TÉCNICO LISBOA

Resource-Centered Concurrency Control

Mechanization of a Type Safety proof

Beatriz Abecasis Gomes Ferreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Ana Gualdina Almeida Matos
Prof. Jan Gunnar Cederquist

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. Ana Gualdina Almeida Matos
Member of the Committee: Prof. Mikoláš Janota

June 2019

Dedicated to my mother.

Acknowledgments

I would like to thank my family and friends for being there to provide all the help when I needed the most. I would also like to thank my supervisors Ana Almeida Matos and Jan Cederquist who were always available to provide help and guidance when necessary as well as Instituto Superior Técnico, FCT and Instituto de Telecomunicações for providing the infrastructure and the financial aid so that I could work. This work is partially funded by FCT within project Elven POCI-01-0145-FEDER-016844, Project 9471 - Reforçar a Investigação, o Desenvolvimento Tecnológico e a Inovação (Project 9471-RIDTI), and by Fundo Comunitário Europeu FEDER, by project DeDuCe (PTDC/CCI-COM/32166/2017), and via NOVA LINCS by project UID/CEC/04516/2019. Last but not least, I would like to thank the people that took the time to read this work.

Resumo

Vivendo num mundo onde a computação concurrente está presente em todo o lado, é difícil garantir que os recursos concurrentes não são acedidos ao mesmo tempo e numa ordem incorrecta. Portanto, é necessário um sistema de controlo de concorrência que assegure que as transações são feitas adequadamente. Existem vários modelos que tentam ter o melhor modelo de concorrência possível, no entanto, cada um com as suas falhas. Devido a esse facto, surgiu um modelo que permite aliviar o fardo do programador e ao mesmo tempo fazer uma análise correcta sobre quais os recursos proteger. O programador só tem de inserir um programa correto (sem erros de tipo) com algumas anotações atómicas coerentes e o modelo irá inferir quais recursos proteger. Este modelo é o *Resource-Centered Concurrency Control* e tem um mecanismo, com várias etapas, que analisa e transforma um programa. Uma parte central destas etapas é o sistema de tipos, e vamos provar que este modelo está de facto correcto, provando a propriedade *type safety*. Isto traduz-se em garantir que o sistema de tipos e as semânticas do modelo satisfazem os lemas de *progress* e *preservation*, o que garante que não aconteça nada de mal aos programas tipificados. Além disso, de forma a assegurar um nível considerável de certeza, pode ser realizada a verificação mecânica num provador de teoremas (neste caso o *Coq*). Nesta tese, formalizamos mecanicamente o sistema de tipos RC^3 no Coq, e provamos que satisfaz o teorema *type safety*. Fazemos isso adaptando o sistema de tipos mecanizado e a prova de *type safety* de uma linguagem semelhante ao Java chamada OOlong.

Palavras-chave: Controlo de concurrência, RC^3, Type Safety, Coq, OOlong, Atomicidade.

Abstract

Living in a world where concurrent computation is happening everywhere, all the time, it is very difficult to ensure that the concurrent resources aren't accessed at the same time and in an incorrect order. Therefore, we need a correct concurrency control to assure that those accesses are performed in a proper fashion. There are several models that try to be the best concurrency control model, each with their own flaws. From that premise, emerged a model that promises to unburden the programmer and still make a correct assessment about which resources to protect. The programmer only has to insert a correct program (without type errors) with some coherent **atomic** annotations, and the model will infer which resources must be protected. This model is the Resource-Centered Concurrency Control model [1], that has a mechanism with several stages which analyze and transform a program. A central part of these stages is the type system and we are going to prove that the RC^3 model is indeed correct by proving its type safety. This means that the model's type system and semantics satisfy the lemmas of progress and preservation, which ensures that typable programs never "go wrong". Furthermore, in order to ensure a very high level of certainty, mechanical verification in a theorem prover (in this case, Coq) can be performed. In this Thesis, we mechanically formalize the type system of the RC^3 model in Coq, and prove that it is type safe. We do that by adapting the mechanized type system and type safety proof of a similar Java-like language, OOlong. The contribution of this Thesis is the formalization in Coq, which gives the assurance that the formal part of the RC^3 mechanism is correct.

