A}[\text{istr}]](\bar{f}')}{\mathbb{A}[\text{fixed}(\mathbb{T} * \text{p} = \&\text{var} + \text{exp})\{\text{istr}\}]](\bar{f}) = \bar{f}''} \\
\\
\frac{\mathbb{A}[\text{istr}_1]](\bar{f}) \rightarrow \bar{f}'}{\mathbb{A}[\text{istr}_1; \text{istr}_2]](\bar{f}) \rightarrow \mathbb{A}[\text{istr}_2]](\bar{f}')}
\end{array}$$

Figure 7.7: The abstract transition semantics for uMSIL

The Abstract Transition Semantics

The abstract semantics is in Fig. 7.7. The abstract function \overline{eval} lifts its concrete counterpart to handle \top . \top values occur for instance when exp contains a variable x whose value is read through a pointer and we do not trace the value for x . \overline{eval} simply propagates \top through all strict operator positions, e.g., $\overline{eval}(5 + \top, f) = \overline{eval}(\top, f) = \top$.

The semantics is a little bit more than the projection of the concrete semantics on its first component: if $\overline{eval}(\text{exp}, f) = \top$, then we cannot decide if $\text{exp} \geq 0$ and hence if a buffer overrun has occurred. In this case, we force the transition to the Ω_γ state, which means that a buffer overrun may occur.

For the fixed instruction, we abstract away (a) the fact that the object is pinned: in our abstract semantics we do not need to model the garbage collector; (b) the aliasing between p and $\&\text{var} + \text{exp}$: we are interested just in checking that memory accesses are valid.

Abstraction and Concretization Function

The concretization function returns the set of all the concrete states such that the first component is compatible with one of the abstract states. If the abstract state contains the unknown state Ω_γ , then all the concrete states are returned, included the error state Ω . As a consequence, in order to show that a program has no memory access violations, it suffices to prove that its abstract semantics in Fig. 7.7 never reduces to Ω_γ .

The next two theorems guarantee the soundness of the approach. The first states that the abstract elements are a correct approximation of the abstract ones. The second one states that no concrete behavior is forgotten in the abstract semantics.

THEOREM 7.6.1 *Soundness of the abstraction. Let $\gamma : [\wp(\overline{\mathbb{C}}) \rightarrow \wp(\mathbb{C})]$ be the concretization function defined as*

$$\begin{aligned} \gamma(\overline{F}) = & \bigcap_{\overline{f} \in \overline{F}} \{(f, g, t) \mid \forall \text{WB}(p) \in \text{dom}(\overline{f}). \overline{f}(\text{WB}(p)) \neq \top \\ & \implies f(\text{WB}(p)) = \overline{f}(\text{WB}(p)) \wedge p \in \text{dom}(f)\} \\ & \cup \{(f, g, t) \mid \Omega_\gamma \in \overline{F}\}. \end{aligned}$$

Then γ is a complete \cap -morphism, so that it exists an abstraction function $\alpha : [\wp(\mathbb{C}) \rightarrow \wp(\overline{\mathbb{C}})]$ such that $\wp(\mathbb{C}) \xrightleftharpoons[\alpha]{\gamma} \wp(\overline{\mathbb{C}})$ as proved by Theorem 2.2.3.

THEOREM 7.6.2 *Soundness of the abstract semantics. Let $\overline{f} \in \overline{\mathbb{C}}$, $(f, g, t) \in \gamma(\overline{f})$ and $\text{ist} \in \text{uMSIL}$. If $\mathbb{A}[\text{ist}](\overline{f}) \rightarrow \overline{f}'$ and $\mathbb{C}[\text{ist}](f, g, t) \rightarrow (f', g', t')$, then $(f', g', t') \in \gamma(\overline{f}')$.*

7.6.2 Generic Memory Access Analysis

If we extend uMSIL with (conditional) jumps, e. g. to enable loops, then the abstract semantics in Fig. 7.7 will no longer be computable. In particular, the expressions used for memory accesses may evaluate to infinitely many values. As a consequence, in order to

$$\frac{check(exp \geq 0, \bar{s}) = \top}{\mathbb{F}[\![T * p = stackalloc T[exp]]\](\bar{f}) \rightarrow \Omega,}$$

$$\frac{check(exp \geq 0, \bar{s}) = true, \bar{s}' = assign(WB(p), size * sizeof(T), \bar{s})}{\mathbb{F}[\![T * p = stackalloc T[exp]]\](\bar{s}) \rightarrow \bar{s}'}$$

$$\frac{check(WB(p) \geq sizeof(x) + exp * sizeof(*p), \bar{s}) = \top \vee check(exp \geq 0, \bar{s}) = \top}{\mathbb{F}[\![*(p + exp) = x]]\](\bar{s}) \rightarrow \Omega,}$$

$$\frac{check(WB(p) \geq sizeof(x) + exp * sizeof(*p), \bar{s}) = true, check(exp \geq 0, \bar{s}) = true}{\mathbb{F}[\![*(p + exp) = x]]\](\bar{s}) \rightarrow \bar{s}}$$

$$\frac{check(WB(p) \geq sizeof(x) + exp * sizeof(*p), \bar{s}) = \top \vee check(exp \geq 0, \bar{s}) = \top}{\mathbb{F}[\![x = *(p + exp)]\](s) \rightarrow \Omega,}$$

$$\frac{check(WB(p) \geq sizeof(x) + exp * sizeof(*p), \bar{s}) = true, check(exp \geq 0, \bar{s}) = true}{\mathbb{F}[\![x = *(p + exp)]\](\bar{s}) \rightarrow \bar{s}}$$

$$\frac{\text{var is a } T \text{ array}}{\bar{s}' = assign(WB(p), (var.length - exp) * sizeof(T), \bar{s}), \mathbb{F}[\![istr]]\](\bar{s}') \rightarrow \bar{s}''}$$

$$\frac{\mathbb{F}[\![istr_1]]\](\bar{s}) \rightarrow \Omega,}{\mathbb{F}[\![istr_1; istr_2]]\](\bar{s}) \rightarrow \Omega,}$$

$$\frac{\text{var is a string}}{\bar{s}' = assign(WB(p), (var.length - exp) * 2, \bar{s}), \mathbb{F}[\![istr]]\](\bar{s}') \rightarrow \bar{s}''}$$

$$\frac{\mathbb{F}[\![istr_1]]\](\bar{s}) \rightarrow \bar{s}'}{\mathbb{F}[\![istr_1; istr_2]]\](\bar{s}) \rightarrow \mathbb{F}[\![istr_2]]\](\bar{s}')}$$

Figure 7.8: The generic abstract semantics for memory access validity checking

cope with a more realistic scenario, we need to perform a further abstraction, to capture the values of index expressions.

We assume a numerical domain \bar{N} which correctly approximates $\wp(\bar{C})$ ($\langle \wp(\bar{C}), \sqsubseteq \rangle \stackrel{\gamma_{\bar{N}}}{\leftarrow} \langle \bar{N}, \sqsubseteq \rangle$) and with two primitives: (a) $assign(x, exp, s) \in \bar{N}$ which is (an overapproximation of) the assignment $x := exp$ in the abstract state $s \in \bar{N}$; (b) $check(exp, s) \in \{true, \top\}$ which checks whether, in the abstract state $s \in \bar{N}$, the expression exp holds ($true$) or it cannot be decided (\top).

The generic abstract semantics for checking memory safety, parameterized by \bar{N} is reported in Fig. 7.8.

7.7 The Right Numerical Abstract Domain

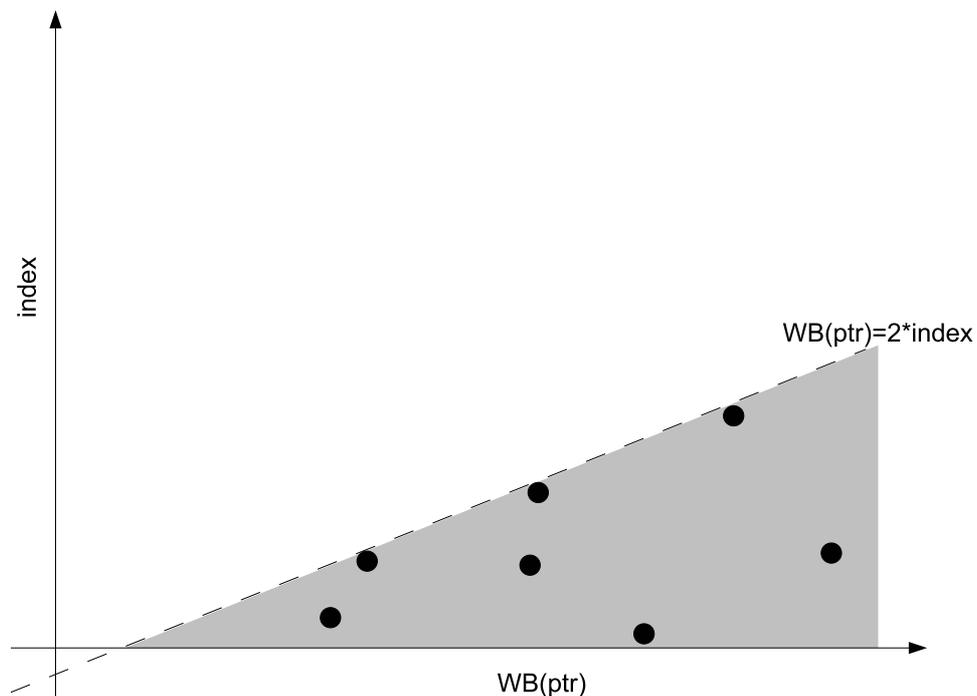


Figure 7.9: The concrete points, and the property of interest

The generic abstract semantics in Fig. 7.8 can be instantiated with any numerical abstract domain containing the primitives `assign` and `check`. As a consequence the problem of checking the validity of memory accesses boils down to the problem of choosing the right abstract domain. In particular, we want to obtain a scalable and precise analysis. In

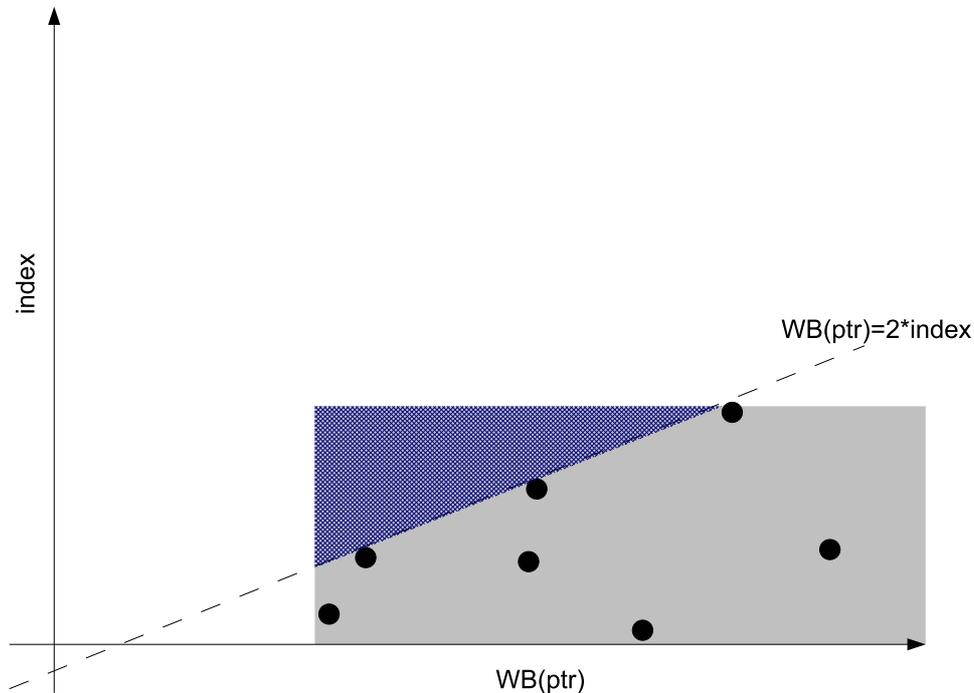


Figure 7.10: The abstraction of Intv

this way, we need to use a domain that is not expensive but enough accurate in order to capture the property of interest.

