Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: TD 12/07

On Abstract Interpretation of Object Calculi

Stefano Cataudella

Supervisor

Prof. Roberto Barbuti

Referee

Prof. David Sands

Referee

Prof. Dennis Volpano

Referee

Dr. Francesco Logozzo

December 2007

Abstract

This thesis presents some original results in the framework of program verification, referred in particular to object-oriented languages. Object-oriented concepts and programs are expressed using object calculi, since they allow to formalize the basic ideas behind the object-oriented approach, without considering the details which are peculiar of each particular language. We will initially present a very simple object calculus, which models the classical object-oriented features, and which has the computational power of Church's untyped lambda-calculus. This calculus has a functional behavior, and in the following of the thesis it will be extended to include types, to formalize the natural imperative behavior of object-oriented languages, and to extend the model to concurrent languages. The verification of the properties of object-oriented programs will be based on abstract interpretation, since in many cases it can be more precise than other widely used techniques of analysis, like for example model checking or type inference. We will present in detail the theory behind this verification method that will be used as a unified approach in the rest of thesis. The approach will be used in three examples of analysis of object calculi, applied to the fields of safety, optimization and security. The first analysis, related to safety, is intended to verify that threads belonging to a unique concurrent program do not access shared resources simultaneously. We will extend the simple calculi presented at the beginning of the thesis with primitives for locking and unlocking objects, and using a very simple abstraction, we will be able to be more precise in our results than other approaches based on type systems. The second analysis, related to optimization, is in some way complementary to the first one. In fact, in this second case, our aim is to avoid the use of unnecessary locks in concurrent programs, since they often cause an overhead in computation. We will further extend our language with constructs for tracing the objects and the threads which acquire locks on them. The analysis will then check the dependencies between the various locks, to see if some of them may be safely deleted from programs without incurring in an erroneous behavior due to simultaneous accesses to shared resources. The third analysis will be intended to check a property of secure information flow in concurrent programs. The language will be further extended to include information levels and the analysis will check if, for each point of the computation, the values contained in variables are, in some ways, dependent of other variables with higher information levels.

Acknowledgments

I would like to thank a lot of people that helped me during these last three years (actually four) of my Ph.D. studies. Please, forgive me if I will forget someone, but you all are really too many!

First of all a great ”thank you” is due to my supervisor, Professor Roberto Barbuti, for his support, help and patience started during my graduation thesis and continued along my Ph.D. course. I would like to thank him for having trusted me, even if in this last period I moved my attention to other fields of research and work.

I thank my external referees for their insightful reading of this thesis, as well as for their comments, suggestions and corrections.

I also thank my internal referees, Professor Gianluigi Ferrari and Professor Francesca Levi, for their corrections and advices during the first two years of my Ph.D. and in particular for their help for my Ph.D. proposal and advancement talks.

I would also like to thank all the anonymous referees which read and corrected all my published and unpublished works in these years, on some of which this thesis is based.

Some of the publications on which this work is based were written with the collaboration and help of Luca Tesei and Paolo Tiberi, I would like to thank them for their contributions as well.

Now let us switch to the people that helped me in a non-technical way, with their personal support: my parents, Angela and Cesare, my brother Enrico, and all the other relatives and friends, in particular Claudio Cherri and Antonella Rizza, Yan He and Jinghua Gao, Daniela Tulone, Maurizio Atzori, Ivan Lanese, Roberto Zunino, Professor Susanna Pelagatti, Professor Paolo Mancarella.

A particular thanks to Antonio Gulli’. We started our Ph.D. together and now we are working together, thank you for the opportunity you gave to me, for accepting me at work for Ask.com without any particular experience in the field of search engines.

And last but not least, the biggest thanks is due to my wife, Carmela, for her strong support in these years and for our baby, Riccardo.

Thank you all for your precious help!

To my family, Carmela and Riccardo

Contents

Introduction **vii**

I Background **1**

1 Preliminaries **3**

1.1 Basic mathematical notions . . . 3
1.1.1 Relations . . . 3
1.1.2 Orderings and Lattices . . . 4
1.1.3 Functions . . . 5
1.1.4 Fixpoints . . . 6
1.2 Semantics of Programming Languages . . . 6
1.2.1 Transition Systems . . . 6
1.2.2 Reduction Semantics . . . 8
1.2.3 Equational Theories and Structural Congruences . . . 9
1.3 Abstract Interpretation Theory . . . 10

2 The ς-calculus **15**

2.1 Syntax . . . 15
2.2 Semantics . . . 16
2.2.1 Reduction semantics . . . 16
2.2.2 Equational theory . . . 17
2.2.3 Operational semantics . . . 18
2.3 Expressive power . . . 19
2.3.1 Encoding λ-terms in ς-calculus . . . 19
2.3.2 Fixpoints and recursive terms . . . 21

3 Extensions to the pure ς-calculus **23**

3.1 The $\mathbf{Ob}_{\mathbf{1}<:}$-calculus . . . 23
3.1.1 Syntax . . . 23
3.1.2 Typing . . . 24
3.1.3 Operational semantics . . . 26
3.2 The **imp**ς-calculus . . . 26

3.2.1 Syntax 27
3.2.2 Lazy fields vs Eager fields 27
3.2.3 Operational semantics 28
3.3 The **conc**ς-calculus 30
3.3.1 Syntax 31
3.3.2 Reduction semantics 32
3.3.3 Synchronization 33

II Abstract Interpretations of Object Calculi 35

4 Abstract Interpretation against Races 37
4.1 The problem of Race Conditions 37
4.2 Type checking against races 38
4.2.1 Extensions to the **conc**ς-calculus 39
4.2.2 The type system 40
4.3 The object calculus **aconc**ς 43
4.3.1 Syntax 43
4.3.2 Semantics of **aconc**ς 44
4.4 Abstract interpretation of **aconc**ς 46
4.5 Correctness of the Abstract Interpretation 52
4.6 Race checking analysis 62
4.7 Examples 66

5 Abstract Interpretation against Unnecessary Locks 69
5.1 The Problem of Lock Overhead 69
5.2 The object calculus **raconc**ς 71
5.2.1 Syntax 72
5.2.2 Semantics of **raconc**ς 73
5.3 Abstract interpretation of **raconc**ς 75
5.4 Correctness of the Abstract Interpretation 79
5.5 Lock elimination analysis 80
5.6 Example 82

6 Abstract Interpretation against Insecure Flows 85
6.1 The Problem of Insecure Information Flow 85
6.2 The object calculus **saconc**ς 87
6.2.1 Syntax 87
6.2.2 Semantics of **saconc**ς 88
6.3 Abstract interpretation of **saconc**ς 89
6.4 Correctness of the Abstract Interpretation 96
6.5 Secure information flow analysis 107
6.6 Examples 112

Conclusions and Future Works **117**

Bibliography **119**

Introduction

Nowadays, the importance of software systems is becoming larger and larger. Software systems are used in almost all human activities, often with crucial tasks. These systems manipulate enormous quantities of data and their complexity, as well as the complexity of the data they manipulate, require sophisticated tools for design and verification. It is impossible to think, in fact, that programmers may check every single aspect of the software they produce, due to the increasing dimension of programs and to the enormous quantity of *issues* that programmers should look for. These *issues* range from correctness of programs, to safety, security, efficiency, and to many other aspects that may be desirable in a state-of-the-art computer system.

Since programs are too complex to be checked manually in a reasonable amount of time and resources, code designers need automatic, flexible and efficient tools for checking a wide range of program properties. This is why the field of formal methods for program verification is having a great impact on industry and on applications.

The main idea behind formal verification methods is to define mathematical models to represent computer systems and their behavior, as well as the properties to check for these computer systems. Then, the precision of mathematical definitions may be exploited to build algorithms that have a model and a property as their input and that check the existence of the property, or enforce the property itself. The main advantage of these approaches lies in the fact that the correctness of these algorithms for program verification is established using mathematical proofs. Obviously, these approaches could fail if the mathematical models do not represent systems correctly.

Several formalisms have been proposed in the past to verify properties of systems. In particular we refer to *Static Analysis* techniques, which analyze the behavior of programs without executing them. Static Analysis directly checks the program code for inferring properties. Computer system properties are undecidable to compute, in general, so static analyses check for *approximated properties*, in order to be computable using a finite amount of time and memory. There are many examples of static analysis techniques, the most notably of which are *Type Checking*, *Model Checking* and *Abstract Interpretation*.

Type Checking techniques extend languages with the concept of *type*. A type system is defined in order to model the properties which have to be analyzed, and type checking algorithms attempt to give a correct type to programs according to the defined type system. If a program may be typed according to the desired type system, then that program has the property enforced by the type system itself. In the opposite case, nothing can be said, due to the approximation introduced by the static analysis. Type Checking techniques may easily be applied as extensions to language compilers.

Model Checking techniques verify if the abstract model for a computer system, often derived from a hardware or software design, satisfies a formal specification. The specification is often written as temporal logic formulas. Computer systems are usually modeled using transition systems (i.e. directed graphs). Each state of the transition system has an associated proposition which represents the basic properties that hold in that state of the computation. In this kind of static analysis, the model has to be chosen very carefully, in order to avoid a combinatorial explosion of the states space.

Abstract Interpretation is a technique used to describe static analyses and to derive their correctness. It is applied in a *systematic* way by modeling the property to analyze in an *abstract domain*, by establishing relations between the two domains of abstract computations and concrete ones (the ones which describe the semantic of the language), and by finally deriving an *abstract semantics* of the language that allows to execute programs in an approximated way, in order to compute the properties modeled by the abstract domain. This kind of analysis is usually more precise (and more complex) of other widely used techniques, due to the fact that programs are *executed*, even if in an approximated way. We will present the mathematical foundations behind abstract interpretation techniques in chapter 1 of this thesis.

The mathematical model of *object calculi* has been chosen in this thesis in order to represent computer systems. This because the object-oriented approach is becoming more and more important in the design, project and development of computer systems. Object calculi have proved useful to us since they represent object-oriented languages in their basic constructs, without considering the peculiarities (and modeling problems) of a specific object oriented language. Obviously there are also other formalisms which allow to perform these kind of analyses. Featherweight Java [38] is a well-known formalism in this sense, but we preferred the use of object calculi since in this way we can abstract from the syntax and semantics of a particular programming language (Java in this case).

Object calculi were first introduced in nineties by Martin Abadi and Luca Cardelli [1], and in some other works [10, 25, 26, 36], also with extensions. These calculi consider the notion of object as primitive, and model in a very simple and natural way the classical object-oriented notions of *methods*, *instance variables*, *self*, and so on. Even if these calculi are so simple, their computational power is equivalent to the one of Church's untyped lambda-calculus and then object calculi have the full expressiveness of Turing machines. Object calculi may have functional or classical imperative behavior, and can easily be extended (and they have been) with features such as types, concurrency [33], and so on. We will present the basic notions for object calculi in chapter 2, starting from a basic untyped object calculus which provides only the notions of objects, self and methods. We will see how we can encode in this calculus the Church's untyped lambda-calculus, with fixpoint and recursion operators. Then we will examine some extensions to object calculi, referred in particular to types, imperative behavior and concurrency in chapter 3.

Recently, the area of static analysis applied to object oriented calculi or languages is becoming more and more important [30, 40]. This thesis is intended to show the abtract interpretation approach to static analysis applied to the case of object calculi. We will then show three applications of the abstract interpretation approach for detecting properties belonging to three cornerstone areas of computer science: *safety, efficiency* and *security*.

The first analysis we show in chapter 4 is a safety analysis that will be used to detect possible *race conditions* in concurrent languages. A race condition happens when two concurrent threads access a shared resource simultaneously, often provoking incorrect behaviors of programs. We will present an extension of the object calculi defined in the initial part of the thesis, define and prove the correctness of an abstract interpretation, and apply this abstract interpretation to detect possible race conditions.

Chapter 5 will present, instead, an analysis belonging to the area of efficiency, and intended to remove unnecessary synchronization operations from a concurrent language. Synchronization operations may be useful to enforce safety in programs, but present a computational overhead which is avoidable if some of this operations may be safely removed without causing any race condition. We will slightly extend the object calculus presented in chapter 4 and, using its same abstract interpretation we will be able to detect some families of unnecessary synchronization operations which often are present also in real programming languages.

Finally, the last analysis we present in chapter 6 pertains to the area of security and addresses one of the classical problems belonging to this area of computer science: the one of *insecure information flows.* We have an insecure flow of information when we are able to detect some information about secret data inside programs using the results of programs themselves. We will present some cases of insecure flows in concurrent programs, as well as another abstract interpretation that will be proved to be correct, and used to detect such cases.

Examples of our abstract interpretations will be presented in all chapters, in order to see the effectiveness of our approach, which can be applied easily to different areas of computer science, as well as to different programming languages, without considering their peculiarities, but keeping in mind only their basic behavior.

Part I

Background

Chapter 1

Preliminaries

Abstract

This chapter presents some basic notions behind the theory of semantics of programming languages, as well as abstract interpretation [21, 19]. We will start from the very beginning with the notions of *relation*, *ordering*, *lattice*, *function*, *chain* and *fixpoint*. We will then continue with the description of some classical approaches to the analysis of the semantics of programming languages, and end with the abstract interpretation theory.

1.1 Basic mathematical notions

In order to fully understand the theory behind the semantics of programming languages, as well as Abstract Interpretation, some basic mathematical notions are needed. In this section we will then introduce the notions of relation, function, ordering and fixpoint. We will review some classical results, such as Tarsky theorem for fixpoints. These concepts will be used later, in the study of semantics of programming languages, as well as in the definition of the Abstract Interpretation theory.

1.1.1 Relations

Let S and T be sets. The *powerset* of a set S, $\wp(S)$, is defined as the set which contains all subsets of S:

$$\wp(S) = \{X | X \subseteq S\}$$

The *cartesian product* between S and T, $S \times T$, is defined as the set which contains all pairs having their first element belonging to S, and second element belonging to T:

$$S \times T = \{(s, t) | s \in S \wedge t \in T\}$$

We can define in an analogous way the cartesian product between k sets $S_1, \ldots, S_k$ as the set of k-uple such that the i-th element belongs to set S_i for each i.

A k-ary *relation* is a subset of the cartesian product $S_1 \times \ldots \times S_k$. In a similar way, a *binary relation* R between two sets S and T is a subset of the cartesian product $S \times T$:

$$R \in \wp(S \times T)$$

Given a binary relation R on $S \times S$, we can define the following properties of R:

- *Reflexivity*: $\forall x \in S.\ (x, x) \in R$
- *Symmetry*: $\forall s, t \in S.\ (s, t) \in R \Leftrightarrow (t, s) \in R$
- *Antisymmetry*: $\forall s, t \in S.\ ((s, t) \in R \wedge (t, s) \in R) \Rightarrow s = t$
- *Transitivity*: $\forall r, s, t \in S.\ ((r, s) \in R \wedge (s, t) \in R) \Rightarrow (r, t) \in R$

When a binary relation is reflexive, symmetric and transitive, it is called *equivalence relation*, or simply equivalence, and it is usually represented by symbols like $=$, $\equiv$, $\leftrightarrow$ and so on. When an equivalence relation $\equiv$ is defined on a set S, we have that S may be partitioned into subsets according to the *equivalence classes* defined by the relation itself. An equivalence class is a maximal subset of S such that all its elements are in the relation $\equiv$ to one another.

When a binary relation is reflexive, antisymmetric and transitive it is called *partial ordering*, as we will see better in the next section.

1.1.2 Orderings and Lattices

A *partial ordering* $\sqsubseteq$ on a set S, is a binary relation on S which is *reflexive, antisymmetric* and *transitive*. We usually use the infix notation $s \sqsubseteq t$ to say that $(s, t) \in \sqsubseteq$. A set S with a partial ordering relation $\sqsubseteq$ is called *partially ordered set* (or, shortly, *poset*), and is usually represented by the couple $(S, \sqsubseteq)$. We will use symbols such as $\sqsubseteq$, $\subseteq$, $\leq$, $\preceq$ for partial orderings. Strict orderings will be denoted by the corresponding symbols $\sqsubset$, $\subset$, $<$, $\prec$ to exclude equality (i.e. $s \sqsubset t \Leftrightarrow ((s \sqsubseteq t) \wedge (s \neq t))$).

Let $(S, \sqsubseteq)$ be a poset and let T be a subset of S, $T \subseteq S$. An *upper bound* of T is an element $s \in S$ which is greater or equal than each element of T, that is $\forall t \in T.\ t \sqsubseteq s$. The *least upper bound* ($lub(T)$ or $\sqcup(T)$) of T is defined as the upper bound $s \in S$ such that for each other upper bound $s' \in S$, we have $s \sqsubseteq s'$. In a similar way, by using $\sqsupseteq$ instead of $\sqsubseteq$, we can define the concepts of *lower bound* and *greatest lower bound* ($glb(T)$ or $\sqcap(T)$) of $T \subseteq S$.

Let us consider the poset $(S, \sqsubseteq)$ and a set $T \subseteq S$. We say that T is an *ascending chain* if every two elements of T are comparable according to $\sqsubseteq$, that is $\forall s, t \in T.\ (s \sqsubseteq t) \vee (t \sqsubseteq s)$. A chain is said to be *finite* if it is composed by a finite number of elements. The elements belonging to a chain may be disposed in a sequence according to the ordering relation $\sqsubseteq$. If every ascending chain in S has a least upper bound, S is said to be a *complete partial order* (*cpo*). The least upper bound of the set containing all the elements of the chain, is called *limit* of the chain.

A *lattice* is a poset $(S, \sqsubseteq)$ such that every pair of elements in S has both *lub* and *glb*. A *complete lattice* is a poset $(L, \sqsubseteq)$ such that every subset of L has both *lub* and *glb*, that is $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq L$. Complete lattices are usually denoted by the t-uple $\langle L, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$, where $\top$ and $\bot$ are respectively the *top* and *bottom* elements and are defined as $\top = \sqcup L$ and $\bot = \sqcap L$.

1.1.3 Functions

A *binary function* $f : S \mapsto T$ is a binary relation between S and T such that if $(s, t) \in f$ and $(s, r) \in f$, then $t = r$. In other words a function is a relation such that for every $s \in S$, there exists at most one element $t \in T$ such that $(s, t) \in f$. We usually write $f(s) = t$ do denote $(s, t) \in f$, when f is a function. In a similar way we can define *k-ary functions* as particular relations (with the same property as above) between sets $S_1, \ldots, S_k, T$, that is $f : S_1, \ldots, S_k \mapsto T$. In this case we say that k is the *arity* (number of arguments) of function f. Given two sets S and X, such that $X \subseteq S$, and a function $f : S \mapsto T$, we denote by $f(X)$ the set $\{f(x)|x \in X\}$. We call this set *image* of X under f.

A function $f : S \mapsto T$ is said to be *total* if the value $f(s)$ is defined for every argument $s \in S$, that is $\forall s \in S.\ \exists t \in T.\ f(s) = t$. If this is not true, the function is said to be *partial*. A function is *injective* when for each pair of arguments $s \in S$ and $s' \in S$, if $s \neq s'$, then $f(s) \neq f(s')$. A function $f : S \mapsto T$ is *surjective* if the image of S under f is equal to T, that is $\forall t \in T.\ \exists s \in S.\ f(s) = t$. When a function is both injective and surjective, it said to be *bijective*, or a bijection.

Given two posets $(S, \sqsubseteq)$ and $(T, \preceq)$, a function $f : S \mapsto T$ is *monotone* if $s \sqsubseteq s'$ implies $f(s) \preceq f(s')$ for all s and s' in S. A function is said to be upper-*continuous* when it preserves the existence of upper bounds for ascending chains, that is when for all $X \subseteq S$, $f(\sqcup_S X) = \sqcup_T f(X)$, where X is an ascending chain. Lower-continuous functions are defined in a similar way, by considering descending chains instead of ascending ones, and *glb* instead of *lub*. Continuity implies monotonicity, since if we consider $s \in S$ and $s' \in S$ such that $s \sqsubseteq s'$, we can build an ascending chain containing only s and s', and whose least upper bound is s'. Then, from continuity, we have $f(s) \sqsubseteq \sqcup\{f(s), f(s')\} = \sqcup f(\{s, s'\}) = f(\sqcup\{s, s'\}) = f(s')$.

Given two functions $f : T \mapsto U$ and $g : S \mapsto T$ we can consider the *composition* between f and g. This is a new function, denoted by $f \circ g : S \mapsto U$ which is defined by $(f \circ g)(x) = f(g(x))$ for all $x \in S$. The composition of functions preservs continuity (if two functions are continue, then their composition is continue), as well as monotonicity.

1.1.4 Fixpoints

Given a function $f : S \mapsto S$, we say that $x \in S$ is a *fixpoint* of f if $f(x) = x$. By Tarsky fixpoint theorem, we have that the set of fixpoints for a monotone function f, that is $\{x \in S | f(x) = x\}$ is a complete lattice. Then there exist both the *least fixpoint* ($lfp(f)$) and the *greatest fixpoint* ($GFP(f)$) for all monotone functions. By the same theorem, if f is defined on a complete lattice, and if f is upper continuous, we have that the least fixpoint of f can be obtained as the limit of the chain $f^n(\bot)$ which starts from the bottom element $\bot$ of S:

$$lfp(f) = \sqcup_{n \geq 0} f^n(\bot)$$

where $f^0(\bot) = \bot$ and $f^{n+1}(\bot) = f(f^n(\bot))$. Then, in general, the least fixpoint of a function f cannot be obtained in a finite number of steps.

1.2 Semantics of Programming Languages

In this thesis we will use various approaches to describe the semantics of programming languages, represented by object calculi. In general we can say that the *syntax* of a programming language is the set of rules governing the formation of expressions in the language. The *semantics* of a programming language is the *meaning* of those expressions.

There are several forms of language semantics. Axiomatic semantics is defined as a set of axiomatic truths about programming language expressions. Denotational semantics involves modeling programs as static mathematical objects, namely as set-theoretic functions with specific properties. In this thesis we will use a different form of semantics called *operational semantics*.

An operational semantics is a mathematical model of programming language *execution*. In practice it is an interpreter of the language defined mathematically. However, an operational semantics is more precise than an interpreter, since its mathematical definition makes the semantics independent from the meaning of the language in which an interpreter should be defined.

In the following of this section we will outline the mathematical foundations, as well as the main operational semantics approaches used in the following chapters.

1.2.1 Transition Systems

Transition systems are one of the basic models used to represent the computational steps performed during the execution of a program. They are at the basis of almost all the definitions of semantics of programming languages in operational style.

A transition system is a triple $\langle \Gamma, T, \rightarrow \rangle$, where:

- Γ is a set which contains elements called *configurations*

- T is a subset of Γ, $T \subseteq \Gamma$, which contains configurations called *terminal configurations*
- $\rightarrow$ is a binary relation between elements of Γ.

We will denote by γ, γ' two generic elements of Γ, and say $\gamma \rightarrow \gamma'$ to represent that the couple $\langle\gamma, \gamma'\rangle$ belongs to the relation $\rightarrow$. Such a couple is called *transition.*

We can think about configurations as the *states* that a system may encounter during a computation. In particular, we have that terminal configurations represent those states where the system ends its execution. The transition relation represents the behavior of the system. If the system moves from configuration γ to configuration γ' we write $\gamma \rightarrow \gamma'$ to show the modification in the state of the system due to the computation.

Given a transition system $S = \langle\Gamma, T, \rightarrow\rangle$, a *finite derivation* in S is a sequence of configurations $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that, for each $i \in \{1, \ldots, n-1\}$, we have that $\gamma_i \rightarrow \gamma_{i+1}$. Finite derivations usually represent the behavior of terminating programs, but in an analogous way we can define *infinite derivations*, to represent the behavior of non-terminating programs, as infinite sequences of configurations where each element is in the relation $\rightarrow$ with its successor. When talking about derivations, it can be useful to consider the transitive and reflexive closure of the relation $\rightarrow$, in order to represent when a program is obtained as the result of a computation starting from another program, without specifying every single step of the derivation. The reflexive and transitive closure of $\rightarrow$ is usually represented by $\rightarrow^*$. Then $\gamma \rightarrow^* \gamma'$ means that there exists a derivation $\gamma_1 \rightarrow \ldots \rightarrow \gamma_n$ where $\gamma = \gamma_1$ and $\gamma' = \gamma_n$.

Given a transition system $S = \langle\Gamma, T, \rightarrow\rangle$, there may exist some configurations $\gamma \in \Gamma\backslash T$ such that it does not exist $\gamma' \in \Gamma$ such that $\gamma \rightarrow \gamma'$. Such configurations are called *blocked*, and represent states where the program cannot further evolve, even if it is not in a terminal configuration (i.e. inconsistent states).

Transition systems defined in this way are used not only in the definition of semantics, but also in the definition of generic computations performed on programs, such as type inference, verification processes, and Abstract Interpretation.

Conditional inference rules

In the following we will often define the transition relation $\rightarrow$ using *conditional inference rules.* These rules are defined in the following way:

$$\frac{\pi_1 \; \pi_2 \; \ldots \; \pi_n}{\gamma_1 \rightarrow \gamma_2} \; \text{(Name of the rule)}$$

In this rule, $\pi_1, \pi_2, \ldots, \pi_n$ are called *premises*, and in the most general case they are logical formulae or transitions between configurations that must be verified before the application of the rule. If the current configuration of the program is γ_1, and all the premises $\pi_1, \pi_2, \ldots, \pi_n$ are true in the current configuration, then, by applying the rule, we can conclude that $\gamma_1 \rightarrow \gamma_2$.

In general free variables are used inside rules, so that a single inference rule actually represents an infinite set of rules that, opportunely instantiated, are applicable to specific program configurations.

Historically, there are two kind of approaches in the definition of transition systems used in the description of the semantics of programming languages. The main difference between these two approaches lies in the level of detail chosen to describe single steps of computation. If we are interested in the description of every single step in the computation, we can use a *small-step* semantics, where the final result of the evaluation of a program is reached at the end of a derivation containing all intermediate steps. This is typical of the so-called *structural operational semantics*, introduced by Plotkin. This approach allows for compact semantic definitions, as well as simple and powerful proof methods, such as induction on rules, or on the length of the derivations. Moreover, the set of rules constitutes a true *abstract machine* to interpret the language under specification.

On the other hand, if we are simply interested in the results of computations we can instead use a *big-step* semantics, where the transition relation directly describes the correspondence between initial and final configurations of programs. In this thesis we will concentrate on the first approach.

Transition systems may be extended to *labeled transition systems* where we add a set of labels to denote the transitions. A labeled transition system is then denoted by $\langle \Gamma, T, L, \rightarrow \rangle$ where L is the set of labels. The transitions between two configurations γ_1 and γ_2 are then labeled using elements $\ell \in L$, writing $\gamma_1 \xrightarrow{\ell} \gamma_2$. As an example, a labeled transition system may be built using as labels the names of the conditional inference rules which define the transition relation $\rightarrow$. In this way we are able to identify which rule has been applied at each step of a derivation.

1.2.2 Reduction Semantics

A *reduction semantics* or rewriting semantics is a small-step operational semantics which defines an evaluation function for programs. In particular, programs are rewritten in other programs using this function. The definition of the evaluation function is usually made using transition systems, with conditional inference rules.

Often a basic reduction function is defined, describing the elementary reduction steps between simple terms. Then this basic reduction function is extended to general terms using the concept of *reduction context*. A reduction context is a term with an hole, usually denoted by $[\cdot]$. This hole can be filled by other terms of the language. Now, if we have a basic reduction rule saying $s \rightarrow t$, we can apply the rule to a general term T, which contains s, by considering $T = \mathcal{C}[s]$ (so that T is represented as a context with its hole filled by s), by applying the basic reduction rule to s, and by rewriting the new term $\mathcal{C}[t]$.

As an example, we can consider the Church's untyped lambda calculus, whose syntax is defined by the following BNF grammar:

$$a, b ::= x \mid \lambda x.b \mid b(a)$$

The above syntax says that the lambda calculus terms are variables, functions with arity one, and function applications. We can define a reduction semantics for the lambda calculus by defining the basic reduction function as the two following rules: β-reduction and η-reduction:

$$(\lambda x.b)(a) \to b\{\{x \leftarrow a\}\}$$

$$(\lambda x.(b(x))) \to b$$

The first reduction means that when we have a function $\lambda x.b$ and we have to apply it to a term a, we have to rewrite the body of the function, b, and replace every free occurence of the variable x in it with the term a. The second rule requires that b is a function and that x does not occur as a free variable in b. It simply states that the λ-abstraction is the inverse operation of functional application.

According to wath said before, we can extend the semantics to general terms by defining evaluation contexts as follows:

$$\mathcal{C} = [\cdot] \mid \lambda x.\mathcal{C} \mid \mathcal{C}(a) \mid b(\mathcal{C})$$

and by using the following rule:

$$\frac{a \to b}{\mathcal{C}[a] \to \mathcal{C}[b]} \quad \text{(Red-Context)}$$

which formalizes the reduction of general terms.

Usually the reduction semantics is completed by considering the reflexive and transitive closure of the transition relation $\to$, denoted by $\to^*$, in order to formalize reductions which require many steps. Moreover, since programs may also be non-terminating, the semantics of the language is defined as the least (or greatest) fixpoint of the relation $\to^*$.

In the following of the thesis this formalism will be the most used to represent the semantics of object calculi.

1.2.3 Equational Theories and Structural Congruences

Equational theories give semantics to terms of programming languages by defining an equivalence relation among terms of the language itself. In this way we can represent the evolution of a system using the fact that the set of terms of the language is partitioned into equivalence classes, and we can simplify a term $\mathcal{C}[s]$ by replacing the subterm s with another term t such that $s \equiv t$. The most straightforward way to define an equational theory is to consider the reduction rules of a reduction semantics as bidirectional, and add rules for reflexivity, symmetry, transitivity and replacement of local subterms which are equivalent according to the equational theory itself. As an example, we can consider the following equational theory for the untyped lambda calculus:

$$\frac{}{\vdash a \leftrightarrow a}\ \text{(Eq Ref)} \qquad \frac{}{\vdash (\lambda x.b)(a) \leftrightarrow b\{\{x \leftarrow a\}\}}\ \text{(Eq } \beta\text{)}$$

$$\frac{\vdash b \leftrightarrow a}{\vdash a \leftrightarrow b}\ \text{(Eq Symm)} \qquad \frac{\vdash a \leftrightarrow b \quad \vdash b \leftrightarrow c}{\vdash a \leftrightarrow c}\ \text{(Eq Trans)}$$

$$\frac{\vdash a \leftrightarrow b}{\vdash \lambda x.a \leftrightarrow \lambda x.b}\ \text{(Eq } \lambda\text{)} \qquad \frac{\vdash a \leftrightarrow b \quad \vdash c \leftrightarrow d}{\vdash c(a) \leftrightarrow d(b)}\ \text{(Eq appl)}$$

$$\frac{b\ \textit{is a function}, x \notin FV(b)}{\vdash \lambda x.(b(x)) \leftrightarrow b}\ \text{(Eq } \eta\text{)}$$

Equational theories define a semantics equivalence, and must not be confused with Structural Congruences. A structural congruence defines an equivalence relation between terms based on the syntax of terms. When two terms are equivalent according to a structural congruence, they are definitely *the same term*, and can be interchanged with one another during the evolution of the program. Structural congruences will be often used in the definition of the semantics of object calculi, in order to make easier the application of reduction rules. In fact, when a rule does not match directly the structure of a term, it may be possible to replace the term with another equivalent one, in order to make the rule match. As an example, we can consider the α-conversion rule for the untyped λ-calculus:

$$\lambda x.b \equiv \lambda y.b\{\{x \leftarrow y\}\}$$

requiring that y does not occur free in b and is not bound by another λ-abstraction when it replaces x. This rule states that two λ-terms are equivalent (are the same term) if names are replaced with other names, provided that the new names are *fresh*.

1.3 Abstract Interpretation Theory

Abstract Interpretation, introduced by Patrick and Radhia Cousot in 1977 [20], is a theory that allows to describe and prove the correctness of static analyses. It is based on *semantic approximation*, which allows to systematically derive program analysis algorithms based on the semantics of programming languages. Nowadays, the abstract interpretation theory is becoming more and more important in the research field applied to object oriented languages and calculi [4, 5, 6, 7, 9, 24, 29, 30, 40, 52, 58].

Abstract Interpretation is primarily a theoretical framework, since it allows to express many techniques used for static analyses into a unified formalism. Its main purpose is to build *Automatic Static Program Analyzers*, in order to be able to analyze dynamical properties of programs without executing them.

Unfortunately, we know that program properties are in general *undecidable*. Early milestone works from Turing, Gödel, Kleene, Church and others showed that a program cannot, in general, tell if another program has a certain property. Maybe the most famous example is the *halting problem*: we know that it doesn't exist a program which can tell, in a finite number of steps, if another program halts on all its inputs or not. This is why static analyses are based on the concept of *approximation*. Properties are analyzed in an approximated way with a correctness constraint: if the static analyzer assigns a property to a program, then that program must have the property when concretely evaluated. Obviously, the viceversa cannot hold, due to undecidability, then there can be (and often there are) programs that have a certain property, even if static analyzers cannot tell anything.

When the properties of a programs are analyzed using Abstract Interpretation, they are modeled in an *abstract domain*, which is a static approximation of the *concrete domain* where programs are executed. Then each property, defined on the concrete semantic domain, is mapped in a corresponding *abstract property*, defined on the abstract domain, so that it can be analyzed using a reasonable amount of time and resources.

Given the concrete domain of execution of programs C, first of all it is necessary to define an opportune abstraction, according to the properties to be checked. An abstract domain A is then defined, where terms are built using features which make easier the properties detection. For the sakes of finiteness and simplicity, we have that each term in the abstract domain represents a set of terms in the concrete domain, so that it is necessary to consider the concrete domain as a powerset.

Each of the two domains is then extended with an ordering relation, and given the structure of a complete lattice. The ordering relations of the domains represent precision: the lower the values, the more precise they are. The two domains are then the following:

$$\langle \wp(C), \subseteq, C, \emptyset, \cup, \cap \rangle \quad \text{Concrete domain}$$

$$\langle A, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle \qquad \text{Abstract domain}$$

As said before, each abstract value represents a set of concrete values, and then the concrete domain must be a powerset, so that a binding can be made between abstract values and sets of concrete values.

Two functions are defined to bind abstract values and concrete ones:

- The abstraction function α maps each element of $\wp(C)$ to an element of A which is said to *describe* it:

$$\alpha : \ \wp(C) \mapsto A$$

- The concretization function γ maps each element $a \in A$ to the set of elements of C which are described by a:

$$\gamma : \ A \mapsto \wp(C)$$

Since the ordering relations in the two domains reflect precision, if we abstract two sets of concrete values S and T with $S \subseteq T$ we would reasonably expect to obtain two abstract values $\alpha(S)$ and $\alpha(T)$ such that the latter one is less precise then the former, since T contains S. Then we require monotonicity for the abstraction function:

$$\forall c_1, c_2 \in \wp(C).\ c_1 \subseteq c_2 \Rightarrow \alpha(c_1) \sqsubseteq \alpha(c_2)$$

The converse must hold for the concretization function, since having two abstract values a_1 and a_2 such that $a_1 \sqsubseteq a_2$ means that a_2 abstracts *more concrete values* than a_1, since it is less precise:

$$\forall a_1, a_2 \in A.\ a_1 \sqsubseteq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2)$$

In order to guarantee the correctness of the analysis, two relations must hold between the abstraction and the concretization functions, which together make the two functions a *Galois connection*:

$$\forall c \in \wp(C).\ c \subseteq \gamma(\alpha(c))$$

$$\forall a \in A.\ \alpha(\gamma(a)) \sqsubseteq a$$

From the two conditions above, we have that the two functions α and γ mutually determine each other. The first condition, in particular, states that there may be loss of information (approximation) in describing an element of $\wp(C)$ by an element of A. If we have $\forall a \in A.\ \alpha(\gamma(a)) = a$ we have a stronger binding between the two posets $\wp(c)$ and A, called *Galois insertion*.

As said in section 1.2.2, the concrete semantics of a language is defined as the least (or greatest) fixpoint of a concrete semantic evaluation function F, defined on the domain C. The concrete semantic evaluation function is defined in terms of primitive semantic operations f_i on C (as an example, we can consider each f_i as one of the conditional inference rules which define the operational semantics of the language). However, since the actual concrete domain is $\wp(C)$, we need first to lift the concrete semantics $lfp(F)$ to a *collecting* semantics defined on $\wp(C)$.

Lifting $lfp(F)$ to the powerset is simply a conceptual operation, since we have that the collecting semantics may be defined as the set $\{lfp(F)\}$. Then we do not need to define a new collecting semantics from scratch on $\wp(C)$, but just need to reason in terms of liftings of all the primitive operations (and of the whole concrete semantic evaluation function F) while designing the abstract operations and establishing their properties. In the following, by abuse of notation, we will use the same notation for the standard and the collecting (lifted) operations.

Since we have to execute abstract programs, it is necessary to define an *abstract* semantic evaluation function, in correspondence of the concrete one. Then, for each concrete operator f_i a correspondent abstract operator $f_i^{\#}$ is defined. This operator will be part of the abstract semantic evaluation function $F^{\#}$. For the abstract semantics to be correct, the following requirement must be satisfied by each couple of concrete and abstract operators:

$$\forall c_1, \ldots, c_n \in \wp(C).\ f_i(c_1, \ldots, c_n) \subseteq \gamma(f_i^{\#}(\alpha(c_1), \ldots, \alpha(c_n)))$$

The concrete computation step must be more precise than the concretization of the "corresponding" abstract computation step. In other words, whenever a concrete computation is mapped to an abstract one, the result of the abstract computation must represent a set of concrete elements in $\wp(C)$ which contains the result of the concrete computation. This is a very weak requirement, which is satisfied, for example, by an abstract operator which always computes the worst abstract value $\top$ for each of its inputs. The real issue in the design of abstract interpretations is, therefore, *precision*.

An abstract semantic operator $f_i^{\#}$ is said to be *optimal* when it is the most precise abstract operator, correct with respect to the corresponding concrete semantic operator f_i. Then the following condition must hold:

$$\forall a_1, \ldots, a_n \in A.\ f_i^{\#}(a_1, \ldots, a_n) = \alpha(f_i(\gamma(a_1), \ldots, \gamma(a_n)))$$

This is more a theoretical bound and a basis for the design, rather than an implementable definition.

When the abstraction of the concrete computation step is exactly the same as the result of the corresponding abstract computation step, we have also completeness of the abstract operation, since we have no loss of information:

$$\forall c_1, \ldots, c_n \in \wp(C).\ \alpha(f_i(c_1, \ldots, c_n)) = f_i^{\#}(\alpha(c_1), \ldots, \alpha(c_n))$$

The abstract semantic evaluation function $F^{\#}$ is obtained by defining, for each concrete semantic operator f_i, a corresponding locally correct abstract semantic operator $f_i^{\#}$. Since the composition of locally abstract operations is locally correct with respect to the composition of concrete operations, we obtain that the abstract semantic evaluation function $F^{\#}$ is locally correct as well:

$$\forall c \in \wp(C).\ F(c) \subseteq \gamma(F^{\#}(\alpha(c)))$$

We have to say, however, that composition does not preserve optimality, so that the composition of optimal abstract operators may be less precise than the optimal abstract version of the composition.