Keywords: Concurrency Control, RC^3, Type Safety, Coq, OOlong, Atomic.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Concurrent computations are taking place in almost every computational system and occur when several devices access the same resources or multiple clients access a server. However, concurrent code is difficult to write and it is hard to verify whether it is correct, being susceptible to errors and leading to many bugs [2]. To guarantee its correctness, it is necessary to place constraints on accesses to shared memory objects, in order to delimit the sequences of instructions that must operate atomically upon such objects. Thus, forbidding two accesses at the same time, that would lead to lost updates. So, we need to have a concurrency mechanism that guarantees absence of errors and absence of non-termination problems (and consequently a non-responding service or access). These can be translated into safety and liveness properties. A safety property means that "something (bad) will not happen" and examples of safety properties are absence of data races, strong atomicity (either all the threads are ran, or none do, assuring that no data is lost) and serializability (ensures that concurrent transactions are equivalent as they were executed serially, without overlapping in time). A liveness property means that "something (good) must happen" such as progress (freedom from deadlock) for example.

There are two main approaches to the design of concurrency control mechanisms: Control-centric approaches (decentralized management, making the programer delimiting explicitly the sequences of instructions that must be protected) and Data-centric approaches (centralizes all concurrency control management on data declaration). Both of the them aim at assuring safety and liveness properties described earlier [3]. Several models of these approaches aim at assuring these security properties in concurrency control by protecting a critical section, but either have many execution errors or are too complex and don't ensure progress and atomicity in all scenarios. To address problems with complexity and insufficient guarantee of progress and atomicity, a new model was created which is called Resource-Centered Concurrency Control (RC^3) [1]. Here, instead of a set of memory objects that are accessed by concurrency, this model builds only upon the individual annotation of the resources that must be protected. This decreases the number of annotations required. So, in addition to the advantages of the data-centric concurrency management, this is a simpler model and has simple semantics that makes

reasoning easier.

1.2 The RC^3 Model

The idea of the RC^3 model is to unburden the programmer, so that he doesn't have to insert all the **atomic** annotations in his program. The model will output a program where all the **atomic** resources are protected, so it has to infer which resources must be atomically accessed. In order to reach this goal, the mechanism for analysis and transforming the program has several stages, where the model must check first if the program satisfies the basis of being a program without type errors. Then, it will gradually add more details into the verification, starting by adding the keyword **atomic** into the program's classes and interfaces, and then, it will analyze the variables to know if they are **atomic** or not. It will also check in a later stage if the code is consistent with the atomic annotations. In a final stage, the model assures that the **atomic** resources are exclusively accessed.

1.3 Proving Type Safety

An important part of the previously mentioned mechanism in Section 1.2 is the type system to infer which variables should be **atomic** or **non-atomic**. In order to ensure that the type system is correct, we need to prove its type safety. If a model has progress and preservation we say that is type-safe, that is, the programs can't "go wrong". *Progress* is the property that ensures a typable program never gets "stuck", so it enables the program to always take a step. Linked to it there is also *preservation*, that is the property that ensures that the program's type is preserved at each step.

The proof of type safety can be made using pen and paper or mechanized using a theorem prover. At first, all the proofs were made on the paper, but with the advance of the technology, new tools to aid the proofs begun to appear. Those tools not only make the proof easier to do, but also increase the assurance of the correctness of the proof. If the proof is only made by a human, it can contain some errors and can be harder to accept than the proofs checked by a computer, that are easier to write. This is due to the fact that the computer can be asked to do much more work to check each step than a human is willing to do, and this allows longer and fewer steps [4]. Thus, the best scenario in a proof is where the machine and a human user work together interactively to produce a formal proof. It can be just the machine acting as a proof checker on a formal proof made by a human, or it can be highly automated and powerful tool with human guidance. In order to guide a machine proof, there needs to be a language for the user to communicate that proof to the machine, this involves introducing the semantics, the type system and theorems.