First of all, we investigate if any existing domain fill these requirements when analyzing unsafe code.

Let us consider the set of points \mathcal{A} of Fig. 7.9 corresponding to all the possible values that $WB(ptr)$ and $index$ assume at some memory access $c = *(ptr + index)$. Supposing that the type of $WB(ptr)$ is `char`, we want that $WB(ptr) \geq 2 * index$ holds always. This is the property of interest: if we access the memory through `ptr` with an `index` that does not respect this constraint, the bounds are not respected. So in this figure we color in grey the area in which the property is respected.

Fig. 7.10 shows that `Intv` alone is not precise enough for our purposes: the best approximation for \mathcal{A} with `Intv` goes outside the grey area, and so false alarms are produced by this domain. Intuitively, this is because `Intv` does not keep relational information, e. g., any relation between $WB(p)$ and $index$ is abstracted away.

Weakly-relational numerical abstract domains such as Octagons [101] (that captures relations in the form $\pm x \pm y \leq a$) or Pentagons [93] ($a \leq x \leq b \wedge x < y$) have been introduced as lightweight solutions for array bounds checking. Fig. 7.11 shows that

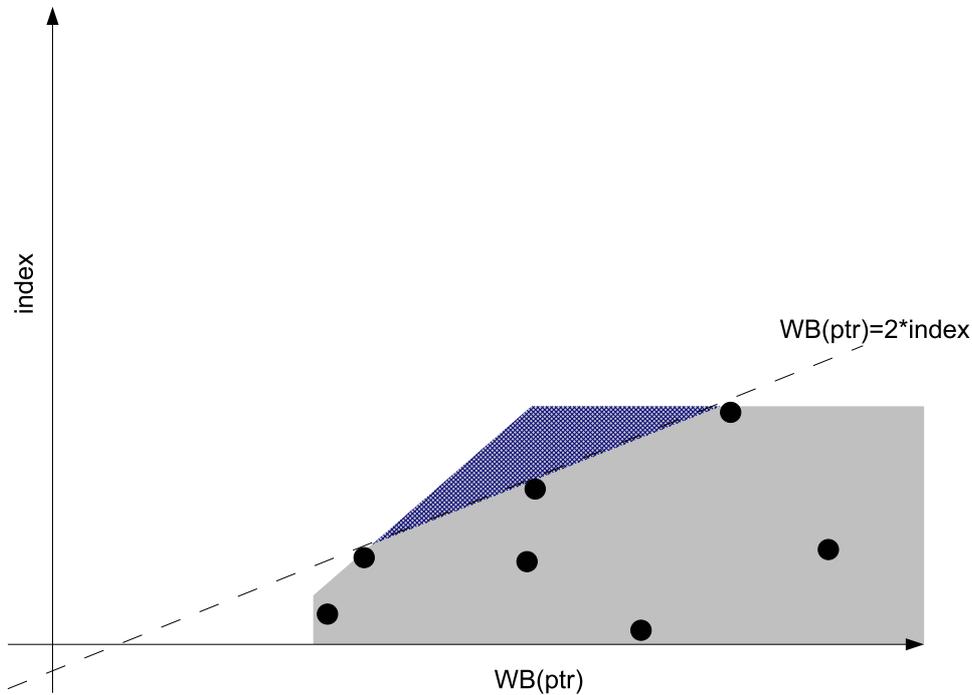


Figure 7.11: The abstraction of Oct

Octagons are more precise than `Intv`, but they are still not precise enough to validate memory accesses due to the multiplicative factor `sizeof(*p)` which makes the slopes in Fig.7.9 possibly non-45°.

Fig. 7.12 shows that `Poly` ($\sum_i a_i \cdot x_i \leq b$) is precise enough, as approximates correctly the concrete points. The main drawback of using `Poly` is its worst case cost, which for the most common operations is exponential in time and space (and this is a lower-bound [77]). But is the worst case common when analyzing unsafe code or it never happens in this context?

We apply the implementation of `Poly` in `Boogie` [8] to the analysis of unsafe code. Although this implementation of `Polyhedra` is not as optimized as for example [7], it has been well debugged and in use for a number of years. The results are shown in Table 7.3 (the experiments were conducted on a 2.4Ghz Intel Core Duo laptop, with 4Gbytes of RAM, running Windows Vista).

In our runs, we used a 2 minute timeout per method. The timeout was reached 23 times on `mscorlib.dll` and 13 times on `System.dll`. In all fairness, the `Parma` library [7] is likely to be much faster than the implementation of `Polyhedra` we used. However, it is unlikely to consistently improve the execution by two orders of magnitude and it would

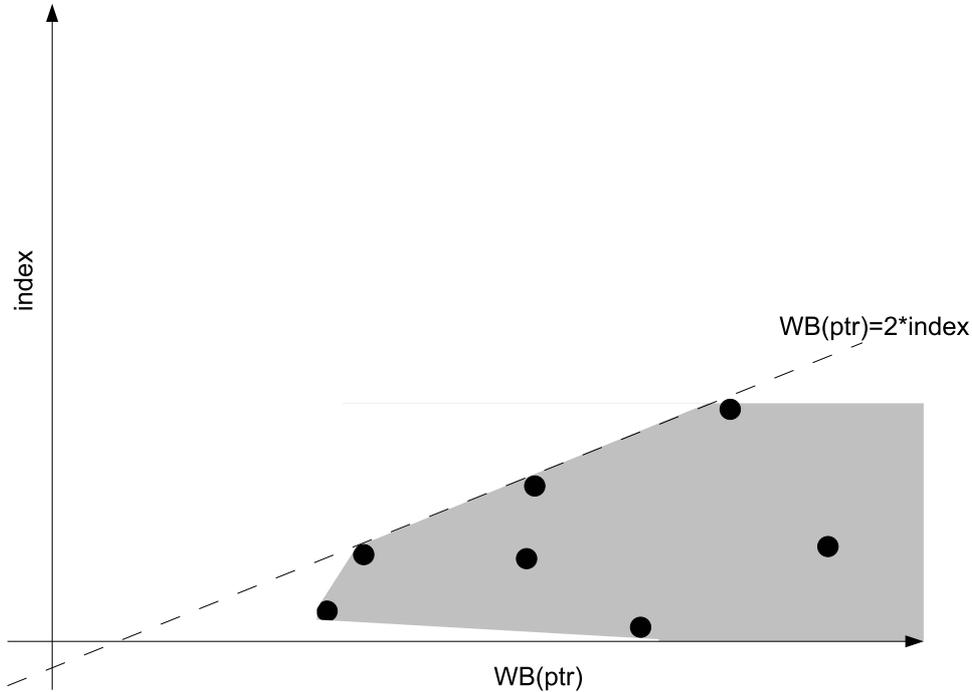


Figure 7.12: The abstraction of Poly

still suffer from exponential behavior on some methods where the 2 minute timeout was reached. When removing the timeout, one method in `mscorlib.dll` took 49 minutes to reach a fixpoint using Polyhedra.

Finally, Poly do not scale to the code that we want analyze. So we chose a different approach: (a) to design abstract domains focused on a particular property; and (b) to combine domains using well-known techniques such as the reduced product. For our analysis, we designed a new numerical abstract domain, `Strp`, and we combined it with `Intv` and `LinEq` to achieve precision without giving up on performance.

In the next sections we present the details of `Strp`, its reduction with `Intv` and `LinEq`, and the results of our practical experiments.

Assembly	Time	# Accesses		%
		Checked	Validated	
<code>mscorlib.dll</code>	125m52s*	3 070	1 610	52.46
<code>System.dll</code>	257m27s*	1 576	744	44.94

Table 7.3: Unsafe code analysis using the Polyhedra domain

7.8 The Stripes Abstract Domain

We introduce a novel, weakly-relational domain Stripes, Strp , focused on the inference and checking of (upper bounds on) memory accesses that use a base, an index, and a multiplicative factor. We define the order, the join, the meet and the widening operators.

7.8.1 Constraints

As a first approximation, Strp captures constraints of the form $\text{WB}(p) - \text{sizeof}(T) * (\text{count}[+\text{base}]) > k$ where $\text{WB}(p)$, count , and optionally base are variables, T is a type, and k is an integer constant. The intuition behind it is that the pointer p is defined at least on $\text{count}[+\text{base}]$ elements of its type, and on k additional bytes.

In practice, these constraints are used in a more generic way: the first element may be any variable (and not only the writable bytes of a pointer) and the $\text{sizeof}(T)$ may be any numerical value (and not only the size of the type of the pointer target). Then the constraints captured by the Stripe domain are $z - k_1 * (x[+y]) > k_2$, where z , x , and y are variables, and k_1 and k_2 are integer values.

7.8.2 Abstract Domain Structure

Abstract Elements

We represent Strp elements as maps from variables to constraints. We chose maps as they allow efficient manipulation of directional constraints:

$$\text{Strp} = [\text{Var}_{\text{WB}} \rightarrow \wp(\text{Var}_{\text{WB}} \times (\text{Var}_{\text{WB}} \cup \perp) \times \mathbb{Z} \times \mathbb{Z})].$$

Intuitively, the domain of the map contains the variable z , the first and second component of the 4-tuple represent the two variables x and y (\perp if it is not present), the third component is k_1 and the last one is k_2 .

Example (Representation of stripes constraints)

The two constraints $z - 4 * y > 0$ and $z - 2 * (x + u) \geq 5$ are represented in Strp by the map $[z \rightarrow \{(y, \perp, 4, 0), (x, u, 2, 4)\}]$.