Local correctness implies global correctness, since correctness is preserved when fixpoints are computed:

$$lfp(F) \subseteq \gamma(lfp(F^{\#}))$$
$$gfp(F) \subseteq \gamma(gfp(F^{\#}))$$

Using the monotonicity of the abstraction function and the fact that α and γ form a Galois connection, we obtain:

$$\alpha(lfp(F)) \sqsubseteq \alpha(\gamma(lfp(F^{\#})) \sqsubseteq lfp(F^{\#})$$
$$\alpha(gfp(F)) \sqsubseteq \alpha(\gamma(gfp(F^{\#})) \sqsubseteq gfp(F^{\#})$$

then the abstraction of the concrete semantics is more precise than the abstract semantics. However the former, in general, is not computable in a finite number of steps, while the latter may be computable in a finite number of steps, if the domain is finite or, at least, noetherian.

Then it is interesting for static program analyses to compute $lfp(F^{\#})$, since the fixpoint computation must terminate, and since, as said before, most program properties are undecidable. Then a loss of precision is accepted, in order to make the analysis feasible.

If the abstract domain is non-noetherian or if the fixpoint computation is too complex, it is usual to use *widening* operators which force termination by computing an upper approximation of $lfp(F^{\#})$, and by guaranteeing termination, even if more approximation is introduced.

Chapter 2

The ς-calculus

Abstract

In this chapter we recall the formal definition of an untyped calculus of objects, presented in [1] which constitutes the kernel for the other calculi used in the thesis. In this calculus objects are primitive and functions are not directly included. However we will show how to encode functions and fixpoints operators in terms of objects.

2.1 Syntax

The basic calculus here presented has a minimal set of syntactic constructs and computation rules, like the untyped version of the λ-calculus. Objects are the only computational structures, and are constituted by collections of named methods. Each method has a bound variable for self and the only operations on objects are method invocation and update. The instance variables inside objects are represented with methods that do not use the self parameter.

The following is the syntax of the pure ς-calculus terms:

a, b ::=	terms
x	variable
$[l_i = \varsigma(x_i)b_i \; ^{i \in 1 \ldots n}]$	object ($\forall i, j.\ l_i \neq l_j$)
$a.l$	method invocation (field selection)
$a.l \Leftarrow \varsigma(x)b$	method update (field update)

Table 2.1: Syntax of the ς-calculus

To complete the formal syntax of the ς-calculus, we give the usual definitions of free variables and substitutions for ς-terms, which will be used later to define the formal semantics of the language:

$fv(\varsigma(y)b)$	$\triangleq$	$fv(b) - \{y\}$
$fv(x)$	$\triangleq$	x
$fv([l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}])$	$\triangleq$	$\bigcup_{i\in 1\ldots n} fv(\varsigma(x_i)b_i)$
$fv(a.l)$	$\triangleq$	$fv(a)$
$fv(a.l \Leftarrow \varsigma(x)b)$	$\triangleq$	$fv(a) \cup fv(\varsigma(x)b)$

Table 2.2: Free variables for ς-calculus

$(\varsigma(y)b)\{\{x \leftarrow c\}\}$	$\triangleq$	$\varsigma(y')(b\{\{y \leftarrow y'\}\}\{\{x \leftarrow c\}\})$ for $y' \notin fv(\varsigma(y)b) \cup fv(c) \cup \{x\}$
$x\{\{x \leftarrow c\}\}$	$\triangleq$	c
$y\{\{x \leftarrow c\}\}$	$\triangleq$	y for $y \neq x$
$[l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}]\{\{x \leftarrow c\}\}$	$\triangleq$	$[l_i = (\varsigma(x_i)b_i)\{\{x \leftarrow c\}\}\ ^{i\in 1\ldots n}]$
$(a.l)\{\{x \leftarrow c\}\}$	$\triangleq$	$(a\{\{x \leftarrow c\}\}).l$
$(a.l \Leftarrow \varsigma(y)b)\{\{x \leftarrow c\}\}$	$\triangleq$	$(a\{\{x \leftarrow c\}\}).l \Leftarrow ((\varsigma(y)b)\{\{x \leftarrow c\}\})$

Table 2.3: Susbtitutions for ς-calculus

As usual, a *closed term* is a term without free variables. Moreover we identify any two objects that differ only in the order of their methods, and also identify $\varsigma(x)b$ with $\varsigma(y)(b\{\{x \leftarrow y\}\})$ for any y not occurring free in b.

2.2 Semantics

In this section we describe the semantics of the ς-calculus, using three different approaches. The first one uses reduction rules, while the second one uses an equational style which groups ς-terms into equivalence classes. The third one, finally, defines an operational semantics for the ς-calculus, which has the advantage of being deterministic and allows us to build an interpreter for the language.

First of all we give an informal semantics to the language saying that method invocation corresponds simply to execute the body of the method binding the objects which calls the method to the self variable. Method update, instead, produces a new object with a method replaced with a new one. This informal notion of semantics will be formalized in the following sections.

2.2.1 Reduction semantics

In the following, three reduction relations are defined: top-level one-step reduction ($\rightarrowtail$), one-step reduction ($\rightarrow$), and general many-step reduction ($\twoheadrightarrow$). These relations capture exactly the informal semantics given before. Error conditions are not made explicit in the relations, since we suppose that objects and method are correctly used.

Definition 2.2.1 (Reduction relations)**.**

- *$a \rightarrowtail b$ if for some $o \equiv [l_i = \varsigma(x_i)b_i\ ^{i \in 1\ldots n}]$ and $j \in 1\ldots n$ either:*

  $a \equiv o.l_j$ and $b \equiv b_j\{\{x_j \leftarrow o\}\}$, or
  $a \equiv o.l_j \Leftarrow \varsigma(x)c$ and $b \equiv [l_j = \varsigma(x)c, l_i = \varsigma(x_i)b_i\ ^{i \in (1\ldots n)-\{j\}}]$.

- *Let a context $\mathbf{C}[-]$ be a term with a single hole and let $\mathbf{C}[d]$ represent the result of filling the hole with the term d (possibly capturing some free variables of d). Then $a \rightarrow b$ if $a \equiv \mathbf{C}[a']$, $b \equiv \mathbf{C}[b']$, and $a' \rightarrowtail b'$, where $\mathbf{C}[-]$ is any context.*

- *$\twoheadrightarrow$ is the reflexive and transitive closure of $\rightarrow$.*

The first relation models the behavior of the operators for selection and update. In particular a selection $o.l_j$ reduces to the body of the selected method where the object o replaces every occurrence of the self variable. An update, instead, reduces to a new object (hence we have a functional behavior) in the intuitive way. The second relation allows to reduce larger terms by reducing smaller components, while the third one models the sequences of reduction steps.

As an example, let us consider the term $o \equiv [l = \varsigma(x)[]].l$ and the context $\mathbf{C}[-] \equiv [k = \varsigma(x)-]$. We have that $\mathbf{C}[o] \equiv [k = \varsigma(x)o]$ and from the above relations we can derive $o \rightarrowtail []$, $\mathbf{C}[o] \rightarrow [k = \varsigma(x)[]]$ and $\mathbf{C}[o].k \twoheadrightarrow []$.

2.2.2 Equational theory

The theory defined in this section is derived from the reduction rules presented before. In particular, the aim of this theory is capturing a notion of equality for ς-terms, useful for saying when two objects behave in the same way. The reduction rules are then rewritten in the following equational theory, and rules are added to have symmetry, transitivity and congruence. This last feature allows to substitute equals for equals inside ς-terms. We have then the following rules:

$$\frac{\vdash b \leftrightarrow a}{\vdash a \leftrightarrow b}\ \text{(Eq Symm)} \qquad \frac{\vdash a \leftrightarrow b \quad \vdash b \leftrightarrow c}{\vdash a \leftrightarrow c}\ \text{(Eq Trans)}$$

$$\frac{}{\vdash x \leftrightarrow x}\ \text{(Eq } x\text{)} \qquad \frac{\vdash b_i \leftrightarrow b'_i \quad \forall i \in 1\ldots n}{\vdash [l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}] \leftrightarrow [l_i = \varsigma(x_i)b'_i\ ^{i\in 1\ldots n}]}\ \text{(Eq Object)}$$

$$\frac{\vdash a \leftrightarrow a'}{\vdash a.l \leftrightarrow a'.l}\ \text{(Eq Select)} \qquad \frac{\vdash a \leftrightarrow a' \quad \vdash b \leftrightarrow b'}{a.l \Leftarrow \varsigma(x)b \leftrightarrow a'.l \Leftarrow \varsigma(x)b'}\ \text{(Eq Update)}$$

$$\frac{a \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n}] \quad j \in 1\ldots n}{\vdash a.l_j \leftrightarrow b_j\{\{x_j \leftarrow a\}\}} \quad \text{(Eval Select)}$$

$$\frac{a \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n}] \quad j \in 1\ldots n}{\vdash a.l_j \Leftarrow \varsigma(x)b \leftrightarrow [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n-\{j\}}]} \quad \text{(Eval Update)}$$

As it is easy to check, the equality notion above defined ($\leftrightarrow$) is the equivalence relation generated by the one-step reduction ($\rightarrow$) defined in the previous section.

2.2.3 Operational semantics

The reductions and equations defined in the previous sections do not impose any specific evaluation order. In this section a reduction system for the closed terms of the ς-calculus is defined. This system is deterministic. The purpose of the reduction system is to reduce every closed expression to a result, which is defined to be a term of the form $[l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n}]$. The reduction system does not reduce the body of methods until they are invoked, hence a *weak* reduction is defined (in the sense that this reduction works only when the binding variables of methods are replaced with the corresponding objects). The weak reduction relation is denoted by $\rightsquigarrow$ and defined by the following rules:

$$\frac{v \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n}]}{\vdash v \rightsquigarrow v} \quad \text{(RO)}$$

$$\frac{v' \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n}] \quad \vdash a \rightsquigarrow v' \quad \vdash b_j\{\{x_j \leftarrow v'\}\} \rightsquigarrow v \quad j \in 1\ldots n}{\vdash a.l_j \rightsquigarrow v} \quad \text{(RS)}$$

$$\frac{\vdash a \rightsquigarrow [l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n}] \quad j \in 1\ldots n}{\vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i \ ^{i \in 1\ldots n-\{j\}}]} \quad \text{(RU)}$$

The first rule simply says that results are not reduced further. The second rule says that in order to evaluate a method selection $a.l_j$ we must first calculate the result of a, and then, if it is in the form of an object, evaluate b_j with the usual binding. Finally the third rule allows to evaluate method updates of the form $a.l_j \Leftarrow \varsigma(x)b$. It requires first to reduce the term a to an object, and then to replace the old method with the new one. Note that the terms b and b_i are not subject to any computation. The reduction system defined above is deterministic: if $\vdash a \rightsquigarrow v$ and $\vdash a \rightsquigarrow v'$, then $v \equiv v'$. This can be trivially seen using an induction on the derivations of $\vdash a \rightsquigarrow v$ and $\vdash a \rightsquigarrow v'$. Moreover, by giving an order in the application of the rules, we can easily build an interpreter for the language such that, given a closed term a, it returns v if and only if $\vdash a \rightsquigarrow v$.

We have two important results about soundness and completeness for this weak reduction relation with respect to the general many-step reduction ($\twoheadrightarrow$). We present simply the results, and omit the proofs.

Proposition 2.2.1 (Soundness of weak reduction).
If $a \rightsquigarrow v$, then $a \twoheadrightarrow v$ and hence $\vdash a \leftrightarrow v$.

This proposition can be easily proved by induction on the derivation of $a \rightsquigarrow v$. The next result is a weak form of completeness, which holds only if a term reduces to a result (an object) using the general many-step reduction $\twoheadrightarrow$.

Theorem 2.2.1 (Completeness of weak reduction).
Let a be a closed term and v be a result. If $a \twoheadrightarrow v$, then there exists v' such that $\vdash a \rightsquigarrow v'$.

It is important to point out that the results v and v' can be syntactically different, because the general many-steps reduction can reduce terms using *contexts*, and then reductions can be applied also *inside* terms. Instead, the weak reduction relation operates only on method invocations and updates which are *outside* of objects. However, by putting together the two results above, we have as a consequence that in the second theorem $\vdash v \leftrightarrow v'$, and then the two objects obtained as results from the reduction semantics and the weak reduction semantics are the same object up to equivalence.

2.3 Expressive power

In this section we show how the ς-calculus can encode the whole untyped Church's λ-calculus. We will show also how to encode the fixpoint operator and, consequently, the recursive λ-terms. This will allow us to write object methods using λ-terms, even if the syntax of the calculus does not provide them directly.

2.3.1 Encoding λ-terms in ς-calculus

We define the following translation $\mathcal{T}$ from λ-terms to ς-terms. We recall that λ-terms consist of variables, functional abstractions and applications.

$$\begin{array}{lcl}
\mathcal{T}(x) & \triangleq & x \\
\mathcal{T}(\lambda x.b) & \triangleq & [arg = \varsigma(x)x.arg, val = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.arg\}\}] \\
\mathcal{T}(b(a)) & \triangleq & ((\mathcal{T}(b)).arg \Leftarrow \varsigma(x)(\mathcal{T}(a))).val \quad \text{for } x \notin fv(\mathcal{T}(a))
\end{array}$$

According to the first row, variables are left unchanged. A functional abstraction is translated into an object which has two methods. The first one will be used to store the argument of the function in the case of an application (the initial value is unimportant), while the second one stores the translation of the body of the function itself. Note that, in this translation, every occurrence of the variable x is replaced with $x.arg$. This substitution allows to retrieve the value of the argument of the function during an application, using the field arg of the object itself. An functional application $b(a)$ is translated into a modification of the first object $\mathcal{T}(b)$ in a way that fills its arg field with the argument for the function $\mathcal{T}(a)$. After that, the method val is invoked. Note that the translation maps nested λ's into nested ς's, then we can emulate functions with multiple parameters using multiple nested ς-binders.

As an example, let's consider the simple λ-term $(\lambda x.x)y$. According to the above definition, its translation into ς-calculus is the following:

$$\begin{aligned}
\mathcal{T}((\lambda x.x)y) &= ((\mathcal{T}(\lambda x.x)).arg \Leftarrow \varsigma(x)(\mathcal{T}(y))).val \\
&= ((\mathcal{T}(\lambda x.x)).arg \Leftarrow \varsigma(x)y).val \\
&= ([arg = \varsigma(x)x.arg, val = \varsigma(x)\mathcal{T}(x)\{\{x \leftarrow x.arg\}\}].arg \Leftarrow \varsigma(x)y).val \\
&= ([arg = \varsigma(x)x.arg, val = \varsigma(x)x.arg].arg \Leftarrow \varsigma(x)y).val
\end{aligned}$$

Using the reduction relations defined in the previous section, we can obtain:

$$([arg = \varsigma(x)x.arg, val = \varsigma(x)x.arg].arg \Leftarrow \varsigma(x)y).val \twoheadrightarrow y$$

which corresponds to the translation of the result of the λ-term $(\lambda x.x)y$.

In general, under this translation we have that both α-conversion and β-conversion are valid. In fact, α-conversion holds trivially because of the renaming properties of ς-binders. For β-conversion we have the following argument:

let $o \equiv [arg = \varsigma(x)(\mathcal{T}(a)), val = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.arg\}\}]$

$$\begin{aligned}
\mathcal{T}((\lambda x.b)a) &= ((\mathcal{T}(\lambda x.b)).arg \Leftarrow \varsigma(x)(\mathcal{T}(a))).val \\
&= (([arg = \varsigma(x)x.arg, \\
&\qquad val = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.arg\}\}]).arg \Leftarrow \varsigma(x)(\mathcal{T}(a))).val \\
&\rightarrow o.val \\
&\rightarrow \mathcal{T}(b)\{\{x \leftarrow x.arg\}\}\{\{x \leftarrow o\}\} \\
&= \mathcal{T}(b)\{\{x \leftarrow o.arg\}\} \\
&\rightarrow \mathcal{T}(b)\{\{x \leftarrow \mathcal{T}(a)\{\{x \leftarrow o\}\}\}\} \\
&= \mathcal{T}(b)\{\{x \leftarrow \mathcal{T}(a)\}\} \qquad \text{since } x \notin fv(\mathcal{T}(a)) \\
&= \mathcal{T}(b\{\{x \leftarrow \mathcal{T}(a)\}\})
\end{aligned}$$

Differently from α-conversion and β-conversion, we have that η-conversion does not hold under this translation. This comes basically from the fact that not every object is the translation of a λ-term. For example in λ-calculus we have that $\lambda y.(xy) = x$, but using our translation we obtain:

$$\begin{array}{rcl}
\mathcal{T}(\lambda y.(xy)) & = & [arg = \varsigma(y)y.arg, val = \varsigma(y)\mathcal{T}(xy)\{\{y \leftarrow y.arg\}\}] \\
& = & [arg = \varsigma(y)y.arg, val = \varsigma(y)((x.arg \Leftarrow y).val)\{\{y \leftarrow y.arg\}\}] \\
& = & [arg = \varsigma(y)y.arg, val = \varsigma(y)((x.arg \Leftarrow y.arg).val)] \\
& \neq & \mathcal{T}(x)
\end{array}$$

2.3.2 Fixpoints and recursive terms

The encoding for λ-calculus presented in the previous section is sufficient to provide an object-oriented versions of all the encodings possible within the λ-calculus, included the definitions of the fixpoint operator and of recursive λ-terms. In this section, however, we will present some definitions that are much simpler than the ones obtainable using directly the translation.

The fixpoint operator, for example, can be defined in the following way:

$$fix \triangleq [arg = \varsigma(x)x.arg, val = \varsigma(x)((x.arg).arg \Leftarrow \varsigma(y)x.val).val]$$

We can verify the fixpoint property: $fix(f) = f(fix(f))$ by adding a constant **fix** to the λ-calculus such that $\mathcal{T}(\mathbf{fix}) = fix$. Using the translation given before and defining $fix_f \triangleq [arg = \varsigma(x)\mathcal{T}(f), val = \varsigma(x)((x.arg).arg \Leftarrow \varsigma(y)x.val).val]$, we obtain:

$$\begin{array}{rcl}
\mathcal{T}(\mathbf{fix}(f)) & = & ([arg = \varsigma(x)x.arg, \\
& & \quad val = \varsigma(x)((x.arg).arg \Leftarrow \varsigma(y)x.val).val].arg \Leftarrow \varsigma(x)\mathcal{T}(f)).val \\
& \rightarrow & [arg = \varsigma(x)\mathcal{T}(f), val = \varsigma(x)((x.arg).arg \Leftarrow \varsigma(y)x.val).val].val \\
& \equiv & fix_f.val \\
& \rightarrow & (((x.arg).arg \Leftarrow \varsigma(y)x.val).val)\{\{x \leftarrow fix_f\}\} \\
& \rightarrow & ((fix_f.arg).arg \Leftarrow \varsigma(y)fix_f.val).val \\
& \rightarrow & (\mathcal{T}(f).arg \Leftarrow \varsigma(y)fix_f.val).val \\
& = & (\mathcal{T}(f).arg \Leftarrow \varsigma(y)\mathcal{T}(\mathbf{fix}(f))).val \\
& \equiv & \mathcal{T}(f(\mathbf{fix}(f)))
\end{array}$$

Recursive terms $\mu x.b$ can be translated into ς-calculus directly from their definition as $\mathcal{T}(\mathbf{fix}(\lambda x.b))$. However, shorter definitions are findable, like the following one:

$$\mathcal{T}(\mu x.b) \triangleq [rec = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.rec\}\}].rec$$

Using this translation we can easily verify that $\mu x.b = b\{\{x \leftarrow \mu x.b\}\}$:

$$\begin{array}{rcl}
\mathcal{T}(\mu x.b) & \equiv & [rec = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.rec\}\}].rec \\
& \rightarrow & \mathcal{T}(b)\{\{x \leftarrow x.rec\}\}\{\{x \leftarrow [rec = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.rec\}\}]\}\} \\
& \equiv & \mathcal{T}(b)\{\{x \leftarrow [rec = \varsigma(x)\mathcal{T}(b)\{\{x \leftarrow x.rec\}\}].rec\}\} \\
& \equiv & \mathcal{T}(b)\{\{x \leftarrow \mathcal{T}(\mu x.b)\}\} \\
& \equiv & \mathcal{T}(b\{\{x \leftarrow \mu x.b\}\})
\end{array}$$

Chapter 3

Extensions to the pure ς-calculus

Abstract

This chapter presents some object calculi derived from the core ς-calculus presented in chapter 2. These calculi are presented because they are more used in the literature than the pure ς-calculus, and because they provide some new useful constructs.

The first extension here presented adds types to the ς-calculus, as well as the notion of subtyping. This calculus is very useful when we need to model some classical object-oriented features such as sub-classing or inheritance. The second extension is an imperative calculus, useful to see the concept of *state* for an object, and how side-effects can modify it. The last extension here presented is a concurrent calculus, which uses some constructs from the π-calculus to model concurrency.

3.1 The $\mathbf{Ob}_{1<:}$-calculus

The calculus presented in this section was defined in [1] in order to add types to the ς-calculus and to provide the notion of subtyping, typical of all object-oriented programming languages. We will present the syntax of the language, followed by a formal system useful to calculate the type of a term of the language. The semantics of the language will be extended from the operational semantics of the ς-calculus, defined in the previous chapter.

3.1.1 Syntax

The syntax of the $\text{Ob}_{1<:}$-calculus is extended from the one of the ς-calculus by adding types and by annotating the definition of methods with the type of the self parameter. The only type available in the following syntax is the object type, built from the corresponding object by giving the return type of each method. However, basic types such as *Bool*, *Nat*, *Int* or others could be added, as an extension, when needed. The syntax of the calculus is defined as follows:

$A, B ::=$	types
$\quad [l_i : B_i \ ^{i \in 1 \ldots n}]$	object type
$a, b ::=$	terms
$\quad x$	variable
$\quad [l_i = \varsigma(x_i : A_i) b_i \ ^{i \in 1 \ldots n}]$	object ($\forall i, j.\ l_i \neq l_j$)
$\quad a.l$	method invocation (field selection)
$\quad a.l \Leftarrow \varsigma(x : A) b$	method update (field update)

3.1.2 Typing

In this section we define a typing system for the $\mathrm{Ob}_{1<:}$-calculus. In the following we will use three kind of typing judgments. A type judgment of the form $E \vdash \diamond$ asserts that E is a well-formed typing environment. An environment is simply a list of variables, annotated with their respective types. A type judgment $E \vdash B$ states that the type B is well-formed in the environment E, while the last kind of judgment, $E \vdash b : B$, states that the term b has type B in the environment E. The formal system defined below is composed by two fragments: the first one, composed by the first three rules, describes how to build environments, and how to extract the type of variables from them. The second part, instead, describes the typing of objects. We have what follows:

$$\frac{}{\emptyset \vdash \diamond}\ (\text{Env } \emptyset) \qquad \frac{E \vdash A \quad x \notin dom(E)}{E, x : A \vdash \diamond}\ (\text{Env } x)$$

$$\frac{E', x : A, E'' \vdash \diamond}{E', x : A, E'' \vdash x : A}\ (\text{Val } x) \qquad \frac{E \vdash B_i \quad \forall i \in 1 \ldots n}{E \vdash [l_i : B_i \ ^{i \in 1 \ldots n}]}\ (\text{Type Object})$$

$$\frac{A \equiv [l_i : B_i \ ^{i \in 1 \ldots n}] \quad E, x_i : A \vdash b_i : B_i \quad \forall i \in 1 \ldots n}{E \vdash [l_i = \varsigma(x_i : A) b_i \ ^{i \in 1 \ldots n}] : A}\ (\text{Val Object})$$

$$\frac{E \vdash a : [l_i : B_i \ ^{i \in 1 \ldots n}] \quad j \in 1 \ldots n}{E \vdash a.l_j : B_j}\ (\text{Val Select})$$

$$\frac{A \equiv [l_i : B_i \ ^{i \in 1 \ldots n}] \quad E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1 \ldots n}{E \vdash a.l_j \Leftarrow \varsigma(x : A) b : A}\ (\text{Val Update})$$

The first rule simply says that the empty environment is well-formed, while the second one explains how to construct environments. According to this rule, in order to add a new variable x in an environment E, we must check that x is not already present in E (to this purpose we write $x \notin dom(E)$) and that the type for x is well-formed. This rules forbids the existence of two or more variables with the same name. The third rule explains how to extract the types of the variables from an environment: it suffices to look for the variable inside the environment itself. The rule "Type Object" asserts that an object type is composed by all the labels of methods, along with their respective well-formed types. The rule "Val Object" explains how to give a type to an object: we must first give a type to all methods, assuming the type of the object for the self variable (note the circularity in assuming something that we have still to determine). Finally, the last two rules explain how to type method invocation and update. In the first case we must give a type to the object which invokes the method, and then extract the type of the invoked method. In the second case we have also to verify that the type of the new method is *exactly* the same as the old one. Then the type for the modified object remain unchanged.

The rules presented so far allow to calculate the type of a $\mathrm{Ob}_{1<:}$-term but do not contain any notion of subtyping. In order to add this notion we define the subtyping relation $<:$ by adding the following typing rules:

$$\frac{E \vdash A}{E \vdash A <: A}\ \text{(Sub Refl)} \qquad \frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}\ \text{(Sub Trans)}$$

$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}\ \text{(Val Subsumption)} \qquad \frac{E \vdash \diamond}{E \vdash \mathit{Top}}\ \text{(Type Top)}$$

$$\frac{E \vdash A}{E \vdash A <: \mathit{Top}}\ \text{(Sub Top)} \qquad \frac{E \vdash B_i \quad \forall i \in 1 \ldots (n+m)}{E \vdash [l_i : B_i\ ^{i \in 1 \ldots (n+m)}] <: [l_i : B_i\ ^{i \in 1 \ldots n}]}\ \text{(Sub Obj)}$$

The fist two rules are for reflexivity and transitivity of the subtyping relation. The third rule allows to subsume an object of type A to type B whenever A is a subtype of B. The next two rules introduce the supertype Top, which would not be strictly necessary, but is useful in many practical situations. The last rule, finally, expresses the classical situation in which a *longer* (with more methods) object type is subtype of a *shorter* one.

We have two interesting properties about uniqueness of typing for $\mathrm{Ob}_{1<:}$-terms. In particular, if we do not consider the subtyping relation just introduced we have the following result:

Proposition 3.1.1 ($\mathbf{Ob_1}$ has unique types)**.**
If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in Ob_1 *(that is in* $\mathrm{Ob}_{1<:}$ *without subtyping), then $A \equiv A'$.*

The above result is provable by induction on the derivation of $E \vdash a : A$. Obviously, the above property cannot hold when we consider the whole $\mathrm{Ob}_{1<:}$-calculus because we have the rules for subsumption. In this case, however, we have the following weaker property:

Proposition 3.1.2 (**$\mathbf{Ob_{1<:}}$** has minimum types)**.**
In $\mathrm{Ob}_{1<:}$*, if* $E \vdash a : A$ *then there exists* B *such that* $E \vdash a : B$ *and, for any* A'*, if* $E \vdash a : A'$ *then* $E \vdash B <: A'$*.*

3.1.3 Operational semantics

The operational semantics defined in the previous chapter for the untyped ς-calculus can be easily extended in the case of a typed calculus like $\mathrm{Ob}_{1<:}$. Actually, in order to adapt the rules previously defined, it is sufficient to ignore any type information as follows:

$$\frac{v \equiv [l_i = \varsigma(x_i : A_i)b_i \ ^{i \in 1 \ldots n}]}{\vdash v \leadsto v} \quad \text{(RO)}$$

$$\frac{v' \equiv [l_i = \varsigma(x_i : A_i)b_i \ ^{i \in 1 \ldots n}] \qquad \vdash a \leadsto v' \qquad \vdash b_j\{\{x_j \leftarrow v'\}\} \leadsto v \qquad j \in 1 \ldots n}{\vdash a.l_j \leadsto v} \quad \text{(RS)}$$

$$\frac{\vdash a \leadsto [l_i = \varsigma(x_i : A_i)b_i \ ^{i \in 1 \ldots n}] \qquad j \in 1 \ldots n}{\vdash a.l_j \Leftarrow \varsigma(x : A)b \leadsto [l_j = \varsigma(x : A_j)b, l_i = \varsigma(x_i : A_i)b_i \ ^{i \in 1 \ldots n - \{j\}}]} \quad \text{(RU)}$$

Reduction in this calculus preserves types. This property is often called *subject reduction* property. This result is stated by the following theorem:

Theorem 3.1.1 (Subject reduction for **$\mathbf{Ob_{1<:}}$**)**.**
Let c *be a closed term and* v *be a result, and assume* $\vdash c \leadsto v$*. If* $\emptyset \vdash c : C$*, then* $\emptyset \vdash v : C$*.*

The proof is obtainable by induction on the derivation of $\vdash c \leadsto v$.

The calculus presented in this section is a first-order calculus. There exist some examples of second-order and higher-order calculi, used to model more complex situations. The calculus here presented is a very simple one, even if it shows how to add type information to the ς-calculus previously presented.

3.2 The impς-calculus

The calculus presented in this section is an imperative variant of the ς-calculus introduced in chapter 2. Object-oriented languages are naturally imperative, and methods often perform side-effects on the internal state of objects. We will see that the semantics of this calculus is, in fact, based on the concept of a *state*, which is modifiable through method invocation and update.

3.2.1 Syntax

The following is the syntax of the impς-calculus. It is extended from the syntax of the ς-calculus by adding a *clone* operations and the usual *let* construct:

a, b ::=	terms
x	variable
$[l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}]$	object ($\forall i, j.\ l_i \neq l_j$)
$a.l$	method invocation (field selection)
$a.l \Leftarrow \varsigma(x)b$	method update (field update)
$clone(a)$	cloning
$let\ x = a\ in\ b$	let

As we will see better describing the semantics of the calculus, there are a few differences between this calculus and the ς-calculus presented before. In particular, method update is now imperative. In fact, while in the ς-calculus the method update operator returned a new object, now the object whose method is updated is modified and returned. Moreover, the cloning construct allows to express an interesting operation characteristic of some languages (called prototype-based languages) in which a new object can be cloned from an existing one with the result of a sharing of methods between the two objects. The let construct, finally, allows to implement side-effects and also to define the sequential evaluation typical of imperative languages as $a; b \triangleq let\ x = a\ in\ b$ for $x \notin fv(b)$.

3.2.2 Lazy fields vs Eager fields

In the syntax of the ς-calculus we have identified fields with methods that do not use their self parameter. This identification is valid also in the impς-calculus here presented. Then a field selection is simply a method invocation, and a field update is simply a method update. These similarities between fields and methods may result useful to make the calculus simpler and compact, but have an unfortunate drawback. In fact, in both field definition and update, the $\varsigma(x)$ binder suspends the evaluation until selection. For what concerns fields, this semantics is both inefficient and inadequate for an imperative calculus, since at every access to an object its suspended fields must be reevaluated, with a consequent repetition of their side-effects. In this sense the impς-calculus here presented may be considered a *lazy* calculus, because fields are not evaluated until they are invoked. To overcome this problem it is possible to define a new calculus with eagerly evaluated fields, impς_f, as follows:

$a, b ::=$	terms
x	variable
$[l_i = b_i \ ^{i \in 1 \ldots n}, l_j = \varsigma(x_j)b_j \ ^{j \in n+1 \ldots n+m}]$	object ($\forall i, j.\ l_i \neq l_j$)
$a.l$	method invocation (field selection)
$a.l := b$	field update
$a.l \Leftarrow \varsigma(x)b$	method update
$clone(a)$	cloning

Now an object is no more simply a collection of methods. In fact fields are now different from methods and objects are no more identifiable up to the order of their components. Evaluation proceeds from left to right, and fields are evaluated (and possibly produce side-effects) when objects are created. The *let* construct and sequencing are no more necessary, since we can define them as follows:

$$\begin{aligned}&let\ x = a\ in\ b \triangleq [def = a, val = \varsigma(x)b\{\{x \leftarrow x.def\}\}].val\\&a; b \triangleq [fst = a, snd = b].snd\end{aligned}$$

In order to better understand the difference between impς and impς$_f$, it is possible to give a translation of the latter calculus into the former as follows:

$$\begin{aligned}&\mathcal{T}(x) \triangleq x\\&\mathcal{T}([l_i = b_i \ ^{i \in 1 \ldots n}, l_j = \varsigma(x_j)b_j \ ^{j \in n+1 \ldots n+m}]) \triangleq let\ y_1 = \mathcal{T}(b_1)\ in \ldots let\ y_n = \mathcal{T}(b_n)\ in\\&\quad [l_i = \varsigma(y_0)y_i \ ^{i \in 1 \ldots n}, l_j = \varsigma(x_j)\mathcal{T}(b_j) \ ^{j \in n+1 \ldots n+m}]\\&\quad y_i \notin fv(b_k \ ^{k \in 1 \ldots n+m}), y_i \text{ distinct}, i \in 0 \ldots n\\&\mathcal{T}(a.l) \triangleq \mathcal{T}(a).l\\&\mathcal{T}(a.l := b) \triangleq let\ y_1 = \mathcal{T}(a)\ in\ let\ y_2 = \mathcal{T}(b)\ in\ y_1.l \Leftarrow \varsigma(y_0)y_2\\&\quad y_i \notin fv(b), y_i \text{ distinct}, i \in 0 \ldots 2\\&\mathcal{T}(a.l \Leftarrow \varsigma(x)b) \triangleq \mathcal{T}(a).l \Leftarrow \varsigma(x)\mathcal{T}(b)\\&\mathcal{T}(clone(a)) \triangleq clone(\mathcal{T}(a))\end{aligned}$$

Then we have two possibilities for an imperative calculus, which are translatable one into the other: the former has a lazy evaluation of fields, but we can emulate side-effects using *let*, while the latter has eager evaluation of fields, field selection and update, and does not require the *let* construct.

3.2.3 Operational semantics

The semantics of the impς-calculus presented in this section is based on the concept of *store*, since the calculus is imperative. In this semantics, object terms reduce to object results consisting of a list of store locations, one location for each object component. The semantics here described uses closures to evaluate methods. In particular, a closure is a pair consisting of a method and a stack. The stack contains the associations between variables and the corresponding values which will be used to evaluate the method. A store maps locations to method closures.

The semantics relies on three kinds of judgment, whose meaning is described hereafter (in the following σ denotes a store, while S denotes a stack):

$\sigma \vdash \diamond$	well-formed store judgment
$\sigma \cdot S \vdash \diamond$	well-formed stack judgment
$\sigma \cdot S \vdash a \rightsquigarrow v \cdot \sigma'$	term reduction judgment

The reduction relation $\rightsquigarrow$ relates a store σ, a stack S, a term a, a result v, and another store σ'. It means that with the store σ and the stack S, the term a reduces to the result v, yielding an updated store σ' in the process; the stack does not change.

In the following we use the symbol ι to denote a generic store location. An object result is represented using the notation $[l_i = \iota_i \ ^{i \in 1\ldots n}]$, to denote the fact that every label is linked to a store location. The store relates locations to method closures and is represented using the notation $\iota_i \mapsto \langle \varsigma(x_i)b_i, S_i\rangle^{i\in 1\ldots n}$. Possibly, we will represent single associations between locations and closures, leaving the rest of the store undetermined. In that case we will use the notation $\sigma, \iota \mapsto \langle\varsigma(x)b, S\rangle$. A modification of the location ι_i in the store σ will be represented as $\sigma.\iota_i \longleftarrow c$, where c is a method closure. Finally, a stack is represented using the notation $x_i \mapsto v_i \ ^{i\in 1\ldots n}$. The following rules describe the operational semantics of the impς-calculus:

$$\frac{}{\emptyset \vdash \diamond}\ \text{(Store } \emptyset) \qquad \frac{\sigma \cdot S \vdash \diamond \quad \iota \notin dom(\sigma)}{\sigma, \iota \mapsto \langle\varsigma(x)b, S\rangle \vdash \diamond}\ \text{(Store } \iota)$$

$$\frac{\sigma \vdash \diamond}{\sigma \cdot \emptyset \vdash \diamond}\ \text{(Stack } \emptyset) \qquad \frac{\sigma \cdot S \vdash \diamond \quad x \notin dom(S) \quad \forall i \in 1\ldots n}{\sigma \cdot (S, x \mapsto [l_i = \iota_i \ ^{i\in 1\ldots n}]) \vdash \diamond}\ \text{(Stack } x)$$

$$\frac{\sigma \cdot (S, x \mapsto v, S') \vdash \diamond}{\sigma \cdot (S, x \mapsto v, S') \vdash x \rightsquigarrow v \cdot \sigma}\ \text{(Red } x)$$

$$\frac{\sigma \cdot S \vdash \diamond \quad \iota_i \notin dom(\sigma) \quad \forall i \in 1\ldots n}{\sigma \cdot S \vdash [l_i = \varsigma(x_i)b_i \ ^{i\in 1\ldots n}] \rightsquigarrow [l_i = \iota_i \ ^{i\in 1\ldots n}] \cdot (\sigma, \iota_i \mapsto \langle\varsigma(x_i)b_i, S\rangle^{i\in 1\ldots n})}\ \text{(RO)}$$

$$\frac{\begin{array}{c}\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i\in 1\ldots n}] \cdot \sigma' \quad \sigma'(\iota_j) = \langle\varsigma(x_j)b_j, S'\rangle \quad x_j \notin dom(S') \\ j \in 1\ldots n \quad \sigma' \cdot (S', x_j \mapsto [l_i = \iota_i \ ^{i\in 1\ldots n}]) \vdash b_j \rightsquigarrow v \cdot \sigma''\end{array}}{\sigma \cdot S \vdash a.l_j \rightsquigarrow v \cdot \sigma''}\ \text{(RS)}$$

$$\frac{\sigma \cdot S \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i\in 1\ldots n}] \cdot \sigma' \quad j \in 1\ldots n \quad \iota_j \in dom(\sigma')}{\sigma \cdot S \vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i \ ^{i\in 1\ldots n}] \cdot (\sigma'.\iota_j \longleftarrow \langle\varsigma(x)b, S\rangle)}\ \text{(RU)}$$

$$\frac{\sigma \cdot S \vdash a \leadsto [l_i = \iota_i\ ^{i \in 1 \ldots n}] \cdot \sigma' \quad \iota_i \in dom(\sigma') \quad \iota'_i \notin dom(\sigma') \quad \forall i \in 1 \ldots n}{\sigma \cdot S \vdash clone(a) \leadsto [l_i = \iota'_i\ ^{i \in 1 \ldots n}] \cdot (\sigma', \iota'_i \mapsto \sigma'(\iota_i)\ ^{i \in 1 \ldots n})} \quad \text{(RC)}$$

$$\frac{\sigma \cdot S \vdash a \leadsto v' \cdot \sigma' \quad \sigma' \cdot (S, x \mapsto v') \vdash b \leadsto v'' \cdot \sigma''}{\sigma \cdot S \vdash let\ x = a\ in\ b \leadsto v'' \cdot \sigma''} \quad \text{(RL)}$$

The first two rules explain how to build stores. Empty stores are well-formed and an association between a location and a closure can be added whenever the location isn't already in the store. The same considerations hold for stacks. When we add a new association to a stack, it is obviously necessary that all labels and locations are distinct. The following rules are reduction rules. They express the core of the semantics of the calculus. According to these rules a variable reduces to the value it has in the current stack. When we reduce an object, we create a new result which is a sequence of locations, and associate each location which a closure, made by coupling each method with the current stack. In order to reduce a method invocation, we have first to reduce the object which invokes the method, then we look in the store for the closure containing the method invoked, we add to the stack in this closure the new association for the self variable, and finally evaluate the body of the method using the new resulting stack. The rule for method update modifies the location where there is the closure for the old method, and stores in it a new closure consisting of the new method and the current stack. The rule for cloning reduces the cloned object, and then creates a new object result, whose locations are associated with the method closures of the cloned object itself (hence we have a sharing of methods). Finally, the rule for *let* reduces the first argument of the construct, associates the result of the reduction to the variable in the current stack, and then reduces the second argument of the construct in the resulting stack.