1.4 Objectives

The aim of this work is to mechanize the type system of the RC^3 model (whose formalization, built on top of the language OOlong, is still under development) and to mechanically verify its type safety by

formalizing the two properties that define it: preservation + progress. This is developed in a theorem prover named Coq.

1.5 Contributions

Starting from a formalization of the OOlong language to the proof of type safety, we adapted it to the RC^3 language. We mechanized all the RC^3 language in the theorem prover Coq, also based in the already mechanized version of the OOlong language for Coq, as well as the type safety to the formal model RC^3. The work was done until the stage of the type verification of the type system.

1.6 Thesis Outline

After introducing the contributions in Chapter 1, this Thesis is organized as follows: Chapter 2 talks about other important work related to this Thesis. We begin by analyzing which concurrency control models have proved the type safety of their model. This is followed by some notions that are going to be important in order to understand the definitions in the rest of the paper. Along with this we also study some proofs about type safety so that we can take advantage of them. Besides this reason, it also serves as a guide to create the proof in the chosen theorem prover. To finish this Section, we talk about the main tools and make a brief description of them. In Chapter 3 we are going to present the formalization of the RC^3 model, that is still in progress, and how it was adapted from the formalization of the OOlong language. We also describe the RC^3 model. The implementation of the RC^3 formalized language and its proof of type safety in Coq is displayed in Chapter 4. The Thesis ends with the conclusion and the future work in Chapter 5.

Chapter 2

Related Work

In this Chapter, we talk about other important related work, which helps the achievement of the goals of this Thesis. We begin by analyzing which concurrency control models have proved the type safety of their model. This is followed by some notions that are going to be important in order to understand the definitions in the rest of the paper, and the early proofs of type safety as well as more recent proofs. To finish this Section, we talk about the main tools and make a brief description of them.

2.1 Concurrency Control

There are two main approaches for controlling access in concurrent computations: Control-centric approaches and Data-centric approaches. Control-centric concurrency has decentralized management rather than centralized management like data-centric approaches [1].

2.1.1 Control-centric approaches.

This is the traditional approach to prevent concurrency-related errors. Due to being the oldest, there are more models that use this approach instead of the other one. However, since it involves local reasoning, the approach is susceptible to the dispersal of concurrency related bugs: it is enough that a single lock/unlock is missing to enable data races to occur. Hence, the more code scales the more missing constraints we'll find. Furthermore, even if all shared data is protected, high-level data races may still happen [5].

In order to withstand this, there are several approaches in which the target is progress and isolation [6–10] and others in which the central point is atomicity and protocol compliance [11, 12].

Regarding the models with focus on progress and isolation, some of the papers early mentioned prove their corresponding desirable properties. The remaining papers don't have the proofs, which can be problematic because it's ambiguous either the models really work correctly or not. In [6, 9], they formalize the definitions, present the type system and prove type safety [1] with operational semantics like

[1] In other papers sometimes type safety is called type soundness or even just soundness. That's because type soundness can be called type safety when it only refers to preservation and progress.

we are going to do. In paper [10], they prove it using a model that helps doing hierarchical correctness proofs of distributed algorithms [13]. Concerning the papers in which the central point is atomicity and protocol compliance, one of them [12] also formalizes the definitions and present the proofs about preservation and progress to prove type safety like we'll do. Nonetheless, despite this the errors remain in great quantity [2].

All the proofs are made in the paper instead of doing it in a theorem prover.

2.1.2 Data-centric approaches.

Data-centric synchronization is an alternative recent approach that unburdens the programmer by offloading some of the work on the language implementation. This approach centralizes all concurrency control management on data declaration. It is more intuitive and can make correct concurrent programming more easily. Atomic Sets is the only model that supports this [14, 15].