Order

An abstract state \bar{s}_1 in Strp is more precise than \bar{s}_2 iff for each constraint in \bar{s}_2 , \bar{s}_1 contains a constraint such that (a) the three variables and the integer constant k_1 are the same; and (b) k_2 is less or equal than the k_2 of \bar{s}_2 since if $x > y$ and $y > z$ then $x > z$ by transitivity of $>$. Formally:

$$\bar{s}_1 \sqsubseteq \bar{s}_2 \iff \forall z \in \text{dom}(\bar{s}_2), \forall (y, x, k_1, k_2^2) \in \bar{s}_2(z) : \\ z \in \text{dom}(\bar{s}_1) \wedge \exists (y, x, k_1, k_2^1) \in \bar{s}_1(z). k_2^2 \leq k_2^1$$

Top and Bottom

The largest element of Strp is a map with no information: $\lambda z. \emptyset$. An abstract state \bar{s} is bottom iff it contains a contradiction: e. g. $[z \mapsto \{(y, \perp, 1, 0)\}, y \mapsto \{(z, \perp, 1, 0)\}]$.

Join

The upper bound operator (a) keeps the constraints that are defined in both operands; (b) takes the smallest lower bound k_2 if it is different in the two constraints since if $\text{exp} > a$, $\text{exp} > b$ and $a \geq b$ then $\text{exp} > b$ is an upper bound for both constraints. Formally:

$$\bar{s}_1 \sqcup \bar{s}_2 = \lambda z. \{(y, x, k_1, k_2) \mid (y, x, k_1, k_2^1) \in \bar{s}_1(z), (y, x, k_1, k_2^2) \in \bar{s}_2(z), k_2 = \min(k_2^1, k_2^2)\}$$

Meet

The lower bound operator traces the constraints of both operands. If both contain a constraint with the same variables x , y , and z , and the same integer value k_1 , the operator keeps the largest integer value for the numerical lower bound.

$$\bar{s}_1 \sqcap \bar{s}_2 = \lambda z. \left\{ \begin{array}{l} (y, x, k_1, k_2) \mid (y, x, k_1, k_2^1) \in \bar{s}_1(z), \\ (y, x, k_1, k_2^2) \in \bar{s}_2(z), \\ k_2 = \max(k_2^1, k_2^2) \end{array} \right\} \cup \left\{ \begin{array}{l} (y, x, k_1, k_2) \mid ((y, x, k_1, k_2) \in \bar{s}_1(z) \wedge \\ (y, x, k_1, -) \notin \bar{s}_2(z)) \\ \vee ((y, x, k_1, k_2) \in \bar{s}_2(z) \wedge \\ (y, x, k_1, -) \notin \bar{s}_1(z)) \end{array} \right\}$$

Widening

Strp does not satisfy the ACC condition. As a consequence, we need to define a widening operator to ensure convergence. Our widening simply drops the constraints that are not stable between two iterations:

$$\bar{s}_1 \nabla \bar{s}_2 = \lambda z. \bar{s}_1(z) \cap \bar{s}_2(z).$$

Concretization

The concretization function $\gamma_{\text{Strp}} : [\text{Strp} \rightarrow \wp(\bar{\mathbb{C}})]$ returns all the possible states that satisfy the constraints represented by the abstract state:

$$\gamma_{\text{Strp}}(\bar{s}) = \{\bar{f} \mid \forall z \in \text{dom}(\bar{s}) \forall (y, x, k_1, k_2) \in \bar{s}(z). \bar{f}(z) - k_1 * (\bar{f}(y) + \bar{f}(x)) > k_2\}.$$

It is immediate to see that γ_{Strp} is monotonic, and furthermore that it is a complete \cap -morphism. Therefore, as the composition of monotonic functions is monotonic, the following theorem stating that Strp is a sound approximation holds:

THEOREM 7.8.1 *Abstraction γ_{Strp} as defined above is a complete \cap -morphism. Therefore, it exists an α_{Strp} such that $\langle \wp(\overline{\mathbf{C}}), \sqsubseteq \rangle \xleftrightarrow[\alpha_{\text{Strp}}]{\gamma_{\text{Strp}}} \langle \text{Strp}, \sqsubseteq \rangle$ as proved by Theorem 2.2.3. As a consequence, $\langle \wp(\mathbf{C}), \sqsubseteq \rangle \xleftrightarrow[\alpha_{\text{Strp} \circ \alpha}]{\gamma \circ \gamma_{\text{Strp}}} \langle \text{Strp}, \sqsubseteq \rangle$ as proved by Theorem 2.2.4.*

7.8.3 Refinement of the Abstract State

A state of the Stripe domain may be internally refined, by carefully propagating information between constraints.

Example (Refinement of constraints)

Consider the two stripes constraints $x - 2 * (y + u) > 4$ and $y - z > 0$. From the first constraint we derive:

$$x - 2 * (y + u) > 4 \iff x - 2 * u - 4 > 2 * y \iff x/2 - u - 2 > y.$$

From the second constraint we derive that $y > z \iff y \geq z + 1$. Combining the two, we derive a new stripe constraint: $x/2 - u - 2 > z + 1 \iff x - 2 * (u + z) > 6$.

The above example can be easily generalized:

LEMMA 7.8.2 *Saturation* If an abstract state contains the two constraints

$$\begin{aligned} x - k_1 * (y[+u]) &> k_2 \\ y - 1 * z &> k_3 \end{aligned}$$

then we can infer the constraint $x - k_1 * (z[+u]) > k_2 + k_1 * (k_3 + 1)$.

The refinement enabled by this Lemma above is important in practice. It allows adding new constraints to the abstract state, without requiring an expensive closure to propagate the information. Of course, Lemma 7.8.2 does not guarantee the completeness of the saturation, but it is sufficient for our purposes, as illustrated by the next example. In addition, when we need to apply widening in order to make convergent the analysis, we do not apply refinement in order to assure that the analyses ends.

Example (Saturation)

Let us consider the example in Fig. 7.3. Inside the loop, we have the abstract state $\bar{s} = \{\text{WB}(a) - 4 * \text{len} > -1, \text{len} - i > 0\}$ ¹. We have to check whether $\text{WB}(a) \geq 4 * i + 4$. We cannot do it directly by inspecting \bar{s} as there is no direct relation between $\text{WB}(a)$ and i . Applying the refinement of Lemma 7.8.2, we infer the constraint $\text{WB}(a) - 4 * i > 3$ which suffices to validate the access: $\text{WB}(a) - 4 * i > 3 \iff \text{WB}(a) > 4 * i + 3 \iff \text{WB}(a) \geq 4 * i + 4$.

In our implementation we perform this refinement only on-demand when we need to check the proof obligations.

¹To simplify the reading, we present a stripe abstract state as a set of constraints.

7.8.4 Transfer Functions

Assignment

When an expression is assigned to a variable, we first drop all the constants that are defined on the assigned variable, and then we add some inferred constraints. Formally:

$$\text{assign}(x, \text{exp}, \bar{s}) = \text{let } \bar{s}' = \text{drop}(x, \bar{s}) \text{ in } \bar{s}' \cup C(x, \text{exp}, \bar{s}')$$

where

$$\text{drop}(x, \bar{s}) = \lambda y. \{(z, u, k_1, k_2) \mid y \neq x, (z, u, k_1, k_2) \in \bar{s}(y) \implies z \neq x \wedge u \neq x\};$$

and C infers new constraints from an assignment and an abstract state. Few representative cases for C follow. In our implementation we consider a richer structure of expressions and cases.

$$\begin{aligned} C(x, y, \bar{s}) &= [x \rightarrow \bar{s}(y)] \cup [v_1 \mapsto \{(x, v_2, k_1, k_2) \mid (y, v_2, k_1, k_2) \in \bar{s}(v_1)\}] \\ C(x, u + v, \bar{s}) &= [v_1 \mapsto \{(u, w, k_1, k_2) \mid (x, \perp, k_1, k_2) \in \bar{s}(v_1)\}] \\ &\dots \end{aligned}$$

Abstract Checking

To check a boolean expression, we first try to normalize it into a form like $x - k_1 * (y + z) > k_2$, and then we check if the abstract state contains a constraint which implies it. Formally:

$$\begin{aligned} \text{check}(\text{exp}, \bar{s}) &= \text{let } (x - k_1 * (y + z) > k_2^1, b) = \text{normalize}(\text{exp}) \text{ in} \\ &\text{if } (b \wedge \exists (y, z, k_1, k_2^2) \in \bar{s}(x). k_2^1 \leq k_2^2) \text{ then true else } \top \end{aligned}$$

We skip the details of `normalize`. Roughly, it applies basic arithmetic identities to rewrite the expression. If it fails to put the expression into a stripe constraint form, it returns a boolean value signaling the failure.

7.8.5 Representation of Strp

Figure 7.13 depicts the area captured by `Strp` when applying it to the set of concrete points depicted by Figure 7.9. The relational information between variables `index` and `WB(ptr)` is sufficient in order to prove that, when accessing the memory, the upper bound is respected. Indeed, `Strp` are not enough in order to prove the lower bound (i.e. that the index used to access memory is positive), and so we need to refine it.

7.9 Refined Abstract Semantics

We refine the information captured by the `Strp` domain with `Intv` and the `LinEq` domain. `Intv` is needed to check lower bounds of accesses. `LinEq` is needed to track linear equalities, and in particular to handle the compilation schema for `fixed` in `C#`.