The calculus presented in this section shows how to give an imperative semantics to the ς-calculus presented in chapter 2. Like the ς-calculus can encode the whole λ-calculus, the calculus here presented can encode a version of the λ-calculus based on procedure, instead of functions. Obviously, we can add type information to this calculus, like we did in the previous section for $\text{Ob}_{1<:}$, to obtain typed imperative calculi of the first order, second order, or higher order.

3.3 The concς-calculus

The calculus presented in this section was introduced by Gordon and Hankin in [33] to add concurrency to the impς-calculus presented in the previous section. The concς-calculus is obtained from the impς-calculus by adding some primitives for concurrency from the π-calculus. Eventually, mutexes can be added to the calculus in order to have operators for synchronization. The semantics of the calculus is given using a chemical-style reduction. To show the expressive power of this calculus, we show an encoding of the asynchronous π-calculus into concς with mutexes.

3.3.1 Syntax

The syntax of the concς-calculus is extended from the one of the impς-calculus by adding two operators for parallel composition ($\Rsh$) and for restriction (ν). Moreover, a new concept of *result* is introduced, in order to force the reduction of objects before method invocation or update. In fact, method invocation or update are possible only when the object which invokes or is updated is reduced to a result. Finally, to formalize the imperative behavior of the calculus, we have also the concept of *reference*, which is an association between a name and an object. The syntax of the concς-calculus is defined as follows:

$u, v \ ::=$	results
$\quad x$	variable
$\quad p$	name
$d \ ::=$	denotations
$\quad [l_i = \varsigma(x_i)b_i \ ^{i \in 1 \dots n}]$	object $(\forall i, j.\ l_i \neq l_j)$
$a, b \ ::=$	terms
$\quad u$	result
$\quad p \mapsto d$	reference
$\quad u.l$	method invocation (field selection)
$\quad u.l \Leftarrow \varsigma(x)b$	method update (field update)
$\quad clone(u)$	cloning
$\quad let\ x = a\ in\ b$	let
$\quad a \Rsh b$	parallel composition
$\quad (\nu p)a$	restriction

The syntax given above is very general. Obviously there are terms which are not well-formed and must be ruled out. For example, we have that names cannot be associated with more than one denotation in the same term. We assume that syntax is used consistently, and do not consider erroneous terms. We have some syntactic conventions for the new terms introduced: $(\nu p)a \Rsh b$ has to be read $((\nu p)a) \Rsh b$; $u.l \Leftarrow \varsigma(x)b \Rsh c$ has to be read $(u.l \Leftarrow \varsigma(x)b) \Rsh c$; and $let\ x = a\ in\ b \Rsh c$ has to be read $(let\ x = a\ in\ b) \Rsh c$. Finally, we omit the formal definition of an operator $fn(a)$ which calculates the *free names* occurring inside a term a, since this definition is analogous to the definition of the operator fv for variables (see chapter 2).

As an informal semantics of the calculus, we have that now the cloning operator produces two distinct copies of an object (then we have no more a sharing of methods like in the impς-calculus). Moreover, a term $a \Rsh b$ means that the two expressions a and b are running in parallel. The result of the whole term is the result of b (hence this parallel composition is not commutative). Finally, the term $(\nu p)a$ generates a new fresh name p, whose scope is a.

3.3.2 Reduction semantics

The semantics here described is based on structural congruence and on reduction relations. In particular, reduction represents individual computation steps, and is defined in terms of structural congruence. Structural congruence allow a syntactical rearrangement of terms, so that the reduction rules may be applied. We begin the formal treatment of the semantics of the concς-calculus by defining the structural congruence as the least congruence which satisfies what follows:

$$\begin{array}{ll}
(a \Rsh b) \Rsh c \equiv a \Rsh (b \Rsh c) & \\
(a \Rsh b) \Rsh c \equiv (b \Rsh a) \Rsh c & \\
(\nu p)(\nu q)a \equiv (\nu q)(\nu p)a & \\
(\nu p)(a \Rsh b) \equiv a \Rsh (\nu p)b & \text{if } p \notin fn(a) \\
(\nu p)(a \Rsh b) \equiv (\nu p)a \Rsh b & \text{if } p \notin fn(b) \\
let\ x = (let\ y = a\ in\ b)\ in\ c \equiv let\ y = a\ in\ (let\ x = b\ in\ c) & \text{if } y \notin fn(c) \\
(\nu p)let\ x = a\ in\ b \equiv let\ x = (\nu p)a\ in\ b & \text{if } p \notin fn(b) \\
a \Rsh let\ x = b\ in\ c \equiv let\ x = (a \Rsh b)\ in\ c &
\end{array}$$

The first equivalence is for associativity of the parallel composition. The second equivalence shows that the parallel composition is commutative if we do not consider the rightmost term. In fact, this is the term that produces the result for the whole parallel composition, and then its position cannot change. Since all the other terms are evaluated for effect, then they may be rearranged freely. The third rule shows that the order of local names is not relevant in a restriction, while the next two rules allow to extract a term out of a restriction, whenever the local name of the restriction does not occur inside it. The next two equivalences allow to rearrange *let* constructs on the basis of unused variables, while the last rule allows to compose a term in parallel with the first argument of a *let*, since the first term of a parallel composition is evaluated only for effect.

Now we can define the reduction relation for this calculus, $\Rightarrow$, as the least relation on terms to satisfy:

$$\text{let } d = [l_i = \varsigma(x_i)b_i\ ^{i \in 1 \ldots n}] \text{ and } d' = [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i\ ^{i \in (1 \ldots n) - \{j\}}]$$

$$\begin{array}{ll}
(p \mapsto d) \Rsh p.l_j \Rightarrow (p \mapsto d) \Rsh b_j\{\{x_j \leftarrow p\}\} & \text{if } j \in 1 \ldots n \\
(p \mapsto d) \Rsh (p.l_j \Leftarrow \varsigma(x)b) \Rightarrow (p \mapsto d') \Rsh p & \text{if } j \in 1 \ldots n \\
(p \mapsto d) \Rsh clone(p) \Rightarrow (p \mapsto d) \Rsh (\nu q)(q \mapsto d \Rsh q) & \text{if } q \notin fn(d) \\
let\ x = p\ in\ b \Rightarrow b\{\{x \leftarrow p\}\} & \\
(\nu p)a \Rightarrow (\nu p)a' & \text{if } a \Rightarrow a' \\
a \Rsh b \Rightarrow a' \Rsh b & \text{if } a \Rightarrow a' \\
a \Rsh b \Rightarrow a \Rsh b' & \text{if } b \Rightarrow b' \\
let\ x = a\ in\ b \Rightarrow let\ x = a'\ in\ b & \text{if } a \Rightarrow a' \\
a \Rightarrow b & \text{if } a \equiv a', a' \Rightarrow b', b' \equiv b
\end{array}$$

The first reduction is for method invocation. As usual, we return the body of the method with the self variable bound to the object which invokes the method. The reduction of method update yields a new reference between the name of the old object and the new denotation for the updated object. The name of the new object is returned as result. The cloning operator produces a copy of the cloned object and returns the name of the new object. The following reduction is for the *let* construct, which works as usual.

The other reductions are quite obvious, and correspond to a generalization of the previous reductions to complex terms, using the basic reductions just explained, as well as the structural congruence rules defined before.

According to the syntax of the calculus, now all method invocation, updates and cloning are done using results instead of objects. Then, if a is not a result, $a.l$ stands for $let\ x = a\ in\ x.l$; while $a.l \Leftarrow \varsigma(x)b$ stands for $let\ x = a\ in\ x.l \Leftarrow \varsigma(x)b$; and $clone(a)$ stands for $let\ x = a\ in\ clone(x)$. Moreover, in contexts expecting a term, let an object $[l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}]$ be short for the term $(\nu p)(p \mapsto [l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}] \upharpoonright p)$, where $p \notin fn([l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}])$.

3.3.3 Synchronization

In this section we show how to add synchronization mechanisms to the concς-calculus. In particular, we add syntax for mutexes (binary semaphores), show a semantics for them, and finally encode the asynchronous π-calculus in the obtained extended calculus, in order to show its expressiveness.

In order to add synchronization, the syntax of the calculus is extended by adding two new denotations: *locked* and *unlocked*; as well as two new terms: $acquire(p)$ and $release(p)$. The semantics of the calculus must be extended consequently by adding the following reduction rules:

$$\begin{array}{ll} (p \mapsto unlocked) \upharpoonright acquire(p) \Rightarrow (p \mapsto locked) \upharpoonright p & \\ (p \mapsto d) \upharpoonright release(p) \Rightarrow (p \mapsto unlocked) \upharpoonright p & \text{for } d \in \{locked, unlocked\} \end{array}$$

The rules are quite simple, and simply show the classical behavior of mutexes. A mutex acquisition, $acquire(p)$, tries to lock the mutex denoted by p. If a reference $p \mapsto unlocked$ is present, then it is changed into $p \mapsto locked$, and p is returned as result. A mutex release, $release(p)$, changes the reference $p \mapsto d$ to $p \mapsto unlocked$, independently of the denotation d (which can be *locked* or *unlocked*), and returns p as result.

Using the new operator just introduced, it is possible to give an encoding of asynchronous communications channels as follows. A channel is an object named p. It has two methods: *read* and *write*, and can be empty, or containing a result. The operation $p.write$ updates the content of the channel, but requires an empty channel in order to work. If the channel is full the write operation blocks. On the other hand, the $p.read$ operation extracts the content of the channel p (and makes the channel empty), requiring a full channel in order to work. If the channel is empty the read operation blocks. Then, at each moment, only one of these two operations may succeed, since the channel may be only in two different states. The term which formalizes what just said is the following one:

$$
\begin{array}{l}
\mathit{Channel} \triangleq \\
\quad \mathit{let}\ \mathit{rd} = \mathit{locked}\ \mathit{in}\ \mathit{let}\ \mathit{wr} = \mathit{unlocked}\ \mathit{in} \\
\quad [\mathit{reader} = \varsigma(s)\mathit{rd},\ \mathit{writer} = \varsigma(s)\mathit{wr},\ \mathit{val} = \varsigma(s)s.\mathit{val}, \\
\quad\ \mathit{read} = \varsigma(s)\mathit{acquire}(s.\mathit{reader});\ \mathit{let}\ x = s.\mathit{val}\ \mathit{in}\ (\mathit{release}(s.\mathit{writer}) \Rsh x), \\
\quad\ \mathit{write} = \varsigma(s)\lambda(x) \\
\qquad (\mathit{acquire}(s.\mathit{writer});\ s.\mathit{val} \Leftarrow \varsigma(s)x;\ \mathit{release}(s.\mathit{reader})) \Rsh x]
\end{array}
$$

In the above term we used the operator for sequential composition which we defined, using *let*, when we described the impς-calculus. Moreover we used the λ-notation in the definition of the method write, since the encoding of λ-calculus defined for the ς-calculus in chapter 2 may be easily extended to the impς-calculus and then to the concς-calculus.

To show the expressive power of this object calculus, we can use the above definition of channel to give the following encoding for the asynchronous π-calculus:

$$
\begin{array}{l}
\mathcal{T}(\overline{x}y) \triangleq x.\mathit{write}(y) \\
\mathcal{T}(x(y).P) \triangleq \mathit{let}\ y = x.\mathit{read}\ \mathit{in}\ \mathcal{T}(P) \\
\mathcal{T}(P|Q) \triangleq \mathcal{T}(P) \Rsh \mathcal{T}(Q) \\
\mathcal{T}((\nu x)P) \triangleq \mathit{let}\ x = \mathit{Channel}\ \mathit{in}\ \mathcal{T}(P) \\
\mathcal{T}(!x(y).P) \triangleq \\
\quad [\mathit{rep} = \varsigma(s)\mathit{let}\ y = x.\mathit{read}\ \mathit{in}\ (\mathcal{T}(P) \Rsh s.\mathit{rep})].\mathit{rep} \quad \text{for } s \notin \{x, y\} \cup fv(P)
\end{array}
$$

The calculus presented in this section shows how to add concurrency to the impς-calculus introduced in the previous section. The calculus is obtained by taking some operators (the parallel composition and the restriction) from the π-calculus. Obviously it is possible to add type information also to this calculus, in the way seen before for the ς-calculus.

Part II

Abstract Interpretations of Object Calculi

Chapter 4

Abstract Interpretation against Races

Abstract

This chapter is a major revision of an article where we investigated the use of abstract interpretation techniques in order to prevent race conditions [7]. A race condition occurs when two or more threads access a shared resource simultaneously, often provoking unexpected behaviors of concurrent programs. To detect race conditions, we present an abstraction of the concurrent object calculus **conc**ς presented in section 3.3 which annotates terms with the set of "locks" held at any time. We use this form of the object calculus to check the absence of race conditions and show that abstract interpretation is more flexible than other type analysis approaches used in this field, allowing to certify as "race free" a larger class of programs.

4.1 The problem of Race Conditions

When programming with multithreaded languages, insidious errors, usually denoted as *race conditions*, can arise [44]. A race condition occurs when two processes access a shared resource simultaneously, often provoking an incorrect and unexpected behavior.

A usual method to avoid such conditions is to provide each resource with a *lock*. A process must acquire the lock on a resource before using it, and a locked resource cannot be used by other processes. Concurrent object oriented languages are often based on this approach: resources are embedded in an object and a lock is attached to the object. Java methods adopt this strategy: a method or a block can be declared `synchronized`. A lock is associated to every object which has a synchronized code [34].

Despite this synchronization method, it is not unusual to write multithreaded programs which access objects without acquiring locks on them, thus creating error conditions. The non-acquisition can be originated by different reasons, the most common being mistakes or the conviction that an object is accessed by a single thread.

We saw in the previous chapters that, in order to avoid the peculiarities of specific programming languages, we can analyze the properties of concurrent object-oriented languages, including race-free conditions, on object calculi like the ones presented in this thesis. Here we refer to the concς-calculus, presented in section 3.3.

Many works have been devoted to the static analysis of programs to find possible race conditions. Such methods are essentially based on type analysis [28, 27, 16, 15]: a program is well-typed iff an object is accessed only when a suitable set of locks, corresponding to a policy of synchronization, is acquired. Obviously, the type correctness can be checked statically by applying a set of typing rules. In particular, in [27] a type analysis checks that an object, in a **conc**ς term, is accessed by a process only if a lock on that object is held by the process itself.

All the mentioned type analyses check a program under correctness assumption which are somewhat rigid. For example this last rule above could be relaxed when no concurrent accesses to the same object can be done during the execution of processes or when objects are accessed by only one process (locks are not necessary in this case and may cause unnecessary overhead). A method for a less rigid analysis can be based on *abstract interpretation* [21, 19]. We saw in fact that, since abstract interpretation executes the program in an abstract (approximated) way to statically check dynamic properties, it can be, in many cases, more precise than type analysis.

We will build an abstraction of the concς-calculus that will embody in the terms the knowledge on the locks held at any time. On the basis of this information the semantic definition can be aware, at the time of an access to an object, whether the lock to that object is held. Thus an analysis can be performed to check that processes accessing an object oare holding the right locks, or that no concurrent accesses to an object are performed at the same time by different processes.

4.2 Type checking against races

In this section we describe the approach adopted in [27] in order to certify a program as race-free. The approach described there is based on type-checking of an extension of the concς-calculus presented in chapter 3. We present the syntax and semantics of the extended calculus, as well as the type checking rules used to perform the static analysis about race-freeness.

4.2.1 Extensions to the concς-calculus

Here we describe the syntax of the calculus used in [27]. The authors define the sets of *results, denotations, lock states* and *terms* as presented in table 4.1:

u	$::=$		results
		x	variable
		p	location
d	$::=$		denotations
		$[\ell_i = \varsigma(x_i)t_i \ ^{i\in 1\ldots n}]^l$	object
l	$::=$		lock states
		$\circ$	unlocked
		$\bullet$	locked
a, b	$::=$		terms
		u	result
		$\nu p.a$	restriction
		$p \mapsto d$	reference
		$u.\ell$	method invocation
		$u.\ell \Leftarrow \varsigma(x)t$	method update
		let $x = a$ in b	let
		$a \Rsh b$	parallel composition
		lock u in a	lock acquisition
		locked p in a	lock acquired

Table 4.1: Syntax of the extension to the **conc**ς-calculus

The main novelties introduced by this new calculus are the elimination of the cloning operation, the introduction of two *lock states* attached to objects ($\circ$ for unlocked objects and $\bullet$ for locked ones), and two new terms which describe the request and the acquisition of a lock on an object.

The semantics of the calculus is largely the same as the one of the concς-calculus. We present the additional rules necessary to reduce the new terms introduced:

$$\begin{array}{lll} (p \mapsto [\ldots]^\circ) \Rsh \textit{lock } p \textit{ in } a & \Rightarrow (p \mapsto [\ldots]^\bullet) \Rsh \textit{locked } p \textit{ in } a & \text{(Red Lock)} \\ (p \mapsto [\ldots]^\bullet) \Rsh \textit{locked } p \textit{ in } u & \Rightarrow (p \mapsto [\ldots]^\circ) \Rsh u & \text{(Red Locked)} \end{array}$$

As it is easy to see, the first rule reduces a lock request on an object with lock state $\circ$ (unlocked) to the same object with lock state $\bullet$ (locked) in parallel with the execution of the term on which the lock was requested. The second rule releases the lock on an object, when the term executed under the lock reduces to a result. There is no need for *unlock* operators then, since locks are released automatically when locked terms reduce to results.

4.2.2 The type system

In this section we describe the type system proposed in [27] to check the absence of race conditions. First of all, in table 4.2, we present the type language:

$A, B ::= [\ell_i : \varsigma(x_i)A_i \cdot r_i \cdot s_i \ ^{i \in 1 \ldots n}] \mid Proc \mid Exp$	Types
$r ::= \{u_1 \ldots u_n\}$	Permissions
$s ::= u \mid +$	Protection annotations

Table 4.2: The type language

An object type $[\ell_i : \varsigma(x_i)A_i \cdot r_i \cdot s_i \ ^{i \in 1 \ldots n}]$ describes an object containing n methods labelled $\ell_1 \ldots \ell_n$. Each method ℓ_i has result type A_i, permission r_i, and protection annotation s_i. The permission r_i is a set of results describing the locks that must be held before the invocation of method ℓ_i. The protection annotation s_i is a result describing the lock that must be held before the update of method ℓ_i. If a method is never updated, then s_i may be the symbol '+'. In addition to these types the type language also includes the types Exp and $Proc$. The type Exp describes the results that may be returned by an expression, while the type $Proc$ (supertype of Exp) is used to cover terms that never return results, such as a reference $p \mapsto d$.

Given a term a, the type system checks that the appropriate locks are held whenever a method is invoked or updated. In addition to this, the system also checks that each lock is held by at most one thread at a time, and that every name introduced is associated with a unique denotation. To do this, the authors provide the definitions of *clean* and *defined* names The following definition is given in terms of *evaluation contexts*, as defined in section 1.2.2. We recall that an evaluation context $\mathcal{E}[\ ]$ is a term with the hole $[\ ]$, that can be filled by a statement. Thus $\mathcal{E}[a]$ means the evaluation context $\mathcal{E}[\ ]$ with the hole filled by the term a.

Definition 4.2.1 (Clean and defined names)**.**
Given a term a, the sets clean(a) and defined(a) are defined by the following rules:

$$\begin{array}{ll} p \in clean(a) & \text{if } a = \mathcal{E}[p \mapsto [\ldots]^\circ] \text{ or } a = \mathcal{E}[locked\ p\ in\ b] \text{ and } p \notin bn(\mathcal{E}) \\ p \in defined(a) & \text{if } a = \mathcal{E}[p \mapsto d] \text{ and } p \notin bn(\mathcal{E}) \end{array}$$

where bn is the function giving the bound names *in a term, and $\mathcal{E}[\ ]$ stands for an evaluation context, defined by the following grammar:*

$$\mathcal{E} ::= [\cdot] \mid let\ x = \mathcal{E}\ in\ b \mid \mathcal{E} \upharpoonright b \mid a \upharpoonright \mathcal{E} \mid (\nu p)\mathcal{E} \mid locked\ p\ in\ \mathcal{E}$$

The type system here defined is based on six judgements, defined in table 4.3. In these judgements, an environment E is a sequence of bindings of results to types, of the form $\emptyset, u_1 : A_1, \ldots, u_n : A_n$.

$E \vdash \diamond$	E is a well-formed environment
$E \vdash A$	given $E,$ A is well-formed
$E \vdash r$	given $E,$ permission r is well-formed
$E \vdash A <: B$	given $E,$ A is a subtype of B
$E \vdash r <: r'$	given $E,$ r is a sub-permission of r'
$E; r \vdash a : A$	given E and $r,$ the term a has type A

Table 4.3: Type judgements

In the following, we define the typing rules used to calculate the typing judgements just described.

$$\frac{}{\emptyset \vdash \diamond}\ \text{(Env } \emptyset\text{)} \qquad \frac{E \vdash A \quad u \notin dom(E)}{E, u : A \vdash \diamond}\ \text{(Env } u\text{)}$$

$$\frac{E \vdash \diamond \quad r \subseteq dom(E)}{E \vdash r}\ \text{(Perm)} \qquad \frac{E \vdash \diamond}{E \vdash Proc}\ \text{(Type Proc)}$$

$$\frac{E \vdash \diamond}{E \vdash Exp}\ \text{(Type Exp)}$$

$$\frac{\begin{array}{c} E \vdash \diamond \quad E, x_i : [\,] \vdash B_i <: Exp \quad E, x_i : [\,] \vdash r_i \\ s_i \in r_i \cup \{+\} \quad \forall i \in 1 \ldots n \end{array}}{E \vdash [\ell_i : \varsigma(x_i) B_i \cdot r_i \cdot s_i \ ^{i \in 1 \ldots n}]}\ \text{(Type Obj)}$$

$$\frac{\begin{array}{c} A = [\ell_i : \varsigma(x_i) B_i \cdot r_i \cdot s_i \ ^{i \in 1 \ldots n}] \quad E = E_1, p : A, E_2 \quad E \vdash \diamond \\ E; r_i\{\{x_i \leftarrow p\}\} \vdash b_i\{\{x_i \leftarrow p\}\} : B_i\{\{x_i \leftarrow p\}\} \\ defined(b_i) = clean(b_i) = \emptyset \quad \forall i \in 1 \ldots n \end{array}}{E; \emptyset \vdash p \mapsto [\ell_i = \varsigma(x_i) b_i \ ^{i \in 1 \ldots n}]^l : Proc}\ \text{(Val Obj)}$$

$$\frac{E, u : A, E' \vdash \diamond}{E, u : A, E'; \emptyset \vdash u : A}\ \text{(Val } u\text{)}$$

$$\frac{E; \emptyset \vdash u : [\ell_i : \varsigma(x_i) B_i \cdot r_i \cdot s_i \ ^{i \in 1 \ldots n}] \quad j \in 1 \ldots n}{E; r_j\{\{x_j \leftarrow u\}\} \vdash u.\ell_j : B_j\{\{x_j \leftarrow u\}\}}\ \text{(Val Select)}$$

$$\frac{\begin{array}{c} A = [\ell_i : \varsigma(x_i) B_i \cdot r_i \cdot s_i \ ^{i \in 1 \ldots n}] \quad E; \emptyset \vdash u : A \\ E; r_j\{\{x_j \leftarrow u\}\} \vdash b\{\{x_j \leftarrow u\}\} : B_j\{\{x_j \leftarrow u\}\} \quad s_j \neq + \\ j \in 1 \ldots n \quad defined(b) = clean(b) = \emptyset \end{array}}{E; \{s_j\{\{x_j \leftarrow u\}\}\} \vdash u.\ell_j \Leftarrow \varsigma(x_j) b : A}\ \text{(Val Update)}$$

$$\frac{\begin{array}{c} E; r \vdash a : A \quad E, x : A; r \vdash b : B \quad E \vdash A <: Exp \quad E \vdash B <: Exp \\ defined(b) = clean(b) = \emptyset \end{array}}{E; r \vdash let\ x = a\ in\ b : B} \text{ (Val Let)}$$

$$\frac{E, p : A; r \vdash a : B \quad E \vdash r \quad E \vdash B \quad p \in defined(a) \quad p \in clean(a)}{E; r \vdash (\nu p)a : B} \text{ (Val Res)}$$

$$\frac{\begin{array}{c} E; \emptyset \vdash a : Proc \quad E; r \vdash b : B \\ defined(a) \cap defined(b) = \emptyset \quad clean(a) \cap clean(b) = \emptyset \end{array}}{E; r \vdash a \Rsh b : B} \text{ (Val Par)}$$

$$\frac{E; \emptyset \vdash u : [\,] \quad E; r \cup \{u\} \vdash a : A \quad defined(a) = clean(a) = \emptyset}{E; r \vdash lock\ u\ in\ a : A} \text{ (Val Lock)}$$

$$\frac{E; \emptyset \vdash p : [\,] \quad E; r \cup \{p\} \vdash a : A \quad p \notin clean(a)}{E; r \vdash locked\ p\ in\ a : A} \text{ (Val Locked)}$$

$$\frac{E; r \vdash a : A \quad E \vdash A <: B \quad E \vdash r <: r'}{E; r' \vdash a : B} \text{ (Val Subsumption)}$$

$$\frac{E \vdash r \quad E \vdash r' \quad r \subseteq r'}{E \vdash r <: r'} \text{ (Subperm)}$$

$$\frac{E \vdash A}{E \vdash A <: A} \text{ (Sub Refl)} \qquad \frac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C} \text{ (Sub Trans)}$$

$$\frac{E \vdash A \quad A \neq Proc}{E \vdash A <: Exp} \text{ (Sub Exp)} \qquad \frac{E \vdash A}{E \vdash A <: Proc} \text{ (Sub Proc)}$$

$$\frac{E \vdash [\ell_i : \varsigma(x_i)B_i \cdot r_i \cdot s_i\ ^{i \in 1 \ldots n+m}]}{E \vdash [\ell_i : \varsigma(x_i)B_i \cdot r_i \cdot s_i\ ^{i \in 1 \ldots n+m}] <: [\ell_i : \varsigma(x_i)B_i \cdot r_i \cdot s_i\ ^{i \in 1 \ldots n}]} \text{ (Sub Obj)}$$

The majority of the rules just seen is very similar to the rules presented in chapter 3 for the $\mathrm{Ob}_{1<:}$-calculus. The most important rules are those which describe the last typing judgement shown in table 4.3. These rules control that every invocation or update of methods is performed while holding the necessary locks. In particular, the rules for method invocation and update give a correct type to the corresponding term *only* if *all* the locks which protect the respective method are held.

In our opinion, this requirement makes this algorithm too restrictive, and in the following sections we will show how to address the same problem using abstract interpretation techniques.

4.3 The object calculus aconcς

This section describes a concurrent object calculus which is based on the concς-calculus described in chapter 3. We will use this calculus in our abstract interpretation to detect race-conditions. Since the calculus is very similar to the ones seen before in this thesis, we will use this section to formally recall the calculus syntax and semantics.

4.3.1 Syntax

The syntax of the calculus is largely the same as the calculus shown in table 4.1. We added numeric results and an *if* construct in order to express terms more easily. The Table 4.4 defines the following syntactic categories: results, denotations and terms.

u	$::=$		<u>results</u>
		x	variable
		p	location
		n	integer number
d	$::=$		<u>denotations</u>
		$[\ell_i = \varsigma(x_i)t_i \ ^{i \in 1 \ldots n}]^l$	object
l	$::=$		<u>lock states</u>
		$\circ$	unlocked
		$\bullet$	locked
r, s, t	$::=$		<u>terms</u>
		u	result
		$\nu p.t$	restriction
		$p \mapsto d$	reference
		$u.\ell$	method invocation
		$u.\ell \Leftarrow \varsigma(x)u$	field update
		lock u in t	lock acquisition
		locked p in t	lock acquired
		let $x = s$ in t	let
		$s \upharpoonright t$	parallel composition
		if r then s else t	if

Table 4.4: Syntax of **aconc**ς

Results are defined as *variables*, *numbers* or *references* to objects. A *denotation* $[\ell_i = \varsigma(x_i)t_i \ ^{i \in 1 \ldots n}]^l$ describes as usual an object with a collections of methods named

ℓ_i. In addition, the object has a *lock state* which can be either $\circ$, meaning that the object is not locked by any process, or $\bullet$, meaning that a process holds a lock on it.

A *term* is a *result*, a *restriction*, a *reference*, a *method invocation*, a *method update*, a *lock acquisition*, a *lock acquired*, a *let expression*, a *parallel composition* of terms or an *if expression*. The reference term says that an object is identified by the reference, p, to it. The reference p is introduced by a restriction, $\nu p.t$, which binds the reference p with scope t. The method invocation and field update are the usual ones, with the only exception that fields are now *constants*. A lock acquisition is a term which describes an execution after having acquired a lock to an object. A *lock acquired term*, locked p in t, says that a lock on p is held and that the subterm t can be executed under this lock. The parallel composition of terms, $s \Rsh t$, indicates the parallel execution of s and t. The result of the construct is, like in the concς-calculus, the result of t; s is evaluated only for effect. Integer expressions, let terms and if terms are usual.

We suppose, as usual, that the syntax is used consistently, so that, for example, the terms used as guards of *if* statements should return integer values.

4.3.2 Semantics of aconcς

The semantics of **aconc**ς is given in terms of a structural congruence and a set of reduction rules. Structural congruence allows to syntactically transform terms in order to apply the reduction rules. The application of a reduction rule corresponds to a computation step.

Both the structural congruence and reduction rules are given in terms of *evaluations contexts*, like shown before for the calculus presented in [27].

The possible holes in a term are given in Table 4.5, where the syntax of contexts is given. $[\cdot]$ means that the context can be the whole term.

$$
\begin{array}{lll}
\mathcal{E} & ::= & \\
& & [\cdot] \\
& & \mathcal{E} \Rsh t \\
& & s \Rsh \mathcal{E} \\
& & \text{locked } p \text{ in } \mathcal{E} \\
& & \text{let } x = \mathcal{E} \text{ in } t \\
& & \text{if } \mathcal{E} \text{ then } s \text{ else } t \\
& & \nu p.\mathcal{E}
\end{array}
$$

Table 4.5: Reduction contexts

The structural congruence rules are given in Table 4.6.

The first rule says that the left term in a parallel composition can be inserted or extracted from an evaluation context, provided that this insertion is capture-free (the definition of free names fn is a straightforward extension of the definition in

$$\begin{array}{lcll} s \Rsh \mathcal{E}[t] & \equiv & \mathcal{E}[s \Rsh t] & \text{if } fn(s) \cap bn(\mathcal{E}) = \emptyset \\ (\nu p)\mathcal{E}[s] & \equiv & \mathcal{E}[(\nu p)s] & \text{if } p \notin fn(\mathcal{E}) \cup bn(\mathcal{E}) \end{array}$$

Table 4.6: Structural congruence rules

table 2.1). The reason of such a congruence is that the left term is only evaluated for effect. From this rule it is possible to prove both the associativity and the commutativity, up to the last parallel term, of the parallel operator $\Rsh$. In fact, by defining $\mathcal{E} = [\cdot] \Rsh t$ we have that:

$$(r \Rsh s) \Rsh t \equiv \mathcal{E}[r \Rsh s] \equiv r \Rsh \mathcal{E}[s] \equiv r \Rsh (s \Rsh t)$$

For the commutativity, instead, by defining $\mathcal{E} = s \Rsh [\cdot]$ we obtain:

$$(r \Rsh s) \Rsh t \equiv r \Rsh (s \Rsh t) \equiv r \Rsh \mathcal{E}[t] \equiv \mathcal{E}[r \Rsh t] \equiv s \Rsh (r \Rsh t) \equiv (s \Rsh r) \Rsh t$$

Moreover, this rule allows to prove also other equivalences, like for example:

$$r \Rsh \text{let } x = s \text{ in } t \equiv \text{ let } x = (r \Rsh s) \text{ in } t$$

The second rule allows to get rid of names that do not bind anything in the current context. This rule can be used to prove the commutativity between names in restrictions. In fact, by defining $\mathcal{E} = (\nu q)[\cdot]$ we have that:

$$(\nu p)(\nu q)t \equiv (\nu p)\mathcal{E}[t] \equiv \mathcal{E}[(\nu p)t] \equiv (\nu q)(\nu p)t$$

Moreover, using this same rule, we are able to prove the following equivalences:

$$\begin{array}{ll} (\nu p)(s \Rsh t) \equiv (\nu p)s \Rsh t & \text{if } p \notin fn(t) \\ (\nu p)(s \Rsh t) \equiv s \Rsh (\nu p)t & \text{if } p \notin fn(s) \\ (\nu p)\text{let } x = s \text{ in } t \equiv \text{let } x = (\nu p)s \text{ in } t & \text{if } p \notin fn(t) \end{array}$$

The reduction rules are given in Table 4.7. Here follows a brief explanation. (Red invoke) requires a term, in parallel, which defines the reference to the object the method of which is called; then the method is invoked with the instantiation of the self parameter to the reference to the object itself. (Red update) updates a method in an object; the result of the term is the reference to the modified object (as in [27, 33]). We point out that the rule for method update allows only to update *instance variables* of objects, assigning constants to them, as it is usual in object-oriented programming languages. In fact, the requirement $\ell_j = \varsigma(x)u_j$ tells that the updated method is actually a constant, and the syntax of method update allows to update using only constants as arguments. (Red lock) acquires a lock to an object and (Red unlock) unlocks the object when a result is computed. (Red let) performs a substitution for the variable x when a result u is reached. (Red if0) and (red ifn) reduce the if term in the standard way. Finally (Red context) says that the reduction rules can be applied to any evaluation context.

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n)}{p \mapsto d \upharpoonright p.\ell_j \longrightarrow p \mapsto d \upharpoonright t_j\{\{x_j \leftarrow p\}\}} \text{ (Red invoke)}$$

$$\frac{\begin{array}{c} d = [\ell_j = \varsigma(x)u_j, \ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)-\{j\}}]^l \\ d' = [\ell_j = \varsigma(x)u, \ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)-\{j\}}]^l \end{array}}{p \mapsto d \upharpoonright p.\ell_j \Leftarrow \varsigma(x)u \longrightarrow p \mapsto d' \upharpoonright p} \text{ (Red update)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)}]^\circ \quad d' = [\ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)}]^\bullet}{p \mapsto d \upharpoonright \text{lock } p \text{ in } t \longrightarrow p \mapsto d' \upharpoonright \text{locked } p \text{ in } t} \text{ (Red lock)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)}]^\bullet \quad d' = [\ell_i = \varsigma(x_i)t_i \;^{i\in(1\ldots n)}]^\circ}{p \mapsto d \upharpoonright \text{locked } p \text{ in } u \longrightarrow p \mapsto d' \upharpoonright u} \text{ (Red unlock)}$$

$$\frac{}{\text{let } x = u \text{ in } t \longrightarrow t\{\{x \leftarrow u\}\}} \text{ (Red let)}$$

$$\frac{}{\text{if } 0 \text{ then } s \text{ else } t \longrightarrow t} \text{ (Red if0)}$$

$$\frac{n \neq 0}{\text{if } n \text{ then } s \text{ else } t \longrightarrow s} \text{ (Red ifn)}$$

$$\frac{s \longrightarrow t}{\mathcal{E}[s] \longrightarrow \mathcal{E}[t]} \text{ (Red context)}$$

Table 4.7: Concrete Reduction rules

4.4 Abstract interpretation of aconcς

In this section we define an abstract interpretation of **aconc**ς. The concrete domain, as usual, is represented by the powerset $\wp(C)$, where C is the set containing all the terms of the **aconc**ς-calculus. The abstract interpretation is given with respect to an abstract calculus which approximates the concrete one. In fact, since the semantics is given using reduction rules, and there is not a concept of *state* of the computation separated from syntax, we have to chose as abstract domain a whole abstract calculus.

The abstraction consists mainly in adding a concept of *lock environment* to the terms of the calculus. A *lock environment* is a set of all the locks held by a term. Using this abstraction, the abstract semantics can be aware wheter a term is accessing an object under a certain lock or not. This knowledge is our basis to detect race conditions in the abstract calculus. Moreover, we will see that recursive calls will be *truncated* in the abstract semantics, so that the set of possible terms which can be generated by reduction and structural congruence, starting from a general abstract term, is finite. We can then build a transition system, the states of which are abstract terms, which can be finitely analyzed to establish properties of the concrete program which originated it.

We call the abstract object calculus **aconc**$\varsigma^\sharp$. Its syntax is given in Table 4.8. We will denote by A the set of all abstract terms, that is the abstract domain of our abstract interpretation.

$u^\sharp$	$::=$	$x \mid p \mid \odot \mid \top^\sharp$
$d^\sharp$	$::=$	$[\ell_i = \varsigma(x_i)t_i^\sharp \ {}^{i\in 1\ldots n}]^l$
l	$::=$	$\circ \mid \bullet$
$r^\sharp, s^\sharp, t^\sharp$	$::=$	$u^\sharp \mid p \mapsto d^\sharp \mid \nu p.t^\sharp \mid u^\sharp.\ell$ $u^\sharp.\ell \Leftarrow \varsigma(x)u^\sharp \mid \text{lock } u^\sharp \text{ in } t^\sharp \mid \text{locked } p \text{ in } t^\sharp$ $\text{let } x = s^\sharp \text{ in } t^\sharp \mid s^\sharp \Rsh t^\sharp \mid \text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp \mid \llbracket t^\sharp \rrbracket^{L,S}$
L	$=$	$\{p_1, \ldots, p_n\}$
S	$=$	$\{p_1.\ell_1, \ldots, p_m.\ell_m\}$ multiset

Table 4.8: Syntax of **aconc**$\varsigma^\sharp$

There are a few differences between the syntax of the concrete and the abstract calculus. In the abstract calculus, for the sake of finiteness, integer values are collapsed to a unique value, denoted by $\odot$. Moreover, a new kind of result has been inserted, $\top^\sharp$. This particular result is the *top* element of the abstract domain A, so that $\forall a \in A.\ a \sqsubseteq \top^\sharp$. It has been inserted among results in order to represent unknown results, as well as non-terminating computations. In fact, since the language allows recursion, we need a way to truncate loops, so that the analysis process can be carried out in a finite number of steps. Finally, there is a new kind of term, $\llbracket t^\sharp \rrbracket^{L,S}$. Terms in this form have a set of locks (the *lock environment*), denoted by L, and a multiset of method labels, denoted by S, attached to them. The set L will be used to know which locks are held by the term, while the multiset S will register method calls, and will be used to find recursive calls during the execution of methods.