In this approach, instead of thinking about the synchronization of execution flows, we need to know what memory segments share consistency properties (e.g. are accessed by concurrency). The idea is to make a set of atomic variables (variables accessed synchronously) by prefixing the declaration of the variables with atomic(s), thus, adding them to the set. Along with this, there are code fragments called units of work, related to the set of atomic variables whose purpose is assuring the consistency properties of it. We can see the proof about type safety in [14], where they prove preservation and progress after presenting their type system and operational semantics. Once more, these proofs are all made on paper, while we are using an interactive theorem prover.

Despite all of this, this solution makes reasoning difficult and is more prone to errors because of the high number of annotations. Also, it does not always guarantee progress (e.g. Deadlocks).

Variants.

One solution to the Deadlock problem is presented in [16], which has an algorithm for detecting possible deadlocks by ordering the locks associated with atomic sets. However, the programmer has to act when the analysis fails to infer about the partial order between sets.

An alternative to Atomic Sets is AJ-lite [17], a lighter version of AJ that decreases the number of annotations by having only one atomic set per java class. However, this is still prone to errors and the programmer is also called to intervene. Besides, currently this approach applies only to libraries and not to full programs.

In [18] the initial step to automate the inference of atomic sets are shown. It processes the execution traces to identify patterns in the access of class fields (currently only those are supported), that help to automatically form atomic sets. Despite the experimental results being mostly correct, it generated false positives (more annotations than required). This approach has also two more problems: the result varies according to the input traces and has longer compilation time.

Our approach is the one we saw before: Resource-Centered Concurrency Control (RC^3). Not only is it simpler than the other solutions, it also grants safety and progress properties.

2.2 Type System

2.2.1 Basic Concepts.

In order to understand the notations that are mentioned in the following Sections, we have to define them first. Therefore, in this Section, we have a brief explanation of them.

Inductive definition of relations.

In simplified terms, a set is inductively defined, that is, a set is determined by the rules that construct its elements. For example in this rule:

$$\frac{x_1, x_2, ...}{y}$$

If the premises $x_1, x_2, ...$ are in the set being defined, then so is the conclusion y. This rule leads to the derivation tree:

$$\frac{\dfrac{\overline{x_1}}{\neg x_1} \quad \dfrac{\overline{x_2} \quad \overline{x_3}}{x_2 \vee x_3}}{\neg x_1 \wedge (x_2 \vee x_3)}$$

It has the form of a tree with the conclusion at the root and with axioms at the leaves. It is built by stacking rules together, matching conclusions with premises [19].

The syntax have the following notation: $A ::= a$, where A is the set and a is an element of the set. The small-step operator represented by $\Rightarrow$ that we are going to see later in this Section, is an inductively defined relation.

Although we are going to see later what a Type System is and it's purpose, we introduce already some notations [19, 20]:

- τ is going to be used for expressions;
- Γ , the typing environment (also called a typing context), is a mapping from variables to types. $\Gamma(x) = \tau$ means that Γ maps x to τ;
- $\vdash$ is called a typing judgment, which means that makes a statement about typability of (part of) a program. The relation $\vdash$ is inductively defined;
- $\Gamma \vdash e : \tau$, is the judgment for typing expressions;
- The rules

  $$\frac{assumption_1 \ ... \ assumption_n}{conclusion}$$

  are a way of deriving judgments from assumptions. Rules with empty premises (assumptions) are called axioms.

Semantics.

Associated with the syntax we just saw, there is the semantics. The syntax is concerned with the grammatical structure of programs. Yet the semantics is concerned with the meaning of grammatically correct programs. [21]

Semantics is the field concerned with the rigorous mathematical study of the meaning of programming language. There are three major approaches to Semantics: Operational, Denotational and Axiomatic Semantics.