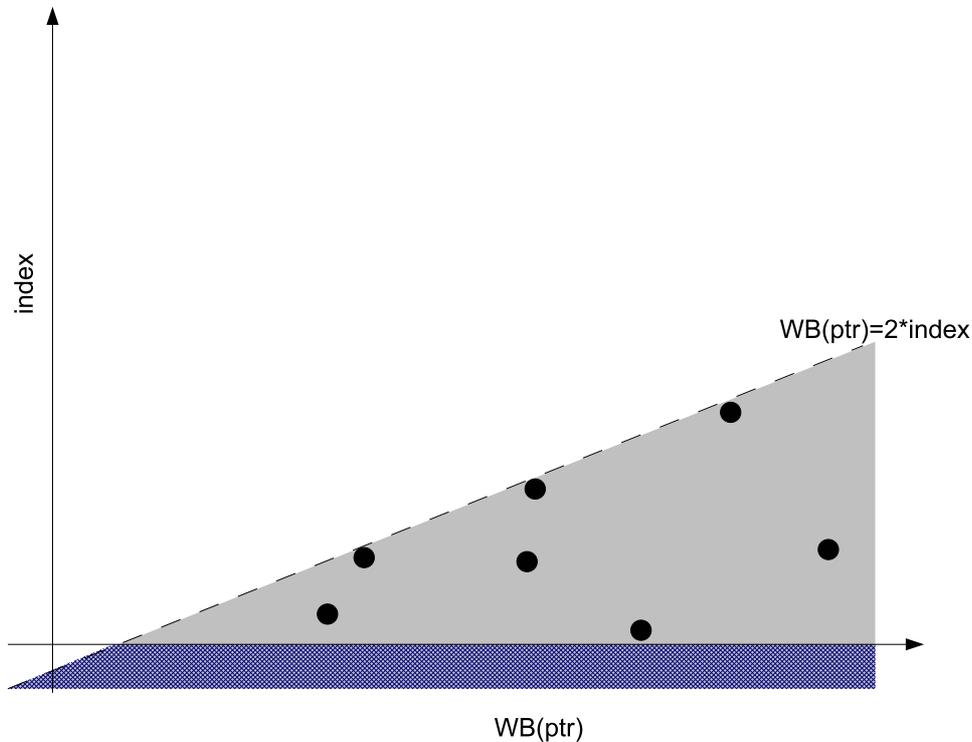


Figure 7.13: The abstraction of Strp

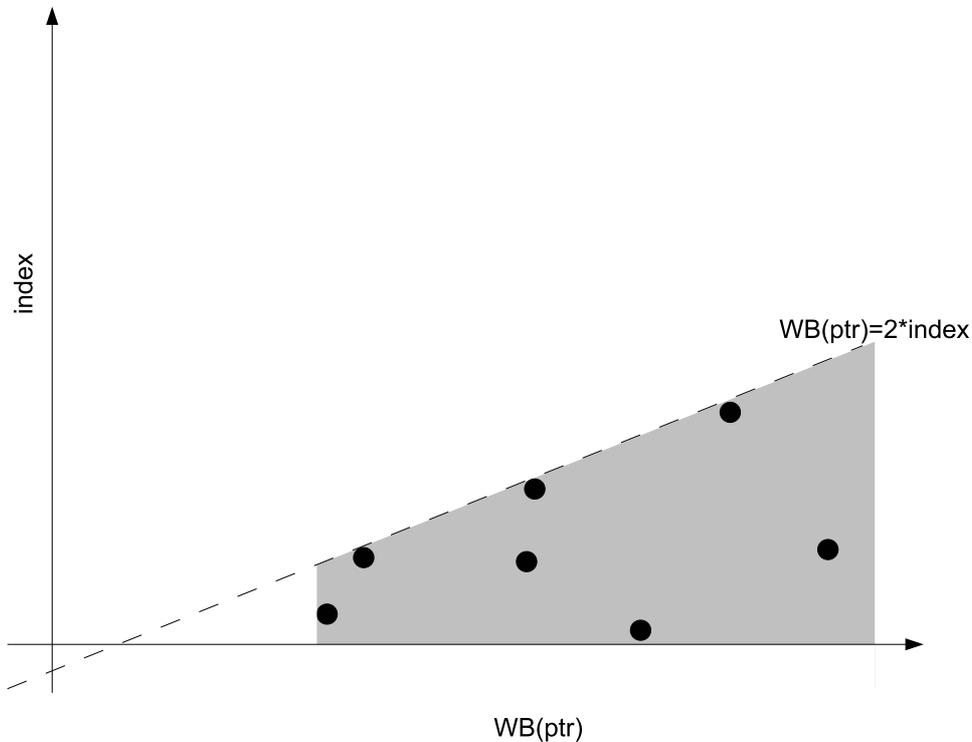
7.9.1 Checking Lower Bounds of Accesses

Strp allows representing just partial numerical bounds on variables. In fact, when $k_1 = 0$, a stripe constraint boils down to a numerical lower bound: $z > k_2$. Nevertheless, in general we need to track numerical upper bounds on variables: Those may appear in expressions that must be evaluated to check under-flow accesses. We use Intv to track the numerical bounds on variables. Figure 7.14 depicts the area captured by the Cartesian product of Strp and Intv. It makes evidence graphically that this domain is sufficient enough in order to precisely analyze both lower and upper bound of memory accesses, as the area that approximates the concrete values is inside the one that proves the property is respected.

Example (Need for numerical bounds)

Let us consider the following code snippet (“...” denotes an arbitrary boolean expression):

```
int *p;
...
// suppose that WB(p) = 12, a = 5
if (...) {
```

Figure 7.14: The abstraction of $\text{Strp} \times \text{Intv}$

```

    b = 3;
}
else {
    b = 4;
}
*(p + (a-b)) = 0; // (*)

```

If we track just lower bounds, at (*) we have $a > 4, b > 2$, so that we cannot prove the memory access correct. If we track both numerical bounds, at (*) we have that $a = 5, b \in [3, 4]$, so that $b - a \in [1, 2]$ which suffices to prove the access correct.

The numerical abstract domain for the analysis is the product domain $\text{Intv} \otimes \text{Strp}$. All the domain operations are lifted pair-wise to the product domain. Sometimes we may want to use the information contained in Intv to refine the information in Strp . For instance, to improve the precision of the join operator, as shown by the next example.

Example (Refinement of Strp with Intv)

Consider the following piece of code:

```

if (arr == null)
    p = null;
else if (arr.Length == 0)
    p = null;
else
    p = &arr[0];

```

Figure 7.15: The code generated by the C# compiler for the statement `fixed(T * p = arr)...`

```

int [] array;
...
// suppose that array.Length - count > 0
if (count == 0)
    array = new int[1];
else
    /* do nothing */ ;

```

Using just `Strp`, at the join point we cannot conclude that `array.Length - count > 0`: inside the conditional, `array` is assigned a new value, so that the entry constraint is dropped.

Using `Intv` \otimes `Strp`, the abstract state after array creation is $\bar{p}_1 = \langle \langle \text{count} \in [0, 0], \text{array.Length} \in [1, 1] \rangle, \lambda z. \emptyset \rangle$; the abstract state at the end of the false branch is $\bar{p}_2 = \langle \emptyset, [\text{array.Length} \rightarrow (\text{count}, 1, 0)] \rangle$. The join is $\langle \emptyset, [\text{array.Length} \rightarrow (\text{count}, 1, 0)] \rangle$, as the interval component of \bar{p}_2 implies that `array.Length - count > 0`.

7.9.2 Compilation of fixed

When the C# compiler compiles a `fixed` statement which assigns an array `arr` of type `T[]` to a pointer `p`, it generates code to check whether the `arr` is null or if its length is 0. If it is the case, then it assigns null to `p`. Otherwise it assigns the address of the first element of `arr` to `p`. Fig. 7.15 depicts this compilation schema.

Without any refinement, the analysis performed by `Clousot` cannot capture that $\text{WB}(p) = \text{sizeof}(T) * \text{array.length}$. There are two main reasons for that: (1) it is not possible to represent a constraint in the form of $x - a * y = 0$ in `Intv` \otimes `Strp`; (2) At the join point, a state where `p` is null is merged with one where $\text{WB}(p) = \text{sizeof}(T) * \text{array.length}$.

For (1), we refine the abstract domain to use `LinEq`, to retain linear equalities: the abstract domain used in the analysis becomes `LinEq` \otimes `Intv` \otimes `Strp`.

For (2), if `arr = null` or `arr.Length = 0`, then $0 = \text{sizeof}(T) * \text{array.Length} = \text{WB}(p)$ trivially holds. As we are performing an over-approximation of the reachable states, we can safely add $\text{WB}(p) = \text{sizeof}(T) * \text{array.length}$ to our abstract state.

Assembly	# Methods	Time	# Accesses		%
			Checked	Validated	
mscorlib.dll	18 084	3m43s	3 069	1 835	59.79
System.dll	13 776	3m18s	1 720	1 048	60.93
System.Data.dll	11 333	3m45s	138	59	42.75
System.Design.dll	11 419	2m42s	16	10	62.50
System.Drawing.dll	3 120	19s	48	29	60.42
System.Web.dll	22 076	3m19s	88	44	50.00
System.Windows.Forms.dll	23 180	4m31s	364	266	73.08
System.XML.dll	10 046	2m41s	772	311	40.28
Average					57.96

Table 7.4: Experimental results

7.10 Experiments

We implemented the analysis for unsafe memory accesses using the Stripes domain in Clousot. We tested extensively our analysis on all the libraries of the .NET framework. Our experiments were conducted on a 2.4Ghz Intel Core Duo laptop, with 4Gbytes of RAM, running Windows Vista (Windows processor score 5.3). The target assemblies are taken from the `%WINDIR%\ Microsoft\ Framework\ v2.0.50727` directory of the test laptop. No pre-processing, manipulation or filtering of the assemblies has been conducted.

A primary goal for Clousot is its use at development time during compilation or even within the integrated development environment. Thus, the performance of the analysis is crucial. Our specialized domains provide us with excellent performance as reported in Tab. 7.4.

The analysis is fast: the average analysis time per method is 12ms. We validate on average 57.96% of the unsafe memory accesses. This may not seem high at first glance. However, consider the burden of human code reviews for unsafe code which is currently a necessary practice. Our analysis cuts down the work load in half, focusing the reviews on accesses that seem non-obvious to prove correct. Nevertheless, we feel that we can improve the precision of the unsafe analysis in two ways:

1. We intend to remove short-comings in the current implementation of the domains, resulting in unnecessary precision loss or inability to prove facts that are implied. We intend to improve the domains as described e.g. in Section 7.8.3.
2. The code we analyzed does not contain contracts. This leads to loss of precision when the proof obligation required in one method is established by the caller of the method, or sometimes several call frames higher on the stack. As a consequence, without contracts on the intermediate methods Clousot reports warnings on those memory accesses.

7.10.1 System.Drawing Case Study

We analyzed the 19 warnings in System.Drawing.dll to determine what contracts need to be written to avoid them, or whether they represent true vulnerabilities.

First, we found the use of two helper methods that required pre-conditions:

```

short GetShort(byte* ptr) {Contract.Requires(Contract.WritableBytes(ptr) >= sizeof(short));
    ...
}
int GetInt(byte* ptr) {Contract.Requires(Contract.WritableBytes(ptr) >= sizeof(int ));
    ...
}

```

These helper methods simply load 16 bits or 32 bits from the given pointer location using little-endian encoding and avoiding unaligned accesses.

With the pre-conditions written as above, Clousot no longer reports warnings within these helper methods. Instead, it reports warnings at 26 call-sites to these methods. The remaining warnings are all located within 5 distinct methods.

1. One method uses an unmanaged heap allocation routine to obtain memory from the marshal heap. Writing an appropriate post-condition for this allocator eliminates the warnings in that method.

```

public static IntPtr AllocHGlobal(int cb) {
    Contract.Ensures(Contract.WritableBytes(Contract.Result<IntPtr>()) == cb);
    ...
}

```

2. The next method we examined actually contained an error leading to buffer overruns on read accesses.
3. The third method uses a complicated invariant on a data structure that involves indexing using a product expression of two variables. Our domains cannot currently track such products (only variables multiplied with constants). However, the code appears to be safe.
4. The fourth method extracts a `byte[]` from an auxiliary data structure and indexes it assuming the array contains 1K elements. Examining the data structure and all its construction sites, we determined that it is built via marshalling from an unmanaged Windows API call and the marshal annotation specifies that the buffer is to be allocated with the fixed size of 1K. Although we can specify this size as an object invariant on the auxiliary structure leading to the removal of the warning by Clousot, our tool chain does not yet understand the marshalling constraints establishing the invariant.
5. Finally, the last function containing most of the accesses and calls to the helper functions `GetShort` and `GetInt`, whose pre-conditions must be validated, exposed

a shortcoming in our implementation. Upon examination, we determined that the analyzer infers a sufficiently strong loop invariant which implies the safety of the memory accesses and pre-conditions. However, our implementation was not able to show this implication automatically.