To formalize the abstract interpretation we define *abstraction functions*, α, and concretization functions, γ, between the concrete and abstract domains. In particular we define an abstract function for each syntactic category, thus we define $\alpha_r : u \to u^\sharp$, $\alpha_d : d \to d^\sharp$, and so on. The definition of the abstraction functions is given in Table 4.9.

$$
\begin{array}{lcl}
\alpha_r(x) & = & x \\
\alpha_r(p) & = & p \\
\alpha_r(n) & = & \odot \\
 & & \\
\alpha_d([\ell_i = \varsigma(x_i)t_i \ ^{i\in 1\ldots n}]^l) & = & [\ell_i = \varsigma(x_i)\alpha(t_i) \ ^{i\in 1\ldots n}]^l \\
 & & \\
\alpha_t(u) & = & \alpha_r(u) \\
\alpha_t(\nu p.t) & = & \nu p.\alpha_t(t) \\
\alpha_t(p \mapsto d) & = & p \mapsto \alpha_d(d) \\
\alpha_t(u.\ell) & = & \alpha_r(u).\ell \\
\alpha_t(u_1.\ell \Leftarrow \varsigma(x)u_2) & = & \alpha_r(u_1).\ell \Leftarrow \varsigma(x)\alpha_r(u_2) \\
\alpha_t(\text{lock } u \text{ in } t) & = & \text{lock } \alpha_r(u) \text{ in } \alpha_t(t) \\
\alpha_t(\text{locked } p \text{ in } t) & = & \text{locked } p \text{ in } \alpha_t(t) \\
\alpha_t(\text{let } x = s \text{ in } t) & = & \text{let } x = \alpha_t(s) \text{ in } \alpha_t(t) \\
\alpha_t(s \Rsh t) & = & \alpha_t(s) \Rsh \alpha_t(t) \\
\alpha_t(\text{if } r \text{ then } s \text{ else } t) & = & \text{if } \alpha_t(r) \text{ then } \alpha_t(s) \text{ else } \alpha_t(t) \\
 & & \\
\alpha(t) & = & \llbracket \alpha_t(t) \rrbracket^{\emptyset,\emptyset}
\end{array}
$$

Table 4.9: Abstraction functions

There are two abstraction functions for terms: α_t and α. The first one collapses all the integer values to the new abstract value $\odot$, and is used to abstract all the subterms of the initial program. The second one, instead, attaches to the initial term (the program to be analyzed) an empty lock enviroment and an empty multiset of method calls, and then calls α_t. The initial lock enviroment and multiset of method calls are empty because we assume (as it is reasonable) that the initial term does not already hold any lock (i.e. there is no subterm of the form: locked p in t), and starts with no already executed method calls.

The concretization functions, γ, are defined in Table 4.10. Note that γ functions produce sets of concrete syntactic objects, since the concrete domain is a powerset. The concretization of an abstract term simply discards the lock environments and the multisets of method calls. Moreover, each abstract term $\odot$ is replaced by the set of all integer values.

The abstract semantics is given, analogously to the concrete one, by means

$\gamma_r(x)$	$=$	$\{x\}$
$\gamma_r(p)$	$=$	$\{p\}$
$\gamma_r(\odot)$	$=$	$\{n \mid n \text{ is an integer number}\}$
$\gamma_r(\top^\sharp)$	$=$	C the set of all concrete terms
$\gamma_d([\ell_i = \varsigma(x_i)t_i^\sharp \;{}^{i\in 1\ldots n}]^l)$	$=$	$\{[\ell_i = \varsigma(x_i)t_i \;{}^{i\in 1\ldots n}]^l \mid t_i \in \gamma(t_i^\sharp)\}$
$\gamma_t(u^\sharp)$	$=$	$\gamma_r(u^\sharp)$
$\gamma_t(\nu p.t^\sharp)$	$=$	$\{\nu p.t \mid t \in \gamma(t^\sharp)\}$
$\gamma_t(p \mapsto d^\sharp)$	$=$	$\{p \mapsto d \mid d \in \gamma_d(d^\sharp)\}$
$\gamma_t(u^\sharp.\ell)$	$=$	$\{u.\ell \mid u \in \gamma(u^\sharp)\}$
$\gamma_t(u_1^\sharp.\ell \Leftarrow \varsigma(x)u_2^\sharp)$	$=$	$\{u_1.\ell \Leftarrow \varsigma(x)u_2 \mid u_1 \in \gamma(u_1^\sharp), u_2 \in \gamma(u_2^\sharp)\}$
$\gamma_t(\text{lock } u^\sharp \text{ in } t^\sharp)$	$=$	$\{\text{lock } u \text{ in } t \mid u \in \gamma(u^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(\text{locked } p \text{ in } t^\sharp)$	$=$	$\{\text{locked } p \text{ in } t \mid t \in \gamma(t^\sharp)\}$
$\gamma_t(\text{let } x = s^\sharp \text{ in } t^\sharp)$	$=$	$\{\text{let } x = s \text{ in } t \mid s \in \gamma(s^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(s^\sharp \upharpoonright t^\sharp)$	$=$	$\{s \upharpoonright t \mid s \in \gamma(s^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(\text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp)$	$=$	$\{\text{if } r \text{ then } s \text{ else } t \mid r \in \gamma(r^\sharp), s \in \gamma(s^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t([\![t^\sharp]\!]^{L,S})$	$=$	$\gamma(t^\sharp)$
$\gamma(t^\sharp)$	$=$	$\gamma_t(t^\sharp)$

Table 4.10: Concretization functions

of evaluation contexts, structural congruence and reduction rules. The evaluation contexts are the same as the ones presented in Table 4.5, opportunely modified to take into consideration the new syntactic categories of abstract terms instead of the concrete ones.

For the congruence rules, we have the same rules presented in Table 4.6, slightly modified to take into account the new abstract syntactic categories. Moreover, we need to take into consideration the new kind of abstract terms, containing the lock environment and the multiset of method calls. The new congruence rules are shown in Table 4.11. Note the new rules which distribute the lock environments to subterms. All rules are quite straightforward, except for the restriction and the parallel. The rule for restriction performs a renaming with a fresh name, in order to avoid capture in the lock environment or in the mulltiset of method calls names. Finally, according to the rule for parallel, the set of locks held by a parallel term may be propagated only to one of the subterms. This because otherwise we would end with two parallel terms holding the same locks.

The abstract reduction rules are largely analogous to the ones defined in Table 4.7, the only conceptual differences being the treatment of the lock environments and multisets of method calls. They are given in Table 4.12.

The first two rules deal with method invocation. We used, in these rules, a

$s^\sharp \Rsh \mathcal{E}[t^\sharp]$	$\equiv$	$\mathcal{E}[s^\sharp \Rsh t^\sharp]$	if $fn(s^\sharp) \cap bn(\mathcal{E}) = \emptyset$
$(\nu p)\mathcal{E}[s^\sharp]$	$\equiv$	$\mathcal{E}[(\nu p)s^\sharp]$	if $p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})$
$[\![\text{locked } p \text{ in } t^\sharp]\!]^{L,S}$	$\equiv$	$\text{locked } p \text{ in } [\![t^\sharp]\!]^{L,S}$	
$[\![s^\sharp \Rsh t^\sharp]\!]^{L,S}$	$\equiv$	$[\![s^\sharp]\!]^{\emptyset,S} \Rsh [\![t^\sharp]\!]^{L,S}$	
$[\![\text{let } x = s^\sharp \text{ in } t^\sharp]\!]^{L,S}$	$\equiv$	$\text{let } x = [\![s^\sharp]\!]^{L,S} \text{ in } [\![t^\sharp]\!]^{L,S}$	
$[\![\text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp]\!]^{L,S}$	$\equiv$	$\text{if } [\![r^\sharp]\!]^{L,S} \text{ then } [\![s^\sharp]\!]^{L,S} \text{ else } [\![t^\sharp]\!]^{L,S}$	
$[\![\nu p.t^\sharp]\!]^{L,S}$	$\equiv$	$\nu p'.[\![t^\sharp\{\{p \leftarrow p'\}\}]\!]^{L,S}$	p' fresh

Table 4.11: New abstract structural congruence rules

function $occ(p.\ell_j, S)$, which is assumed to return the number of occurrences of the method call $p.\ell_j$ inside the multiset S. We omit its definition, since it is straightforward. Methods are replaced with their bodies, and the usual substitutions are performed. If methods are recursive, this expansion is done until the *second* recursive call (then the third call from the beginning of the method) is performed. The first call of the method is regularly expanded, as well as the first recursive call. This can be seen from rule (Red invoke$^{\sharp 1}$). In fact, this rule expands method calls with bodies of methods, and it is executed until the number of occurrences of the same call is less than two. Since we add an element to the multiset S each time we expand a method call, we have that $occ(p.\ell_j, S) = 2$ when the method is called recursively twice. When this happens (and since all execution paths are examined, this happens for all recursive methods), we are forced to execute rule (Red invoke$^{\sharp 2}$). This rule simply replaces the method call with the new abstract result $\top^\sharp$, truncating the sequence of recursive calls. To understand clearly why two recursive calls are needed, we can apply the following reasoning. A recursive method returns when the sequence of its recursive calls (which may also be empty) reaches the base cases of the recursive definition. The results of the base cases may be returned as they are, or may be further manipulated by the method, before returning to the caller. In the first case, we need only the initial method call to execute all the base cases, and all recursive calls could be directly replaced by $\top^\sharp$. When, instead, the recursive results $r_1, \ldots, r_n$ are further manipulated by the method to produce the final result r_f, we need to know what kind of results $r_1, \ldots, r_n$ we may receive from the recursion, in order to produce correctly the kind of the final result r_f. Then the results from the base cases are passed to recursive calls, so that we can produce the results of the recursive cases. Then we allow the expansion of two successive calls of the same method: the first as usual, and the second as a first recursive call. In this way we are able to examine every path of a recursive definition. This is also true because of the two rules for the *if* term. We point out in fact that, differently from the concrete rules, the abstract reduction of an if term produces non-deterministically two different results, so that all execution paths may be correctly examined. Finally, note that to examine all possible execution paths in the abstract calculus, we will use the evaluation contexts $\mathcal{E} \Rsh b$ and $a \Rsh \mathcal{E}$ in order to analyze *all possible interleavings* of

$$\frac{d^\sharp = [\ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n) \quad occ(p.\ell_j, S) \leq 1}{p \mapsto d^\sharp \upharpoonleft [\![p.\ell_j]\!]^{L,S} \longrightarrow^\sharp p \mapsto d^\sharp \upharpoonleft [\![t_j^\sharp\{\{x_j \leftarrow p\}\}]\!]^{L,S\uplus\{p.\ell_j\}}} \;\text{(Red invoke}^{\sharp 1}\text{)}$$

$$\frac{d^\sharp = [\ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n) \quad occ(p.\ell_j, S) > 1}{p \mapsto d^\sharp \upharpoonleft [\![p.\ell_j]\!]^{L,S} \longrightarrow^\sharp p \mapsto d^\sharp \upharpoonleft \top^\sharp} \;\text{(Red invoke}^{\sharp 2}\text{)}$$

$$\frac{\begin{array}{c} d^\sharp = [\ell_j = \varsigma(x)u_j^\sharp, \ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)-\{j\}}]^l \\ d^{\sharp\prime} = [\ell_j = \varsigma(x)u^\sharp, \ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)-\{j\}}]^l \end{array}}{p \mapsto d^\sharp \upharpoonleft [\![p.\ell_j \Leftarrow \varsigma(x)u^\sharp]\!]^{L,S} \longrightarrow p \mapsto d^{\sharp\prime} \upharpoonleft p} \;\text{(Red update}^\sharp\text{)}$$

$$\frac{d^\sharp = [\ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)}]^\circ \quad d^{\sharp\prime} = [\ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)}]^\bullet}{p \mapsto d^\sharp \upharpoonleft [\![\text{lock } p \text{ in } t^\sharp]\!]^{L,S} \longrightarrow p \mapsto d^{\sharp\prime} \upharpoonleft [\![\text{locked } p \text{ in } t^\sharp]\!]^{L\cup\{p\},S}} \;\text{(Red lock}^\sharp\text{)}$$

$$\frac{d^\sharp = [\ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)}]^\bullet \quad d^{\sharp\prime} = [\ell_i = \varsigma(x_i)t_i^\sharp \;\; ^{i\in(1\ldots n)}]^\circ}{p \mapsto d \upharpoonleft \text{locked } p \text{ in } u^\sharp \longrightarrow p \mapsto d^{\sharp\prime} \upharpoonleft u^\sharp} \;\text{(Red unlock}^\sharp\text{)}$$

$$\frac{}{\text{let } x = u^\sharp \text{ in } [\![t]\!]^{L,S} \longrightarrow [\![t\{\{x \leftarrow u^\sharp\}\}]\!]^{L,S}} \;\text{(Red let}^\sharp\text{)}$$

$$\frac{}{\text{if } \odot \text{ then } s^\sharp \text{ else } t^\sharp \longrightarrow^\sharp s^\sharp} \;\text{(Red if}^{\sharp 1}\text{)}$$

$$\frac{}{\text{if } \odot \text{ then } s^\sharp \text{ else } t^\sharp \longrightarrow^\sharp t^\sharp} \;\text{(Red if}^{\sharp 2}\text{)}$$

$$\frac{s^\sharp \longrightarrow t^\sharp}{\mathcal{E}[s^\sharp] \longrightarrow \mathcal{E}[t^\sharp]} \;\text{(Red context}^\sharp\text{)}$$

$$\frac{}{[\![u^\sharp]\!]^{L,S} \longrightarrow u^\sharp} \;\text{(Red res}^\sharp\text{)}$$

$$\frac{}{[\![p \mapsto d^\sharp]\!]^{L,S} \longrightarrow p \mapsto d^\sharp} \;\text{(Red den}^\sharp\text{)}$$

$$\frac{}{\mathcal{E}[\top^\sharp] \longrightarrow \top^\sharp} \;\text{(Red }\top^\sharp\text{)}$$

Table 4.12: New abstract reduction rules

parallel threads. The other rules are quite similar to the ones presented in Table 4.7, so we will explain only the differences between the concrete and the abstract case for these rules. Rule (Red update$^\sharp$) simply discards the lock enviroment and the multiset of method calls. This because the update construct returns a result, and then all locks are released and results represent finished computations, so we are not interested in L or S. Rule (Red lock$^\sharp$) accumulates the newly acquired lock into the lock environment, while rule (Red unlock$^\sharp$) releases all locks ad done in the concrete case. Rules for let and contexts are analogous to the concrete cases. Finally, note the new rules for results, denotations, and $\top^\sharp$. The first two rules simply discard the lock environment and the method calls, because results or denotations are returned, and in those cases locks are released and we have a finished computation, so also the method calls are not considered. The rule for $\top^\sharp$ simply reduces each context where $\top^\sharp$ occurs to $\top^\sharp$. This is done in order to assure the termination of the static analysis of the program by truncating recursive method calls.

4.5 Correctness of the Abstract Interpretation

We can state the correctness of the abstract interpretation by the following results. First we have to show that α and γ form a Galois connection between the concrete and abstract domains. We recall that the concrete domain is the powerset of the set containing all concrete **aconcς**-calculus terms $\wp(C)$, with ordering relation $\subseteq$ between sets. Thus it is a lattice, with bottom element $\emptyset$ and top element C. The abstract domain is instead represented by the set A containing all abstract **aconcς**$^\sharp$-calculus terms. Its ordering relation is defined by the conditions in table 4.13.

$[\![t^\sharp]\!]^{L_1,S_1} \sqsubseteq [\![t^\sharp]\!]^{L_2,S_2}$	$\Leftrightarrow$	$L_1 \subseteq L_2 \wedge S_1 \subseteq S_2$
$t_1^\sharp \sqsubseteq t_2^\sharp$	$\Rightarrow$	$(\nu p.t_1^\sharp) \sqsubseteq (\nu p.t_2^\sharp)$
$t_1^\sharp \sqsubseteq t_2^\sharp$	$\Rightarrow$	$(\text{locked } p \text{ in } t_1^\sharp) \sqsubseteq (\text{locked } p \text{ in } t_2^\sharp)$
$(s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp)$	$\Rightarrow$	$(\text{let } x = s_1^\sharp \text{ in } t_1^\sharp) \sqsubseteq (\text{let } x = s_2^\sharp \text{ in } t_2^\sharp)$
$(s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp)$	$\Rightarrow$	$(s_1^\sharp \Rsh t_1^\sharp) \sqsubseteq (s_2^\sharp \Rsh t_2^\sharp)$
$(r_1^\sharp \sqsubseteq r_2^\sharp) \wedge (s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp)$	$\Rightarrow$	$(\text{if } r_1^\sharp \text{ then } s_1^\sharp \text{ else } t_1^\sharp) \sqsubseteq (\text{if } r_2^\sharp \text{ then } s_2^\sharp \text{ else } t_2^\sharp)$

Table 4.13: Abstract ordering relation

In practice, the elements of A are compared according to the lock environments and the multisets of method calls. When two abstract terms have recursively the same structure of subterms, we have that the more precise is the one which has smaller lock environments and multisets of method calls. Using this ordering relation, we have that the least upper bound operator $\sqcup$, given two terms which have the same structure, computes the *union* of the corresponding lock environments and multisets in the two terms, while the greatest lower bound operator, $\sqcap$, computes the intersection. All other couples of elements of A, are not comparable with each other (apart, obviously, reflexivity).

A is extended as a lattice by adding a top and a bottom element, which are respectively greater and lesser than all the other elements of the domain. We will denote the top and bottom elements of the abstract domain, respectively by $\top^\sharp$ and $\bot^\sharp$. We recall that the top element of the abstract domain is already used in the syntax of the **aconc**$\varsigma^\sharp$-calculus. So, if any two terms do not share a common structure as shown in Table 4.13, the two operators $\sqcup$ and $\sqcap$ return, respectively, the top and bottom elements of the abstract domain.

Proposition 4.5.1. *Let us consider two abstract terms $t_1^\sharp$ and $t_2^\sharp$ such that $t_1^\sharp \sqsubseteq t_2^\sharp$. If both $t_1^\sharp$ and $t_2^\sharp$ are different from $\bot^\sharp$ and $\top^\sharp$, we can conclude that $\gamma(t_1^\sharp) = \gamma(t_2^\sharp)$.*

Proof: The proof is done by induction on the rules of Table 4.13. For the base case we must consider the rule

$$[\![t^\sharp]\!]^{L_1,S_1} \sqsubseteq [\![t^\sharp]\!]^{L_2,S_2} \Leftrightarrow L_1 \subseteq L_2 \wedge S_1 \subseteq S_2$$

In this case we have:

$$\gamma(t_1^\sharp) = \gamma([\![t^\sharp]\!]^{L_1,S_1}) = \gamma_t([\![t^\sharp]\!]^{L_1,S_1}) = \gamma(t^\sharp)$$

$$\gamma(t_2^\sharp) = \gamma([\![t^\sharp]\!]^{L_2,S_2}) = \gamma_t([\![t^\sharp]\!]^{L_1,S_1}) = \gamma(t^\sharp)$$

and then $\gamma(t_1^\sharp) = \gamma(t_2^\sharp)$.

For the inductive case, we have to consider all the other rules in Table 4.13. Note that the premises of all these rules allow to use immediately the inductive hypothesis, so that each of these proofs is trivial. As an example, we can consider the proof for the rule regarding the *let* construct:

$$(s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp) \Rightarrow (\text{let } x = s_1^\sharp \text{ in } t_1^\sharp) \sqsubseteq (\text{let } x = s_2^\sharp \text{ in } t_2^\sharp)$$

In this case we can immediately apply the inductive hypothesis on $s_1^\sharp$ and $s_2^\sharp$ and on $t_1^\sharp$ and $t_2^\sharp$ obtaining that:

$$\gamma(s_1^\sharp) = \gamma(s_2^\sharp)$$
$$\gamma(t_1^\sharp) = \gamma(t_2^\sharp)$$

Now we can apply the concretization function on the right part of the rule obtaining:

$$\gamma(\text{let } x = s_1^\sharp \text{ in } t_1^\sharp) = \{\text{let } x = s \text{ in } t \mid s \in \gamma(s_1^\sharp), t \in \gamma(t_1^\sharp)\}$$

$$\gamma(\text{let } x = s_2^\sharp \text{ in } t_2^\sharp) = \{\text{let } x = s \text{ in } t \mid s \in \gamma(s_2^\sharp), t \in \gamma(t_2^\sharp)\}$$

and since $\gamma(s_1^\sharp) = \gamma(s_2^\sharp)$ and $\gamma(t_1^\sharp) = \gamma(t_2^\sharp)$ we can conclude that the proposition is true for the *let* term, that is:

$$\gamma(\text{let } x = s_1^\sharp \text{ in } t_1^\sharp) = \gamma(\text{let } x = s_2^\sharp \text{ in } t_2^\sharp)$$

□

Proposition 4.5.2. *Let $t^\sharp$ be an abstract term, and $S \in \wp(C)$ be a set of concrete terms. α and γ form a Galois connection between the two domains $\wp(C)$ and A. That is:*

- *α and γ are monotonic,*
- *$S \subseteq \gamma(\alpha(S))$, where α and γ are applied pointwise,*
- *$\alpha(\gamma(t^\sharp)) \sqsubseteq t^\sharp$.*

Proof: First of all, we have to address the abuse of notation used on the abstraction function α. In fact we have that $\alpha : C \mapsto A$ while the two domains are respectively $\wp(C)$ and A. Then the abstraction function defined on concrete terms should be considered as defined on *sets* of terms. The extension is straightforward, since given a set S of concrete terms, its abstraction is represented by the least upper bound on the domain A of the abstractions $\alpha(t_i)$ performed for each element t_i in S. Let us denote this extended abstraction function as α^{Set}:

$$\alpha^{Set}(S) = \bigsqcup_{t_i \in S} \alpha(t_i)$$

Now, all the proofs have to be done for the couple of functions α^{Set} and γ.

- **Monotonicity**: For the concretization function γ, we have that the premise of the monotonicity condition:

  $$t_1^\sharp \sqsubseteq t_2^\sharp \Rightarrow \gamma(t_1^\sharp) \subseteq \gamma(t_2^\sharp)$$

  is verified only for $t_1^\sharp = \bot^\sharp$ or $t_2^\sharp = \top^\sharp$, and for the cases treated in Table 4.13. We have already that $\gamma(\top^\sharp) = C$, so that if $t_2^\sharp = \top^\sharp$ we have obviously $\gamma(t_1^\sharp) \subseteq \gamma(t_2^\sharp) = \gamma(\top^\sharp) = C$ for all abstract terms $t_1^\sharp$. When $t_1^\sharp = \bot^\sharp$, it is sufficient to extend γ so that $\gamma(\bot^\sharp) = \emptyset$ in order to have $\gamma(t_1^\sharp) = \gamma(\bot^\sharp) = \emptyset \subseteq \gamma(t_2^\sharp)$ for all abstract terms $t_2^\sharp$. Finally, the result of proposition 4.5.1 covers the cases of Table 4.13.

  For the abstraction function, α^{Set}, since we have that $\alpha^{Set}(S) = \bigsqcup_{t_i \in S} \alpha(t_i)$, we can conclude that:

  $$\begin{array}{ll} & S_1 \subseteq S_2 \\ \Rightarrow & S_2 = S_1 \cup S_3 \\ \Rightarrow & \alpha^{Set}(S_1) \sqsubseteq \alpha^{Set}(S_1) \sqcup \alpha^{Set}(S_3) = \alpha^{Set}(S_1 \cup S_3) = \alpha^{Set}(S_2) \end{array}$$

- $\mathbf{S} \subseteq \gamma(\alpha^{\mathbf{Set}}(\mathbf{S}))$: let us split this proof in three cases:
  - $\alpha^{\mathbf{Set}}(\mathbf{S}) = \bot^\sharp$: this case is trivial, since $\alpha^{Set}(S)$ computes the least upper bound of the abstractions of all terms in S. If this upper bound results to be $\bot^\sharp$, then S must be empty. Then: $\gamma(\alpha^{Set}(S)) = \gamma(\bot^\sharp) = \emptyset$ and $S = \emptyset \subseteq \emptyset$.

  - $\alpha^{\mathbf{Set}}(\mathbf{S}) = \top^\sharp$: this case is trivial, since $\gamma(\alpha^{Set}(S)) = \gamma(\top^\sharp) = C$ and for all S in $\wp(C)$ we have that $S \subseteq C$.
  - $\bot^\sharp \sqsubset \alpha^{\mathbf{Set}}(\mathbf{S}) = \mathbf{t}^\sharp \sqsubset \top^\sharp$: in this case, consider again that the α^{Set} function computes the least upper bound of the abstractions of the elements of S. Now, the only cases where $t_1^\sharp \sqcup t_2^\sharp = t^\sharp \neq \top^\sharp$ are the ones following the definitions of Table 4.13. However, since $\alpha(t) = [\![\alpha_t(t)]\!]^{\emptyset,\emptyset}$ (note that both L and S are *always* empty as an abstraction result), we have that the first rule of Table 4.13:

$$[\![t^\sharp]\!]^{L_1,S_1} \sqsubseteq [\![t^\sharp]\!]^{L_2,S_2} \Leftrightarrow L_1 \subseteq L_2 \wedge S_1 \subseteq S_2$$

    is satisfied as an *equality*, since L_1, L_2, S_1 and S_2 are all empty. All the following rules are built inductively and this first rule is the basis of the induction, so we can conclude (with a proof analogous to the one presented for proposition 4.5.1) that:

$$\forall t_i, t_j \in S.\ \alpha(t_i) = \alpha(t_j) = t^\sharp$$

    Now, we have that the definition of γ is such that, for each concrete term t, $t \in \gamma(\alpha(t))$ (this comes directly from the definition of α and γ and can be verified straightforwardly). Then we have that:

$$\forall t_i \in S.\ t_i \in \gamma(\alpha(t_i)) = \gamma(t^\sharp)$$

    and thus we can conclude $S \subseteq \gamma(t^\sharp) = \gamma(\alpha^{Set}(S))$.

- $\alpha^{\mathbf{Set}}(\gamma(\mathbf{t}^\sharp)) \sqsubseteq \mathbf{t}^\sharp$: let us split this proof in three cases:
  - $\mathbf{t}^\sharp = \bot^\sharp$: this case is trivial, since we have that:

$$\alpha^{Set}(\gamma(t^\sharp)) = \alpha^{Set}(\gamma(\bot^\sharp)) = \alpha^{Set}(\emptyset) = \bot^\sharp \sqsubseteq \bot^\sharp = t^\sharp$$

  - $\mathbf{t}^\sharp = \top^\sharp$: this case is trivial, since we have that:

$$\alpha^{Set}(\gamma(t^\sharp)) = \alpha^{Set}(\gamma(\top^\sharp)) = \alpha^{Set}(C) = \top^\sharp \sqsubseteq \top^\sharp = t^\sharp$$

  - $(\mathbf{t}^\sharp \neq \top^\sharp) \wedge (\mathbf{t}^\sharp \neq \bot^\sharp)$: in this case, the cases of the definitions of α and γ are such that, given the abstract term $t^\sharp$, for all concrete terms $t \in \gamma(t^\sharp)$, we have $\alpha(t) \sqsubseteq t^\sharp$. This because the γ function discards all the lock environments and the multisets attached to terms, while the α_t function rebuilds empty lock environments and multisets of method calls, which are then distributed to subterms by the abstract structural congruence (considering α-convertion for the rule of restriction). Then the two abstract terms $\alpha(t)$ and $t^\sharp$ are in relation according to Table 4.13. Thus we can conclude that $\alpha^{Set}(\gamma(t^\sharp)) \sqsubseteq t^\sharp$.

$\square$

Proposition 4.5.3. *Let t_1 and t_2 be concrete terms, if $t_1 \longrightarrow^n t_2$ then there exists an abstract term $t^\sharp$ such that $\alpha(t_1) \longrightarrow^{\sharp m} t^\sharp$ and $t_2 \in \gamma(t^\sharp)$. $\longrightarrow$ and $\longrightarrow^\sharp$ include the congruence rule applications which make possible the reduction. $\longrightarrow^n$ denotes then a sequence of applications of n consecutive reduction rules and congruences.*

Proof: The proof is by induction on n. For the basis of the induction, $n = 1$, let us consider all the different cases, according to the possible couples of concrete statements t_1 and t_2 taken from the concrete reduction rules, and from the concrete congruence definition:

- $t_1 = p.\ell_j;\ t_2 = t_j\{\{x_j \leftarrow p\}\}$:

  This case comes from the concrete (Red invoke) rule. Here, we have

  $$\alpha(t_1) = [\![p.\ell_j]\!]^{\emptyset,\emptyset}$$

  Here we can use the (Red invoke$^{\sharp 1}$) rule to obtain

  $$t^\sharp = [\![\alpha_t(t_j)\{\{x_j \leftarrow p\}\}]\!]^{\emptyset,\{p.\ell_j\}}$$

  From this, by applying the concretization function we have

  $$\gamma(t^\sharp) = \{t \mid t \in \gamma(\alpha_t(t_j)\{\{x_j \leftarrow p\}\})\}$$

  Then we have clearly $t_2 \in \gamma(t^\sharp)$, directly from the fact that α and γ form a Galois connection.

- $t_1 = p.\ell_j \Leftarrow \varsigma(x)u;\ t_2 = p$:

  This case comes from the concrete (Red update) rule. Here

  $$\alpha(t_1) = [\![p.\ell_j \Leftarrow \varsigma(x)\alpha_r(u)]\!]^{\emptyset,\emptyset}$$

  and using the (Red update$^\sharp$) rule we obtain

  $$t^\sharp = p$$

  Using the concretization function we have

  $$\gamma(t^\sharp) = \{p\}$$

  and clearly $t_2 \in \{p\}$.

- $t_1 = \text{lock } p \text{ in } t;\ t_2 = \text{locked } p \text{ in } t$:

  This case comes from the concrete (Red lock) rule. Now

  $$\alpha(t_1) = [\![\text{lock } p \text{ in } \alpha_t(t)]\!]^{\emptyset,\emptyset}$$

  and using the (Red lock$^\sharp$) rule we have

  $$t^\sharp = [\![\text{locked } p \text{ in } \alpha_t(t)]\!]^{\{p\},\emptyset}$$

  Again, using the concretization function

  $$\gamma(t^\sharp) = \{\text{locked } p \text{ in } t \mid t \in \gamma(\alpha_t(t))\}$$

  and thanks to the Galois connection, $t_2 \in \gamma(t^\sharp)$.

- $t_1 = \text{locked } p \text{ in } u;\ t_2 = u$:

  This case comes from the concrete (Red unlock) rule. In this case, we have

  $$\alpha(c_1) = [\![\text{locked } p \text{ in } \alpha_r(u)]\!]^{\emptyset,\emptyset}$$

  Note that the lock environment is incorrect, in this case, but this never happens in real terms, since a real term should not start with some already locked names. However, this fact does not influence this proof. Here we can use a structural congruence step, the (Red res$^\sharp$) rule and the (Red unlock$^\sharp$) rule to obtain

  $$t^\sharp = \alpha_r(u)$$

  From this, by applying the concretization function we have

  $$\gamma(t^\sharp) = \gamma_r(\alpha_r(u))$$

  and again the fact that $t_2 \in \gamma(t^\sharp)$ comes directly from the Galois connection.

- $t_1 = \text{let } x = u \text{ in } t;\ t_2 = t\{\{x \leftarrow u\}\}$:

  This case comes from the concrete (Red let) rule. Here

  $$\alpha(t_1) = [\![\text{let } x = \alpha_r(u) \text{ in } \alpha_t(t)]\!]^{\emptyset,\emptyset}$$

  and using an abstract structural congruence step, the (Red res$^\sharp$) rule and the (Red let$^\sharp$) rule we obtain

  $$t^\sharp = [\![\alpha_t(t)\{\{x \leftarrow \alpha_r(u)\}\}]\!]^{\emptyset,\emptyset}$$

  Using the concretization function we have

  $$\gamma(t^\sharp) = \{t \mid t \in \gamma_t(\alpha_t(t)\{\{x \leftarrow \alpha_r(u)\}\})\}$$

  Again, the conclusion $t_2 \in \gamma(t^\sharp)$ follows from the Galois connection.

- $t_1 = \text{if } 0 \text{ then } s \text{ else } t;\ t_2 = t$:

  This case comes from the concrete (Red if0) rule. Now

  $$\alpha(t_1) = [\![\text{if } \odot \text{ then } \alpha_t(s) \text{ else } \alpha_t(t)]\!]^{\emptyset,\emptyset}$$

  and, after an abstract congruence step and the (Red res$^\sharp$) rule, we have two possible rules to apply: (Red if$^{\sharp 1}$) and (Red if$^\sharp$2). By choosing rule (Red if$^\sharp$1) we obtain

  $$t^\sharp = [\![\alpha_t(t)]\!]^{\emptyset,\emptyset}$$

  and using the concretization function

  $$\gamma(t^\sharp) = \gamma(\alpha_t(t))$$

  Again, since γ and α form a Galois connection, we obtain the conclusion $t_2 \in \gamma(\alpha(t))$. This case is exactly the same as the one for the other branch of the *if* construct, so we will not show that case.

- $t_1 = s \upharpoonright \mathcal{E}[t];\ t_2 = \mathcal{E}[s \upharpoonright t]$ where $fn(s) \cap bn(\mathcal{E}) = \emptyset$:

  This case comes from the first concrete congruence rule. Here

  $$\alpha(t_1) = [\![\alpha_t(s) \upharpoonright \alpha_t(\mathcal{E}[t])]\!]^{\emptyset,\emptyset} \equiv [\![\alpha_t(s)]\!]^{\emptyset,\emptyset} \upharpoonright [\![\alpha_t(\mathcal{E}[t])]\!]^{\emptyset,\emptyset}$$

  Now we have that $\mathcal{E}$ is a term with an hole, so that its abstraction produces an abstract term with the same hole (the abstraction function should be opportunely extended, but it is straightforward). This because the α_t function simply replaces each integer value with $\odot$. So we can write

  $$\alpha_t(\mathcal{E}[t]) = \alpha_t(\mathcal{E})[\alpha_t(t)]$$

  where $\alpha_t(\mathcal{E}) = \mathcal{C}$ is an abstract context identical to $\mathcal{E}$, with the only exception of integer values collapsed to $\odot$. Now we can use one of the abstract congruence rules of Table 4.11 to distribute the $[\![\ ]\!]^{\emptyset,\emptyset}$ structure inside the $\mathcal{C}$ context, obtaining

  $$\alpha(t_1) \equiv [\![\alpha_t(s)]\!]^{\emptyset,\emptyset} \upharpoonright \mathcal{C}'[[\![\alpha_t(t)]\!]^{\emptyset,\emptyset}]$$

  where $\mathcal{C}'$ is identical to $\mathcal{C}$, apart from the fact that its subterms are surrounded by $[\![\ ]\!]^{\emptyset,\emptyset}$. Then $\mathcal{C}'$ has the same names of $\mathcal{E}$, and $\alpha_t(s)$ has the same names of s, thus we can use the first abstract congruence rule from Table 4.11 to obtain

  $$\alpha(t_1) \equiv \mathcal{C}'[[\![\alpha_t(s)]\!]^{\emptyset,\emptyset} \upharpoonright [\![\alpha_t(t)]\!]^{\emptyset,\emptyset}] = t^\sharp$$

  Now, since the concretization function discards the lock environments and the multisets of method calls, we obtain

  $$\gamma(t^\sharp) = \{\mathcal{F}[x \upharpoonright y] \mid \mathcal{F} \in \gamma(\mathcal{C}') = \gamma(\mathcal{C}) = \gamma(\alpha_t(\mathcal{E})),\ x \in \gamma(\alpha_t(s)),\ y \in \gamma(\alpha_t(t))\}$$

  and $t_2 \in \gamma(t^\sharp)$ comes again from the Galois connection between α and γ. The other side of this congruence rule is very similar to this one, so we do not show it.

- $t_1 = (\nu p)\mathcal{E}[s];\ t_2 = \mathcal{E}[(\nu p)s]$ where $p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})$:

  This case comes from the second concrete congruence rule. In this case we can use the same reasoning as before, obtaining

  $$\alpha(t_1) = [\![(\nu p)\alpha_t(\mathcal{E}[s])]\!]^{\emptyset,\emptyset} = [\![(\nu p)\alpha_t(\mathcal{E})[\alpha_t(s)]]\!]^{\emptyset,\emptyset} \equiv (\nu p)[\![\alpha_t(\mathcal{E})[\alpha_t(s)]]\!]^{\emptyset,\emptyset}$$

  where this last equivalence was obtained by applying the abstract congruence rule for restriction. Note that, according to the rule, we should change the name p with a fresh name, but since there is no possibility of capture (both L and S are empty), we did not rename it. Now, using again the same reasoning as before, we can obtain

  $$\alpha(t_1) \equiv (\nu p)\mathcal{C}'[[\![\alpha_t(s)]\!]^{\emptyset,\emptyset}] \equiv \mathcal{C}'[(\nu p)[\![\alpha_t(s)]\!]^{\emptyset,\emptyset}] = t^\sharp$$

  where $\alpha_t(\mathcal{E}) = \mathcal{C}$ and $\mathcal{C}'$ is obtained from $\mathcal{C}$ by distributing the lock environment and the multiset of method calls on the subterms. The last equivalence was obtained by applying the abstract congruence rule for restriction. Now using the concretization function on $t^\sharp$ we obtain:

  $$\gamma(t^\sharp) = \{\mathcal{F}[(\nu p)x] \mid \mathcal{F} \in \gamma(\mathcal{C}') = \gamma(\mathcal{C}) = \gamma(\alpha_t(\mathcal{E})),\ x \in \gamma(\alpha_t(s))\}$$

  and again $t_2 \in \gamma(t^\sharp)$ comes from the Galois connection result.

We should continue with the inductive cases by supposing the proposition true for $\longrightarrow^{n-1}$. All the cases are identical to the ones shown before, with the only difference that we have a possible non-empty lock environment L and a possible non-empty multiset S of method calls. Since both the lock environment and the multiset of method calls are discarded by the concretization function, we have that also the proofs of the inductive cases for the above rules are exactly the same as before, and thus will be omitted.

The only cases which remain, for the inductive step, are the ones concerning the (Red context) concrete rule, as well as the (Red invoke) concrete rule, when we have a non-empty multiset of method calls (and then the (Red invoke$^{\sharp 2}$) rule may be triggered).

- $t_1 = \mathcal{E}[r];\ t_2 = \mathcal{E}[s]$ where $r \longrightarrow s$:

  This case comes from the (Red context) rule. Here, we have that the (Red context) rule allows to conclude $\mathcal{E}[r] \longrightarrow \mathcal{E}[s]$ for any concrete context, starting from the $r \longrightarrow s$ hypothesis. We can use the inductive hypothesis on $r \longrightarrow s$, to conclude that

  $$\alpha(r) \longrightarrow^{\sharp m} s^\sharp \quad , \quad s \in \gamma(s^\sharp)$$

  Now let us apply the abstraction function on $\mathcal{E}[r]$ obtaining (using the same reasoning as before)

  $$\alpha(\mathcal{E}[r]) = [\![\alpha_t(\mathcal{E}[r])]\!]^{\emptyset,\emptyset} = [\![\alpha_t(\mathcal{E})[\alpha_t(r)]]\!]^{\emptyset,\emptyset} = [\![\mathcal{C}[\alpha_t(r)]]\!]^{\emptyset,\emptyset} \equiv \mathcal{C}'[[\![\alpha_t(r)]\!]^{\emptyset,\emptyset}]$$

Now, since $\alpha(r) = [\![\alpha_t(r)]\!]^{\emptyset,\emptyset}$, we can apply the (Red context$^\sharp$) rule on our inductive hypothesis, obtaining

$$\alpha(\mathcal{E}[r]) \longrightarrow^\sharp \mathcal{C}'[s^\sharp] = t^\sharp \quad , \quad s \in \gamma(s^\sharp)$$

Finally, we can apply the γ function on $t^\sharp$ obtaining

$$\gamma(t^\sharp) = \{\mathcal{F}[x] \mid \mathcal{F} \in \gamma(\mathcal{C}') = \gamma(\mathcal{C}) = \gamma(\alpha_t(\mathcal{E})),\ x \in \gamma(s^\sharp)\}$$

and $t_2 = \mathcal{E}[s] \in \gamma(t^\sharp)$ comes from the inductive hypothesis ($s \in \gamma(s^\sharp)$) and from the Galois connection between α and γ ($\mathcal{E} \in \gamma(\alpha_t(\mathcal{E}))$).