- *Operational Semantics:* Describes the effect of each statement on the state. It is of interest how the effect of a computation is produced.
- *Denotational Semantics*: Each phrase in the language is interpreted as a mathematical object that represent the effect of executing the constructs. Thus only the effect is of interest, not how is obtained.
- *Axiomatic Semantics:* Gives meaning to phrases by describing the logical axioms that apply to them. Thus there may be aspects of the executions that are ignored.

In this work we use Operational Semantics, which is now described in more detail.
In an operational semantics we are concerned with how to execute programs and not merely what the results of execution are. More precisely, we are interested in how the states are modified during the execution of the statement. We have two different approaches to operational semantics: Small-step semantics, which is used here, and Big-step semantics (or Natural semantics). The main difference between them is that the purpose of the Big-step semantics is to describe how the *overall* results of executions are obtained, contrarily to Small-step semantics that it's purpose is to describe how the *individual steps* of the computations take place [21]. In small-step semantics, a transition encodes only one step of computation [22]. The transition relation has the form:

$$\langle S, s\rangle \Rightarrow \gamma$$

where γ either is of the form $\langle S', s'\rangle$ or of the form s'. The transition express the first step of the execution of S from state s.

Using the definition explained in the last paragraph, the rules specify the meaning, or semantics, of arithmetic expressions in an operational way, and the rules are said to give an operational semantics of such expressions [23].

Type Systems.

According to [24], a type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. This means that a type system associates a type with each computed value and, by examining the flow of these values, attempts to ensure or prove that no type errors can occur. It can also be defined as a set

of rules that assign a property called type to the various constructs of a computer program, such as variables, expressions, functions or modules. A type system defines how a programming language classifies values and expressions into types, how it manipulates those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations (operations expecting a certain kind of value from being used with values for which that operation does not make sense). This way it reduces the possibility of bugs in computer programs.

Not only can the type systems detect errors, allowing an early detection and an immediately fixed, they can also provide a form of documentation, abstraction (that leads to a better design), efficiency and language safety (one that protects its own abstractions). Type systems are also applied in proof assistants, where they are used to represent logical propositions and proofs.

Several popular proof assistants, like Coq and other theorem provers seen in Subsection 2.3, are based directly on type theory.

Type Safety.

In order to guarantee that the type system don't have run-time type errors, we need type safety. This notion started to be defined by Robin Miller in 1978 [25] as: Well-typed programs cannot "go wrong". In 1994, Andrew Wright and Matthias Felleisen [26] formulated what is now the standard definition and proof technique for type safety in languages defined by operational semantics. Under this approach, type safety is determined by two properties of the semantics of the programming language: preservation (or subject reduction) + progress. The first property says that programs remain invariant under the transition rules, and the second one that the program never gets into an undefined state where no further transitions are possible, i.e. the program never gets "stuck".

These properties don't exist by themselves, they are linked to the semantics of the programming language they describe. The notion of "well typed" program is part of the static semantics of the programming language and the notion of "getting stuck" (or "going wrong") is a property of its dynamic semantics.

Consequently, in order to assure that well-typed programs do not incur run-time type errors (that means, no well-formed program gets stuck), we have to prove type safety. Therefore it is necessary to prove the following two lemmas, regarding the operational semantics:

Lemma 1 (Preservation)**.** If $\Gamma \vdash e : \tau$ and $e \Rightarrow e'$ then $\Gamma \vdash e' : \tau$

The first lemma says that if an expression e, with type τ, in the next step is e', then e' also has type τ. That is, *preservation* (also called subject reduction as seen before) says that every step preserves the type.

Lemma 2 (Progress)**.** If $\Gamma \vdash e : \tau$ then $e = v \in$ Value or $\exists e''$ t.q. $e \Rightarrow e''$

As for second lemma, it says that if an expression e has type τ, then e is a value or there exists an e'' such that the next step of e is e''. So Progress ensures that e does not stay stuck.