With the above contracts and fixes, Clousot would validate 3 additional methods, but report false warnings in one method due to an index expression we cannot handle, and another false warning in a new method due to the lack of support for marshal annotations.

7.10.2 Summary

Overall, the analysis is fast enough to use in integrated development environments. It achieves a higher level of automation and scalability than existing tools. In fact, we found that the tool rarely fails to infer the necessary loop invariants to validate the memory accesses. More often, it is the lack of contracts that limits our modular intra-procedural analysis. The use of contracts not only allows reducing the false positive rate, the contracts furthermore serve as checked documentation on important safety invariants. Clousot can catch code changes or additions that fail to live up to the existing specifications and thereby provide excellent static regression checking.

7.11 Related Work

We developed a sound and scalable analysis to statically check memory safety in unsafe code. Scalability, without giving up precision, was a main goal for the analysis. Similar work for C does not fulfill these two requirements. For instance the analysis introduced by Wagner et al. [139] is not precise enough to check memory accesses that involve a pointer, a base and an offset, which we found to be pervasive in `mscorlib.dll`, the main library of the .NET framework. On the other hand, the analysis of Dor et al. [33, 34] is precise enough to capture these relations, but it is based on the use of the Polyhedra (Poly) abstract domain [31] which is known to have severe scalability problems². The work of Simon and King [128, 129] improved on that by using an abstraction of Poly, where linear inequalities were restricted to buckets of two variables. However, we did not find it precise enough to match the programming style adopted in the code we analyzed. In particular, we found that a common pattern in unsafe code of .NET libraries uses both a base and a number of element to be accessed starting from a pointer. This requires to deal with constraints with three variables, and this is not supported by the domain developed by Simon and King. Our approach differs from earlier work in that it is based on the combination of lightweight and focused abstract domains, instead of a monolithic, precise domain. Each abstract domain is specialized (and optimized) toward a particular

²The worst case complexity of Poly is exponential. To the best of our knowledge, at the moment of writing, the most optimized implementations do not scale to more than 40 variables [7, 8]. In the analysis of .NET assemblies, we need to capture up to 965 variables.

program property, and their combination provides a powerful analysis without sacrificing performance.

Bounds Analysis for C

Rinard and Rugina published a powerful analysis of C programs to determine aliasing, bounds, and sharing of memory, enabling bounds optimizations, and parallelization [123, 124]. Their analysis infers a set of polynomial bounds on variables that are solved using a linear programming problem to minimize the spread of the bound. The reported analysis times are fast (in the same range as ours), but they only report results for small examples. Their technique based on solving a linear programming problem is quite different from using symbolic abstract domains, but equally promising. A benefit of their approach is that it performs inter-procedural analysis by inferring relations for function inputs and outputs using a bottom up call graph approach. However, this is also a major drawback, as for strongly connected components of functions (recursively calling each other), their analysis needs to compute a fixpoint. It is well known that call-graphs built for very large applications (in particular object-oriented programs) are imprecise, leading to very large components [32], making such an approach unlikely to scale.

Das et. al. describe buffer overflow checking and annotation inference on large Microsoft C/C++ code bases [59]. Few details of the used numerical domains are public, but from the paper it is apparent that for precision, their analysis performs path splitting, meaning it analyzes paths separately through a function whenever the abstract state at join points disagrees. The Stripes domain described in this paper and the associated transfer functions and join operations are geared towards providing precision without path splitting (our analyzer does not perform path splitting).

Analysis of JNI

A few analyses for Java handle programs using the Java Native Interface (JNI) [88]. Furr and Foster in [50] present a restricted form of dependent types used to infer and type-check the types passed to foreign functions via the JNI. Tan et al proposed a mixed dynamic/static approach to guarantee type safety in Java programs that interface with C. We are not interested in type safety: in unsafe C#, type errors are less common than with the JNI, since the unsafe context is integrated in C#, so that (a) the compiler can still perform most type checking and (b) types do not need to be serialized as strings (the most common type error in using the JNI). Instead our analysis focuses directly on memory usage via pointers, whereas previous work did not.

Interoperability of Languages

Recent work focuses on language interoperability. Tan and Morrisett, [135], advocate an approach in which the Java bytecode language is extended with a few instructions useful to model C code. Hirzel and Grimm, [65], take an alternative approach with Jeannie,

which is a language which subsumes Java and C, and the burden of creating the “right” JNI for interfacing the two languages is left to the compiler. Matthews and Findler, [97], give an operational semantics for multi-language programs which uses contracts as glue for the inter-operating languages. The MSIL instruction set is rich enough to allow an agile compilation of several languages: our analysis, working at the MSIL level does not need to take into account inter-operability issues.

Static Analyzers

ESC/Java 2 [22] and Spec# [10] use automatic theorem provers to check programs. Automatic theorem provers provide a strong engine for symbolic reasoning (e. g. quantifiers handling). The drawbacks are that: (a) they require the programmer to provide loop invariants and (b) they present scalability problems. Analysis times close to the one we obtain in Clousot on shipped code are well beyond the state-of-the art in automatic theorem proving.

7.12 Discussion

We presented a new static analysis for checking memory accesses in unsafe code in .NET. The core of the analysis is a new abstract domain, *Strp*, which combined with *Intv* and *LinEq*, allows the analysis to scale to hundreds of thousands of lines of code. We proved the soundness of the approach by designing the static analysis using stepwise abstraction of a concrete transition semantics.

In this way we applied Clousot, an industrial generic static analyzer, to the study of a property of practical interest. The analysis is both scalable (we are in position to analyze .NET libraries in a couple of minutes) and precise (we found bugs on shipped code analyzing only a small case study). This chapter makes evidence that generic analyzers seem to be a promising way to build up tools to debug real applications: the idea of re-using the most part of the analysis and of focusing only on the property of interest and on the numerical domain puts us in position to precisely and efficiently analyze industrial code.

8

Conclusions

In this thesis we presented a generic approach to the analysis of multithreaded programs, we applied it to Java programs, we implemented it, and we extended an industrial generic analyzer proving the practical and industrial interest of this type of analyzers.

The first contribution is the development of a generic theoretical framework in order to define a static analysis sound with respect to the happens-before memory model. Memory models define which behaviors are allowed during the execution of a multithreaded program. In particular they define at a given point of the execution which values can be seen through shared memory. We defined the concrete semantics of the happens-before memory model in a fixpoint form, and then we abstracted it with a computable semantics proving formally the soundness of our approach.

A second contribution is the definition of a new deterministic property. The most part of static analyses of multithreaded programs is focused on particular properties like data races and deadlocks. Proving the absence of data races and deadlocks is not sufficient for developers to prove the correctness of a multithreaded program. In fact, even if a program is data races and deadlocks free, it may still expose nondeterministic behaviors because of arbitrary interleaving of threads. Starting from these considerations we developed a deterministic property aimed at checking directly the nondeterministic behaviors because of unordered communications of threads through shared memory. We defined it on a concrete semantics, we abstracted it in two steps proving formally the correctness, we proposed the new idea of weak determinism, we proposed other two ways of projecting this property (on states and on traces) defining a global hierarchy, we related the data race condition to the deterministic property, and finally we sketched how this property may be used in order to semi-automatically parallelize sequential programs. We believe that the deterministic property, dealing with the effects of unordered communications through the shared memory instead of its causes (as data races do), provides a more expressive and flexible instrument in order to debug multithreaded programs.

In order to apply this generic framework to Java, we defined a domain particularly focused on the main features of multithreading and an operational semantics of bytecode statements. Our approach supports all the main features of Java programs, e.g. arrays, strings, overloading and overriding of methods, and dynamic creation of threads, shared locations, and monitors. In this context, we proposed a specific alias analysis that precisely approximates threads' identifiers, monitors, and accesses to the shared memory.

All these features were implemented in *Checkmate*, a new generic static analyzer of Java multithreaded programs. We developed some well-known non-relational numerical domains, some properties of interest (included determinism and weak determinism), and some memory models (included the happens-before one). The experimental results were deeply studied both in term of precision and efficiency obtaining encouraging conclusions.

Finally, we extended an industrial generic analyzer (*Clousot*) to the study of a property of interest, i.e. the detection of buffer overruns. In order to obtain a precise and scalable analysis, we developed a new relational domain, *Strp*, and we combined it with *Intervals*, and *Linear Equalities*. The analysis was proved to be scalable and precise: we are able to analyze about twenty thousands of methods in a couple of minutes, and we found bugs on shipped code analyzing a really small case study only. In this way, we showed the industrial interest of generic analyzers in order to develop powerful and useful tools to debug programs.

Future work: The main challenge is the application of the overall approach developed by this thesis to the analysis of industrial software. In this context, we will need to deeply study in which way our analysis has to be refined in order to keep precise results also when dealing with large size programs and commercial software.

First of all, we would check if the happens-before memory model is precise enough, or if we need to take into account other features of the Java one. In addition, our approach considers as synchronization primitives only launchings of threads and mutual exclusion on threads. It is clear that we will need to consider more synchronization actions, but we believe that our framework can be easily extended to support the most part of them. Then we want to investigate the relaxations of the deterministic property that may be interesting in order to find bugs on real software and in order to semi-automatically parallelize sequential programs. It is clear that both these two issues will require to refine our domain and semantics of Java bytecode. In particular, we will need to eliminate the stack representing the code in the 3-address form, and to reconstruct expressions.

It is important to notice that modularity remains an issue that cannot be dealt with at the moment for the analysis of multithreaded programs, because of actual limits in the semantics of modern programming languages supporting multithreading as pointed out in Section 3.6.3. In the last decade modularity has been the most appealing feature of object-oriented languages. We think that in order to correctly develop large multithreaded applications, it is necessary to provide additional primitives that allow developers to reason in a modular way. In order to achieve this goal, it will be necessary to deeply investigate what developers may be interested to specify, and the best way they can express it.