- $t_1 = p.\ell_j;\ t_2 = t_j\{\{x_j \leftarrow p\}\}$:

  This case comes from the (Red invoke) rule. In particular we are interested in the case concerning the abstract rule (Red invoke$^{\sharp 2}$). Let us suppose, that from an initial statement r we reached after $n-1$ steps a statement s, having $r \longrightarrow^{n-1} s$. Then by inductive hypothesis, we have that there exists an abstract statement $s^\sharp$ such that $\alpha(r) \longrightarrow^{\sharp m} s^\sharp$ and $s \in \gamma(s^\sharp)$. Now we consider $s = t_1 = p.\ell_j$ and reduce it to $t_2 = t_j\{\{x_j \leftarrow p\}\}$. On the abstract side, we consider only the (Red invoke$^\sharp$2) rule since, as already said, the case corresponding to the (Red invoke$^\sharp$1) rule is identical as before. We have, from the inductive hypothesis, that $s \in \gamma_s(s^\sharp)$. We can have two cases for $s^\sharp$:

  - $\mathbf{s}^\sharp = \top^\sharp$: in this case the abstract computation remains blocked since, according to the (Red context$^\sharp$) rule, we have $\top^\sharp \longrightarrow^\sharp \top^\sharp$. Then this case is trivial, since we have $\alpha(r) \longrightarrow^{\sharp m} s^\sharp \longrightarrow \top^\sharp$ and since $t \in \gamma(\top^\sharp) = C$ for each concrete term t (then also for $t_2 = t_j\{\{x_j \leftarrow p\}\}$).
  - $\mathbf{s}^\sharp \neq \top^\sharp$: in this case, the fact that $s \in \gamma(s^\sharp)$ implies that $\{s\} \subseteq \gamma(s^\sharp)$. Thus, using the fact that α never returns $\bot^\sharp$, the monotonicity of α, the Galois connection between α and γ and the hypothesis $s^\sharp \neq \top^\sharp$ we obtain:

    $$\bot^\sharp \sqsubset \alpha^{Set}(\{s\}) \sqsubseteq \alpha^{Set}(\gamma(s^\sharp)) \sqsubseteq s^\sharp \sqsubset \top^\sharp$$

    This implies that $\alpha^{Set}(\{s\})$ and $s^\sharp$ are in relation according to Table 4.13, and then they differ only in their lock environments and multisets of method calls. So $s^\sharp = [\![p.\ell_j]\!]^{L,S}$ and by applying rule (Red invoke$^{\sharp 2}$) we obtain that

    $$[\![p.\ell_j]\!]^{L,S} \longrightarrow^\sharp \top^\sharp$$

    so that we can conclude that $t_2 \in \gamma(\top^\sharp)$ since $\gamma(\top^\sharp) = C$.

□

This proposition states that the abstract reduction correctly approximates the concrete one. That is every concrete computation has a corresponding abstract one.

Thus, if a property is verified for all the reductions of an abstract term $\alpha(t)$ then it is verified also for t.

In the following proposition we address the problem of termination for the abstract reduction process. Termination must be intended in the sense that the abstract reduction graph, whose nodes are represented by intermediate terms encountered during abstract reduction (which includes the abstract congruence), is finite. In fact, since the abstract calculus is non-deterministic, we can have different paths of reduction, given a single abstract term as our starting point. The following property tells that each path of reduction has a finite number of intermediate terms. Using this property it is possible to examine the abstract reduction graph in a finite number of steps, in order to establish properties of the concrete reductions, thanks to the correctness of the abstract interpretation shown above.

Proposition 4.5.4. *Given a concrete term t, the abstract interpretation process of $\alpha(t)$ always terminates.*

Proof: Trivial, since the only possibility of non-termination is given by recursive methods and they reduce to $\top^\sharp$ after the second recursive call. We treat all cases of recursion, both direct (a method which call itself) and indirect (methods which call each other in a circular way), since we look only at name of methods contained in the current *stack* of method calls, represented by the multiset S attached to terms.

$\square$

Finally, we give a brief proposition explaining the meaning and the correctness of lock environments attached to the abstract calculus terms. The proposition takes into consideration a concrete term t where we assume to have no already locked names. This can be done without loosing in generality, since if we want to consider a term s which already holds a lock on p, we can always replace it by a *lockpins* term, and start the examination without any already held lock.

Proposition 4.5.5. *Given a concrete term t, without any already locked name, consider its abstraction $\alpha(t) = [\![\alpha_t(t)]\!]^{\emptyset,\emptyset}$. The first set attached to each subterm during the reduction of $\alpha(t)$ (the* lock environment*) registers all locks on objects held while executing that subterm.*

Proof: The proof may be easily done using induction on the number of steps which compose the various reduction paths of the abstract term $\alpha(t)$. Let us consider a generic reduction path and denote the number of abstract steps in this path by n.

- The base case of the induction is represented by $n = 0$ and is trivial, since at the beginning we have $\alpha(t) = [\![\alpha_t(t)]\!]^{\emptyset,\emptyset}$, and since we suppose that we have no already locked names, the proposition is true.

- For the inductive case, let us consider a path composed by n steps of reduction and abstract structural congruence, and let us apply one more step of reduction or abstract structural congruence. So we have

$$\alpha(t) \longrightarrow^{\sharp n} t^\sharp$$

  and we can suppose, by inductive hypothesis, that for all subterms of $t^\sharp$ the lock environment registers all locks held (acquired and not yet released) during their reduction. By applying one more step of computation, we can have the following cases affecting the lock environments:

  – The lock environment of a subterm s is distributed on the subterms of s using the abstract structural congruence. In this case the proposition is trivially true, since no new locks are acquired and no locks are released, so each subterm of s must be executed under the same locks as s itself. The only exception of this reasoning is represented by the following abstract structural congruence rule:

$$[\![s^\sharp \upharpoonright t^\sharp]\!]^{L,S} \equiv [\![s^\sharp]\!]^{\emptyset,S} \upharpoonright [\![t^\sharp]\!]^{L,S}$$

    In this case, however, it is correct to have two different lock environments, since we cannot have two parallel terms holding the same locks (this would be incorrect, since there would be concurrent access to shared resources).

  – A reduction rule modifies the lock environment of a term. In this case we have that all modifications to the lock environments happen when a result or a denotation are obtained from a reduction (rules (Red invoke$^\sharp$2), (Red update$^\sharp$), (Red unlock$^\sharp$), (Red res$^\sharp$) and (Red den$^\sharp$)) and when some new locks are acquired (rule (Red lock$^\sharp$)). All these cases are trivially correct, since when we have a result or a denotation all the locks are released in the concrete semantics, and when we acquire a new lock using the (Red lock$^\sharp$) rule, we opportunely extend the lock environment of the term which is acquiring the lock.

□

4.6 Race checking analysis

In this section we formally define our notion of race-freeness.

As a consequence of the correctness of our abstract interpretation, we can check that every access to an object is done while holding the lock to that object. This analysis corresponds to the one in [27]. In this case it would be sufficient to check that when an object is used in a term t for a method call or a field modification, the name for that object is included in the lock environment attached to t.

However, we can apply a less rigid analysis to detect races. In particular we can check that, during the reduction of a term t, no *parallel* accesses to an object, referred by p, can be performed. That is a (sub)statement of the form $r \upharpoonright s$, where $r, s \in \{p.\ell_i\} \cup \{p.\ell_j \Leftarrow \varsigma(x)u\}$, is never reached.

This can be done by analyzing the abstraction of t: if in every possible reduction path of $\alpha(t)$, the abstraction of such a kind of statement is never introduced, we can conclude that it cannot be introduced in the concrete computation of t as well. Of course, given the approximation introduced by abstract interpretation, the viceversa does not hold in general.

Let us state the result formally with the following two definitions.

Definition 4.6.1 (Concrete race free term)**.** *A concrete term t is* race free *iff, for all terms s reached during its reduction, $t \longrightarrow^* s$, s does not contain a (sub)term of the form $p \mapsto d \upharpoonright \mathcal{E}'[r'] \upharpoonright \mathcal{E}''[r'']$, where $d = [\ell_i = \varsigma(x_i)t_i\ ^{i\in(1\ldots n)}]^l$ and $r', r'' \in \{p.\ell_i \mid i \in (1\ldots n)\} \cup \{p.\ell_j \Leftarrow \varsigma(x)u \mid j \in (1\ldots n)\}$.*

Definition 4.6.2 (Abstract race free term)**.** *An abstract term $t^\sharp$ is* race free *iff, for all terms $s^\sharp$ reachable in its reduction graph excluding $\top^\sharp$, $t^\sharp \longrightarrow^{\sharp *} s^\sharp \neq \top^\sharp$, $s^\sharp$ does not contain a (sub)term of the form $p \mapsto d \upharpoonright \mathcal{E}'[r^{\sharp\prime}] \upharpoonright \mathcal{E}''[r^{\sharp\prime\prime}]$, where $d = [\ell_i = \varsigma(x_i)t_i^\sharp\ ^{i\in(1\ldots n)}]^l$ and $r^{\sharp\prime}, r^{\sharp\prime\prime} \in \{[\![p.\ell_i]\!]^{L_1,S_1} \mid i \in (1\ldots n)\} \cup \{[\![p.\ell_j \Leftarrow \varsigma(x)u]\!]^{L_2,S_2} \mid j \in (1\ldots n)\}$.*

Note that the lock environments are not directly used in the definition of a race-free term, but they can be very useful in detecting which locks are held in each point of the program, and so may help to fix programs that present possible races, by showing which locks are already held by a term, and which ones should be acquired to make the term race free.

Note also that the abstract definition of race free term rules out the $\top^\sharp$ case. This because the abstract $\top^\sharp$ term is obtained after expanding method calls up to the first recursion (included), and so if we have a potential race during a reduction path leading to $\top^\sharp$, it certainly appears in the reduction graph before the $\top^\sharp$ term, since recursive calls are expanded once and since all possible paths are followed during the reduction. The $\top^\sharp$ term is used *only* to assure termination, but does not preclude the precision of the analysis. This consideration will be made more clear in the proof for the following proposition.

Proposition 4.6.1. *Given a term t, if $\alpha(t) = t^\sharp$ is race free, then t is race free.*

Proof: Since the abstract interpretation is correct, we have that all possible concrete executions are synthesized in the abstract reduction graph [21, 19].

If the abstract term t can not perform any recursive call then the property is trivially true. In fact, since we never apply the rule (Red invoke$^{\sharp 2}$), the abstract term can not reduce to $\top^\sharp$. Then we can apply the result of proposition 4.5.3 to say that for each reachable concrete term s we have that $s \in \gamma(s^\sharp)$ for some $s^\sharp$ in the abstract reduction graph. Then we can apply again the reasoning that:

$$\bot^\sharp \sqsubset \alpha(s) \sqsubseteq \alpha(\gamma(s^\sharp)) \sqsubseteq s^\sharp \sqsubset \top^\sharp$$

to say that $\alpha(s)$ and $s^\sharp$ differ only in their lock environments and multisets of method calls (since the ones of $\alpha(s)$ are all empty). Then, since those sets and multisets are not used in detecting the race-free property, we conclude that in all concrete computations, the initial term is race-free.

If, on the contrary, the abstract term can perform recursive calls, we surely have some abstract paths which reduce to $\top^\sharp$. In fact, since we explore all possible paths (see the rules for the *if* and parallel terms), and since, according to rule (Red invoke$^{\sharp 1}$), each method call is replaced by the corresponding body, we have that we surely will reach again the recursive call, and this process will repeat until we will be forced to use rule (Red invoke$^{\sharp 2}$) and reduce to $\top^\sharp$. Now, let us consider each recursive method. A recursive method may have multiple method calls in its body, and some of them (maybe all) will lead to a recursive behavior (note that recursion may also be indirect, so that method a calls method b which calls method a again). The recursive method may then have three main behaviors:

- There are no exit points (paths which do not end in a recursive call) during all the body of the method
- There are some exit points, and some of the method calls which originate the recursion are placed as first term of an *if* or *let* construct, so that we have some (sub)terms of the two following forms:

  $$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\text{if } [\![p.\ell]\!]^{L,S} \text{ then } s^{\sharp\prime} \text{ else } s^{\sharp\prime\prime}]$$

  $$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\text{let } x = [\![p.\ell]\!]^{L,S} \text{ in } s^{\sharp\prime}]$$

  while all the other calls are not placed as first term of an *if* or *let* construct.
- The method may fork, and each of the resulting terms falls in one of these three cases.

In the first case we have that the method can not terminate in the concrete case, since it has no exit points. This is represented in the abstract by the fact that all abstract reduction paths end in $\top^\sharp$. However, the recursive calls have been substituted by the corresponding method body twice, since in the first two calls we apply the rule (Red invoke$^{\sharp 1}$). Then, if a race happens in the concrete reduction after the second recursive call, we have that the same race must also happen before that call, since we always replace the call with the *same* method body (if the recursion is indirect the reasoning is the same since method bodies can be easily composed). This because in order to have a race we must have two parallel terms which access a shared resource simultaneously. Since the method body which replaces the method call is always the same, we have that if it introduces a term causing a race, it does that in each call, since all possible interleavings with all other parallel terms are examined. Then there is a concrete case of race before the second recursive call and then this case of race is present in the abstract reduction graph, since before the second recursive call we never apply rule (Red invoke$^\sharp$2) and since the abstract interpretation is correct.

In the second case, the exit point corresponding to the base cases of the recursion are reached during the first abstract call of the method (note that we are talking of the first call, not the first *recursive* call), since we follow all possible paths in the reduction of the body. For those recursive calls which are not placed as first term of an *if* or *let* construct, these results represent a possible termination of the recursive method, since the method body eventually falls in one of the following cases:

$$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\nu p.[\![p.\ell]\!]^{L,S}]$$

$$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\text{locked } p \text{ in } [\![p.\ell]\!]^{L,S}]$$
$$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\text{let } x = u^\sharp \text{ in } [\![p.\ell]\!]^{L,S}]$$
$$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\text{if } u^\sharp \text{ then } [\![p.\ell]\!]^{L,S} \text{ else } s^{\sharp\prime}]$$
$$p.\ell \longrightarrow^{\sharp *} s^\sharp = \mathcal{E}[\text{if } u^\sharp \text{ then } s^{\sharp\prime} \text{ else } [\![p.\ell]\!]^{L,S}]$$

Now, if there is a race condition during a path leading to an exit point after the second recursive call, then the *same* race condition happens also during the first non-recursive call, since the body of the method which replaces the call is always the same, and then, as an example, we would have

$$s^\sharp = \mathcal{E}[\text{let } x = u^\sharp \text{ in } [\![p.\ell]\!]^{L,S}] \longrightarrow^{\sharp *} \mathcal{E}'[\text{let } x = u^\sharp \text{ in } [\![p.\ell]\!]^{L',S'}]$$

For the recursive calls placed as first terms of *if* or *let* terms, we have that those calls, if they end, give a result useful for the following computation of the *if* or the *let* term. We are in the following cases:

$$p.\ell \longrightarrow^{*} s^\sharp = \mathcal{E}[\text{let } x = [\![p.\ell]\!]^{L,S} \text{ in } s^{\sharp\prime}]$$

$$p.\ell \longrightarrow^{*} s^\sharp = \mathcal{E}[\text{if } [\![p.\ell]\!]^{L,S} \text{ then } s^{\sharp\prime} \text{ else } s^{\sharp\prime\prime}]$$

Then, since the exit points are reached during the first non-recursive call, and since we explore all possible execution paths, we have that those terms will appear in the abstract graph once for each possible result of the recursive method. Then those terms will be executed, checking the absence of races also in $s^{\sharp\prime}$ and $s^{\sharp\prime\prime}$ in the above examples. Since we execute $s^{\sharp\prime}$ and $s^{\sharp\prime\prime}$, if there are race conditions after the second recursive call, this means again that the same race conditions appear also after the first recursive call for $s^{\sharp\prime}$ and $s^{\sharp\prime\prime}$, and after the first (non-recursive) call for all the terms between the first call to $p.\ell$ and $s^\sharp$.

Finally, in the last case, the recursive method will fork twice, since two calls are performed for each method and all paths are explored. Then we can apply an inductive reasoning for the two new terms to detect race conditions. Moreover, note that since we fork twice, we cover also the case where the new created threads may have race conditions with each other.

□

4.7 Examples

In this section we will give two examples of analysis. The first example shows the behavior of a non-recursive term, while the second one shows how recursion is treated, and then how the multisets of method calls attached to environments work.

As a first simple example, consider the following term t_1:

$$\begin{aligned} t_1 = \ & \nu p.p \mapsto [l = \varsigma(x)5]^{\circ} \Rsh \\ & \text{let } x = p.l \text{ in } \ \text{if } n \text{ then } p.l \text{ else } (\text{lock } p \text{ in } p.l \Rsh \ \text{lock } p \text{ in } p.l \Leftarrow \varsigma(x)6) \end{aligned}$$

The abstraction $\alpha(t_1)$ is analogous to t_1 apart from the substitution of $\odot$ for every integer value and from the attaching of an empty lock environment and an empty multiset of method calls to the whole term. The graph of all possible abstract reductions for $\alpha(t_1)$ is shown in Figure 4.1. For the sake of readability we removed the restriction νp and the reference term $p \mapsto [l = \varsigma(x)\odot]^{\circ}$ from the figure. Moreover, because it is clear from the context, we removed the word "locked" from the terms when a lock is acquired. The structural congruence rules are applied implicitly.

$$\begin{tikzcd}
& \llbracket \text{let } x = p.l \text{ in } \text{if } \odot \text{ then } p.l \text{ else } (\text{lock } p \text{ in } p.l \Rsh \text{ lock } p \text{ in } p.l \Leftarrow \varsigma(x)\odot) \rrbracket^{\emptyset,\emptyset} \arrow[d] & \\
& \text{let } x = \llbracket p.l \rrbracket^{\emptyset,\emptyset} \text{ in } \llbracket \text{ if } \odot \text{ then } p.l \text{ else } (\text{lock } p \text{ in } p.l \Rsh \text{ lock } p \text{ in } p.l \Leftarrow \varsigma(x)\odot) \rrbracket^{\emptyset,\emptyset} \arrow[d] & \\
& \llbracket \text{ if } \odot \text{ then } p.l \text{ else } (\text{lock } p \text{ in } p.l \Rsh \text{ lock } p \text{ in } p.l \Leftarrow \varsigma(x)\odot) \rrbracket^{\emptyset,\emptyset} \arrow[dl] \arrow[dr] & \\
\llbracket p.l \rrbracket^{\emptyset,\emptyset} \arrow[d] & & \llbracket \text{lock } p \text{ in } p.l \rrbracket^{\emptyset,\emptyset} \Rsh \llbracket \text{ lock } p \text{ in } p.l \Leftarrow \varsigma(x) \odot \rrbracket^{\emptyset,\emptyset} \arrow[dl] \arrow[d] \\
\odot & \llbracket p.l \rrbracket^{\{p\},\emptyset} \Rsh \llbracket \text{ lock } p \text{ in } p.l \Leftarrow \varsigma(x) \odot \rrbracket^{\emptyset,\emptyset} \arrow[d] & \llbracket \text{lock } p \text{ in } p.l \rrbracket^{\emptyset,\emptyset} \Rsh \llbracket p.l \Leftarrow \varsigma(x) \odot \rrbracket^{\{p\},\emptyset} \arrow[d] \\
& \odot \Rsh \llbracket p.l \Leftarrow \varsigma(x) \odot \rrbracket^{\{p\},\emptyset} \arrow[d] & \llbracket p.l \rrbracket^{\{p\},\emptyset} \Rsh p \arrow[d] \\
& \odot \Rsh p & \odot \Rsh p
\end{tikzcd}$$

Figure 4.1: Abstract reduction graph for $\alpha(t_1)$

It is easy to check that, in the abstract computation graph, all the terms are race free. Thus we can conclude that the concrete computation is race free as well.

Let us remark that the term t_1 is not certified by current type-based analyses, because the object referred by p is accessed, in two cases, without locking it. However such accesses are safe because they are performed without any other access composed in parallel.

Our second example is much more complex than the first one, so we will give just a sketch of its evolving to demonstrate how our analysis always terminates, since all recursions are blocked in correspondence of the second recursive call. The term that we consider is the following:

$$\begin{array}{lll} t_2 = & \nu p.p \mapsto [& \begin{array}{ll} \mathit{res} & = \varsigma(x)0 \\ \mathit{mul} & = \varsigma(x)\lambda y.\ \text{let } z = y * x.\mathit{res} \text{ in } x.\mathit{res} \Leftarrow z \\ \mathit{fatt} & = \varsigma(x)\lambda y, z. \\ & \quad \text{if } z \text{ then} \\ & \quad\quad \text{if } y \text{ then } 0 \\ & \quad\quad \text{else}((\text{lock } x \text{ in } x.\mathit{fatt}(y-1, 0)) \upharpoonleft \\ & \quad\quad\quad (\text{lock } x \text{ in } x.\mathit{mul}(y)) \upharpoonleft \\ & \quad\quad\quad 0) \\ & \quad \text{else let } w = x.\mathit{res} \Leftarrow 1 \text{ in } x.\mathit{fatt}(y, 0) \end{array} \\ &]^{\circ} \upharpoonleft p.\mathit{fatt}(3,1) & \end{array}$$

The object denoted by p has three methods: the first one, res, is simply an instance variable that will contains results of computations. The second method, mul, reads the value from res and multiplies it for its argument. The third method, $fatt$, allows to compute the factorial of its first argument, y, by creating y threads in parallel. Each of these threads receives as argument an integer between 1 and y and calls mul on the instance variable with this argument. In the syntax of the object we used the λ-notation, since we know how to codify λ-terms into object calculi. The whole term computes the factorial of 3, and stores it in the instance variable $p.res$.

The abstraction of the term t_2 is obtained by simply replacing all numbers by $\odot$ and by attaching an empty lock environment and an empty multiset of method calls to the resulting term. In the following we describe (part of) the evolution of the abstract term. As before, we removed unnecessary terms from the reductions.

The analysis starts with first step

$$[\![p.\mathit{fatt}(\odot, \odot)]\!]^{\emptyset,\emptyset} \tag{4.1}$$

and continues by substituting the body of the method $p.fatt$ as follows:

$$\begin{array}{c} [\![\text{if } \odot \text{ then if } \odot \text{ then } \odot \text{ else}((\text{lock } p \text{ in } p.\mathit{fatt}(\odot,\odot)) \upharpoonleft (\text{lock } p \text{ in } p.\mathit{mul}(\odot)) \upharpoonleft \odot) \\ \text{else let } w = p.\mathit{res} \Leftarrow \odot \text{ in } p.\mathit{fatt}(\odot,\odot)]\!]^{\emptyset,\{p.\mathit{fatt}\}} \end{array} \tag{4.2}$$

Now, in the abstract reduction graph we have two different ways to continue, according to the two branches of the *if* term. Taking the *then* branch leads to:

$$[\![\text{if } \odot \text{ then } \odot \text{ else}((\text{lock } p \text{ in } p.fatt(\odot,\odot)) \Lsh (\text{lock } x \text{ in } p.mul(\odot)) \Lsh \odot)]\!]^{\emptyset,\{p.fatt\}} \tag{4.3}$$

which can continue directly in $\odot$ (*then* branch) or evolve in:

$$[\![\text{lock } p \text{ in } p.fatt(\odot,\odot)]\!]^{\emptyset,\{p.fatt\}} \Lsh [\![\text{lock } p \text{ in } p.mul(\odot)]\!]^{\emptyset,\{p.fatt\}} \Lsh \odot$$

Now, according to which lock is acquired before on the object denoted by p, we can proceed with the evolution of the first or the second parallel term. By choosing the former, we acquire the lock and then return in the same situation as step 4.1, but with method call multiset equal to $\{p.fatt\}$. Then we follow the same steps from the beginning, ending eventually in $\odot$ or $\top^\sharp$ (when we arrive to the third recursive call, the term reduces to $\top^\sharp$). Locks are released thanks to the third parallel term above (simply a result), using the evaluation context $\mathcal{E}[\cdot] =$ locked p in $[\cdot]$, the congruence $\mathcal{E}[s \Lsh t] \equiv s \Lsh \mathcal{E}[t]$, and the rule (Red unlock$^\sharp$).

By choosing the latter term, instead, we follow the execution of method *p.mul* which eventually terminates returning p (since the object denoted by p is modified at the end of *p.mul*). Then we have to release the lock and continue with the former parallel term, which as already said, eventually ends in $\top^\sharp$.

Taking the *else* branch of the *if* term in 4.2 leads to the execution of the term:

$$[\![\text{let } w = p.res \Leftarrow \odot \text{ in } p.fatt(\odot,\odot)]\!]^{\emptyset,\{p.fatt\}} \tag{4.4}$$

which executes the first part of the let construct (an initialization in the concrete term) and then reduces to 4.1 with method call multiset equal to $\{p.fatt\}$. From there we can follow all the paths examined till here, and eventually end in $\odot$ or $\top^\sharp$, since the number of equal calls in the method call multisets is increasing.

We have parallel terms in this term only during the path following the *else* branch of 4.3. In this case we never find two parallel terms which match the definition 4.6.2, so we can say that the term is race-free. Let us note again that this term is not certified as race-free looking only at the locks before the accesses to objects, since we have an unlocked access in 4.4, since those path is a single-thread part of the term.

Chapter 5

Abstract Interpretation against Unnecessary Locks

Abstract

This chapter is a revision of an article [6] where we present a use of abstract interpretation techniques for reducing synchronization overhead in a concurrent object calculus. The approach followed here is the same used in the rest of the thesis, but the goal can be seen as complementary to the one pursued in chapter 4. In fact those chapter was devoted to *safety* requirements, while this one is devoted to *efficiency*. We will first define a new calculus, **raconc**ς, which extends the **aconc**ς calculus presented in chapter 4, for supporting reentrant locks. Then we will use an abstract form of this extension to check when synchronization operations may be safely eliminated from statements. Thus our approach may be used to improve performance in object oriented languages by eliminating locks, without the risks caused by "manual" optimizations performed by programmers.

5.1 The Problem of Lock Overhead

We have seen in chapter 4 as many concurrent object oriented languages provide synchronization operations to avoid interferences among concurrent threads. The aim of such synchronization operations is to avoid erroneous computations caused by concurrent accesses and modifications to the same (shared) objects performed simultaneously by concurrent threads. Essentially, a synchronization operation puts a "lock" on an object, and no other concurrent process can access the object until the lock is released. In chapter 4 we saw a technique which can be exploited to analyze programs in order to check that no concurrent accesses to the same objects can be done.

There is obviously a price to pay for adding safety requirements to programs, usually translated into a computational overhead. Locks attached to objects present two main forms of computational overhead:

- The lock acquisition and release operations usually perform some checks on shared variables (to see if objects can freely be accessed) and set other variables to lock or unlock the access to objects. These operations are usually executed instantaneously, but if requested very often can nevertheless constitute a cause of poor performance of programs.

- The lock acquisition operation may force a thread to wait for other threads to release certain locks. Obviously, this is the major source of computational overhead and comes when a thread requires a lock that is already held by another thread. In fact, in this case, the thread which is requesting the lock cannot proceed in its execution until the lock is released.

Many works have been devoted to find useful static analyses which allow the elimination of unnecessary synchronization from concurrent programs [2, 11, 12, 17, 46, 51, 57]. The majority of these works concentrate on the Java programming language [34]. In this chapter, the generality of our approach based on object calculi will then become more evident, since, as already said, object calculi do not concentrate on peculiarities of languages, but on the main ideas behind the object-oriented paradigm, which are common among the totality of object oriented languages.

The above works investigate on the possible elimination of three categories of locks, described hereafter:

- **Reentrant Locks:** This is the most simple type of unnecessary synchronization operation, and happens when a process acquires the same lock (on the same object) more than once in a nested way. Nowadays reentrant locks are provided by many object-oriented programming languages (i.e. Java), in order to avoid possible cases of deadlock, and may be a significant source of overhead. In fact, many times programmers reuse code without knowing its details, and then it is not unusual to require a lock on an object before the call of a (reused) subroutine which locks that object again. The problem of this kind of lock has been analyzed in [2] for the case of the Java language, with a set of static analyses performed on the code of the program to examine.

- **Single Threaded Locks:** These locks are requested by only one thread at a time, and so are not necessary, since, assuming the program safe, there are no other parallel threads accessing the shared object at the same time as the thread requiring the lock. This kind of lock is definitely the most common among unnecessary locks, so that all the works above address this kind of problem. Again this kind of lock may come from the use of thread-safe libraries without any thread in concurrency with the ones generated by the libraries themselves.

- **Enclosed locks:** In this case we have a lock i which is requested always after another lock j has been acquired. It is clear that the lock i is not necessary, since lock j already forbids concurrency. This is a more subtle kind of unnecessary synchronization operation, and is analyzed only in [2] (we borrowed the name for this kind of lock from that paper).

The approach used in our analysis is based, as in chapter 4, on abstract interpretation techniques, as we saw that they are, in general, more precise than other approaches because they are based on the execution, on an approximated domain, of programs. For this kind of problem, we are more precise of the other kinds of analyses above since, for example, we are able to identify *real* single threaded locks. In fact all the works above consider a lock as single threaded when objects are locked by only one thread. This is obviously true, but a lock is single threaded when an object is accessed by only one thread *at a time*. Then if we have two threads which are not concurrent and access the same object, the above analyses do not classify the locks on this object as single threaded, because those analyses do not *execute* the program and then cannot say if the two threads are concurrent or not.

In order to apply abstract interpretation techniques to analyze the kind of locks above defined, we further extend the object calculus **aconc**ς presented in chapter 4 to support reentrant locks. In order to do this, we include the lock environments used in the abstraction of the **aconc**ς calculus presented in chapter 4 in the concrete semantics of the language. This because to formalize reentrant locks we must know, for each term, which locks are held in any moment during execution. We redefine the concrete semantics of the language and, using the same technique of abstract interpretation of chapter 4 we will analyze the concrete statements of the language. In particular, since the concrete syntax already includes the lock environments, the abstract interpretation approach will be used in this chapter only for its ability to run the program in an approximated way, but there will not be a *real* abstraction of concrete terms, apart for the modification of lock environments (from sets to multisets) and the truncation of recursive terms already seen in chapter 4. Finally, we will define formally the three kind of locks described above and check, for each lock of the concrete statements analyzed, if it fits in one of the above categories.

5.2 The object calculus raconcς

In this section we define the syntax and semantics of the object calculus **raconc**ς, derived from the **aconc**ς calculus presented and analyzed in chapter 4. The **raconc**ς calculus has been obtained as an extension to allow the use of *reentrant* locks. As said before, a lock is reentrant when a process may acquire it more than once in sequence. The extension, in practice, consists in adding to the syntax of the language the lock environments used in the abstraction of the **aconc**ς calculus presented in chapter 4. In this way we can know, for each term and for each program point, which locks are held and so we can formalize correctly the concept of reentrant lock.

5.2.1 Syntax

Table 5.1 defines the syntactic categories of results, denotations, terms and statements. Results, denotations and terms are the same as the ones defined for the **aconc**ς calculus, with the addition of lock environments $[\![t]\!]^L$ already seen in chapter 4 for the abstraction of the **aconc**ς calculus. As already said, using lock environments we can be aware of all the locks held by each (sub)term at any program point, so we can model correctly the concept of reentrant lock, which are acquired multiple times by the same thread (here is why we need lock environments) in a nested way.

u	$::=$		<u>results</u>
		x	variable
		p	location
		n	integer number
d	$::=$		<u>denotations</u>
		$[\ell_i = \varsigma(x_i)t_i \; ^{i \in 1 \ldots n}]^l$	object
l	$::=$		<u>lock states</u>
		$\circ$	unlocked
		$\bullet$	locked
r, s, t	$::=$		<u>terms</u>
		u	result
		$\nu p.t$	restriction
		$p \mapsto d$	reference
		$u.\ell$	method invocation
		$u.\ell \Leftarrow \varsigma(x)u$	field update
		lock u in t	lock acquisition
		locked p in t	lock acquired
		let $x = s$ in t	let
		$s \Rsh t$	parallel composition
		if r then s else t	if
		$[\![t]\!]^L$	lock environment
a	$::=$		<u>statements</u>
		$[\![t]\!]^{\emptyset}$	initial program
L	$=$	$\{p_1, \ldots, p_n\}$	<u>set of locations</u>

Table 5.1: Syntax of **raconc**ς

As already done in chapter 4, we assume that terms are used consistently. In particular. in addition to what already said in chapter 4, we forced the initial program to have an empty lock environment in the syntax of statements (this, as already said, can be done without loosing generality), and suppose that in the initial statement $[\![t]\!]^{\emptyset}$ the subterm t does not contain any lock environment.

5.2.2 Semantics of raconcς

The semantics of **raconc**ς is given as usual in terms of a structural congruence and a set of reduction rules. Structural congruence allows to syntactically transform statements in order to apply the reduction rules. The application of a reduction rule corresponds to a computational step.

As in the **aconc**ς calculus, both the structural congruence and reduction rules are given in terms of *evaluations contexts*. The syntax of evaluation contexts is given in Table 5.2, while the structural congruence rules are given in Table 5.3. They are identical to the ones defined for the **aconc**$\varsigma^{\sharp}$ calculus, apart for the absence of the multisets of method calls, so we refer to chapter 4 for a complete explanation.

$$\begin{array}{lll}
\mathcal{E} & ::= & \\
 & & [\cdot] \\
 & & \mathcal{E} \upharpoonright t \\
 & & s \upharpoonright \mathcal{E} \\
 & & \text{locked } p \text{ in } \mathcal{E} \\
 & & \text{let } x = \mathcal{E} \text{ in } t \\
 & & \text{if } \mathcal{E} \text{ then } s \text{ else } t \\
 & & \nu p.\mathcal{E}
\end{array}$$

Table 5.2: Reduction contexts

$s \upharpoonright \mathcal{E}[t]$	$\equiv$	$\mathcal{E}[s \upharpoonright t]$	if $fn(s) \cap bn(\mathcal{E}) = \emptyset$
$(\nu p)\mathcal{E}[s]$	$\equiv$	$\mathcal{E}[(\nu p)s]$	if $p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})$
$[\![\text{locked } p \text{ in } t]\!]^L$	$\equiv$	$\text{locked } p \text{ in } [\![t]\!]^L$	
$[\![s \upharpoonright t]\!]^L$	$\equiv$	$[\![s]\!]^{\emptyset} \upharpoonright [\![t]\!]^L$	
$[\![\text{let } x = s \text{ in } t]\!]^L$	$\equiv$	$\text{let } x = [\![s]\!]^L \text{ in } [\![t]\!]^L$	
$[\![\text{if } r \text{ then } s \text{ else } t]\!]^L$	$\equiv$	$\text{if } [\![r]\!]^L \text{ then } [\![s]\!]^L \text{ else } [\![t]\!]^L$	
$[\![\nu p.t]\!]^L$	$\equiv$	$\nu p'.[\![t\{\{p \leftarrow p'\}\}]\!]^L$	p' fresh

Table 5.3: Structural congruence rules

The reduction rules for the **raconc**ς calculus are given in Table 5.4. They are analogous to the ones presented in chapter 4 for the **aconc**$\varsigma^{\sharp}$ calculus, so we report the rules and explain only the modifications.

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n)}{p \mapsto d \upharpoonright [\![p.\ell_j]\!]^L \longrightarrow p \mapsto d \upharpoonright [\![t_j\{\{x_j \leftarrow p\}\}]\!]^L} \text{ (Red invoke)}$$

$$\frac{\begin{array}{c} d = [\ell_j = \varsigma(x)u_j, \ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)-\{j\}}]^l \\ d' = [\ell_j = \varsigma(x)u, \ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)-\{j\}}]^l \end{array}}{p \mapsto d \upharpoonright [\![p.\ell_j \Leftarrow \varsigma(x)u]\!]^L \longrightarrow p \mapsto d' \upharpoonright p} \text{ (Red update)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)}]^\circ \quad d' = [\ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)}]^\bullet}{p \mapsto d \upharpoonright [\![\text{lock } p \text{ in } t]\!]^L \longrightarrow p \mapsto d' \upharpoonright [\![\text{locked } p \text{ in } t]\!]^{L\cup\{p\}}} \text{ (Red lock 1)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)}]^\bullet \quad p \in L}{p \mapsto d \upharpoonright [\![\text{lock } p \text{ in } t]\!]^L \longrightarrow p \mapsto d \upharpoonright [\![\text{locked } p \text{ in } t]\!]^L} \text{ (Red lock 2)}$$

$$\frac{d = [\ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)}]^\bullet \quad d' = [\ell_i = \varsigma(x_i)t_i \; {}^{i\in(1\ldots n)}]^\circ}{p \mapsto d \upharpoonright \text{locked } p \text{ in } u \longrightarrow p \mapsto d' \upharpoonright u} \text{ (Red unlock)}$$

$$\frac{}{[\![\text{let } x = u \text{ in } t]\!]^L \longrightarrow [\![t\{\{x \leftarrow u\}\}]\!]^L} \text{ (Red let)}$$

$$\frac{}{\text{if } 0 \text{ then } s \text{ else } t \longrightarrow t} \text{ (Red if0)}$$

$$\frac{n \neq 0}{\text{if } n \text{ then } s \text{ else } t \longrightarrow t} \text{ (Red ifn)}$$

$$\frac{s \longrightarrow t}{\mathcal{E}[s] \longrightarrow \mathcal{E}[t]} \text{ (Red context)}$$

$$\frac{}{[\![u]\!]^L \longrightarrow u} \text{ (Red res)}$$

$$\frac{}{[\![p \mapsto d]\!]^L \longrightarrow p \mapsto d} \text{ (Red den)}$$

Table 5.4: Concrete Reduction rules

Differently from the case of **aconc**ς calculus, here we have two rules used to acquire a lock. (Red lock 1) acquires a lock on an unlocked object as before, while (Red lock 2) allows to reacquire a lock on an already locked object when the same lock is already owned (reentrant lock). In this rule it is possible to see why lock environments are needed in the concrete syntax. In fact, before acquiring a reentrant lock, we check if the name to be locked already appears inside the current lock environment.

5.3 Abstract interpretation of raconcς

In this section we present the definition of the abstract interpretation for the **raconc**ς calculus. The abstract interpretation is very similar to the one already presented in chapter 4 for the **aconc**ς calculus, and is given with respect to an abstract calculus which approximates the concrete one. In particular, this abstract interpretation modifies the lock environments in order to have multisets instead of simple sets of names. This because we want to identify multiple occurrences of the same name in lock environments in order to detect reentrant locks. Moreover, to force termination of the abstract reduction process, we attach to terms a multiset of method calls, as already done in chapter 4.

As it can be easily seen, this is not a *real* abstraction of the language, since the concrete semantics already has all we need to perform our analysis correctly. The abstract interpretation approach is used here essentially for its peculiarity of *running* the analyzed programs, even if in an approximated way. As already seen, this interpretation is such that given a statement in the abstract calculus, the set of possible statements which can be generated from it by abstract reduction and abstract structural congruence is finite. This allows to build a finite transition system, whose states are statements, which can be finitely analyzed to establish properties of it. We will use such analysis for lock optimizations.

As before, the concrete and abstract domains are represented, respectively be the two lattices $\langle \wp(C), \subseteq, C, \emptyset, \cup, \cap \rangle$ and $\langle A, \sqsubseteq, \top^\sharp, \bot^\sharp, \sqcup, \sqcap \rangle$. We recall that C is the domain containing all concrete statements of the **raconc**ς calculus, and that the concrete domain is represented by its powerset, with the usual ordering relation of containment between sets and with top and bottom elements, respectively, the whole set C and the empty set, $\emptyset$. The least upper bound operator is given by the union between sets, $\cup$, while the greatest lower bound is given by the intersection, $\cap$. The abstract domain, instead, is represented by the set A containing all abstract statements as they will be redefined in what follows. We will recall also the definition of the abstract ordering relation, $\sqsubseteq$. As usual, the top and bottom abstract elements will be represented respectively by $\top^\sharp$ and $\bot^\sharp$.