The prior two lemmas are typically proven by induction on the type derivation of e.

2.2.2 Proof of Type Safety.

There are two ways of proving type safety: by pen and paper or by using a theorem prover. In this part we are going to focus on the former one technique, given that it was the first being used to prove type safety. We'll start by seeing the beginning of this kind of proofs.

Origins.

The first proofs were made with the pen and paper technique. At that time, stable theorem provers didn't exist so this was the only way of doing it. As we saw before, the notion of type safety appeared in 1978 by Robin Miller in [25], where he did the proof by induction in a simple functional language Exp, which has denotational semantics.

Later, in 1987, Mads Tofte [27] used Milner's functional language but with operational semantics instead of denotational semantics.

Any proof of type safety is intimately tied to the formation of the semantics of the language. The earlier proofs relied on denotational semantics and later proofs used operational semantics.

However, each one of these proofs used a different technique. As they are often unrelated, they provide no guidance in proving type safety for new languages or languages features. Therefore, Andrew Wright and Matthias Felleisen presented a new approach of doing the proof [26], formulating what is now the reference technique for type safety proofs. They introduced additional expression forms and typing rules instead of additional semantic objects (which would require an additional semantic relation between semantic objects and types), thus simplifying the proof. They also defined type safety as being preservation plus progress as seen previously.

For Java-like languages.

Java is the language chosen for the implementation of the model we are going to evaluate [1]. Here, we'll see proofs of type safety made for Java-like languages as the model is made in a Java-like language similar to OOlong, which in turn is similar to Welterweight. The Java language is rather complicated and there is no need to have so many complex features to prove type safety. Therefore, there are several studies that introduced lightweight versions of Java, reducing the language to enable rigorous arguments about key properties. This way we can have rigorous and easy proofs.

We'll see four kinds of a minimal core calculus.

- *Featherweight:* The smallest proposed candidate for a core Java calculus is probably Featherweight Java (FJ) [28], which omits all forms of assignment and object state, focusing on a functional core of Java. It has classes, methods, fields, inheritance, and dynamic typecasts, with semantics similar to Java's, giving it the same computational feeling. In Featherweight paper [28], they define the language syntax, typing and reduction rules. The semantics used are operational semantics using

a small-step reduction relation. The type safety theorem is proved by using the standard technique of preservation and progress lemmas, and the proof is done by induction.

- *Welterweight:* While in many cases FJ is sufficient, it does not model threads and synchronization, which is a significant omission in this day and age. In order to correct this, a new minimal core Java calculus was made: Welterweight Java (WJ) [29]. WJ is imperative and stateful (keeps track of the state of interaction), which is a frequent extension of Featherweight Java. It models Java-style threads and concurrency control, is based on statements rather than expressions and uses explicit casts in favor of implicit subsumption (allows to apply subtyping to their types implicitly). Because of its easy extensibility, WJ is a suitable core Java formalism for extensions to Java-like languages that need imperative features and threads, just like the OOlong language. In [29], they make the proof of type safety by structural induction, using also the preservation and progress lemmas. They define the language syntax, static and dynamic semantics, which are formalized as a small-step reduction semantics.

- *OOlong:* A similar object calculus, to the Welterweight Java is OOlong [30]. OOlong is more lightweight than WJ by omitting mutable variables and using a single flat stack frame rather than modeling the call stack. Given that the proof of type safety for this language is made in a theorem prover and not in pen and paper, we will continue later in the next Subsection 2.2.3.

- *Middleweight:* Middleweight Java (MJ), a minimal imperative core calculus for Java, can be seen as an extension of FJ big enough to include the essential imperative features of Java, yet small enough that formal proofs are still effortless [31]. MJ extends FJ with imperative features such as field assignment, mutable lexical environments, null pointers, and block scopes. We have the definition of the language syntax, type system and small-step operational semantics in [31], as well as the proof of type safety. Once again, they prove in the style of Wright and Felleisen, and the proof of each lemma is made by induction.