Source Code of Examples Taken from [85]

A.1 ExpandableArray

```
public class ExpandableArray {
    private Object[] data_;
    private int size_;

    public ExpandableArray(int cap) {
        data_ = new Object[cap];
        size_ = 0;
    }

    public synchronized int size() {
        return size_;
    }

    public synchronized Object at(int i) throws Exception {
        if (i < 0 || i >= size_)
            throw new Exception();
        else return data_[i];
    }

    public synchronized void append(Object x) {
        if (size_ >= data_.length) {
            Object[] olddata = data_;
            data_ = new Object[3 * (size_ + 1) / 2];
            for (int i = 0; i < size_; ++i)
                data_[i] = olddata[i];
        }
        data_[size_++] = x;
    }
}
```

```
public synchronized void removeLast() throws Exception {  
    if (size_==0)  
        throw new Exception();  
    else  
        data_[--size_]=null;  
}
```

```
private static class RemoveLastThread extends Thread {  
    ExpandableArray array;
```

```
    public RemoveLastThread(ExpandableArray array) {  
        this.array=array;  
    }
```

```
    public void run() {  
        try{array.removeLast();}  
        catch(Exception e) {}  
    }  
}
```

```
private static class AppendThread extends Thread{  
    ExpandableArray array;
```

```
    public AppendThread(ExpandableArray array) {  
        this.array=array;  
    }
```

```
    public void run() {  
        try{array.append(new Object());}  
        catch(Exception e) {}  
    }  
}
```

```
static class ReadWriteConflict {  
    public static void main(String[] args) {  
        ExpandableArray array=new ExpandableArray(10);  
        array.append(new Object());  
        array.append(new Object());  
        array.append(new Object());  
        new RemoveLastThread(array).start();  
        try{array.at (1);}  
        catch(Exception e) {}  
    }  
}
```

```
static class WriteWriteConflict {  
  public static void main(String[] args) {  
    ExpandableArray array=new ExpandableArray(10);  
    array.append(new Object());  
    array.append(new Object());  
    array.append(new Object());  
    new AppendThread(array).start();  
    try{array.append(new Object());}  
    catch(Exception e) {}  
  }  
}
```

A.2 LinkedCell

```
public class LinkedCell {  
  protected double value_;  
  protected LinkedCell next_;  
  
  public LinkedCell(double v, LinkedCell t) {  
    value_=v;  
    next_=t;  
  }  
  
  public synchronized double value() {  
    return value_;  
  }  
  
  public synchronized void setValue(double v) {  
    value_=v;  
  }  
  
  public LinkedCell next() {  
    return next_;  
  }  
  
  public double sum() {  
    double v=value();  
    if (next() !=null)  
      v+=next().sum();  
    return v;  
  }  
}
```

```

public boolean includes(double x) {
    synchronized(this) {
        if (value_==x)
            return true;
    }
    if (next()==null)
        return false;
    else return next().includes(x);
}

double ineffectivelyUnsynchedSum() {
    double v=value_;
    return v+nextSum();
}

double nextSum() {
    return (next()==null) ? 0 : next().sum();
}

private static class SynchronizedSumThread
                                extends Thread{
    LinkCell list ;

    public SynchronizedSumThread(LinkCell list) {
        this. list = list ;
    }

    public void run() {
        list .sum();
    }
}

static class SynchronizedSum {
    public static void main(String[] args) {
        LinkCell list =new LinkCell(1.0, null);
        LinkCell list1 =new LinkCell(2.0, list );
        LinkCell list2 =new LinkCell(3.0, list1 );
        LinkCell list3 =new LinkCell(4.0, list2 );
        new SynchronizedSumThread(list3).start();
        list1 .setValue(5.0);
    }
}

private static class NotSynchronizedSumThread

```

```
                                extends Thread{
LinkedCell list ;

public NotSynchronizedSumThread(LinkedCell list) {
    this. list = list ;
}

public void run() {
    list .ineffectivelyUnsynchedSum();
}
}

static class NotSynchronizedSum {
public static void main(String[] args) {
    LinkedCell list =new LinkedCell(1.0, null);
    LinkedCell list1 =new LinkedCell(2.0, list );
    LinkedCell list2 =new LinkedCell(3.0, list1 );
    LinkedCell list3 =new LinkedCell(4.0, list2 );
    new NotSynchronizedSumThread(list3).start();
    list3 .setValue(5.0);
}
}
}
```

A.3 Document

```
public class Document {
    Document otherPart_;

synchronized void print() {
    // print something
}

synchronized void printAll() {
    otherPart_. print ();
    this. print ();
}

private static class PrintThread extends Thread{
    Document doc;

public PrintThread(Document doc) {
    this.doc=doc;
}
```

```
    }

    public void run() {
        doc.printAll ();
    }
}

public static void main(String[] args) {
    Document letter=new Document();
    Document enclosure=new Document();
    letter .otherPart_=enclosure;
    enclosure.otherPart_=letter ;
    new PrintThread(letter). start ();
    enclosure.printAll ();
}
}
```

A.4 Dot

```
public class Dot {

    class Point {
        private int x_, y_;

        public Point(int x, int y) {
            x_=x;
            y_=y;
        }

        public int x() {
            return x_;
        }

        public int y() {
            return y_;
        }

    }

    protected Point loc_;

    public Dot(int x, int y) {
```

```
loc_=new Point(x,y);
}

public Point location () {
    return loc_;
}

protected synchronized void updateLoc(Point newLoc) {
    loc_=newLoc;
}

public synchronized void moveTo(int x, int y) {
    updateLoc(new Point(x, y));
}

public synchronized void shiftX(int deltaX) {
    Point currentLoc=location();
    updateLoc(new Point(currentLoc.x()+deltaX, currentLoc.y()));
}

private static class ShiftXThread extends Thread{
    Dot dot;

    public ShiftXThread(Dot dot) {
        this.dot=dot;
    }

    public void run() {
        dot.shiftX (1);
    }
}

public static void main(String[] args) {
    Dot dot=new Dot(0, 0);
    new ShiftXThread(dot).start();
    dot.moveTo(1, 1);
}
}
```

A.5 Cell

```
public class Cell {
    private int value_;
```

```
public synchronized int getValue() {  
    return value_;  
}  
  
public synchronized void setValue(int v) {  
    value_=v;  
}  
  
public synchronized void swapContents(Cell other) {  
    int newValue=other.getValue();  
    other.setValue(this.getValue());  
    this.setValue(newValue);  
}  
  
private static class SwapThread extends Thread{  
    Cell x, y;  
  
    public SwapThread(Cell x, Cell y) {  
        this.x=x;  
        this.y=y;  
    }  
  
    public void run() {  
        x.swapContents(y);  
    }  
}  
  
public static void main(String[] args) {  
    Cell x=new Cell(), y=new Cell();  
    new SwapThread(x, y).start();  
    y.swapContents(x);  
}  
}
```

A.6 TwoLockQueue

```
public class TwoLockQueue {  
    private TLQNode head_;  
    private TLQNode last_;  
    private Object lastLock_;
```

```
public TwoLockQueue() {
    head_=last_=new TLQNode(null, null);
    lastLock_=new Object();
}

public void put(Object x) {
    TLQNode node=new TLQNode(x, null);
    synchronized(lastLock_) {
        last_.next=node;
        last_=node;
    }
}

public synchronized Object take() {
    Object x=null;
    TLQNode first=head_.next;
    if ( first !=null) {
        x= first .value;
        head_=first ;
    }
    return x;
}

private final class TLQNode {
    Object value;
    TLQNode next;

    TLQNode(Object x, TLQNode n) {
        value=x;
        next=n;
    }
}

private static class TakeThread extends Thread{
    TwoLockQueue queue;

    public TakeThread(TwoLockQueue queue) {
        this.queue=queue;
    }

    public void run() {
        ((IntWrapper) queue.take()).val++;
    }
}
```

```
static class IntWrapper {  
    int val;  
    IntWrapper(int val) {this.val=val;}  
}  
  
public static void main(String[] args) {  
    TwoLockQueue queue=new TwoLockQueue();  
    new TakeThread(queue).start();  
    queue.put(new IntWrapper(1));  
    queue.put(new IntWrapper(2));  
    queue.put(new IntWrapper(3));  
}  
}
```

A.7 Account package

Account

```
package Account;  
  
public interface Account {  
    public long balance();  
}
```

AccountHolder

```
package Account;  
  
public class AccountHolder {  
    private UpdatableAccount acct_  
        = new UpdatableAccountObject(0);  
    private AccountRecorder recorder_;  
  
    public AccountHolder(AccountRecorder r) {  
        recorder_=r;  
    }  
  
    public void acceptMoney(long amount) {  
        try{  
            acct_.credit(amount);  
            recorder_.recordBalance(new ImmutableAccount(acct_));  
        }  
    }
```

```
    catch(InsufficientFunds ex) {}
}

public void acceptMoneyWithoutImmutable(long amount) {
    try{
        acct_.credit(amount);
        recorder_.recordBalance(acct_);
    }
    catch(InsufficientFunds ex) {}
}
}
```

AccountRecorder

```
package Account;

public class AccountRecorder {
    public void recordBalance(Account a) {
        a.balance();
    }
}
```

EvilAccountRecorder

```
package Account;

public class EvilAccountRecorder extends AccountRecorder {
    private long embezzlement_;

    public void recordBalance(Account a) {
        if (a instanceof UpdatableAccount) {
            UpdatableAccount u=(UpdatableAccount) a;
            try {
                u.debit(10);
                embezzlement_+=10;
            }
            catch(InsufficientFunds e) {}
        }
        super.recordBalance(a);
    }
}
```

Immutable

```
package Account;

public interface Immutable {}
```

ImmutableAccount

```
package Account;

public class ImmutableAccount
    implements Account, Immutable {
    private Account delegate_;

    public ImmutableAccount(long initialBalance) {
        delegate_ = new UpdatableAccountObject(initialBalance);
    }

    ImmutableAccount(Account delegate) {
        delegate_ = delegate;
    }

    public long balance() {
        return delegate_.balance();
    }
}
```

InsufficientFunds

```
package Account;

public class InsufficientFunds extends Exception {
    public InsufficientFunds() {}
}
```

UpdatableAccount

```
package Account;

public interface UpdatableAccount extends Account {
```

```
public void credit(long amount) throws InsufficientFunds;  
public void debit(long amount) throws InsufficientFunds;  
}
```

UpdatableAccountObject

```
package Account;  
  
public class UpdatableAccountObject  
    implements UpdatableAccount {  
    private long currentBalance_  
  
    public UpdatableAccountObject(long initialBalance) {  
        currentBalance_=initialBalance;  
    }  
  
    public long balance() {  
        return currentBalance_  
    }  
  
    public void credit(long amount) throws InsufficientFunds {  
        if (amount >=0 || currentBalance_>=-amount)  
            currentBalance_+=amount;  
        else throw new InsufficientFunds();  
    }  
  
    public void debit(long amount) throws InsufficientFunds {  
        credit(-amount);  
    }  
}
```

MainClass

```
package Account;  
  
public class MainClass {  
  
    private static class Balance extends Thread {  
        AccountRecorder recorder;  
  
        public Balance(AccountRecorder recorder) {  
            this.recorder=recorder;  
        }  
    }  
}
```

```
}

public void run() {
    recorder.recordBalance(new UpdatableAccountObject(100));
}
}

static class CorrectExample extends Thread {
    public static void main(String[] args) {
        AccountRecorder recorder=new AccountRecorder();
        AccountHolder holder=new AccountHolder(recorder);
        new Balance(recorder).start();
        holder.acceptMoney(100);
    }
}

static class FirstBadExample extends Thread {
    public static void main(String[] args) {
        AccountRecorder recorder=new AccountRecorder();
        AccountHolder holder=new AccountHolder(recorder);
        new Balance(recorder).start();
        holder.acceptMoneyWithoutImmutable(100);
    }
}

static class SecondBadExample extends Thread {
    public static void main(String[] args) {
        AccountRecorder recorder=new EvilAccountRecorder();
        AccountHolder holder=new AccountHolder(recorder);
        new Balance(recorder).start();
        holder.acceptMoney(100);
    }
}

static class WrongExample extends Thread {
    public static void main(String[] args) {
        AccountRecorder evilrecorder=new EvilAccountRecorder();
        AccountHolder holder=new AccountHolder(evilrecorder);
        new Balance(evilrecorder).start ();
        holder.acceptMoneyWithoutImmutable(100);
    }
}
}
```