The syntax of the abstract object calculus **raconc**$\varsigma^\sharp$ is defined in Table 5.5. We recall the differences between the syntax of the concrete and the abstract calculus. All integer values are collapsed in the abstract calculus to the unique abstract value

$u^\sharp$	$::=$	$x \mid p \mid \odot \mid \top^\sharp$
$d^\sharp$	$::=$	$[\ell_i = \varsigma(x_i)t_i^\sharp \; {}^{i \in 1\ldots n}]^l$
l	$::=$	$\circ \mid \bullet$
$r^\sharp, s^\sharp, t^\sharp$	$::=$	$u^\sharp \mid p \mapsto d^\sharp \mid \nu p.t^\sharp \mid u^\sharp.\ell$ $u^\sharp.\ell \Leftarrow \varsigma(x)u^\sharp \mid \text{lock } u^\sharp \text{ in } t^\sharp \mid \text{locked } p \text{ in } t^\sharp$ $\text{let } x = s^\sharp \text{ in } t^\sharp \mid s^\sharp \upharpoonright t^\sharp \mid \text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp \mid [\![t^\sharp]\!]^{L,S}$
L	$=$	$\{p_1, \ldots, p_n\}$ multiset
S	$=$	$\{p_1.\ell_1, \ldots, p_m.\ell_m\}$ multiset

Table 5.5: Syntax of **raconc**$\varsigma^\sharp$

$\odot$. This is because the abstraction must be finitely analyzable. For the same reason, we have to truncate infinite computations and then a new abstract value $\top^\sharp$ is introduced in the abstract syntax, to represent unknown or non-terminating computations represented by recursions. Finally, in order to be able to find recursive calls, the syntax is extended with multisets containing method calls. These multisets are attached to terms, together with lock environments, now transformed from sets to multisets, in order to find cases of reentrant locks.

The *abstraction functions* α and *concretization functions* γ, between the concrete and abstract calculus are almost the same as the ones defined in chapter 4 for the **aconc**$\varsigma^\sharp$-calculus. They are reported in Tables 5.6 and 5.7 for further clarity.

We recall that γ functions produce sets of concrete syntactic objects, since the concrete domain is a powerset.

The abstract semantics is given, analogously to the concrete one, by means of structural congruence and reduction rules. Because of their similarity to the concrete case, we redefine only what differs. The rest is almost identical, apart from the fact that the concrete syntactic categories should be substituted by the abstract ones, that lock environments should be considered as multisets, instead of sets as in the concrete case, and that there are also the multisets of method calls attached to terms. These last multisets are handled exactly in the same way as shown in Table 4.12 for the **aconc**$\varsigma^\sharp$ calculus. Then, as done for the **aconc**$\varsigma^\sharp$-calculus in chapter 4, we have evaluation contexts in Table 5.2, structural congruences in Table 5.8, and new abstract reduction rules in Table 5.9.

Let us note again how method calls and *if* statements are dealt with. For method calls we recall that, thanks to the function $occ(p.\ell_j, S)$, we truncate recursive functions after two recursive calls. A detailed explanation of the reasoning behind such truncation may be found in chapter 4. Here we only say that such a truncation

$\alpha_r(x)$	$=$	x
$\alpha_r(p)$	$=$	p
$\alpha_r(n)$	$=$	$\odot$
$\alpha_d([\ell_i = \varsigma(x_i)t_i \ ^{i\in 1\ldots n}]^l)$	$=$	$[\ell_i = \varsigma(x_i)\alpha(t_i) \ ^{i\in 1\ldots n}]^l$
$\alpha_t(u)$	$=$	$\alpha_r(u)$
$\alpha_t(\nu p.t)$	$=$	$\nu p.\alpha_t(t)$
$\alpha_t(p \mapsto d)$	$=$	$p \mapsto \alpha_d(d)$
$\alpha_t(u.\ell)$	$=$	$\alpha_r(u).\ell$
$\alpha_t(u_1.\ell \Leftarrow \varsigma(x)u_2)$	$=$	$\alpha_r(u_1).\ell \Leftarrow \varsigma(x)\alpha_r(u_2)$
$\alpha_t(\text{lock } u \text{ in } t)$	$=$	$\text{lock } \alpha_r(u) \text{ in } \alpha_t(t)$
$\alpha_t(\text{locked } p \text{ in } t)$	$=$	$\text{locked } p \text{ in } \alpha_t(t)$
$\alpha_t(\text{let } x = s \text{ in } t)$	$=$	$\text{let } x = \alpha_t(s) \text{ in } \alpha_t(t)$
$\alpha_t(s \Rsh t)$	$=$	$\alpha_t(s) \Rsh \alpha_t(t)$
$\alpha_t(\text{if } r \text{ then } s \text{ else } t)$	$=$	$\text{if } \alpha_t(r) \text{ then } \alpha_t(s) \text{ else } \alpha_t(t)$
$\alpha(t)$	$=$	$\llbracket \alpha_t(t) \rrbracket^{\emptyset,\emptyset}$

Table 5.6: Abstraction functions

$\gamma_r(x)$	$=$	$\{x\}$
$\gamma_r(p)$	$=$	$\{p\}$
$\gamma_r(\odot)$	$=$	$\{n \mid n \text{ is an integer number}\}$
$\gamma_r(\top^\sharp)$	$=$	C the set of all concrete terms
$\gamma_d([\ell_i = \varsigma(x_i)t_i^\sharp \ ^{i\in 1\ldots n}]^l)$	$=$	$\{[\ell_i = \varsigma(x_i)t_i \ ^{i\in 1\ldots n}]^l \mid t_i \in \gamma(t_i^\sharp)\}$
$\gamma_t(u^\sharp)$	$=$	$\gamma_r(u^\sharp)$
$\gamma_t(\nu p.t^\sharp)$	$=$	$\{\nu p.t \mid t \in \gamma(t^\sharp)\}$
$\gamma_t(p \mapsto d^\sharp)$	$=$	$\{p \mapsto d \mid d \in \gamma_d(d^\sharp)\}$
$\gamma_t(u^\sharp.\ell)$	$=$	$\{u.\ell \mid u \in \gamma(u^\sharp)\}$
$\gamma_t(u_1^\sharp.\ell \Leftarrow \varsigma(x)u_2^\sharp)$	$=$	$\{u_1.\ell \Leftarrow \varsigma(x)u_2 \mid u_1 \in \gamma(u_1^\sharp), u_2 \in \gamma(u_2^\sharp)\}$
$\gamma_t(\text{lock } u^\sharp \text{ in } t^\sharp)$	$=$	$\{\text{lock } u \text{ in } t \mid u \in \gamma(u^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(\text{locked } p \text{ in } t^\sharp)$	$=$	$\{\text{locked } p \text{ in } t \mid t \in \gamma(t^\sharp)\}$
$\gamma_t(\text{let } x = s^\sharp \text{ in } t^\sharp)$	$=$	$\{\text{let } x = s \text{ in } t \mid s \in \gamma(s^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(s^\sharp \Rsh t^\sharp)$	$=$	$\{s \Rsh t \mid s \in \gamma(s^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(\text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp)$	$=$	$\{\text{if } r \text{ then } s \text{ else } t \mid r \in \gamma(r^\sharp), s \in \gamma(s^\sharp), t \in \gamma(t^\sharp)\}$
$\gamma_t(\llbracket t^\sharp \rrbracket^{L,S})$	$=$	$\llbracket \gamma(t^\sharp) \rrbracket^L$
$\gamma(t^\sharp)$	$=$	$\gamma_t(t^\sharp)$

Table 5.7: Concretization functions

$s^\sharp \Rsh \mathcal{E}[t^\sharp]$	$\equiv$	$\mathcal{E}[s^\sharp \Rsh t^\sharp]$	if $fn(s^\sharp) \cap bn(\mathcal{E}) = \emptyset$
$(\nu p)\mathcal{E}[s^\sharp]$	$\equiv$	$\mathcal{E}[(\nu p)s^\sharp]$	if $p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})$
$[\![\text{locked } p \text{ in } t^\sharp]\!]^{L,S}$	$\equiv$	$\text{locked } p \text{ in } [\![t^\sharp]\!]^{L,S}$	
$[\![s^\sharp \Rsh t^\sharp]\!]^{L,S}$	$\equiv$	$[\![s^\sharp]\!]^{\emptyset,S} \Rsh [\![t^\sharp]\!]^{L,S}$	
$[\![\text{let } x = s^\sharp \text{ in } t^\sharp]\!]^{L,S}$	$\equiv$	$\text{let } x = [\![s^\sharp]\!]^{L,S} \text{ in } [\![t^\sharp]\!]^{L,S}$	
$[\![\text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp]\!]^{L,S}$	$\equiv$	$\text{if } [\![r^\sharp]\!]^{L,S} \text{ then } [\![s^\sharp]\!]^{L,S} \text{ else } [\![t^\sharp]\!]^{L,S}$	
$[\![\nu p.t^\sharp]\!]^{L,S}$	$\equiv$	$\nu p'.[\![t^\sharp\{\{p \leftarrow p'\}\}]\!]^{L,S}$	p' fresh

Table 5.8: New abstract structural congruence rules

$$\frac{d^\sharp = [\ell_i = \varsigma(x_i)t_i^\sharp{}^{\;i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n) \quad occ(p.\ell_j, S) \leq 1}{p \mapsto d^\sharp \Rsh [\![p.\ell_j]\!]^{L,S} \longrightarrow^\sharp p \mapsto d^\sharp \Rsh [\![t_j^\sharp\{\{x_j \leftarrow p\}\}]\!]^{L,S\uplus\{p.\ell_j\}}} \quad (\text{Red invoke}^{\sharp 1})$$

$$\frac{d^\sharp = [\ell_i = \varsigma(x_i)t_i^\sharp{}^{\;i\in(1\ldots n)}]^l \quad j \in (1,\ldots,n) \quad occ(p.\ell_j, S) > 1}{p \mapsto d^\sharp \Rsh [\![p.\ell_j]\!]^{L,S} \longrightarrow^\sharp p \mapsto d^\sharp \Rsh \top^\sharp} \quad (\text{Red invoke}^{\sharp 2})$$

$$\frac{}{\text{if } \odot \text{ then } s^\sharp \text{ else } t^\sharp \longrightarrow^\sharp s^\sharp} \quad (\text{Red if}^{\sharp 1})$$

$$\frac{}{\text{if } \odot \text{ then } s^\sharp \text{ else } t^\sharp \longrightarrow^\sharp t^\sharp} \quad (\text{Red if}^{\sharp 2})$$

Table 5.9: New abstract reduction rules

is needed so that the static analysis can always terminate. For the *if* statement, we note again that it produces two results, differently from the concrete case. We recall again that in the abstract reduction we examine all possible path of execution, due to the above rules for the *if* statement, and due to the fact that the evaluation contexts $\mathcal{E} \Rsh b$ and $a \Rsh \mathcal{E}$ are used to obtain all possible interleavings between parallel threads.

Finally, lte us remark that, as already seen and proved in chapter 4, the lock environments model correctly all the locks held by each (sub)term at each program point. Using lock environments, then, it is possible to know under which locks each term or subterm is executed.

5.4 Correctness of the Abstract Interpretation

In this section we resume the correctness results about the abstract interpretation proved in chapter 4.

First of all we recall the formal definition of the ordering relation of the abstract domain. Abstract statements are ordered according to the rules presented in Table 5.10. Note that, differently from chapter 4, now the abstract terms are compared only according to their multisets of method calls. This because, since the lock environments occur also in the concrete syntax, we require equality on them when comparing abstract terms.

$[\![t^\sharp]\!]^{L,S_1} \sqsubseteq [\![t^\sharp]\!]^{L,S_2}$	$\Leftrightarrow$	$S_1 \subseteq S_2$
$t_1^\sharp \sqsubseteq t_2^\sharp$	$\Rightarrow$	$(\nu p.t_1^\sharp) \sqsubseteq (\nu p.t_2^\sharp)$
$t_1^\sharp \sqsubseteq t_2^\sharp$	$\Rightarrow$	$(\text{locked } p \text{ in } t_1^\sharp) \sqsubseteq (\text{locked } p \text{ in } t_2^\sharp)$
$(s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp)$	$\Rightarrow$	$(\text{let } x = s_1^\sharp \text{ in } t_1^\sharp) \sqsubseteq (\text{let } x = s_2^\sharp \text{ in } t_2^\sharp)$
$(s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp)$	$\Rightarrow$	$(s_1^\sharp \Rsh t_1^\sharp) \sqsubseteq (s_2^\sharp \Rsh t_2^\sharp)$
$(r_1^\sharp \sqsubseteq r_2^\sharp) \wedge (s_1^\sharp \sqsubseteq s_2^\sharp) \wedge (t_1^\sharp \sqsubseteq t_2^\sharp)$	$\Rightarrow$	$(\text{if } r_1^\sharp \text{ then } s_1^\sharp \text{ else } t_1^\sharp) \sqsubseteq (\text{if } r_2^\sharp \text{ then } s_2^\sharp \text{ else } t_2^\sharp)$

Table 5.10: Abstract ordering relation

The first result shows that α and γ form a Galois connection between the concrete and abstract domains, that is the concrete and abstract syntax respectively (since we deal with a calculus). Its proof is largely analogous to the one presented for proposition 4.5.2, with the only difference that the concretization function discards only the multisets of method calls, and does not discards lock environments. However, since we require equality between lock environments when comparing terms, the proofs remain valid.

Proposition 5.4.1. *Let $t^\sharp$ be an abstract term, and $S \in \wp(C)$ be a set of concrete terms. α and γ form a Galois connection between the two domains $\wp(C)$ and A. That is:*

- *α and γ are monotonic,*
- *$S \subseteq \gamma(\alpha(S))$, where α and γ are applied pointwise,*
- *$\alpha(\gamma(t^\sharp)) \sqsubseteq t^\sharp$.*

□

Proposition 5.4.2. *Let a and t be respectively a concrete statement and a concrete term, if $a \longrightarrow^n t$ then there exists an abstract term $t^\sharp$ such that $\alpha(a) \longrightarrow^{\sharp m} t^\sharp$ and $t \in \gamma(t^\sharp)$. $\longrightarrow$ and $\longrightarrow^\sharp$ include the congruence rule applications which make possible the reduction. $\longrightarrow^n$ denotes then a sequence of applications of n consecutive reduction rules and congruences.*

□

This proposition states that the abstract reduction correctly approximates the concrete one. That is every concrete computation has a corresponding abstract one, and then if a property is verified for all the reductions of an abstract statement $\alpha(a)$, it is verified also for a.

Proposition 5.4.3. *Given a statement a, the abstract interpretation process of $\alpha(a)$ always terminates.*

□

5.5 Lock elimination analysis

In this section we consider three categories of locks that can be eliminated from a concrete statement. They were presented informally in section 5.1 as reentrant, single threaded, and enclosed locks. We will define formally those categories in the following.

Reentrant, single threaded and enclosed locks may be eliminated from a statement in the **raconc**ς calculus using the information contained inside lock environments. In fact, given a concrete statement a, first we calculate the reduction graph of the corresponding abstract term $t^\sharp = \alpha(a)$. After this we use a *labeling function*, as defined below, to give a unique label to each lock of the initial statement and propagate these labels along all the abstract reduction graph.

Definition 5.5.1 (labeling function)**.** *A labeling function Λ is an injective function from program points to $\mathbb{N}$.*

In the following, given a program point *lock p in t* such that $\Lambda(\mathit{lock}\ p\ \mathit{in}\ t) = i$, we say for short that the lock has label i, and write lock^i to identify such lock. Given a lock with label i, we consider all its acquisitions in the graph and the respective lock environments before these acquisitions. Note that, since the labeling function is injective, we have that if a lock labeled by i is requested more than once, than it is always requested on the same reference (essentially because this case happens only when a lock appears in the definition of a method which is called more than once). A lock can be eliminated if it satisfies one of the following definitions:

Definition 5.5.2 (reentrant lock)**.** *Let i be the label for a lock requested n times on the reference p in the abstract reduction graph, and let $L_1, \ldots, L_n$ be the lock environments before the n acquisitions of lock^i. The lock^i is reentrant if:*

$$\forall j \in \{1, \ldots, n\}.\ p \in L_j$$

The above definition simply requires that the name for which the lock i is requested already appears inside the lock environment before the acquisition.

Definition 5.5.3 (single threaded lock)**.** *Let i be the label for a lock requested n times on the reference p in the abstract reduction graph, and let $s_j = [\![\mathit{lock}^i\ p\ \mathit{in}\ t]\!]^{L_j}$ with $j \in \{1, \ldots, n\}$ be the terms in the graph where the lock^i is requested. Now consider all m terms $t_h = [\![\mathit{lock}^k\ p\ \mathit{in}\ t]\!]^{L_h}$ for all k (note that $\{t_h\} \supseteq \{s_j\}$). The lock^i is single threaded if we do not have any (sub)statement of the form*

$$t_h \upharpoonright s_j$$

appearing in the abstract reduction graph.

The above definition requires that the lock i on the reference p is never requested in parallel with other locks on the same name p in all the reduction graph.

Definition 5.5.4 (enclosed lock)**.** *Let i be the label for a lock requested on the reference p, let L be the lock environment before its acquisition, and let $L_1, \ldots, L_n$ be the lock environments before the acquisition of all the other locks on p in the graph. The lock^i is enclosed if:*

$$\forall j \in \{1, \ldots, n\}.\ L \cap L_i \neq \emptyset$$

The above definition requires that, considering a lock labeled by i on the reference p, it may be eliminated if the lock environment that we have before its acquisition has at least one reference in common with all the other lock environments for locks on p. This means that we have requested some other locks before requesting the lock on p, and these locks actually protect from concurrent accesses.

Proposition 5.5.1. *Given a concrete statement a, if a lock may be eliminated from its abstraction $\alpha(a)$, then the same lock may also be eliminated from a.*

Proof: The argument is analogous to the one presented in chapter 4 for the race checking analysis.

□

5.6 Example

Let us give a simple example. Consider the following statement a, where the numbers near each lock are the unique labels given for the analysis:

$$a = \llbracket \nu p.\nu q.p \mapsto [\ell = \varsigma(x)\ \text{lock}^1\ x \text{ in } 5]^\circ \Rsh q \mapsto [\ell = \varsigma(x)6]^\circ \Rsh$$
$$\text{if } n \text{ then lock}^2\ p \text{ in } p.\ell \text{ else } (\text{lock}^3\ p \text{ in lock}^4\ q \text{ in } p.\ell \Rsh\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell)\rrbracket^{\emptyset}$$

$\alpha_s(a)$ is analogous to a apart from the substitution of $\odot$ for every integer value and the attachment of empty multisets of method calls to environments. The graph of all possible abstract reductions for $\alpha(a)$ is shown in Figure 5.1. For the sake of readability we removed the restrictions and the reference statements from the figure. Moreover, because it is clear from the context, we removed the word "locked" from the terms when a lock is acquired. The structural congruence rules are applied implicitly. Note that we removed also the rightmost part of the graph, representing it by $\ldots$, since it is equal to the central part, exchanging the order of the execution of the two terms in parallel.

$$\begin{tikzcd}
 & \boxed{\llbracket \text{if } \odot \text{ then lock}^2\ p \text{ in } p.\ell \text{ else } (\text{lock}^3\ p \text{ in lock}^4\ q \text{ in } p.\ell \Rsh\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell)\rrbracket^{\emptyset,\emptyset}} \arrow[dl] \arrow[d] & \\
\boxed{\llbracket \text{lock}^2\ p \text{ in } p.\ell\rrbracket^{\emptyset,\emptyset}} \arrow[d] & \boxed{\llbracket (\text{lock}^3\ p \text{ in lock}^4\ q \text{ in } p.\ell \Rsh\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell)\rrbracket^{\emptyset,\emptyset}} \arrow[d] & \\
\boxed{\llbracket p.\ell\rrbracket^{\{p\},\emptyset}} \arrow[d] & \boxed{\llbracket \text{lock}^3\ p \text{ in lock}^4\ q \text{ in } p.\ell\rrbracket^{\emptyset,\emptyset} \Rsh \llbracket\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell\rrbracket^{\emptyset,\emptyset}} \arrow[d] \arrow[dr] & \\
\boxed{\llbracket \text{lock}^1\ p \text{ in } \odot \rrbracket^{\{p\},\{p.\ell\}}} \arrow[d] & \boxed{\llbracket \text{lock}^4\ q \text{ in } p.\ell\rrbracket^{\{p\},\emptyset} \Rsh \llbracket\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell\rrbracket^{\emptyset,\emptyset}} \arrow[d] & \cdots \\
\boxed{\llbracket \odot \rrbracket^{\{p,p\},\{p.\ell\}}} \arrow[d] & \boxed{\llbracket p.\ell\rrbracket^{\{p,q\},\emptyset} \Rsh \llbracket\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell\rrbracket^{\emptyset,\emptyset}} \arrow[d] & \\
\boxed{\odot} & \boxed{\llbracket \text{lock}^1\ p \text{ in } \odot \rrbracket^{\{p,q\},\{p.\ell\}} \Rsh \llbracket\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell\rrbracket^{\emptyset,\emptyset}} \arrow[d] & \\
 & \boxed{\llbracket \odot \rrbracket^{\{p,p,q\},\{p.\ell\}} \Rsh \llbracket\ \text{lock}^5\ p \text{ in lock}^6\ q \text{ in } q.\ell\rrbracket^{\emptyset,\emptyset}} \arrow[d] & \\
 & \boxed{\odot \Rsh \llbracket \text{lock}^6\ p \text{ in } q.\ell\rrbracket^{\{p\},\emptyset}} \arrow[d] & \\
 & \boxed{\odot \Rsh \llbracket q.\ell\rrbracket^{\{p,q\},\emptyset}} \arrow[d] & \\
 & \boxed{\odot \Rsh \odot} &
\end{tikzcd}$$

Figure 5.1: Abstract reduction graph for $\alpha_s(a)$

Following our analysis, we have that the locks 1,2,4 and 6 may be eliminated from the statement a. In fact the lock number 1 is acquired three times (one is in the removed rightmost part of the graph), and for each acquisition we have a lock environment which contains p. Since this lock is required on the reference p, we have a reentrant lock. For lock number 2, it is clearly a single threaded lock, since we have no threads in parallel with the one requesting it. For the locks 4 and 6 we have that they are acquired twice (again, the second acquisitions appear in the rightmost part of the graph), and for each acquisition we have a lock environment equal to $\{p\}$. Note that these locks are requested on the reference q and that we do not have any other request for locks on q in the graph. Following the definition of enclosed locks, one of these locks may be eliminated because the intersection between the lock environments is not empty, being equal to $\{p\}$. After this first elimination we would have only one lock requested on the name q. Again, all the lock environmets before the acquisition of this lock are equal to $\{p\}$ and then we have another enclosed lock that can be eliminated safely. The resulting statement after these eliminations is then:

$$\begin{gathered} a_1 = [\![\nu p.\nu q.p \mapsto [\ell = \varsigma(x)5]^\circ \Rsh q \mapsto [\ell = \varsigma(x)6]^\circ \Rsh \\ \text{if } n \text{ then } p.\ell \text{ else } (\text{lock}^3\ p \text{ in } p.\ell \Rsh\ \text{lock}^5\ p \text{ in } q.\ell)]\!]^\emptyset \end{gathered}$$

Since we have removed four locks, the abstract reduction graph for this statement is shorter than the one seen above. This means that we have reduced the computation steps and that in a real language we would have a great advantage in terms of performance.

Note that the abstract reduction graph should be recomputed each time we eliminate a lock, since the lock environments could be affected by such elimination. A way to avoid this inconvenient is to annotate each reference inside a lock environment with the label of the lock that acquired it. In this way, each time we remove a lock we can simply update the lock environments affected by this deletion, and go on with further eliminations, if possible.

Chapter 6

Abstract Interpretation against Insecure Flows

Abstract

In this chapter we will adapt our approach based on abstract interpretation to another framework of analysis, the one of security. In particular, we will investigate in this chapter the use of abstract interpretation techniques to ensure a property of *non-interference* in concurrent programs. The approach followed in this chapter will be unified with the rest of the thesis, starting from the definition of a concurrent object calculus (we will simplify the **aconc**ς calculus removing locks, since they are irrelevant for our analysis), continuing with a definition of abstract interpretation (which will be modified from the preceding ones to introduce security levels), and ending with a security analysis.

6.1 The Problem of Insecure Information Flow

Given a program, we may want to assign to variables a *security level.* This approach is often used to encode situations where we want that some information stored in a variable belonging to a certain security level, may not be read or investigated using variables belonging to lower security levels. In the most general sense, we can say that a program has a property of *secure information flow* if the information contained in each of its variables, when termination is reached, does not depend from the initial values contained inside variables with higher information level. If we suppose to have only two security levels, H (for "high") and L (for "low"), and two instance variables of an object in **aconc**ς calculus $p.x$ and $p.y$ with levels respectively H and L, then the following programs *do not* satisfy the secure information flow property:

$$\nu p.\ p \mapsto [x = \varsigma(x)5, y = \varsigma(x)7] \vec{\ } \text{let } z = p.x \text{ in } (p.y \Leftarrow z)$$

$$\nu p.\ p \mapsto [x = \varsigma(x)5, y = \varsigma(x)7] \Rsh \text{if } p.x \text{ then } p.y \Leftarrow 1 \text{ else } p.y \Leftarrow 0$$

In the first case we have a *direct* information flow starting from a variable of level H ($p.x$) and ending in a variable of level L ($p.y$). In fact, after the execution, the variable with initial level L now contains a value which had H as its initial level. In the second case, instead, we do not have an explicit violation of the secure information flow as in the previous case. Nevertheless we have that the final value of a variable with initial level L is dependent from the value of a variable with initial level H, since, as an example, if the final value of $p.y$ is 0, we know that the initial value of $p.x$ was 0 as well.

This problem is also known as *non-interference*. The non-interference property has been introduced in the eighties in some works by Goguen and Meseguer [31, 32]. More recently, this problem has been studied both in sequential and parallel languages using both type checking [56, 54, 35, 43, 55, 45, 49, 53, 13, 14, 58, 41, 37] and semantics techniques [4, 5, 9, 29, 30, 39, 42, 50, 52].

The fundamental difference between sequential and parallel languages, in this context, lies in the fact that in sequential programs, we have that indirect information flows (like the second one above) cannot extend beyond the scope of commands. As an example, let us consider a statement similar to one of the two seen before:

$$\begin{array}{l}\nu p.\ p \mapsto [x = \varsigma(x)5, y = \varsigma(x)7] \Rsh \\ \text{let } z = (\text{if } p.x \text{ then } p.y \Leftarrow 0 \text{ else } p.y \Leftarrow 1) \text{ in } (p.y \Leftarrow 10)\end{array} \tag{6.1}$$

If we consider $p.x$ as having information level H, and $p.y$ as having information level L, we can see that there is no information flow from $p.x$ to $p.y$. In fact, we have that at the end of the execution of the program, the value stored in $p.y$ is always 10, independently of the value stored in variable $p.x$.

When we use multithreaded languages, instead, we have new kind of information flows due to synchronizations [48]. Consider, as an example, the following statement:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a = \varsigma(x)5, b = \varsigma(x)1, c = \varsigma(x)5, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } 1) \text{ in } x.c \Leftarrow 1] \Rsh \\ & (\text{if } p.a \text{ then } p.b \Leftarrow 1 \text{ else let } z = p.b \text{ in } p.b \Leftarrow z) \Rsh p.l\end{array}$$

and suppose that an attacker cannot observe the program non-termination or total execution time. If we suppose to start the execution with a and b of level H, and c of level L, we have that the first of the two parallel threads:

$$\text{if } p.a \text{ then } p.b \Leftarrow 0 \text{ else let } z = p.b \text{ in } p.b \Leftarrow z$$

sets the variable $p.b$ to 0 if the value contained in $p.a$ is not 0, otherwise does nothing. This is an indirect information flow from variable $p.a$ to variable $p.b$, but it is not insecure, since we said that the $p.b$ variable starts with information level H. Look now at the second parallel thread:

$$p.l \longrightarrow (\text{let } y = (\text{if } p.b \text{ then } p.l \text{ else } 1) \text{ in } p.c \Leftarrow 1)$$

This thread looks at the value contained in $p.b$ and loops, calling again $p.l$, until $p.b$ is set to 0 by some other thread. Thus the first thread, with the above values for instance variables, unblocks the second one, passing indirectly to it some information about the value of the H variable $p.a$. In fact, we see that the value of $p.c$ changes to 1, and looking at the final value of $p.c$, we may say that the initial value of $p.a$ was different from 0. Then, in this case, we have an indirect information flow from the H variable $p.a$ to the L variable $p.c$, which is insecure and which extends *beyond* the scope of the command which tests the H variable $p.b$. We would like to remark, in fact, that the two above commands, taken as single threaded programs, are *secure*, (if non-termination or total execution time are not observable) while taken together they present this problem of insecure information flow.

In other words, whenever in a concurrent program we allow synchronization between concurrent threads based on shared variables and loops which test them, we have possible information flows from the variables guarding the loops, to the variables both *inside* the loops and *after* the loops.

In this chapter we will present a static analysis for checking the absence of insecure information flows based on abstract interpretation, as done in chapters 4 and 5. This is useful also in this case, since we gain in precision with respect to other kinds of analyses based on other approaches. As an example, many of the analyses for secure information flows based on type checking techniques suffer of the same problem: they cannot give a correct type to programs like the (6.1) shown before. In fact, type checking techniques often try to give correct types to programs by dealing singularly with each command. In program 6.1 we do have an insecure flow in the *if* statement, but this flow is completely deleted from the following instruction, because we have a constant assignment to the variable $p.y$. Many type checking techniques simply discard the program as non-secure when the first statement is analyzed. Obviously, there are also advantages in using type systems to perform these kind of analyses, namely their computational efficience, and the fact that many type systems can provide principal types for expressions, which subsume all other possible typings.

As done in previous chapters, we will present an object calculus, derived from a simplification of the **aconc**ς calculus presented in chapter 4, which will be used to model the basics of object oriented programming languages. We will then present an abstract interpretation for this calculus, adding security levels to methods, and we will use it to check the above properties of security.

6.2 The object calculus saconcς

6.2.1 Syntax

Table 6.1 defines the syntactic categories of results, denotations, terms and information levels. They are derived from the ones defined for the **aconc**ς calculus, removing all the constructs which are not necessary for our analysis of secure information flow. The main difference, between this calculus and the ones presented in chapters 4 and 5 is that now each method has an *initial level* of information associated to it. This level, without loss of generality, will assume one of the two values H or L, with the assumption that $L \leq H$. The language, however, can be easily extended (as well as the analysis we will show in the following) to support general lattices of security levels. This level codifies what kind of information is stored inside instance variables. We suppose that all other methods have initial level of information equal to L.

u	$::=$		<u>results</u>
		x	variable
		p	location
		n	integer number
d	$::=$		<u>denotations</u>
		$[\ell_i^{\Phi_i} = \varsigma(x_i)t_i \ ^{i\in 1\ldots n}]$	object
r, s, t	$::=$		<u>terms</u>
		u	result
		$\nu p.t$	restriction
		$p \mapsto d$	reference
		$u.\ell$	method invocation
		$u.\ell \Leftarrow \varsigma(x)u$	field update
		let $x = s$ in t	let
		$s \upharpoonright t$	parallel composition
		if r then s else t	if
Φ	$::=$		<u>initial information levels</u>
		H	high
		L	low

Table 6.1: Syntax of **saconc**ς

6.2.2 Semantics of saconcς

As done in the previous chapters, we will define the semantics of the **saconc**ς calculus using a structural congruence to transform terms, and some reduction rules to model computational steps.

Again, both the structural congruence and reduction rules are given in terms of evaluations contexts. We show the syntax of evaluation contexts and the structural congruence rules in Tables 6.2 and 6.3.

$$
\begin{array}{lll}
\mathcal{E} & ::= & \\
 & & [\cdot] \\
 & & \mathcal{E} \Rsh t \\
 & & s \Rsh \mathcal{E} \\
 & & \text{let } x = \mathcal{E} \text{ in } b \\
 & & \text{if } \mathcal{E} \text{ then } a \text{ else } b \\
 & & \nu p.\mathcal{E}
\end{array}
$$

Table 6.2: Reduction contexts

$$
\begin{array}{rcll}
s \Rsh \mathcal{E}[t] & \equiv & \mathcal{E}[s \Rsh t] & \text{if } fn(s) \cap bn(\mathcal{E}) = \emptyset \\
(\nu p)\mathcal{E}[s] & \equiv & \mathcal{E}[(\nu p)s] & \text{if } p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})
\end{array}
$$

Table 6.3: Structural congruence rules

The reduction rules for the **saconc**ς calculus are given in Table 6.4. They are analogous to the ones presented in chapter 4 for the **aconc**ς calculus.

6.3 Abstract interpretation of saconcς

In this section we present an abstract interpretation for the calculus shown before. Since the analysis about secure information flow requires a different kind of knowledge with respect to the previous analyses, we will have to redefine the abstraction and the concretization functions, as well as the abstract reduction rules. Moreover, we have to address the problem of what is observable, so to understand what can generate those insecure information flows that our analysis will detect. This chapter will focus only on the absence of what is know in the literature as *strong dependency* [18] between low and high level variables. So we will look only at information flows caused by direct assignments, by the control structure of programs or by synchronization between parallel threads. We will not consider, as an example, those flows due to missing termination or longer computation after testing a high variable, also known, respectively, as *termination channels* and *timing channels*, or those flows due to resource exhaustion or power consumption observability [47]. Possible extensions of the work presented in this chapter could include then the analyses related to insecure information flows caused by resource contention, since these flows arise in any practical implementation of concurrent object oriented languages.

First of all, we have to consider that our notion of secure information flow, when analyzed using abstract interpretation techniques, requires to know which is the information level of the *contents* of variables, and to distinguish it from the initial information levels of the variables themselves. This will help us in the detection of flows of information caused by assignments of high values to low variables. Another notion needed for our analysis will be attached to commands, since we want to

$$\frac{d = [\ell_i^{\Phi_i} = \varsigma(x_i)t_i\ ^{i\in(1\ldots n)}] \quad j \in (1,\ldots,n)}{p \mapsto d \Rsh p.\ell_j \longrightarrow p \mapsto d \Rsh t_j\{\{x_j \leftarrow p\}\}} \text{ (Red invoke)}$$

$$\frac{\begin{array}{c} d = [\ell_j^{\Phi_j} = \varsigma(x)u_j, \ell_i^{\Phi_i} = \varsigma(x_i)t_i\ ^{i\in(1\ldots n)-\{j\}}] \\ d' = [\ell_j^{\Phi_j} = \varsigma(x)u, \ell_i^{\Phi_i} = \varsigma(x_i)t_i\ ^{i\in(1\ldots n)-\{j\}}] \end{array}}{p \mapsto d \Rsh p.\ell_j \Leftarrow \varsigma(x)u \longrightarrow p \mapsto d' \Rsh p} \text{ (Red update)}$$

$$\frac{}{\text{let } x = u \text{ in } t \longrightarrow t\{\{x \leftarrow u\}\}} \text{ (Red let)}$$

$$\frac{}{\text{if } 0 \text{ then } s \text{ else } t \longrightarrow t} \text{ (Red if0)}$$

$$\frac{n \neq 0}{\text{if } n \text{ then } s \text{ else } t \longrightarrow s} \text{ (Red ifn)}$$

$$\frac{s \longrightarrow t}{\mathcal{E}[s] \longrightarrow \mathcal{E}[t]} \text{ (Red context)}$$

Table 6.4: Concrete Reduction rules

capture also indirect information flows due to the control structure of programs and to synchronizations. We will attach to each term of the language, a *level of knowledge*. This level will represent the amount of knowledge (in term of information level) from which the current term depends. As an example, by considering an *if* term which reads a variable with high information level, we should have a high level of knowledge in both branches of the *if* term itself, so that we will be able to capture indirect information flows. Finally, we will need some more information about recursive calls. In fact, in order to detect flows due to synchronizations, we will modify the multisets of method calls used in the previous chapters to force the termination of the abstract reduction. We will attach to each annotation of method call the level of knowledge that the abstract term had when the call was performed. In this way we will be able to detect those cases when recursion happens with a high level of knowledge. This is completely equivalent to the detection of loops which test information with level H, since recursion is the only way which allows our terms to loop.

The syntax of the abstract calculus reflects the new types of information attached to terms, and is shown in Table 6.5. According to what said above, we have that now results are variables, $\top^\sharp$ to truncate recursions (as in the previous chapters), and the two abstract values $\odot^\varphi$ and p^φ, corresponding to numbers and references with attached their information level, which can be h (for high) and l (for low). In this way we will be able to investigate, at the end of programs, if we have cases when the level of values inside instance variables is greater than the one that those variables had at the beginning of the program. In this case we will have that an insecure information flow happened. To detect cases of indirect information flows we added a new abstract term $[\![t^\sharp]\!]^{\varphi,S}$ with attached a level of knowledge φ which is the least upper bound (according to the ordering relation $l \leq h$) among all the values from which the term depends in its execution. Finally in this new kind of term we can see also the usual multiset of method calls used in the previous chapters to truncate recursions. As said before, this set will also be used to register the level of knowledge of each method call, in order to detect recursions (loops) depending on high level values. If the level of a recursion is h, all subsequent computations (branches of *if* terms or bodies of *let* terms) will have to consider it, because assignments to low-level variables may result in insecure information flows due to synchronizations.

The abstraction function attaches all these kinds of information about levels to concrete terms, giving initial information level l to all constants and references found inside terms, and translating the initial information levels for the values inside instance variables, so that if the initial information level is H, then the value stored in the variable will have information level h, while if the initial information level is L, then the value stored in the variable will have information level l. In the definition of the abstraction function this transformation will be done using a function $\mathcal{T}$ such that $\mathcal{T}(H) = h$ and $\mathcal{T}(L) = l$. We recall again our hypothesis that all methods have initial information level L. As usual, we have different functions for each syntactic category of the grammar describing the abstract syntax. The abstraction functions

$$
\begin{array}{lcl}
u^\sharp & ::= & x \mid p^\varphi \mid \odot^\varphi \mid \top^\sharp \\
\\
d^\sharp & ::= & [\ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ ^{i \in 1 \dots n}] \\
\\
r^\sharp, s^\sharp, t^\sharp & ::= & u^\sharp \mid \nu p.t^\sharp \mid p \mapsto d^\sharp \mid u^\sharp.\ell \mid u^\sharp.\ell \Leftarrow \varsigma(x)u^\sharp \mid \\
 & & \text{let } x = s^\sharp \text{ in } t^\sharp \mid s^\sharp \Rsh t^\sharp \mid \text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp \mid \llbracket t^\sharp \rrbracket^{\varphi, S} \\
\\
\Phi & ::= & H \mid L \\
\\
\varphi & ::= & h \mid l \\
\\
S & = & \{(p_1.\ell_1, \varphi_1), \dots, (p_m.\ell_m, \varphi_m)\}
\end{array}
$$

Table 6.5: Syntax of **saconc**$\varsigma^\sharp$

are shown in Table 6.6.

$$
\begin{array}{lcl}
\alpha_r(x, \varphi) & = & x \\
\alpha_r(p, \varphi) & = & p^\varphi \\
\alpha_r(n, \varphi) & = & \odot^\varphi \\
\\
\alpha_d([\ell_i^{\Phi_i} = \varsigma(x_i)t_i \ ^{i \in 1 \dots n}], \varphi) & = & [\ell_i^{\Phi_i} = \varsigma(x_i)\alpha_t(t_i, \mathcal{T}(\Phi_i)) \ ^{i \in 1 \dots n}] \\
\\
\alpha_t(u, \varphi) & = & \alpha_r(u, \varphi) \\
\alpha_t(\nu p.t, \varphi) & = & \nu p.\alpha_t(t, \varphi) \\
\alpha_t(p \mapsto d, \varphi) & = & p \mapsto \alpha_d(d, \varphi) \\
\alpha_t(u.\ell, \varphi) & = & \alpha_r(u, \varphi).\ell \\
\alpha_t(u_1.\ell \Leftarrow \varsigma(x)u_2, \varphi) & = & \alpha_r(u_1, \varphi).\ell \Leftarrow \varsigma(x)\alpha_r(u_2, \varphi) \\
\alpha_t(\text{let } x = s \text{ in } t, \varphi) & = & \text{let } x = \alpha_t(s, \varphi) \text{ in } \alpha_t(t, \varphi) \\
\alpha_t(s \Rsh t, \varphi) & = & \alpha_t(s, \varphi) \Rsh \alpha_t(t, \varphi) \\
\alpha_t(\text{if } r \text{ then } s \text{ else } t, \varphi) & = & \text{if } \alpha_t(r, \varphi) \text{ then } \alpha_t(s, \varphi) \text{ else } \alpha_t(t, \varphi) \\
\\
\alpha(t) & = & \llbracket \alpha_t(t, l) \rrbracket^{l, \emptyset}
\end{array}
$$

Table 6.6: Abstraction functions

Once the abstraction functions are defined, the concretization ones are uniquely determined. In this case the concretization functions simply discard all information concerning information levels of values, levels of knowledge of terms, and multisets of method calls. The concretization functions are shown in Table 6.7.