2.2.3 Verification of Type Safety.

As was previously mentioned, there are many proofs of type safety made using pen and paper. In this part we are going to see the same proofs, but now using theorem provers. In the following Section we'll talk about each of the theorem provers that are going to appear in this part.

Origins.

There are many papers that describe a type safety proof of some language using a theorem prover. For example in 1995, [32] proved preservation (one of the two necessary properties to prove type safety) of a small typed functional language in a proof assistant called ALF. They present the syntax and the type system just like the other papers previously seen, but the semantics used are different. Although is also operational semantics, in this case the approach used is big-step semantics. Regarding the preservation proof, it's done by induction on the derivation of the transaction $e \Rightarrow e'$ in Lemma 1, and with an auxiliary

lemma. All this is formalized in ALF. In 2000, [33] verified the type safety of Meta Language (ML) within the Coq proof assistant. ML is a functional programming language and its types and pattern matching make it well-suited and commonly used to operate on other formal languages, such as in automated theorem proving and formal verification. They also use operational semantics but with both approaches: small-step and big-step semantics. For small-step semantics they present the Coq formalization and prove the preservation property. Keeping in mind that the latter states that reductions preserve the type of expressions, this proof moves forward by case analysis. Concerning big-step semantics, the same proof proceeds by induction in Coq and requires several supplementary definitions and new lemmas.

Because the correct definition and implementation of non-trivial type systems is difficult and requires expert knowledge, there are studies that try to simplify this development. Veritas [34] is one of them. It automatically tries to derive type safety proofs with the automated first-order theorem prover Vampire. In 2017, [35] study what is the best way to automate classic steps within type safety proofs. To achieve this, they compare two different approaches for proving one of the properties of type safety, progress, against each other: their own Veritas tool (which calls Vampire), and the programming language Dafny. Similar to what they did in 2015 in [34] where they verified the type safety of a statically typed variant of SQL, here they model a subset of typed SQL to study the two approaches. They separately model our subset of SQL in Veritas and in Dafny. The proof is done by induction, the typical proving way. Both Vampire and Dafny are equally well-suited for automatically proving simple steps within progress proofs, however for their progress theorem, Dafny does not seem to be able to figure out the induction scheme on its own.

For Java-like languages.

Resembling the prior segment, in this paragraph we have proofs of type safety made for Java-like languages, but now in a theorem prover. There are several subsets of Java, like μJava [36] and Jinja [37], that prove type safety with big-step and small-step operational semantics respectively, in the theorem prover Isabelle/HOL, but we'll focus on the same lightweight versions of Java as before.

- *Featherweight:* In the smallest core Java calculus, similar to the pen and paper proof, [38] also formalizes the type system, use small-step semantics and prove type safety of the Featherweight Java type system with the same standard technique. However this time, all the formalizations and verifications are made in the theorem prover Isabelle/HOL, as well as any auxiliary lemma. Therefore, this paper mechanically verifies the type safety proof made with pen and paper technique as the lemmas and theorems correspond to the same results.

- *Welterweight:* There isn't any mechanized version of the type safety proof of the Welterweight Java language.

- *OOlong:* This is the closest Java-like language to the RC^3 model language. In this paper [30], they not only define the formal semantics and prove type safety on paper, they also provide a mechanized version of the full semantics and type safety proof, written in the theorem prover Coq. They use

the same semantics the WJ paper use: small-step semantics. The proof of type safety is done in the typical way, by proving the two lemmas, preservation and progress, by induction.

- *Middleweight:* Middleweight Java was never mechanized. There is however a simplification of MJ: Lightweight Java (LJ) [39]. This minimal imperative core of Java does not include type casts, local variables, interfaces, method overloading, or any of the more advanced language features. LJ's syntax, small-step operational semantics and type system are defined rigorously in Isabelle/HOL, as well as the mechanically proof of type safety. This proof is done once more in the usual way: by proving preservation and progress lemmas by case, with helper lemmas.