B

Incremental application

B.1 Account

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class Account {  
    protected int creditRate;  
    protected int debitRate;  
    protected int amount;  
  
    public Account() {  
        synchronized(this) {  
            creditRate=2;  
            debitRate=10;  
            amount=0;  
        }  
    }  
}
```

B.2 ATM

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class ATM {  
  
    protected void perform(Card c, int pin, int action, int amount) {  
        new ThreadATM(c, pin, action, amount).start();  
    }  
}
```

B.3 Bank

```
package name.ferrara.pietro.checkmate.bankapplication;

public class Bank {

    public Person openAccount(int money) {
        BankAccount account1=new BankAccount(1000);
        Card card1=new Card(account1, 1234);
        return new Person(card1, account1);
    }

}
```

B.4 BankAccount

```
package name.ferrara.pietro.checkmate.bankapplication;

public class BankAccount {

    private Account account;

    public BankAccount(int money) {
        account=new Account();

        synchronized(account) {
            account.amount=money;
        }
    }

    protected void withdraw(int money) {
        synchronized(account) {
            account.amount-=money;
        }
    }

    protected int getBalance() {
        synchronized(account) {
            return account.amount;
        }
    }

    protected void deposit(int money) {
        synchronized(account) {
```

```

        account.amount+=money;
    }
}

protected void deposit(Money money) {
    synchronized(account) {
        account.amount+=money.getValue();
        money.destroy();
    }
}

protected void action(int choice) {
    if (choice==1) this.withdraw(100);
    if (choice==2) this.deposit(100);
}

protected void calculateInterests() {
    synchronized(account) {
        int temp=account.amount;
        if (temp>=0)
            temp=temp+temp*account.creditRate;
        else temp=temp-temp*account.debitRate;
        account.amount=temp;
    }
}

protected boolean thereAreMoney() {
    synchronized(account) {
        return account.amount>0;
    }
}
}

```

B.5 Card

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class Card {

    private int code;
    private BankAccount account;

    protected Card(BankAccount b, int pin) {
        synchronized(this) {
            code=pin;

```

```

        account=b;
    }
}

protected Money action(int pin, int action, int amount) {
    synchronized(this) {
        if (pin==code) {
            if (action==0) {
                this.show();
                return new Money(0);
            }
            if (action==1) {
                this.withdraw(amount);
                return new Money(amount);
            }
            return new Money(0);
        }
    }
    return new Money(0);
}

protected void show() {
    account.getBalance();
}

protected void withdraw(int amount) {
    new ThreadWithdraw(account, amount).start();
}
}

```

B.6 Cheque

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class Cheque {
    BankAccount account;
    int amount;

    protected Cheque(BankAccount account, int amount) {
        this.account=account;
        this.amount=amount;
    }

    protected void executes(BankAccount to) {

```

```
        new ThreadWithdraw(account, amount).start();
        new ThreadDeposit(to, amount).start();
        amount=0;
    }

    protected Money executes() {
        new ThreadWithdraw(account, amount).start();
        int temp=amount;
        amount=0;
        return new Money(temp);
    }
}
```

B.7 Money

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class Money {
    protected int amount;

    protected Money(int amount) {
        this.amount=amount;
    }

    protected void destroy() {
        amount=0;
    }

    protected int getValue() {
        return amount;
    }
}
```

B.8 Person

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class Person {

    Card card;
    BankAccount account;
    Money cash;

    public Person(Card c, BankAccount acc) {
        this.account=acc;
    }
}
```

```

        card=c;
        cash=new Money(0);
    }

    public void withdraw(int amount, int pin) {
        new ATM().perform(card, pin, 1, amount);
        cash=new Money(cash.getValue()+amount);
    }

    public void checkMoney(int pin) {
        new ATM().perform(card, pin, 0, 0);
    }

    public void closeYear() {
        new ThreadInterests(account).start();
    }

    public void pay(int amount) {
        cash=new Money(cash.getValue()-amount);
    }

    public void transferFound(Person receiver, int amount) {
        new TransferFunds(this.account, receiver.account, amount).executes();
    }

    public void receiveFound(Person receiver, int amount) {
        new TransferFunds(receiver.account, this.account, amount).executes();
    }

    public void withdrawCheque(Cheque c) {
        cash=new Money(cash.getValue()+c.executes().getValue());
    }

    public void depositCheque(Cheque c) {
        c.executes(account);
    }

    public Cheque giveCheque(int amount) {
        return new Cheque(account, amount);
    }
}

```

B.9 ThreadATM

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class ThreadATM extends Thread {  
  
    Card c;  
    int pin;  
    int action;  
    int amount;  
  
    public ThreadATM(Card c, int pin, int action, int amount) {  
        this.c=c;  
        this.pin=pin;  
        this.action=action;  
        this.amount=amount;  
    }  
  
    public void run() {  
        c.action(pin, action, amount);  
    }  
}
```

B.10 ThreadDeposit

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class ThreadDeposit extends Thread {  
    BankAccount bank;  
    int amount;  
  
    public ThreadDeposit(BankAccount b, int amount) {  
        bank=b;  
        this.amount=amount;  
    }  
  
    public void run() {  
        bank.deposit(amount);  
    }  
}
```

B.11 ThreadInterests

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class ThreadInterests extends Thread {
```

```
        BankAccount bank;

    public ThreadInterests(BankAccount b) {
        bank=b;
    }

    public void run() {
        bank.calculateInterests ();
    }
}
```

B.12 ThreadWithdraw

```
package name.ferrara.pietro.checkmate.bankapplication;

public class ThreadWithdraw extends Thread {
    BankAccount bank;
    int amount;

    public ThreadWithdraw(BankAccount b, int amount) {
        bank=b;
        this.amount=amount;
    }

    public void run() {
        bank.withdraw(amount);
    }
}
```

B.13 TransferFunds

```
package name.ferrara.pietro.checkmate.bankapplication;

public class TransferFunds {
    BankAccount sending, receiving;
    int amount;

    public TransferFunds(BankAccount sending, BankAccount receiving, int amount) {
        this.sending=sending;
        this.receiving=receiving;
        this.amount=amount;
    }
}
```

```
        public void executes() {
            new ThreadWithdraw(sending, amount).start();
            new ThreadDeposit(receiving, amount).start();
            amount=0;
        }
    }
```

B.14 Test

```
package name.ferrara.pietro.checkmate.bankapplication;
```

```
public class Test {
    public static void main(String[] args) {
        Bank bank=new Bank();
        Person person1=bank.openAccount(1000);
        Person person2=bank.openAccount(100);