Again, we recall the usual definition of the concrete domain as the powerset of the set C containing all concrete terms of the **saconc**ς-calculus, with the ordering

$\gamma_r(x)$	$=$	$\{x\}$
$\gamma_r(p^\varphi)$	$=$	$\{p\}$
$\gamma_r(\odot^\varphi)$	$=$	$\{n \mid n$ is an integer number$\}$
$\gamma_r(\top^\sharp)$	$=$	C the set of all concrete statements
$\gamma_d([\ell_i^\Phi = \varsigma(x_i)t_i^\sharp\ {}^{i\in 1\ldots n}])$	$=$	$\{[\ell_i^\Phi = \varsigma(x_i)t_i\ {}^{i\in 1\ldots n}] \mid t_i \in \gamma_t(t_i^\sharp)\}$
$\gamma_t(u^\sharp)$	$=$	$\gamma_r(u^\sharp)$
$\gamma_t(\nu p.t^\sharp)$	$=$	$\{\nu p.t \mid t \in \gamma_t(t^\sharp)\}$
$\gamma_t(p \mapsto d^\sharp)$	$=$	$\{p \mapsto d \mid d \in \gamma_d(d^\sharp)\}$
$\gamma_t(u^\sharp.\ell)$	$=$	$\{u.\ell \mid u \in \gamma_r(u^\sharp)\}$
$\gamma_t(u_1^\sharp.\ell \Leftarrow \varsigma(x)u_2^\sharp)$	$=$	$\{u_1.\ell \Leftarrow \varsigma(x)u_2 \mid u_1 \in \gamma_r(u_1^\sharp), u_2 \in \gamma_r(u_2^\sharp)\}$
$\gamma_t(\text{let } x = s^\sharp \text{ in } t^\sharp)$	$=$	$\{\text{let } x = s \text{ in } t \mid s \in \gamma_t(s^\sharp), t \in \gamma_t(t^\sharp)\}$
$\gamma_t(s^\sharp \upharpoonright t^\sharp)$	$=$	$\{s \upharpoonright t \mid s \in \gamma_t(s^\sharp), t \in \gamma_t(t^\sharp)\}$
$\gamma_t(\text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp)$	$=$	$\{\text{if } r \text{ then } s \text{ else } t \mid r \in \gamma_t(r^\sharp), s \in \gamma_t(s^\sharp), t \in \gamma_t(t^\sharp)\}$
$\gamma_t([\![t^\sharp]\!]^{\varphi,S})$	$=$	$\gamma_t(t^\sharp)$
$\gamma(t^\sharp)$	$=$	$\gamma_t(t^\sharp)$

Table 6.7: Concretization functions

relation of containment between sets, $\subseteq$. The abstract domain is instead defined as the set A containing all abstract terms. The ordering relation for the abstract domain will be formally defined in the following, when we will prove the correctness of the abstract interpretation.

As done in the previous chapters we will define the abstract semantics using evaluation contexts, structural congruences and reduction rules. The evaluation contexts are the same as in the concrete case, and can be seen in Table 6.2. We have to add some congruences to deal with the new term $[\![t^\sharp]\!]^{\varphi,S}$ introduced in the abstract syntax. The new abstract congruence rules are shown in Table 6.8. According to the new rules, levels of knowledge and multisets of method calls are simply distributed among subterms.

$s^\sharp \upharpoonright \mathcal{E}[t^\sharp]$	$\equiv$	$\mathcal{E}[s^\sharp \upharpoonright t^\sharp]$	if $fn(s^\sharp) \cap bn(\mathcal{E}) = \emptyset$
$(\nu p)\mathcal{E}[s^\sharp]$	$\equiv$	$\mathcal{E}[(\nu p)s^\sharp]$	if $p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})$
$[\![s^\sharp \upharpoonright t^\sharp]\!]^{\varphi,S}$	$\equiv$	$[\![s^\sharp]\!]^{\varphi,S} \upharpoonright [\![t^\sharp]\!]^{\varphi,S}$	
$[\![\text{let } x = s^\sharp \text{ in } t^\sharp]\!]^{\varphi,S}$	$\equiv$	$\text{let } x = [\![s^\sharp]\!]^{\varphi,S} \text{ in } [\![t^\sharp]\!]^{\varphi,S}$	
$[\![\text{if } r^\sharp \text{ then } s^\sharp \text{ else } t^\sharp]\!]^{\varphi,S}$	$\equiv$	$\text{if } [\![r^\sharp]\!]^{\varphi,S} \text{ then } [\![s^\sharp]\!]^{\varphi,S} \text{ else } [\![t^\sharp]\!]^{\varphi,S}$	
$[\![\nu p.t^\sharp]\!]^{\varphi,S}$	$\equiv$	$\nu p'.[\![t^\sharp\{\{p \leftarrow p'\}\}]\!]^{\varphi,S}$	p' fresh

Table 6.8: New abstract structural congruence rules

$$\frac{d^\sharp = [\ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ ^{i \in (1 \ldots n)}] \quad j \in (1, \ldots, n) \quad occ(p.\ell_j, S) \leq 1}{p \mapsto d^\sharp \Rsh [\![p^\varphi.\ell_j]\!]^{\psi,S} \longrightarrow^\sharp p \mapsto d^\sharp \Rsh [\![t_j^\sharp\{\{x_j \leftarrow p^\varphi\}\}]\!]^{\psi,S \uplus \{(p.\ell_j,\psi)\}}} \ (\text{Red invoke}^{\sharp 1})$$

$$\frac{d^\sharp = [\ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ ^{i \in (1 \ldots n)}] \quad j \in (1, \ldots, n) \quad occ(p.\ell_j, S) > 1}{p \mapsto d^\sharp \Rsh [\![p^\varphi.\ell_j]\!]^{\psi,S} \longrightarrow^\sharp p \mapsto d^\sharp \Rsh \top^\sharp} \ (\text{Red invoke}^{\sharp 2})$$

$$\frac{\begin{array}{c} d = [\ell_j^{\Phi_j} = \varsigma(x)u_j^\sharp, \ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ ^{i \in (1 \ldots n) - \{j\}}] \\ d' = [\ell_j^{\Phi_j} = \varsigma(x)u^{\varphi \sqcup \psi \sqcup \vartheta}, \ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ ^{i \in (1 \ldots n) - \{j\}}] \end{array}}{p \mapsto d \Rsh [\![p^\varphi.\ell_j \Leftarrow \varsigma(x)u^\psi]\!]^{\vartheta,S} \longrightarrow p \mapsto d' \Rsh [\![p^\varphi]\!]^{\vartheta,S}} \ (\text{Red update}^\sharp)$$

$$\frac{\theta = \Re(S)}{\text{let } x = [\![u^\varphi]\!]^{\psi,S} \text{ in } [\![t^\sharp]\!]^{\vartheta,S'} \longrightarrow [\![t^\sharp\{\{x \leftarrow u^{\varphi \sqcup \psi}\}\}]\!]^{\vartheta \sqcup \theta, S'}} \ (\text{Red let}^\sharp)$$

$$\frac{\theta = \Re(S) \sqcup \varphi \sqcup \psi \sqcup \vartheta}{\text{if } [\![\odot^\varphi]\!]^{\psi,S} \text{ then } [\![s^\sharp]\!]^{\vartheta,S'} \text{ else } [\![t^\sharp]\!]^{\vartheta,S'} \longrightarrow^\sharp [\![s^\sharp]\!]^{\theta,S'}} \ (\text{Red if}^{\sharp 1})$$

$$\frac{\theta = \Re(S) \sqcup \varphi \sqcup \psi \sqcup \vartheta}{\text{if } [\![\odot^\varphi]\!]^{\psi,S} \text{ then } [\![s^\sharp]\!]^{\vartheta,S'} \text{ else } [\![t^\sharp]\!]^{\vartheta,S'} \longrightarrow^\sharp [\![t^\sharp]\!]^{\theta,S'}} \ (\text{Red if}^{\sharp 2})$$

$$\frac{s^\sharp \longrightarrow t^\sharp}{\mathcal{E}[s^\sharp] \longrightarrow \mathcal{E}[t^\sharp]} \ (\text{Red context}^\sharp)$$

$$\frac{}{[\![p \mapsto d^\sharp]\!]^{\varphi,S} \longrightarrow p \mapsto d^\sharp} \ (\text{Red den}^\sharp)$$

$$\frac{}{\mathcal{E}[\top^\sharp] \longrightarrow \top^\sharp} \ (\text{Red } \top^\sharp)$$

Table 6.9: New abstract reduction rules

The reduction rules for the abstract calculus are shown in Table 6.9. Since we have to deal with levels of information and levels of knowledge attached to terms, we are forced to redefine all concrete rules. We will give a brief explanation in what follows. According to rule (Red invoke$^{\sharp 1}$), we have a method invocation of the form $p^\varphi.\ell_j$. The reference p has an information level associated to it, since, as an example, it can come as a result of the reduction of a term with an high level of knowledge (i.e. an *if* term or a recursion testing an high variable), and this would mean that simply using the value p inside a term we could have some leak of high information. The whole term $[\![p^\varphi.\ell_j]\!]^{\psi,S}$ means also that the current level of knowledge is ψ (if this value is high, maybe we are in the scope of an *if* term which tests an high variable), and that the current multiset of method calls (this is quite similar to the ones seen in the previous chapters) is S. The method invocation is, as usual, replaced by the method body with the instantiation of the *self* variable by p^φ, and the new level of knowledge remains the same as the level we had when the call was performed, ψ. The set S is updated with the new call, as well as with the level ψ, to indicate that this call was executed when having a level of knowledge of ψ. This set, as in previous chapters will be used to truncate recursions, but here we have another important function. The knowledge levels attached to each calls will help to determine when a recursion is performed having an high level of knowledge. When this happens we may have an attempt of synchronization between parallel threads which can leak some high information. As in previous chapters, we used in this rule the function $occ(p.\ell_j, S)$ which determines if we have recursion (it is slightly different from the ones used previously, since S is now a multiset of couples), regardless of the levels of knowledge of the calls in S. The rule (Red invoke$^\sharp$2) allows to truncate recursions, as in previous chapters. Note that the information level of the reference p^φ is simply discarded in this rule. This is correct since recursive calls are expanded twice and then we already used φ in the previous recursions. The rule (Red update$^\sharp$) allows to update instance variables. According to the rule, we have that the new level of information of the updated variable is obtained as the least upper bound between the level of information of the result which is used to update the variable itself, the level of knowledge of the update term and the level of information of the reference to the updated object. The level of the result u must be included since that result directly updates the variable. The level of knowledge of the update term must be included since the term may be executed, as an example, inside an *if* term which tests an high variable. Finally, the information level of the reference has to be included in the new information level of the variable since that reference may be the result of the evaluation of a term with high level of knowledge. The returned reference $[\![p^\varphi]\!]^{\vartheta,S}$ keeps its level of information, as well as the level of knowledge of the update term. Then, according to the (Red res$^\sharp$) rule, the reference may change level of information to $p^{\varphi\sqcup\vartheta}$, since it is the result of an update term, and this latter term may have a high level of knowledge. The rule (Red let$^\sharp$) allows to reduce *let* terms. We already said that *let* terms allow to simulate sequences of commands, and then in this rule we have to consider the level of knowledge of high recursivions. In fact, if we have a recursion in the first subterm of a *let* term, and if this recursion has high level of knowledge, we may have an insecure synchronization between parallel threads.

Thus we have to pass this high level of knowledge to the second term of the *let*, in order to check all assignments performed *after* this synchronization. The rule binds the variable x to the result of the first part of the construct u, with information level equal to the least upper bound between the level of the returned value, φ, and the level of knowledge of the term which returns it, ψ. Note that we do not consider the level of recursive calls in the binding, since this level updates the level of knowledge of the whole term where the binding is performed. The new level of knowledge is obtained as the least upper bound between the previous level of knowledge and the level of recursion of recursive calls. This level can be obtained by a special function $\Re$, used in the premise of the rule. This function extracts the least upper bound between all levels of recursive calls in S (a call is recursive if it appears more than once), and for our case where we have only two levels of information, it can be defined as follows:

$$\Re(\emptyset) = l$$

$$\Re(S) = \begin{cases} h & \text{if } \exists (p.\ell, h), (p.\ell, \varphi) \in S \\ l & \text{otherwise} \end{cases}$$

The same function is used also in the two rules for the *if* term. They are analogous to each other, so we will describe only the first one. According to it, we have a result $[\![\ \odot^{\varphi}\ ]\!]^{\psi,S}$ returned by the guard of the *if* term and we have to consider the level φ, since the result may be taken directly from a high variable, and the level ψ, since the result can come from the reduction of a term with high level of knowledge. Moreover, we have to consider the recursion level $\Re(S)$ of the guard of the *if* term, since this term evaluates two subterms in sequence, and the first one may be recursive (maybe leading to insecure synchronizations). Finally, we have to compute the least upper bound between all these levels and the previous level of knowledge for the subterm of the *if* term that is going to be reduced, ϑ, in order to obtain the new level of knowledge. As done in the previous chapters, all possible paths of abstract reduction will be examined. This can be seen from the existence of two (non-deterministic) rules for the *if* term, and from the evaluation contexts of the parallel term, $\mathcal{E} \upharpoonright t$ and $s \upharpoonright \mathcal{E}$, which allow to reduce all possible interleavings of parallel threads. This is useful because when we have parallel threads, we could have insecure flows according to which thread is executed before. The last two abstract rules are analogous to the ones seen in previous chapters.

6.4 Correctness of the Abstract Interpretation

We will prove in this section some results about correctness of the abstract interpretation just presented. The propositions are analogous to the ones shown in the previous chapters, however the proofs are different, since the abstract interpretation includes new concepts about terms.

First of all, we recall the structures of the concrete and abstract domains. The concrete domain, as usual, is represented by the powerset $\langle \wp(C), \subseteq, C, \emptyset, \cup, \cap \rangle$, where

C is the set containing all concrete terms of the **saconc**ς-calculus. The abstract domain is defined using the set A containing all abstract terms, with the ordering relation shown in Table 6.10. We use the usual relation between information levels, which states that $l \leq h$ and $L \leq H$.

$u^{\varphi} \sqsubseteq u^{\psi}$	$\Leftrightarrow$	$\varphi \leq \psi$
$u^{\sharp} \sqsubseteq u^{\sharp}_1$	$\Rightarrow$	$u^{\sharp}.\ell \sqsubseteq u^{\sharp}_1.\ell$
$(u^{\sharp} \sqsubseteq u^{\sharp}_1) \wedge (u^{\sharp}_2 \sqsubseteq u^{\sharp}_3)$	$\Rightarrow$	$(u^{\sharp}.\ell \Leftarrow \varsigma(x)u^{\sharp}_2) \sqsubseteq (u^{\sharp}_1.\ell \Leftarrow \varsigma(x)u^{\sharp}_3)$
$\forall i.\ (s^{\sharp}_i \sqsubseteq t^{\sharp}_i) \vee$ $(\Phi^i = H \wedge s^{\sharp}_i = u^{\varphi} \wedge t^{\sharp}_i = u^{\psi})$	$\Rightarrow$	$[\ell_i^{\Phi_i} = \varsigma(x_i)s^{\sharp}_i \ {}^{i \in (1 \ldots n)}] \sqsubseteq [\ell_i^{\Phi_i} = \varsigma(x_i)t^{\sharp}_i \ {}^{i \in (1 \ldots n)}]$
$d^{\sharp} \sqsubseteq d^{\sharp}_1$	$\Rightarrow$	$(p \mapsto d^{\sharp}) \sqsubseteq (p \mapsto d^{\sharp}_1)$
$t^{\sharp} \sqsubseteq t^{\sharp}_1$	$\Rightarrow$	$(\nu p.t^{\sharp}) \sqsubseteq (\nu p.t^{\sharp}_1)$
$(s^{\sharp} \sqsubseteq s^{\sharp}_1) \wedge (t^{\sharp} \sqsubseteq t^{\sharp}_1)$	$\Rightarrow$	$(\text{let } x = s^{\sharp} \text{ in } t^{\sharp}) \sqsubseteq (\text{let } x = s^{\sharp}_1 \text{ in } t^{\sharp}_1)$
$(r^{\sharp} \sqsubseteq r^{\sharp}_1) \wedge (s^{\sharp} \sqsubseteq s^{\sharp}_1) \wedge (t^{\sharp} \sqsubseteq t^{\sharp}_1)$	$\Rightarrow$	$(\text{if } r^{\sharp} \text{ then } s^{\sharp} \text{ else } t^{\sharp}) \sqsubseteq (\text{if } r^{\sharp}_1 \text{ then } s^{\sharp}_1 \text{ else } t^{\sharp}_1)$
$(s^{\sharp} \sqsubseteq s^{\sharp}_1) \wedge (t^{\sharp} \sqsubseteq t^{\sharp}_1)$	$\Rightarrow$	$(s^{\sharp} \Rsh t^{\sharp}) \sqsubseteq (s^{\sharp}_1 \Rsh t^{\sharp}_1)$
$(t^{\sharp} \sqsubseteq t^{\sharp}_1) \wedge (\varphi \leq \psi) \wedge (S \subseteq S_1)$	$\Rightarrow$	$\llbracket t^{\sharp} \rrbracket^{\varphi, S} \sqsubseteq \llbracket t^{\sharp}_1 \rrbracket^{\psi, S_1}$

Table 6.10: Abstract ordering relation

According to this ordering relation, abstract terms are compared according both to multisets of method calls (which are compared using containment between multisets, as in the previous chapters) and to information levels (which are compared using $l \leq h$ and $L \leq H$). Given two abstract terms $s^{\sharp}$ and $t^{\sharp}$ they are compared with each other by comparing their corresponding subterms. The base elements in the comparison are results, which are compared according to their information levels. After results we have that method calls are compared by comparing what calls the method (a variable or a reference). Instance variable updates are compared recursively on both the reference of the updated object and the result which update the instance variable itself. Given two denotations, each method of the first one is compared with the correspondent one of the second. Note that, in order to form a Galois connection between the abstraction and concretization functions, we cannot compare instance variables which have H as their initial information level. In fact, since the concretization function simply discards multisets and levels, and since the abstraction function sets to h the information level of the values contained in the instance variables ℓ_i having $\Phi_i = H$, we could have the following problem. Consider the term:

$$t = \nu p \mapsto [a^H = \varsigma(x)5, b^L = \varsigma(x)2] \Rsh (\text{ let } y = (p.a \Leftarrow 2) \text{ in let } z = p.a \text{ in } p.b \Leftarrow z)$$

The corresponding abstract one is:

$$\begin{aligned} \alpha(t) = \llbracket \nu p.\ \ & p \mapsto [a^H = \varsigma(x)\odot^h, b^L = \varsigma(x)\odot^l] \Rsh \\ & (\text{let } y = (p^l.a \Leftarrow \odot^l) \text{ in let } z = p^l.a \text{ in } p^l.b \Leftarrow z) \rrbracket^{l, \emptyset} \end{aligned}$$

which reduces to:

$$\begin{array}{lcl} \alpha(t) & \longrightarrow^{\sharp *} & \nu p.\ \ p \mapsto [a^H = \varsigma(x)\odot^h, b^L = \varsigma(x)\odot^l] \Rsh \\ & & \qquad (\text{let } y = [\![p^l.a \Leftarrow \odot^l]\!]^{l,\emptyset} \text{ in } [\![\text{ let } z = p^l.a \text{ in } p^l.b \Leftarrow z]\!]^{l,\emptyset}) \\ & & \\ & \longrightarrow^{\sharp} & \\ s^\sharp & = & \nu p.\ \ p \mapsto [a^H = \varsigma(x)\odot^l, b^L = \varsigma(x)\odot^l] \Rsh \\ & & \qquad ([\![\text{ let } z = p^l.a \text{ in } p^l.b \Leftarrow z]\!]^{l,\emptyset}) \end{array}$$

Note that, at this point, the value stored in the variable $p.a$ has a *low* level of information, since the variable $p.a$ has just been assigned using a constant which starts with a low level of information. Since the abstraction and the concretization functions must form a Galois connection, we must have $\alpha(\gamma(a^\sharp)) \sqsubseteq a^\sharp$ for each abstract term $a^\sharp$. Using the concretization function on this last term we obtain the following (set of) concrete term(s):

$$\gamma(s^\sharp) = \nu p.p \mapsto [a^H = \varsigma(x)n, b^L = \varsigma(x)m] \Rsh (\text{let } z = p.a \text{ in } p.b \Leftarrow z)$$

where n and m represent generic integers. By abstracting again we obtain:

$$\begin{array}{rcl} \alpha(\gamma(s^\sharp)) & = & [\![\nu p.p \mapsto [a^H = \varsigma(x)\odot^h, b^L = \varsigma(x)\odot^l] \Rsh (\text{let } z = p^l.a \text{ in } p^l.b \Leftarrow z)]\!]^{l,\emptyset} \\ & \equiv & \nu p.p \mapsto [a^H = \varsigma(x)\odot^h, b^L = \varsigma(x)\odot^l] \Rsh ([\![\text{ let } z = p^l.a \text{ in } p^l.b \Leftarrow z]\!]^{l,\emptyset}) \end{array}$$

and in order to obtain $\alpha(\gamma(s^\sharp)) \sqsubseteq s^\sharp$ we have that the value inside the instance variable $p.a$, which has initial information level H, must *not* be compared between the two abstract terms. This explains the somehow strange premise of the fourth rule in Table 6.10. The other rules in the table are straightforward extensions of the base cases here explained.

Given this ordering relation, the least upper bound operator $\sqcup$ of the abstract domain computes the union between multisets and the maximum between corresponding information levels (note that this is not required for instance variables such that $\Phi_i = H$ inside denotations, as said above, but it is not incorrect). The greatest lower bound operator, $\sqcap$, on the other hand, computes the intersection between multisets and the minimum between corresponding information levels. As usual, the abstract domain A is extended with a bottom and a top element $\bot^\sharp$ and $\top^\sharp$ (the top element is also used in the abstract syntax).

In the following we present some results about correctness of this abstract interpretation. They are analogous to the ones presented in the previous chapters, but the proofs are slightly different, since the abstract interpretation changed.

Proposition 6.4.1. *Let us consider two abstract terms $t_1^\sharp$ and $t_2^\sharp$ such that $t_1^\sharp \sqsubseteq t_2^\sharp$. If both $t_1^\sharp$ and $t_2^\sharp$ are different from $\bot^\sharp$ and $\top^\sharp$, we can conclude that $\gamma(t_1^\sharp) = \gamma(t_2^\sharp)$.*

Proof: The proof is done by induction on the rules of Table 6.10. For the base case we must consider the rule:

$$u^\varphi \sqsubseteq u^\psi \Leftrightarrow \varphi \leq \psi$$

We can have two forms of the abstract result u^φ: $\odot^\varphi$ and p^φ. In both cases, using the definition of the concretization function, we have:

$$\gamma(\odot^\varphi) = \{n \mid n \text{ is an integer number}\} = \gamma(\odot^\psi)$$

$$\gamma(p^\varphi) = \{p\} = \gamma(p^\psi)$$

Many of the other rules allow to immediately apply the inductive hypothesis and then the correspondent proofs are straightforward. We report here only the more complex ones. Consider the rule for denotations:

$$\begin{gathered}\forall i.\ (s_i^\sharp \sqsubseteq t_i^\sharp) \vee (\Phi^i = H \wedge s_i^\sharp = u^\varphi \wedge t_i^\sharp = u^\psi) \Rightarrow \\ \left[\ell_i^{\Phi_i} = \varsigma(x_i)s_i^\sharp \ {}^{i\in(1\ldots n)}\right] \sqsubseteq \left[\ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ {}^{i\in(1\ldots n)}\right]\end{gathered}$$

By applying the concretization function we obtain:

$$\gamma\left(\left[\ell_i^{\Phi_i} = \varsigma(x_i)s_i^\sharp \ {}^{i\in(1\ldots n)}\right]\right) = \left\{\left[\ell_i^{\Phi_i} = \varsigma(x_i)s_i \ {}^{i\in(1\ldots n)}\right] \mid s_i \in \gamma_t(s_i^\sharp)\right\}$$

$$\gamma\left(\left[\ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \ {}^{i\in(1\ldots n)}\right]\right) = \left\{\left[\ell_i^{\Phi_i} = \varsigma(x_i)t_i \ {}^{i\in(1\ldots n)}\right] \mid t_i \in \gamma_t(t_i^\sharp)\right\}$$

Now, using the premise of the rule we have that, for some values of i, $s_i^\sharp \sqsubseteq t_i^\sharp$. For these values of i we can use immediately the inductive hypothesis to conclude that $\gamma_t(s_i^\sharp) = \gamma_t(t_i^\sharp)$. For the other values of i, we have that $s_i^\sharp = u^\varphi$ and $t_i^\sharp = u^\psi$, then using the concretization function we have $\gamma_t(s_i^\sharp) = \gamma_t(u^\varphi)$ and $\gamma_t(t_i^\sharp) = \gamma_t(u^\psi)$. Finally, if $u = p$ we have $\gamma_t(u^\varphi) = \{p\} = \gamma_t(u^\psi)$, while if $u = \odot$ we have $\gamma_t(u^\varphi) = \mathbb{Z} = \gamma_t(u^\psi)$, where $\mathbb{Z}$ is the set containing all integer numbers.

Finally, consider the last rule:

$$(t^\sharp \sqsubseteq t_1^\sharp) \wedge (\varphi \leq \psi) \wedge (S \subseteq S_1) \Rightarrow [\![t^\sharp]\!]^{\varphi,S} \sqsubseteq [\![t_1^\sharp]\!]^{\psi,S_1}$$

Here we have that $\gamma_t([\![t^\sharp]\!]^{\varphi,S}) = \gamma_t(t^\sharp)$, and $\gamma_t([\![s^\sharp]\!]^{\varphi,S}) = \gamma_t(s^\sharp)$. Then we can immediately apply the inductive hypothesis to conclude the proof.

□

Proposition 6.4.2. *Let $s^\sharp$ be an abstract term, and $S \in \wp(C)$ be a set of concrete terms. α and γ form a Galois connection between the two domains $\wp(C)$ and A. That is:*

- *α and γ are monotonic,*
- *$S \subseteq \gamma(\alpha(S))$, where α and γ are applied pointwise,*
- *$\alpha(\gamma(t^\sharp)) \sqsubseteq t^\sharp$.*

Proof: As done in chapter 4, we have to consider the abuse of notation used on the abstraction function α_p. In fact, since the abstraction function must take as arguments *sets* of concrete values, we have to consider in our proof the following abstraction function:

$$\alpha^{Set}(S) = \bigsqcup_{c_i \in S} \alpha(c_i)$$

Now, all the proofs have to be done for the couple of functions α^{Set} and γ.

- **Monotonicity**: For the concretization function γ, we have that the premise of the monotonicity condition:

  $$t_1^\sharp \sqsubseteq t_2^\sharp \Rightarrow \gamma(t_1^\sharp) \subseteq \gamma(t_2^\sharp)$$

  is verified only for $t_1^\sharp = \bot^\sharp$ or $t_2^\sharp = \top^\sharp$, and for the cases treated in Table 6.10. We have already that $\gamma(\top^\sharp) = C$, so that if $t_2^\sharp = \top^\sharp$ we have obviously $\gamma(t_1^\sharp) \subseteq \gamma(t_2^\sharp) = C$ for all abstract terms $t_1^\sharp$. When $t_1^\sharp = \bot^\sharp$, it is sufficient to extend γ so that $\gamma(\bot^\sharp) = \emptyset$ in order to have $\gamma(t_1^\sharp) = \gamma(\bot^\sharp) \subseteq \gamma(t_2^\sharp)$ for all abstract terms $t_2^\sharp$. Finally, the result of proposition 6.4.1 covers the cases of Table 6.10.

  For the abstraction function, α^{Set}, since we have that $\alpha^{Set}(S) = \bigsqcup_{c_i \in S} \alpha(c_i)$, we can conclude that:

  $$\begin{array}{rl} & S_1 \subseteq S_2 \\ \Rightarrow & S_2 = S_1 \cup S_3 \\ \Rightarrow & \alpha_p^{Set}(S_1) \sqsubseteq \alpha^{Set}(S_1) \sqcup \alpha^{Set}(S_3) = \alpha^{Set}(S_1 \cup S_3) = \alpha^{Set}(S_2) \end{array}$$

- $\mathbf{S} \subseteq \gamma(\alpha^{\mathbf{Set}}(\mathbf{S}))$: let us split this proof in three cases:

  - $\alpha^{\mathbf{Set}}(\mathbf{S}) = \bot^\sharp$: this case is trivial, since $\alpha^{Set}(S)$ computes the least upper bound of the abstractions of all statements in S. This upper bound results to be $\bot^\sharp$, then S must be empty. Then: $\gamma(\alpha^{Set}(S)) = \gamma(\bot^\sharp) = \emptyset$ and $S = \emptyset \subseteq \emptyset$.
  - $\alpha^{\mathbf{Set}}(\mathbf{S}) = \top^\sharp$: this case is trivial, since $\gamma(\alpha^{Set}(S)) = \gamma(\top^\sharp) = C$ and for all S in $\wp(C)$ we have that $S \subseteq C$.
  - $\bot^\sharp \sqsubset \alpha^{\mathbf{Set}}(\mathbf{S}) = \mathbf{t}^\sharp \sqsubset \top^\sharp$: in this case, consider again that the α^{Set} function computes the least upper bound of the abstractions of the elements of S. Now, the only cases where $t_1^\sharp \sqcup t_2^\sharp = t^\sharp \neq \top^\sharp$ are the ones following the definitions of Table 6.10. Now, we have that the abstraction function sets the information level of all constants inside the term to l. This can be easily seen from the definition of α:

    $$\alpha(t) = [\![\alpha_t(t, l)]\!]^{l,\emptyset}$$

and from the structure of the function α_t, which passes the level l in the successive calls to the subterms of t. The only information levels which can be set to h are those of the instance variables with initial information level H. Again, this can be easily seen from the definition of the function α_d where instance variables are abstracted using the information level $\mathcal{T}(H) = h$:

$$\alpha_d([\ell_i^{\Phi_i} = \varsigma(x_i)t_i \ ^{i\in 1\dots n}], \varphi) = [\ell_i^{\Phi_i} = \varsigma(x_i)\alpha_t(t_i, \mathcal{T}(\Phi_i)) \ ^{i\in 1\dots n}]$$

Moreover, from the rule $\alpha(t) = [\![\alpha_t(t,l)]\!]^{l,\emptyset}$ we can also say that the initial level of knowledge of the abstract term is always set to l, and the initial multiset of method calls is always set to $\emptyset$. From these conclusions, and from the fact that for all $t_i \in S$ we have that all the $\alpha(t_i) = t_i^\sharp$ are in relation according to Table 6.10, we have that all $t_i^\sharp$ share the same structure of subterms and the same initial information levels of correspondent objects. This implies that:

$$\forall t_i, t_j \in S.\ \alpha(t_i) = \alpha(t_j) = t^\sharp$$

Now, we have that the definition of γ is such that, for each concrete term t, $t \in \gamma(\alpha(t))$. Then we have that:

$$\forall t_i \in S.\ t_i \in \gamma(\alpha(t_i)) = \gamma(t^\sharp)$$

and thus we can conclude $S \subseteq \gamma(t^\sharp) = \gamma(\alpha^{Set}(S))$.

- $\alpha^{\mathbf{Set}}(\gamma(\mathbf{t}^\sharp)) \sqsubseteq \mathbf{t}^\sharp$: let us split this proof in three cases:
  - $\mathbf{t}^\sharp = \bot^\sharp$: this case is trivial, since we have that:

    $$\alpha^{Set}(\gamma(t^\sharp)) = \alpha^{Set}(\gamma(\bot^\sharp)) = \alpha^{Set}(\emptyset) = \bot^\sharp \sqsubseteq t^\sharp$$

  - $\mathbf{t}^\sharp = \top^\sharp$: this case is trivial, since we have that:

    $$\alpha^{Set}(\gamma(t^\sharp)) = \alpha^{Set}(\gamma(\top^\sharp)) = \alpha^{Set}(C) = \top^\sharp \sqsubseteq t^\sharp$$

  - $(\mathbf{t}^\sharp \neq \top^\sharp) \wedge (\mathbf{t}^\sharp \neq \bot^\sharp)$: in this case, the cases of the definitions of α and γ are such that, given the abstract statement $t^\sharp$, for all concrete statements $t \in \gamma(t^\sharp)$, we have $\alpha(t) \sqsubseteq t^\sharp$. This because the γ function discards all the multisets of method calls and all information levels of values, while the α function rebuilds empty multisets and sets information levels of constants to l (we recall that the values inside instance variables with initial information level H are compared only according to their type, and not according to their level). Then the two abstract terms $\alpha(t)$ and $t^\sharp$ are in relation according to Table 6.10. Thus we can conclude that $\alpha^{Set}(\gamma(t^\sharp)) \sqsubseteq t^\sharp$.

□

Proposition 6.4.3. *Let t_1 and t_2 be concrete terms, if $t_1 \longrightarrow^n t_2$ then there exists an abstract term $t^\sharp$ such that $\alpha(t_1) \longrightarrow^{\sharp m} t^\sharp$ and $t_2 \in \gamma(t^\sharp)$. $\longrightarrow$ and $\longrightarrow^\sharp$ include the congruence rule applications which make possible the reduction. $\longrightarrow^n$ denotes then a sequence of applications of n consecutive reduction rules and congruences.*

Proof: The proof is by induction on n. For the basis of the induction, $n = 1$, let us consider all the different cases, according to the possible couples of concrete statements t_1 and t_2 taken from the concrete reduction rules, and from the concrete congruence definition:

- $t_1 = p.\ell_j; t_2 = t_j\{\{x_j \leftarrow p\}\}$:

  This case comes from the concrete (Red invoke) rule. Here, we have

  $$\alpha(t_1) = [\![p^l.\ell_j]\!]^{l,\emptyset}$$

  Here we can use the rule (Red invoke$^{\sharp 1}$) to obtain

  $$t^\sharp = [\![\alpha_t(t_j)\{\{x_j \leftarrow p^l\}\}]\!]^{l,\{(p.\ell_j,l)\}}$$

  From this, by applying the concretization function we have

  $$\gamma(t^\sharp) = \{t |\ t \in \gamma(\alpha_t(t_j)\{\{x_j \leftarrow p\}\})\}$$

  Then we have clearly $t_2 \in \gamma(t^\sharp)$, directly from the fact that α and γ form a Galois connection.

- $t_1 = p.\ell_j \Leftarrow \varsigma(x)u; t_2 = p$:

  This case comes from the concrete (Red update) rule. Here

  $$\alpha(t_1) = [\![p^l.\ell_j \Leftarrow \varsigma(x)\alpha_r(u,l)]\!]^{l,\emptyset}$$

  and using the (Red update$^\sharp$) rule we obtain

  $$t^\sharp = [\![p^l]\!]^{l,\emptyset}$$

  Using the concretization function we have

  $$\gamma(t^\sharp) = \{p\}$$

  and clearly $t_2 \in \{p\}$.

- $t_1 = \text{let } x = u \text{ in } t; t_2 = t\{\{x \leftarrow u\}\}$:

  This case comes from the concrete (Red let) rule. Here

  $$\alpha(t_1) = [\![\text{let } x = \alpha_r(u, l) \text{ in } \alpha_t(t, l)]\!]^{l,\emptyset}$$

  and using an abstract structural congruence step we obtain

  $$\text{let } x = [\![u^l]\!]^{l,\emptyset} \text{ in } [\![\alpha_t(t, l)]\!]^{l,\emptyset}$$

  Using the rule (Red let$^\sharp$) we can reduce this term to

  $$t^\sharp = [\![\alpha_t(t, l)\{\{x \leftarrow u^l\}\}]\!]^{l,\emptyset}$$

  Finally, using the concretization function we have

  $$\gamma(t^\sharp) = \{t|\ t \in \gamma_t(\alpha_t(t, l)\{\{x \leftarrow u\}\})\}$$

  Again, the conclusion $t_2 \in \gamma(t^\sharp)$ follows from the Galois connection.

- $t_1 = \text{if } 0 \text{ then } s \text{ else } t; t_2 = t$

  This case comes from the concrete (Red if0) rule. Now

  $$\alpha(t_1) = [\![\text{if } \odot^l \text{ then } \alpha_t(s, l) \text{ else } \alpha_t(t, l)]\!]^{l,\emptyset}$$

  and, after an abstract congruence step we obtain the following abstract term

  $$\text{if } [\![\ \odot^l\ ]\!]^{l,\emptyset} \text{ then } [\![\alpha_t(s, l)]\!]^{l,\emptyset} \text{ else } [\![\alpha_t(t, l)]\!]^{l,\emptyset}$$

  At this point we have two possible rules to apply: (Red if$^{\sharp 1}$) and (Red if$^\sharp$2). By choosing the rule (Red if$^\sharp$1) we obtain

  $$t^\sharp = [\![\alpha_t(t, l)]\!]^{l,\emptyset}$$

  and using the concretization function we obtain

  $$\gamma(t^\sharp) = \gamma_t(\alpha_t(t, l))$$

  Again, since α and γ form a Galois connection, we obtain the conclusion $t_2 \in \gamma(\alpha(t, l))$. This case is exactly the same as the one for the other branch of the *if* construct, so we will not show that case.