        person1.checkMoney(1234);
        //end of 1st test
        person2.withdraw(100, 1234);
        //end of 2nd test
        Cheque cheque=person1.giveCheque(100);
        person2.depositCheque(cheque);
        //end of 3rd test
        person1.transferFound(person2, 100);
        person2.receiveFound(person1, 100);
        //end of 4th test
        person1.closeYear();
        person2.closeYear();
        //end of 5th test
        person1.withdraw(100, 1234);
        //end of 6th test
        person2.depositCheque(person1.giveCheque(100));
        //end of 7th test
        person2.withdraw(200, 1234);
        //end of 8th test
        person1.withdrawCheque(person1.giveCheque(200));
        //end of 9th test
        person1.checkMoney(1234);
        person2.checkMoney(1234);
        //end of 10th test
    }
}
```

Bibliography

- [1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. In *Proceedings of TOPLAS '06*. ACM Press, 2006.
- [2] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of ISCA '90*, pages 2–14, 1990.
- [3] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *Proceedings of PADTAD '05*. Springer-Verlag, 2005.
- [4] American National Standards Institute. *ANSI/ISO/IEC 9075: Information Technology — Database Languages — SQL*. ANSI, 1999.
- [5] David Aspinall and Jaroslav Sevcik. Java memory model examples: Good, bad and ugly. In *Proceedings of VAMP '07*, 2007.
- [6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26:345–420, 1994.
- [7] R. Bagnara, P.M. Hill, and E. Zaffanella. The Parma Polyhedra Library. <http://www.cs.unipr.it/ppl/>.
- [8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for Object-Oriented programs. In *Proceedings of FMCO '05*. Springer-Verlag, November 2005.
- [9] M. Barnett, M. Fähndrich, and F. Logozzo. Foxtrot and Clousot: Language Agnostic Dynamic and Static Contract Checking for .Net. Technical Report MSR-TR-2008-105, Microsoft Research, Redmond, WA, August 2008.
- [10] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS '04*, 2004.
- [11] Mike Barnett, Robert Deline, Manuel Fahndrich, K. Rustan, M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3:2004, 2004.
- [12] M. C. Bell and P. H. Madden. On the marketing of multicore. In *Proceedings of EDPS 06*, 2006.

- [13] S. Bensalem, J. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of PADTAD '06*. ACM Press, 2006.
- [14] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multithreaded programs. In *Proceedings of PADTAD '05*. ACM Press, 2005.
- [15] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of SIGMOD '95*. ACM Press, 1995.
- [16] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *Proceedings of CAV 07*, LNCS. Springer-Verlag, 2007.
- [17] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proceedings of POPL '09*. ACM Press, 2009.
- [18] G. P. Brat and A. Venet. Precise and scalable static program analysis at NASA. In *Proceedings of IEEE Aerospace Conference*. IEEE Computer Society Press, 2005.
- [19] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *Proceedings of ESOP 07*, LNCS. Springer-Verlag, 2007.
- [20] S. Chaumette and A. Ugarte. A formal model of the java multi-threading system and its validation on a known problem. In *Proceedings of IPDPS '01*. IEEE Computer Society, 2001.
- [21] R. Chugh, J. W. Vounq, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of PLDI '08*. ACM Press, 2008.
- [22] D. R. Cok and J. Kiniry. ESC/Java 2: Uniting ESC/Java and JML. In *Proceedings of CASSIS '04*, 2004.
- [23] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.
- [24] P Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
- [25] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '77*. ACM Press, 1977.

- [26] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.
- [27] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL '79*. ACM Press, 1979.
- [28] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.
- [29] P. Cousot and R. Cousot. Abstract interpretation frameworks. In *Journal of Logic and Computation*. Oxford University Press, 1992.
- [30] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proceedings of ESOP '05*, LNCS. Springer-Verlag, 2005.
- [31] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL '78*. ACM Press, 1978.
- [32] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of PLDI '00*. ACM Press, 2000.
- [33] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI'03*. ACM Press, 2003.
- [34] N. Dor, M. Rodeh, and S. Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In *Proceedings of SAS '01*, LNCS. Springer-Verlag, 2001.
- [35] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of ISCA '98*. ACM Press, 1998.
- [36] Standard ECMA-335. *Common Language Infrastructure (CLI)*. ECMA, 4th edition, June 2006.
- [37] The Economist. Software that makes software better. *The Economist Technology Quarterly*, 8th March:20–21, 2008.
- [38] Perry A. Emrath and David A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of PADD '88*. ACM Press, 1988.
- [39] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurr. Comput. : Pract. Exper.*, 19(3), 2007.
- [40] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA '03*. ACM Press, 2003.

- [41] A. Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *Proceedings of TACAS '07*, LNCS. Springer-Verlag, 2007.
- [42] P. Ferrara. JAIL: Firewall analysis of java card by abstract interpretation. In *Proceedings of EAAI '06*, 2006.
- [43] P. Ferrara. A fast and precise analysis for data race detection. In *Proceedings of Bytecode '08*, ENTCS. Elsevier, 2008.
- [44] P. Ferrara. Static analysis of the determinism of multithreaded programs. In *Proceedings of SEFM '08*. IEEE Computer Society, 2008.
- [45] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *Proceedings of TAP '08*, LNCS. Springer-Verlag, 2008.
- [46] P. Ferrara. Checkmate: a generic static analyzer of java multithreaded programs. In *Proceedings of SEFM '09*. IEEE Computer Society, 2009.
- [47] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .net. In *Proceedings of OOPSLA '08*. ACM Press, 2008.
- [48] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of PLDI '02*. ACM Press, 2002.
- [49] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of PLDI '08*. ACM Press, 2008.
- [50] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *Proceedings of ESOP '06*, LNCS. Springer-Verlag, 2006.
- [51] Paul Gustin and Michael W. Mislove. A truly concurrent semantics for a simple parallel programming language. In *Proceedings of CSL '99*. Springer-Verlag, 1999.
- [52] David Geer. For programmers, multicore chips mean multiple challenges. *Computer*, 40(9):17–19, 2007.
- [53] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proceedings of VMCAI '05*, LNCS. Springer-Verlag, 2005.
- [54] Samir Genaim and Fausto Spoto. Constancy analysis. In *Proceedings of FTfJP '08*, 2008.
- [55] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA '90*, 1990.

- [56] P. Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings TAPSOFT '91*, LNCS. Springer-Verlag, 1991.
- [57] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
- [58] Rajiv Gupta, Santosh Pande, Kleanthis Psarris, and Vivek Sarkar. Compilation techniques for parallel systems. *Parallel Computing*, 25:1741–1783, 1999.
- [59] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *ACM ICSE'06*. ACM Press, 2006.
- [60] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of PPOPP '05*. ACM Press, 2005.
- [61] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of PLSI '04*, 2004.
- [62] Manuel Hermenegildo. Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming. *Parallel Computing*, 26(13–14):1685–1708, 2000.
- [63] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of TRANSACT '06*, 2006.
- [64] Patricia M. Hill and Fausto Spoto. Deriving Escape Analysis by Abstract Interpretation. *Higher-Order and Symbolic Computation*, 19:415–463, 2006.
- [65] M. Hirzel and R. Grimm. Jeannie: granting Java native interface developers their wishes. In *Proceedings of OOPSLA '07*. ACM Press, 2007.
- [66] R. N. Horspool and J. Vitek. Static analysis of postscript code. In *Proceedings of ICCL '92*. IEEE Computer Society Press, 1992.
- [67] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Not.*, 24(7):28–40, 1989.
- [68] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of PASTE '07*. ACM Press, 2007.
- [69] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of FMOODS '08*. Springer-Verlag, 2008.
- [70] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of PLDI '94*. ACM Press, 1994.

- [71] Thuan Quang Huynh and Abhik Roychoudhury. A memory model sensitive checker for c#. In *Proceedings of FM '06*, LNCS. Springer-Verlag, 2006.
- [72] Intel. Teraflops research chip, <http://techresearch.intel.com/articles/tera-scale/1449.htm>.
- [73] Java Grande Forum Benchmark Suite. <http://www.epcc.ed.ac.uk/research/activities/java-grande/>.
- [74] Jlint: Java program checker. <http://artho.com/jlint/>.
- [75] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of CAV '07*, LNCS. Springer-Verlag, 2007.
- [76] M. Karr. On affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, July 1976.
- [77] L. Khachiyan, E. Boros, K. Borys, K. M. Elbassioni, and M. Gurvich. Generating all vertices of a polyhedron is hard. In *Proceedings of SODA '06*. ACM Press, 2006.
- [78] Johannes Kinder, Helmut Veith, and Florian Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of VMCAI '09*, LNCS. Springer-Verlag, 2009.
- [79] G. Koch. Discovering multi-core: extending the benefits of Moore's law. In *Technology Intel Magazine*. Intel, July 2005.
- [80] A. Lal, T. Touili, N. Kidd, and T. W. Reps. Interprocedural analysis of concurrent programs under a context bound. In *Proceedings of TACAS '08*, LNCS. Springer-Verlag, 2008.
- [81] P. Lammich and M. Müller-Olm. Conflict Analysis of Programs with Procedures, Dynamic Thread Creation, and Monitors. In *Proceedings of SAS 08*, LNCS. Springer-Verlag, 2008.
- [82] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Commun. ACM*. ACM Press, 1978.
- [83] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. Computers*, 1979.
- [84] V. Laviro and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Proceedings of VMCAI '09*, LNCS. Springer-Verlag, 2009.
- [85] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.

-
- [86] E. A. Lee. The problem with threads. In *Computer*. IEEE Computer Society Press, 2006.
- [87] Rustan Leino and Peter Muller. A basis for verifying multi-threaded programs. In *Proceedings of ESOP '09*, LNCS. Springer-Verlag, 2009.
- [88] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [89] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [90] F. Logozzo. *Modular Static Analysis of Object-oriented Languages*. PhD thesis, Ecole Polytechnique, France, 2004.
- [91] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *Proceedings of VMCAI '07*, LNCS. Springer-Verlag, 2007.
- [92] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Proceedings of CC '08*, LNCS. Springer-Verlag, 2008.
- [93] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational domain for the efficient validation of array accesses. In *Proceedings of SAC '08*. ACM Press, 2008.
- [94] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. journal version, to appear.
- [95] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proceedings of POPL '05*. ACM Press, 2005.
- [96] Ami Marowka. Parallel computing on any desktop. *Commun. ACM*, 50(9):74–78, 2007.
- [97] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of POPL '07*. ACM Press, 2007.
- [98] A Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986*. Springer-Verlag, 1987.
- [99] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. Efficient, parametric fixpoint algorithm for analysis of java bytecode. In *Proceedings of Bytecode '07*, ENTCS. Elsevier, 2007.
- [100] B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.

-
- [101] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.
- [102] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of PLDI '07*. ACM Press, 2007.
- [103] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of POPL '07*. ACM Press, 2007.
- [104] S. Nanz, F. Nielson, and H. R. Nielson. Modal Abstractions of Concurrent Behaviour. In *Proceedings of SAS '08*, LNCS. Springer-Verlag, 2008.
- [105] J. Navas, M. Mndez-Lojo, and M. Hermenegildo. A generic, context sensitive analysis framework for object oriented programs. In *Proceedings of FTfJP '07*, 2007.
- [106] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. In *ACM Lett. Program. Lang. Syst.* ACM Press, 1992.
- [107] Piotr Nienaltowski. Efficient data race and deadlock prevention in concurrent object-oriented programs. In *Proceedings of OOPSLA '04*. ACM Press, 2004.
- [108] Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. In *Proceedings of CORDIE '06*, 2006.
- [109] J. K. Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, 1996.
- [110] É. Payet and F. Spoto. Magic-Sets Transformation for the Analysis of Java Bytecode. In *Proceedings of SAS '07*, LNCS. Springer-Verlag, 2007.
- [111] I. Pollet. *Towards a Generic Framework for the Abstract Interpretation of Java*. PhD thesis, Department of Computing Science and Engineering, Catholic University of Louvain, 2004.
- [112] I. Pollet and B. Le Charlier. Towards a complete static analyser for java: an abstract interpretation framework and its implementation. In *Proceedings of AIOOL '05*, ENTCS. Elsevier, 2005.
- [113] W. Pugh. The Java memory model is fatally flawed. In *Concurrency - Practice and Experience 12(6)*. Wiley, 2000.
- [114] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proceedings of TACAS '05*, LNCS. Springer-Verlag, 2005.
- [115] S. Qadeer and D. Wu. KISS: keep it simple and sequential. *SIGPLAN Not.*, 39:14–24, 2004.

- [116] Zvonimir Rakamaric and Alan Hu. A scalable memory model for low-level code. In *Proceedings of VMCAI '09*, 2009.
- [117] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
- [118] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22:416–430, 2000.
- [119] Mohammad Raza, Cristiano Calcagno, and Philippa Gardner. Automatic parallelization with separation logic. In *Proceedings of ESOP '09*, LNCS. Springer-Verlag, 2009.
- [120] J. C. Reynolds. Towards a grainless semantics for shared-variable concurrency. In *Proceedings of FSTTCS '04*, LNCS. Springer-Verlag, 2004.
- [121] M. C. Rinard. Analysis of multithreaded programs. In *Proceedings of SAS '01*, LNCS. Springer-Verlag, 2001.
- [122] Abhik Roychoudhury and Tulika Mitra. Specifying multithreaded java semantics for program verification. In *Proceedings of ICSE '02*. ACM Press, May 2002.
- [123] R. Rugina and C. R. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of PLDI '01*. ACM Press, 2000.
- [124] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, 27(2):185–235, 2005.
- [125] Theo C. Ruys and Niels H.M. Aan de Brugh. Mmc: the mono model checker. In *Proceedings of Bytecode '07*, ENTCS. Elsevier, 2007.
- [126] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proceedings of PPOPP '07*. ACM Press, 2007.
- [127] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*. ACM Press, 1995.
- [128] A. Simon and A. King. Analyzing string buffers in c. In *Proceedings of AMAST '02*, LNCS. Springer-Verlag, 2002.
- [129] A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*, LNCS. Springer-Verlag, September 2002.
- [130] F. Spoto. The JULIA Generic Static Analyser. <http://profs.sci.univr.it/~spoto/julia/>.

-
- [131] F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proceedings of FTfJP'2005*, 2005.
- [132] F. Spoto. Nullness analysis in boolean form. In *Proceedings of SEFM '08*. IEEE Computer Society Press, 2008.
- [133] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.
- [134] H. Sutter and J. Larus. Software and the concurrency revolution. In *ACM Queue*. ACM Press, 2005.
- [135] G. Tan and G. Morrisett. Ilea: inter-language analysis across java and c. In *Proceedings of OOPSLA '07*. ACM Press, October 2007.
- [136] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
- [137] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proceedings of PPOPP 06*. ACM Press, 2006.
- [138] C. Von Praun and T. R. Gross. Object race detection. In *Proceedings of OOPSLA '01*. ACM Press, 2001.
- [139] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS '00*, 2000.
- [140] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proceedings of ECOOP '05*, LNCS. Springer-Verlag, 2005.
- [141] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Rigorous concurrency analysis of multithreaded programs. In *Proceedings of CSJP '04*, 2004.