- $t_1 = s \Rsh \mathcal{E}[t];\ t_2 = \mathcal{E}[s \Rsh t]$ where $fn(s) \cap bn(\mathcal{E}) = \emptyset$:

  This case comes from the first concrete congruence rule. Here

  $$\alpha(t_1) = [\![\alpha_t(s, l) \Rsh \alpha_t(\mathcal{E}[t], l)]\!]^{l,\emptyset} \equiv [\![\alpha_t(s, l)]\!]^{l,\emptyset} \Rsh [\![\alpha_t(\mathcal{E}[t], l)]\!]^{l,\emptyset}$$

Now we have that $\mathcal{E}$ is a term with an hole, so that its abstraction produces an abstract term with the same hole (the abstraction function should be opportunely extended, but it is straightforward). This because the α_t function simply replaces each integer value with $\odot$ and adds the l security level to each result. So we can write

$$\alpha_t(\mathcal{E}[t], l) = \alpha_t(\mathcal{E}, l)[\alpha_t(t, l)]$$

where $\alpha_t(\mathcal{E}, l) = \mathcal{C}$ is an abstract context identical to $\mathcal{E}$, with the only exception of integer values collapsed to $\odot$ and the l level attached to each result inside $\mathcal{C}$. Now we can use one of the abstract congruence rules of Table 6.8 to distribute the $[\![\ ]\!]^{l,\emptyset}$ structure inside the $\mathcal{C}$ context, obtaining

$$\alpha(t_1) \equiv [\![\alpha_t(s, l)]\!]^{l,\emptyset} \upharpoonright \mathcal{C}'[[\![\alpha_t(t, l)]\!]^{l,\emptyset}]$$

where $\mathcal{C}'$ is identical to $\mathcal{C}$, apart from the fact that its subterms are surrounded by $[\![\ ]\!]^{l,\emptyset}$. Then $\mathcal{C}'$ has the same names of $\mathcal{E}$, and $\alpha_t(s, l)$ has the same names of s, thus we can use the first abstract congruence rule from Table 6.8 to obtain

$$\alpha(t_1) \equiv \mathcal{C}'[[\![\alpha_t(s, l)]\!]^{l,\emptyset} \upharpoonright [\![\alpha_t(t, l)]\!]^{l,\emptyset}] = t^\sharp$$

Now, since the concretization function discards the levels of knowledge and the multisets of method calls, we obtain

$$\gamma(t^\sharp) = \{\mathcal{F}[x \upharpoonright y] \mid \mathcal{F} \in \gamma(\mathcal{C}') = \gamma(\alpha_t(\mathcal{E}, l)),\ x \in \gamma(\alpha_t(s, l)),\ y \in \gamma(\alpha_t(t, l))\}$$

and $t_2 \in \gamma(t^\sharp)$ comes again from the Galois connection between α and γ. The other side of this congruence rule is very similar to this one, so we do not show it.

- $t_1 = (\nu p)\mathcal{E}[s];\ t_2 = \mathcal{E}[(\nu p)s]$ where $p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})$:

  This case comes from the second concrete congruence rule. In this case we can use the same reasoning as before, obtaining

$$\alpha(t_1) = [\![(\nu p)\alpha_t(\mathcal{E}[s], l)]\!]^{l,\emptyset} = [\![(\nu p)\alpha_t(\mathcal{E}, l)[\alpha_t(s, l)]]\!]^{l,\emptyset} \equiv (\nu p)[\![\alpha_t(\mathcal{E}, l)[\alpha_t(s, l)]]\!]^{l,\emptyset}$$

  where this last equivalence was obtained by applying the abstract congruence rule for restriction. Note that, according to the rule, we should change the name p with a fresh name, but since there is no possibility of capture (S is empty), we did not rename it. Now, using again the same reasoning as before, we can obtain

$$\alpha(t_1) \equiv (\nu p)\mathcal{C}'[[\![\alpha_t(s, l)]\!]^{l,\emptyset}] \equiv \mathcal{C}'[(\nu p)[\![\alpha_t(s, l)]\!]^{l,\emptyset}] = t^\sharp$$

  where $\alpha_t(\mathcal{E}, l) = \mathcal{C}$ and $\mathcal{C}'$ is obtained from $\mathcal{C}$ by distributing the level of knowledge l and the multiset of method calls on the subterms. The last equivalence was obtained by applying the abstract congruence rule for restriction. Now using the concretization function on $t^\sharp$ we obtain:

$$\gamma(t^\sharp) = \{\mathcal{F}[(\nu p)x] \mid \mathcal{F} \in \gamma(\mathcal{C}') = \gamma(\mathcal{C}) = \gamma(\alpha_t(\mathcal{E}, l)),\ x \in \gamma(\alpha_t(s, l))\}$$

and again $t_2 \in \gamma(t^\sharp)$ comes from the Galois connection result.

We should continue with the inductive cases by supposing the proposition true for $\longrightarrow^{n-1}$. All the cases are identical to the ones shown before, with the only difference that we have a multiset S of method calls, instead of an empty multiset, and that the information levels may increase or decrease during the evaluation. Since the concretization function simply discards both the multisets and the information levels, we have that the inductive cases for the above rules are almost the same as above, and may be omitted.

The only cases which remain, for the inductive step, are the ones concerning the (Red context) concrete rule, as well as the (Red invoke) concrete rule, when we have a non-empty multiset of method calls (and then the (Red invoke$^{\sharp 2}$) rule may be triggered).

- $t_1 = \mathcal{E}[r];\ t_2 = \mathcal{E}[s]$ where $r \longrightarrow s$:

  This case comes from the (Red context) rule. Here, we have that the (Red context) rule allows to conclude $\mathcal{E}[r] \longrightarrow \mathcal{E}[s]$ for any concrete context, starting from the $r \longrightarrow s$ hypothesis. We can use the inductive hypothesis on $r \longrightarrow s$, to conclude that

  $$\alpha(r) \longrightarrow^{\sharp m} s^\sharp \quad , \quad s \in \gamma(s^\sharp)$$

  Now let us apply the abstraction function on $\mathcal{E}[r]$ obtaining (using the same reasoning as before)

  $$\alpha(\mathcal{E}[r]) = [\![\alpha_t(\mathcal{E}[r], l)]\!]^{l,\emptyset} = [\![\alpha_t(\mathcal{E}, l)[\alpha_t(r, l)]]\!]^{l,\emptyset} = [\![\mathcal{C}[\alpha_t(r, l)]]\!]^{l,\emptyset} \equiv \mathcal{C}'[[\![\alpha_t(r, l)]\!]^{l,\emptyset}]$$

  Now, since $\alpha(r) = [\![\alpha_t(r, l)]\!]^{l,\emptyset}$, we can apply the (Red context$^\sharp$) rule on our inductive hypothesis, obtaining

  $$\alpha(\mathcal{E}[r]) \longrightarrow^\sharp \mathcal{C}'[s^\sharp] = t^\sharp \quad , \quad s \in \gamma(s^\sharp)$$

  Finally, we can apply the γ function on $t^\sharp$ obtaining

  $$\gamma(t^\sharp) = \{\mathcal{F}[x] \mid \mathcal{F} \in \gamma(\mathcal{C}') = \gamma(\mathcal{C}) = \gamma(\alpha_t(\mathcal{E}, l)),\ x \in \gamma(s^\sharp)\}$$

  and $t_2 = \mathcal{E}[s] \in \gamma(t^\sharp)$ comes from the inductive hypothesis ($s \in \gamma(s^\sharp)$) and from the Galois connection between α and γ ($\mathcal{E} \in \gamma(\alpha_t(\mathcal{E}, l))$).

- $t_1 = p.\ell_j; t_2 = t_j\{\{x_j \leftarrow p\}\}$:

  This case comes from the (Red invoke) rule. In particular we are interested in the case concerning the abstract rule (Red invoke$^{\sharp 2}$). Let us suppose, that from an initial statement r we reached after $n-1$ steps a statement s, having $r \longrightarrow^{n-1} s$. Then by inductive hypothesis, we have that there exists an abstract statement $s^\sharp$ such that $\alpha(r) \longrightarrow^{\sharp m} s^\sharp$ and $s \in \gamma(s^\sharp)$. Now we consider $s = t_1 = p.\ell_j$ and reduce it to $t_2 = t_j\{\{x_j \leftarrow p\}\}$. On the abstract side, we consider only the (Red invoke$^\sharp$2) rule since, as already said, the case corresponding to the (Red invoke$^\sharp$1) rule is identical as before. We have, from the inductive hypothesis, that $s \in \gamma_s(s^\sharp)$. We can have two cases for $s^\sharp$:

– $\mathbf{s}^\sharp = \top^\sharp$: in this case the abstract computation remains blocked since, according to the (Red context$^\sharp$) rule, we have $\top^\sharp \longrightarrow^\sharp \top^\sharp$. Then this case is trivial, since we have $\alpha(r) \longrightarrow^{\sharp m} s^\sharp \longrightarrow \top^\sharp$ and since $t \in \gamma(\top^\sharp) = C$ for each concrete term t (then also for $t_2 = t_j\{\{x_j \leftarrow p\}\}$).

– $\mathbf{s}^\sharp \neq \top^\sharp$: in this case, the fact that $s \in \gamma(s^\sharp)$ implies that $\{s\} \subseteq \gamma(s^\sharp)$. Thus, using the fact that α never returns $\perp^\sharp$, the monotonicity of α, the Galois connection between α and γ and the hypothesis $s^\sharp \neq \top^\sharp$ we obtain:

$$\perp^\sharp \sqsubset \alpha^{Set}(\{s\}) \sqsubseteq \alpha^{Set}(\gamma(s^\sharp)) \sqsubseteq s^\sharp \sqsubset \top^\sharp$$

This implies that $\alpha^{Set}(\{s\})$ and $s^\sharp$ are in relation according to Table 6.10, and then they differ only in the method call multisets attached to environments and in their information levels. So we can write $s^\sharp = [\![p^\varphi.\ell_j]\!]^{\psi,S}$ and by applying rule (Red invoke$^{\sharp 2}$) we obtain that

$$[\![p^\varphi.\ell_j]\!]^{\psi,S} \longrightarrow^\sharp \top^\sharp$$

so that we can conclude that $t_2 \in \gamma(\top^\sharp)$ since $\gamma(\top^\sharp) = C$.

□

This proposition, as in previous chapters states that the abstract reduction correctly approximates the concrete one, so that every concrete computation has a corresponding abstract one, and if a property is verified for all the reductions of an abstract term $\alpha(t)$ then it is verified also for t.

Proposition 6.4.4. *Given a term t, the abstract interpretation process of $\alpha(t)$ always terminates.*

Proof: The proof is the same as the one seen in chapter 4.

□

6.5 Secure information flow analysis

In this section we define our notion of non-interference. We said before that in the most general case a program is non-interferent when at the end of its execution the variables with initial level of information L *do not depend* from the variables with initial level of information H. Actually, the notion of non-interference depends from what can be observed during or after the execution of a program. As an example, if we agreed that non-termination could be observable, we would have that also those programs which do not terminate because of loops controlled by high variables should be considered as non-secure.

Our notion of non-interference is limited to the contents of instance variables inside objects. Moreover we suppose that we can not observe values during the execution of a program, and thus we are only interested in what the initially low variables contain at the end of a program. Using these hypothesis we can define in the following our concrete and abstract notions for non-interference.

First of all, we need to define two notions of equivalence for concrete terms. The first, stronger, notion, considers two concrete terms as equivalent only if they are completely coincident, except for the values of high level variables. So, in this first notion we consider also the structure of the term in the equivalence. The second notion of equivalence is a less demanding notion which considers two terms as equivalent when the values contained in their initially low-level variables coincide, without considering, now, the structure of terms.

Definition 6.5.1 (strong low-equivalence for concrete terms)**.** *Let s and t be two concrete terms. We have a strong low-equivalence between s and t, written $s =_l t$, when s and t differ only in the values of the instance variables of initial level H contained inside their objects.*

Definition 6.5.2 (light low-equivalence for concrete terms)**.** *Let s and t be two concrete terms. We have a light low-equivalence between s and t, written $s \approx_l t$ when the denotations appearing in s and t differ only in the values of the instance variables of initial level H.*

In the following definition, we use the symbol $\not\longrightarrow$ to represent the absence of a possible reduction for a term. So, if we have a term t such that $t \not\longrightarrow$ we consider to have a terminal configuration, since t cannot be further reduced.

Definition 6.5.3 (Concrete non-interference)**.** *Let t be a concrete term. t is said to be non-interferent iff for each concrete term s such that $s =_l t$, we have that if both s and t reach termination, that is $s \longrightarrow^* s' \not\longrightarrow$ and $t \longrightarrow^* t' \not\longrightarrow$, then $s' \approx_l t'$.*

Definition 6.5.4 (Abstract non-interference)**.** *Let $t^\sharp$ be an abstract term, and let us consider all the denotations occurring in it: $p_1 \mapsto d_1^\sharp, \ldots, p_n \mapsto d_n^\sharp$. The term $t^\sharp$ is non-interferent if for all terminating reductions in the abstract reduction graph,* *the final configuration is such that for all denotations* $d_i^\sharp = [\ell_j^{\Phi_j} = \varsigma(x_j)t_j^\sharp \ {}^{j\in(1\ldots m)}]$ *we have that:*

$$\nexists j.\ \ell_j^L = \varsigma(x_j)u^h$$

From the point of view of the concrete calculus, we have that the above abstract notion of non-interference, together with the abstract semantics rules shown in Table 6.9, allow to check the concrete security of terms with respect to direct and indirect flows of information, as well as flows caused by synchronizations among parallel threads. This result comes directly from the structure of the abstract reduction rules and from the definition of the abstraction function. In fact, as already said in section 6.3, we have that the abstraction function sets to h the information level for all constants inside instance variables with initial level H, while it sets to l the information level of all other constants. Moreover, the initial term is extended with an empty multiset of method calls and with l as its initial level of knowledge.

From the rules of Table 6.9 we can see that the level of knowledge of terms increases only in three cases:

- When the second subterm of a *let* term has to be reduced after a recursion with high level of knowledge
- When one of the branches of an *if* term has to be reduced:
  - after a recursion with high level of knowledge
  - after the evaluation of the first subterm of the *if*, which returns an high level value (this can derive both from the level of the value itself, or from another high level of knowledge reached during the evaluation of the guard subterm)

Since recursion is the only way to have a loop in this calculus, we have that the first two cases capture the notion of insecure information flow due to synchronizations performed having knowledge of some value with level of information h. The third case, instead, captures the cases of indirect information flows, since all assignments performed inside the branch will be evaluated with an high level of knowledge.

Direct flows of information, instead, are captured by the rules (Red update$^\sharp$) and (Red let$^\sharp$). The former rule updates an instance variable obtaining the new information level by the level of the new value, the level of knowledge of the term which performs the update (this captures also indirect flows) and the level of information of the reference to the updated object. This level has to be considered since this reference may be returned as a result from a previous term, and then can represent a possible source of information flows. The latter rule, finally, captures direct flows since it replaces variables inside the second subterm using the information levels of the result of the first subterm (both the level of the result itself and the level of knowledge of the term which produces the result).

Using the same reasoning done in previous chapters, we have also that truncated method calls do not add any information to the abstract reduction graph, and then the analyzed information flows may be detected using only the first two calls of recursive methods.

The above informal reasoning may be formalized in the following proposition:

Proposition 6.5.1. *Given a term t, if $\alpha(t) = t^\sharp$ is an abstract non-interferent term, then t is a concrete non-interferent term.*

Proof: We can suppose that all variables having initial level H contain an integer value. This is useful for two different reasons:

- Usually private values are not references (which are subject to dynamic relocation in real programming languages), but sensible constants which are independent from the program itself
- From the abstraction point of view, we have that the abstraction function leaves the references as they are, so if we had two programs which differ in a reference assigned to an instance variable, those two programs should *not* be strongly low-equivalent as they, instead, are. This because the two programs may have completely different reduction graphs, since references are used to call methods, and then can influence the behavior of programs.

Given the concrete term t, we have that for all other terms s such that $s =_l t$, $\alpha(s) = \alpha(t)$. This is true because of the previous hypothesis, and because the abstraction function collapses all integer values to the uniqe abstract value $\odot$. So, since the abstract interpretation is correct, we have that all reduction paths of s and t are represented by some abstract reduction paths in the reduction graph of $\alpha(t)$.

Let us prove a statement which is slightly different from the one above, but implies our correctness result. We want to prove that:

Given s and t such that $s =_l t$ and an abstract reduction path

$$\alpha(s) = \alpha(t) = t^\sharp \longrightarrow t_1^\sharp \longrightarrow \ldots \longrightarrow t_n^\sharp$$

of length n, if a result $u^\sharp$ has information level equal to l in the term $t_n^\sharp$, then for all concrete reduction paths

$$t \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n$$

$$s \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_n$$

such that $\forall i.\ s_i, t_i \in \gamma(t_i^\sharp)$, the corresponding concrete result u has the same value both in s_n and in t_n.

If this statement is valid, we can prove our proposition by applying the same reasoning to all abstract reduction paths and to all the results contained in the low instance variables occurring in t (and in s, since $s =_l t$). The information level of an abstract result $[\![u^\varphi]\!]^{\psi,S}$ can be obtained as $\varphi \sqcup \psi$ (as in the abstract rule (Red res$^\sharp$)).

In order to prove our statemente we will use an induction on the number of steps of the abstract reduction path $t^\sharp \longrightarrow t_1^\sharp \longrightarrow \ldots \longrightarrow t_n^\sharp$. Consider that the $\longrightarrow$ and $\longrightarrow^\sharp$ relations include the congruences, so the concrete paths may also have equal terms one after another.

The base cases of our induction concern those abstract reductions of length one. All cases are quite straightforward to prove. In fact, for all the abstract congruence rules, they only distribute the (initially low) level of knowledge to the subterms. Since the level of knowledge is initially low and since it is only distributed, it does not contribute to increase any information level. Moreover the abstract congruences do not change any result, so the results after the congruence are equal to the results before the congruence. Finally, since the initial concrete terms are such that $s =_l t$, all concrete results are coincident between s and t, apart the ones assigned to initially high instance variables (not consideret by the statement, since their abstract information level is h). We can apply the same reasoning also to all the reduction rules. In fact, the initial level of all constants occurring in the initial abstract term $t^\sharp$ is always l (because α distributes the initial l level to all subterms). None of the abstract rules, using only one reduction step, is able to modify any result and assign to it a high value. This because in order to access a high value we need a method call (so at least two steps, the first one for the call, the second one to modify a result). In fact high values are only contained in instance variables. Another way to have a high value is to have a term with high level of knowledge, but this is impossible at the beginning of the program, since the initial level of knowledge is set to l by the abstraction function. Then, all abstract results which have a low information level after one abstract reduction step are identical to the results occurring in the initial program. Finally, since initially we have $s =_l t$, we can easily conclude the equality between the corresponding concrete results.

For the inductive cases, we can use the inductive hypothesis on the reductions of length n, and prove our result for the $(n+1)$-th step of reduction or abstract structural congruence. For the congruences, the reasoning is exactly the same as above. In fact, since the congruences do not modify any results, we have that all results are identical to the previous reduction step, and thus the inductive hypothesis guarantees the equality between the concrete results. For the reduction rules, we have the following cases:

- (Red invoke$^{\sharp 1}$): In this case the only part of the abstract term which changes is the method call, which is replaced by the body of the method, with the usual binding. In fact, according to the rule, we have

$$p \mapsto d^\sharp \upharpoonright [\![p^\varphi.\ell_j]\!]^{\psi,S} \longrightarrow^\sharp p \mapsto d^\sharp \upharpoonright [\![t_j^\sharp\{\{x_j \leftarrow p^\varphi\}\}]\!]^{\psi,S \uplus \{(p.\ell_j,\psi)\}}$$

  Since the term $t_j^\sharp$ is taken from a denotation existing in the last reduction step, we have that the results appearing in the current abstract term are exactly the same as the ones appearing in the last reduction step (the level of information ψ does not influence the proof, since if it is high and if $t_j^\sharp$ is a result, we will ignore this new term in our statement). Thus we can use the inductive hypothesis on the denotation containing the method body and on the p^φ reference, to prove our statement in this case.

- (Red invoke$^{\sharp 2}$): This case is trivial, since the method call $p^\varphi.\ell_j$ is replaced by $\top^\sharp$. Thus no new results are introduced, and we can apply directly the inductive hypothesis on the results existing in the last reduction step. Note again that, as said in previous chapters, the truncation of recursions does not alter the precision of the analysis. In fact each time a recursive method is called, the *same* method body is replaced, so that we have the same assignments performed in the first calls, which are regularly expanded in the abstract reduction graph.

- (Red update$^\sharp$): In this case, according to the abstract rule, we have:

$$\frac{\begin{array}{c} d = [\ell_j^{\Phi_j} = \varsigma(x)u_j^\sharp, \ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \quad {}^{i\in(1\ldots n)-\{j\}}] \\ d' = [\ell_j^{\Phi_j} = \varsigma(x)u^{\varphi\sqcup\psi\sqcup\vartheta}, \ell_i^{\Phi_i} = \varsigma(x_i)t_i^\sharp \quad {}^{i\in(1\ldots n)-\{j\}}] \end{array}}{p \mapsto d \upharpoonright [\![p^\varphi.\ell_j \Leftarrow \varsigma(x)u^\psi]\!]^{\vartheta,S} \longrightarrow p \mapsto d' \upharpoonright [\![p^\varphi]\!]^{\vartheta,S}} \quad \text{(Red update}^\sharp)$$

Then we can use the inductive hypothesis on each subterm apart the $u^{\varphi\sqcup\psi\sqcup\vartheta}$ result which updates the instance variable. If this result has a level of information of l, this implies that $\varphi = \psi = \vartheta = l$. But since the level of information for u before the update was ψ, and since the result is copied and not modified by the rule, we can use the inductive hypothesis to conclude that the new result is the same as the result which existed before the reduction, and so our statement is proved also in this case.

- (Red let$^\sharp$): In this case, according to the abstract rule, we have:

$$\frac{\theta = \Re(S)}{\text{let } x = [\![u^\varphi]\!]^{\psi,S} \text{ in } [\![t^\sharp]\!]^{\vartheta,S'} \longrightarrow [\![t^\sharp\{\{x \leftarrow u^{\varphi\sqcup\psi}\}\}]\!]^{\vartheta\sqcup\theta,S'}} \quad \text{(Red let}^\sharp)$$

So the only result which changes, with respect to the previous reduction step, is $u^{\varphi\sqcup\psi}$. If this result has a level of information of l, this implies that $\varphi = \psi = l$ and then it had a low information level also before the reduction. Thus we can again use directly our inductive hyphotesis to prove our statement in this case. Note that the level of the recursion does not influence this case of the proof, since it takes part only in the following reductions, after the binding of the let term has already happened.

- (Red if$^{\sharp 1}$) and (Red if$^{\sharp 2}$): These cases are very similar, so we will show only one of them. According to the (Red if$^{\sharp 1}$) rule we have:

$$\frac{\theta = \Re(S) \sqcup \varphi \sqcup \psi \sqcup \vartheta}{\text{if } [\![\odot^{\varphi}]\!]^{\psi,S} \text{ then } [\![s^\sharp]\!]^{\vartheta,S'} \text{ else } [\![t^\sharp]\!]^{\vartheta,S'} \longrightarrow^\sharp [\![s^\sharp]\!]^{\theta,S'}} \quad (\text{Red if}^{\sharp 1})$$

So we have simply a replacement of the current if term with one of its branches. Then the results appearing after the reduction are the same results appearing before the reduction itself. The new level of knowledge of the resulting term can only be equal or higher than the previous level of knowledge, so that, if it is l, we can again apply directly the inductive hypothesis to prove our statement in this case.

- The last cases for the (Red context$^\sharp$), (Red den$^\sharp$) and (Red $\top^\sharp$) rules are trivial, since no results are modified in those rules.

□

6.6 Examples

In this section we will show some examples of programs presenting various insecure information flows between variables, and show how this flows are detected by our abstract interpretation. Let us begin with a simple example of single threaded program that is not certified as secure by many type checking techniques:

$$\begin{array}{l} \nu p.\ p \mapsto [x^H = \varsigma(x)5, y^L = \varsigma(x)7] \Rsh \\ \text{let } z = (\text{if } p.x \text{ then } p.y \Leftarrow 0 \text{ else } p.y \Leftarrow 1) \text{ in } (p.y \Leftarrow 10) \end{array}$$

This program was presented at the beginning of this chapter as a secure program. In fact, we have that although a test is performed on the high variable $p.x$ and an indirect information flow occurs from $p.x$ to $p.y$, we have that in the following this flow disappears, since the variable $p.y$ is assigned using a constant, and then at the end of the execution of the program $p.y$ is completely independent from $p.x$. The abstraction of the term above is given by:

$$\begin{array}{l} [\![\nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^l] \Rsh \\ \text{let } z = (\text{if } p^l.x \text{ then } p^l.y \Leftarrow \odot^l \text{ else } p^l.y \Leftarrow \odot^l) \text{ in } (p^l.y \Leftarrow \odot^l)]\!]^{l,\emptyset} \end{array}$$

As we can easily see, all constants have been tagged with a low level of information, unless the constant stored in the variable $p.x$, since it starts with initial information level H. The abstract term above is congruent to:

$$\begin{array}{l} \nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^l] \Rsh \\ \text{let } z = (\text{if } [\![p^l.x]\!]^{l,\emptyset} \text{ then } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset} \text{ else } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset}) \text{ in } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset} \end{array}$$

which, using the reduction rule (Red invoke$^{\sharp 1}$) leads to the following term:

$$\begin{array}{l}\nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^l] \upharpoonright \\ \text{let } z = (\text{if } [\![\ \odot^h\ ]\!]^{l,\{(p.x,l)\}} \text{ then } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset} \text{ else } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset}) \text{ in } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset}\end{array}$$

Here we can see that the guard of the *if* term has been reduced to a result having an high information level. This is correct, since we are testing the variable $p.x$ which started with H as its initial level. At this point we have to apply both rules (Red if$^\sharp$1) and (Red if$^\sharp$2) to execute both possible execution paths. However, since the the two branches of the *if* term are equal one to another, we can reduce to:

$$\begin{array}{l}\nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^l] \upharpoonright \\ \text{let } z = [\![p^l.y \Leftarrow \odot^l]\!]^{h,\emptyset} \text{ in } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset}\end{array}$$

The level of knowledge of the branch of the previous *if* term has been increased to h, following the abstract reduction rule, since we tested an high value. Using the rule (Red update$^\sharp$) we have that now we are performing an incorrect assignment, since this assignment depends from the previous test on the variable $p.x$. This can be seen from the fact that the level of knowledge of the assignment term is h. Our term, then, reduces to:

$$\begin{array}{l}\nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^h] \upharpoonright \\ \text{let } z = [\![p^l]\!]^{h,\emptyset} \text{ in } [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset}\end{array}$$

and we can see our incorrect assignment from the fact that the variable $p.y$ has an initial information level of L, while its value as information level h. At this point, many type checking techniques would have rejected this term as insecure, but if we continue in our reduction we obtain:

$$\nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^h] \upharpoonright [\![p^l.y \Leftarrow \odot^l]\!]^{l,\emptyset}$$

which performs an assignment on $p.y$ using a low value, which deletes the high value copied before. In fact, using the rule (Red update$^\sharp$) and a congruence we obtain our last reduced term:

$$\nu p.\ p \mapsto [x^H = \varsigma(x)\odot^h, y^L = \varsigma(x)\odot^l] \upharpoonright p^l$$

which does not show any insecure flow.

Let us now consider an insecure term also shown at the beginning of this chapter, when talking about insecure synchronizations between parallel threads:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)5, b^H = \varsigma(x)1, c^L = \varsigma(x)5, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } 1) \text{ in } x.c \Leftarrow 1] \upharpoonright \\ & (\text{if } p.a \text{ then } p.b \Leftarrow 1 \text{ else let } z = p.b \text{ in } p.b \Leftarrow z) \upharpoonright p.l\end{array}$$

As said at the beginning of the chapter, in this case we have two parallel threads which are secure if considered singularly. In fact the first thread updates an high variable after a test on another high variable, while the second one updates a low variable after a loop which tests an high variable (but *not inside* the loop). When those threads are taken together, however, an insecure flow happens. Let us check how this is captured by our abstract interpretation. The above term is abstracted in:

$$\begin{array}{ll}[\![\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l,\\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \upharpoonright\\ & (\text{if } p^l.a \text{ then } p^l.b \Leftarrow \odot^l \text{ else let } z = p^l.b \text{ in } p^l.b \Leftarrow z) \upharpoonright p^l.l]\!]^{l,\emptyset}\end{array}$$

which is congruent to:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l,\\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \upharpoonright\\ & (\text{if } [\![p^l.a]\!]^{l,\emptyset} \text{ then } [\![p^l.b \Leftarrow \odot^l]\!]^{l,\emptyset} \text{ else } [\![\text{let } z = p^l.b \text{ in } p^l.b \Leftarrow z]\!]^{l,\emptyset}) \upharpoonright [\![p^l.l]\!]^{l,\emptyset}\end{array}$$

By reducing the first term (remember that the abstract interpretation performs all possible reductions, so we may chose one of them) we perform a test on an high variable, which causes an increasing in the level of knowledge of the branches of the *if* term:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l,\\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \upharpoonright\\ & (\text{if } [\![\ \odot^h\ ]\!]^{l,\emptyset} \text{ then } [\![p^l.b \Leftarrow \odot^l]\!]^{l,\emptyset} \text{ else } [\![\text{let } z = p^l.b \text{ in } p^l.b \Leftarrow z]\!]^{l,\emptyset}) \upharpoonright [\![p^l.l]\!]^{l,\emptyset}\end{array}$$

Both branches of the *if* term perform assignments on the variable b which starts with level of information H. So these assignments can not be insecure and the correspondent reductions will not be shown. Both branches of the *if* term reduce by returning the reference to the modified object, p, as follows:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l,\\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \upharpoonright\\ & [\![p^l]\!]^{h,\emptyset} \upharpoonright [\![p^l.l]\!]^{l,\emptyset}\end{array}$$

Now, let us reduce the second parallel term, by applying the rule (Red invoke$^{\sharp 1}$) and some congruences:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l,\\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \upharpoonright\\ & p^h \upharpoonright \text{let } y = (\text{if } [\![p^l.b]\!]^{l,\{(p.l,l)\}} \text{ then } [\![p^l.l]\!]^{l,\{(p.l,l)\}} \text{ else } [\![\ \odot^l\ ]\!]^{l,\{(p.l,l)\}})\\ & \ \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}}\end{array}$$

The subterm guarding the *if* term reduces to $[\![\ \odot^h\ ]\!]^{l,\{(p.l,l)\}}$, since at this time in the execution the value stored inside the variable $p.h$ has an high level of information.

From now on, then, both branches of the *if* term will have an high level of knowledge, and we will have to reduce the following two terms:

$$\begin{array}{rl} \nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh \text{let } y = [\![p^l.l]\!]^{h,\{(p.l,l)\}} \text{ in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}} \end{array}$$

$$\begin{array}{rl} \nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh \text{let } y = [\![\odot^l]\!]^{h,\{(p.l,l)\}} \text{ in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}} \end{array}$$

The second term reduces to an assignment to the low variable $p.c$ with a low value, since y does not occur in the second part of the *let* term, and since all levels in this second part are low. This path emulates the concrete case when the second thread does not synchronize with the first one, being the guard of the *if* term immediately false. We continue the reduction of the first term applying the first recursive call, thanks to the rule (Red invoke$^{\sharp 1}$):

$$\begin{array}{rl} \nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh \text{let } y = \\ & \quad \text{let } y = (\text{if } [\![p^l.b]\!]^{h,\{(p.l,l),(p.l,h)\}} \text{ then } [\![p^l.l]\!]^{h,\{(p.l,l),(p.l,h)\}} \\ & \qquad\qquad \text{else } [\![\odot^l]\!]^{h,\{(p.l,l),(p.l,h)\}}) \\ & \qquad \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{h,\{(p.l,l),(p.l,h)\}} \\ & \ \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}} \end{array}$$

Again, the guard of the internal *if* construct reduces to $[\![\odot^h]\!]^{h,\{(p.l,l),(p.l,h)\}}$, since the variable $p.b$ has an high level of information. At this point, the *then* branch reduces to another recursive call, which leads the reduction to $\top^\sharp$. Then let us reduce the *else* branch:

$$\begin{array}{rl} \nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh \text{let } y = \\ & \quad \text{let } y = [\![\odot^l]\!]^{h,\{(p.l,l),(p.l,h)\}} \\ & \qquad \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{h,\{(p.l,l),(p.l,h)\}} \\ & \ \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}} \end{array}$$

At this point we can apply the rule (Red let$^\sharp$) to the internal *let* term. We have that $\Re(\{(p.l,l),(p.l,h)\}) = h$, so we should increase the level of knowledge of the second part of the internal *let* term, since it follows a recursion with high level of knowledge. Since the level of this term is already high, we have what follows:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^l, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh \text{let } y = [\![p^l.c \Leftarrow \odot^l]\!]^{h,\{(p.l,l),(p.l,h)\}} \\ & \ \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}}\end{array}$$

Now we have a sequence of two insecure assignments to the variable $p.c$. The first one is performed by the first part of the *let* term, since we have a level of knowledge equal to h. We apply the rule (Red update$^\sharp$) to obtain:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^h, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh \text{let } y = [\![p^l]\!]^{h,\{(p.l,l),(p.l,h)\}} \\ & \ \text{in } [\![p^l.c \Leftarrow \odot^l]\!]^{l,\{(p.l,l)\}}\end{array}$$

Using again the rule (Red let$^\sharp$) and the fact that $\Re(\{(p.l,l),(p.l,h)\}) = h$, we can reduce the last part of the term to obtain:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^h, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh [\![p^l.c \Leftarrow \odot^l]\!]^{h,\{(p.l,l)\}}\end{array}$$

which is another insecure assignment to the variable $p.c$, leading to:

$$\begin{array}{ll}\nu p.\ p \mapsto & [a^H = \varsigma(x)\odot^h, b^H = \varsigma(x)\odot^h, c^L = \varsigma(x)\odot^h, \\ & \ l = \varsigma(x)\text{let } y = (\text{if } x.b \text{ then } x.l \text{ else } \odot^l) \text{ in } x.c \Leftarrow \odot^l] \Rsh \\ & p^h \Rsh p^h\end{array}$$

The fact that there are insecure information flows can be easily seen from the fact that the initial level of information of the variable $p.c$ was L, while there is a path of execution (the one just shown), where the level of its final value is h.

Conclusions and Future Works

This thesis presented some innovative results in two very interesting fields of research, abstract interpretation and Object Calculi, which rarely were put together. The first three chapters are an important contribution of this thesis, since we presented all the technical background necessary to follow the remaining part.

We chose the model of object calculi since it permits, with great simplicity, to deal with the world of object oriented languages, without focusing the attention of peculiarities of particular programming languages. Moreover, object calculi may be easily extended, in such a way that we may study particular aspects which are considered interesting from time to time. The work presented in this thesis can then be applied to a broad variety of object oriented languages, and possible future works could include the adaptation of (some of) the analyses presented to a (subset of a) real object oriented language. The major challenge in this sense will be the modeling of the semantics of the considered programming language, using the simple basic constructs provided by object calculi.

The approach of abstract interpretation was chosen in this thesis for the sake of precision, since we noted the limits of other techniques of static analysis. Obviously, abstract interpretation as it was used in this thesis may be very inefficient if implemented, so possible future works could include, as an example, the definition of other abstract interpretations to analyze properties without exploring all possible paths of execution. Another source of complexity which may be further explored is given by the use of structural congruences in the semantics of objet calculi. Future works in this sense could include the definition of a directed semantics for our calculi, in order to avoid the use of congruences, which were included in this thesis for the sake of simplicity.

To give some examples of how the object calculi and abstract interpretation areas of research combine well together, we presented three analyses in three cornerstone fields of computer science: safety (chapter 4), efficiency (chapter 5) and security (chapter 6). All these examples show the simplicity of adaptation of object calculi to very different areas of research, as well as the power of abstract interpretation techniques.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996. ABA m 96:1 1.Ex.

[2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan J. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Static Analysis Symposium*, pages 19–38, 1999.

[3] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76, 1980.

[4] R. Barbuti, C. Bernardeschi, and N. De Francesco. Abstract interpretation of operational semantics for secure information flow. *Inf. Process. Lett.*, 83(2):101–108, 2002.

[5] Roberto Barbuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Checking security of java bytecode by abstract interpretation. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 229–236, New York, NY, USA, 2002. ACM Press.

[6] Roberto Barbuti and Stefano Cataudella. Abstract interpretation of an object calculus for synchronization optimizations. *Fundam. Inform.*, 67(1-3):1–12, 2005.

[7] Roberto Barbuti, Stefano Cataudella, and Luca Tesei. Abstract interpretation against races. *Fundam. Inform.*, 60(1-4):67–79, 2004.

[8] David Bell and Leonard LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, 1973.

[9] Cinzia Bernardeschi, Nicoletta De Francesco, and Giuseppe Lettieri. Concrete and abstract semantics to check secure information flow in concurrent programs. *Fundam. Inform.*, 60(1-4):81–98, 2004.

[10] Juan Bicarregui, Kevin Lano, and T. S. E. Maibaum. Formalizing object-oriented models in the object calculus. In *ECOOP '97: Proceedings of the*

Workshops on Object-Oriented Technology, pages 155–160, London, UK, 1998. Springer-Verlag.

[11] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34, New York, NY, USA, 1999. ACM Press.

[12] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. *ACM SIGPLAN Notices*, 34(10):35–46, 1999.

[13] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 382–395, London, UK, 2001. Springer-Verlag.

[14] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[15] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[16] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, FL, October 2001.

[17] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[18] Ellis Cohen. Information transmission in computational systems. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles*, pages 133–139, New York, NY, USA, 1977. ACM Press.

[19] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[22] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[23] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[24] Jérôme Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63(1):59–130, April 2005.

[25] José Luiz Fiadeiro and T. S. E. Maibaum. Describing, structuring and implementing objects. In *REX Workshop*, pages 274–310, 1990.

[26] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nord. J. Comput.*, 1(1):3–37, 1994.

[27] Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR*, pages 288–303, 1999.

[28] Cormac Flanagan and Martin Abadi. Types for safe locking. In *European Symposium on Programming*, pages 91–108, 1999.

[29] Nicoletta De Francesco, Antonella Santone, and Luca Tesei. Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundam. Inform.*, 54(2-3):195–211, 2003.

[30] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. *SIGPLAN Not.*, 39(1):186–197, 2004.

[31] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[32] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[33] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. *Electr. Notes Theor. Comput. Sci.*, 16(3), 1998.

[34] James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification.* Addison-Wesley, 1996.

[35] Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[36] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP*, pages 133–147, 1991.

[37] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, pages 79–90, 2006.

[38] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[39] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[40] Francesco Logozzo. *Modular Static Analysis of Object-oriented Languages*. PhD thesis, Ecole Polytechnique, France, June 2004.

[41] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, pages 129–145, 2004.

[42] Masaaki Mizuno and David A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[43] Andrew C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[44] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992.

[45] François Pottier and Sylvain Conchon. Information flow inference for free. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 46–57, New York, NY, USA, 2000. ACM Press.

[46] Erik Ruf. Effective synchronization removal for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 208–218, New York, NY, USA, 2000. ACM Press.

[47] A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.

[48] Andrei Sabelfeld. The impact of synchronization on secure information flow in concurrent programs. In *Proceedings Andrei Ershov 4th International Conference on Perspective of System Informatics, Novosibirsk, LNCS, Springer-Verlag*, 2001.

[49] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW '00: Proceedings of the 13th IEEE workshop on Computer Security Foundations*, page 200, Washington, DC, USA, 2000. IEEE Computer Society.

[50] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1):59–91, 2001.

[51] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multi-threaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[52] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[53] Geoffrey Smith. A new type system for secure information flow. In *CSFW14, IEEE Computer Society Press*, pages 115–125, 2001.

[54] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, 1998.

[55] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3):231–253, 1999.

[56] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[57] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

[58] Mirko Zanotti. Security typings by abstract interpretation. In *SAS '02: Proceedings of the 9th International Symposium on Static Analysis*, pages 360–375, London, UK, 2002. Springer-Verlag.