2.3 Theorem Provers

In the Section before we just saw several theorem provers that were used to prove type safety of different languages. Those tools are an alternative to the older pen and paper technique that not only make the proof easier to do, but also increase the assurance of the correctness of the proof. There is a range of theorem provers from interactive proof checkers to fully automatic theorem provers. With a pure proof checkers one can, in principle, prove any theorem, but it requires detailed and precise guidance even for simple problems. A fully automatic theorem prover can prove non-trivial properties but it also fails to prove many properties. Interactive theorem provers (proof assistants) often provide proof automation, and aims at cooperation between human and machine.

The earliest work on computer-assisted proof was dedicated to pure automated theorem proving. Nevertheless the automation of type soundness proofs is a long-standing open problem as there are practical limitations. "How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine checked proofs?" [40] is the question of the POPLmark challenge from 2005. This challenge consists of a benchmark for measuring progress of automated reasoning. Despite several existing solutions to the POPLmark challenge, to date there is no fully automated solution. So far, the interactive proof is likely to be the only way to formalize most of the non-trivial theorems.

2.3.1 Generic theorem provers

Theorem prover is a software tool whose purpose is to assist with the development of formal proofs. This involves some sort of interactive proof editor, or other interface, with which a human can guide the machine. The probable earliest interactive theorem prover must be Automath, that start being developed in 1967. It is the predecessor of several important current provers. In [4] we can see more about the history of interactive theorem proving, but here we'll see what were the candidates for verifying the type safety of the RC^3 model.

- *Agda:* Agda's predecessor is ALF (Another Logical Framework), whose first version is from 1990, that is why the paper [32] from 1995, seen in the Section before, calls it ALF instead of Agda. Like its predecessors, Agda supports a wide range of inductive data types, pattern matching (act

of checking a given sequence of tokens for the presence of the constituents of some pattern), termination checking, and comes with an interface for programming and proving by direct manipulation of proof terms [41]. However, the current Agda has more attributes like: flexibility of pattern matching, more powerful module system, attractive and flexible concrete syntax (using unicode). It has many similarities with other proof assistants based on dependent types, like Coq. Howsoever, unlike it, has no support for tactics, and proofs are written in a functional programming style. Nevertheless, Agda is primarily being developed as a programming language and not as a proof assistant. That and the fact that there are better proof assistants are the reasons that we didn't choose this one.

- *Isabelle/HOL:* The Isabelle theorem prover [42] is a Higher Order Logic (HOL) theorem prover. A Higher Order Logic is a form of predicate logic that is distinguished from first-order logic by additional quantifiers and, sometimes, stronger semantics. Though interactive, Isabelle also features efficient automatic reasoning tools, such as a term rewriting engine and a tableaux prover (a decision procedure that resorts to a tree structure), as well as various decision procedures. Isabelle/HOL does not support dependent types. Keeping in mind that the powerful means that can express and prove more theorems than others, CIC (see the theorem prover Coq) is more powerful than HOL.

- *Coq:* Coq [43] is an interactive theorem prover which provides interactive proof methods, decision and semi-decision algorithms, and a tactic language for letting the user define its own proof methods. It implements a program specification and mathematical higher-level language called Gallina that is based on an expressive formal language called the Calculus of Inductive Constructions (CIC), which aims at representing both functional programs in the style of the ML language and proofs in higher-order logic. Programs written in Gallina have the weak normalization property, which means they always terminate, and therefore this will avoid the halting problem (i.e., the program would continue to run forever).

  Coq is written in the OCaml language, with a bit of C. Coq is not an automated theorem prover but includes automatic theorem proving tactics and various decision procedures.

- *Vampire:* Vampire [44] is an automated theorem prover, that is, a system able to prove theorems. More precisely, it proves theorems in first-order logic that started being developed in 1994. All together Vampire won 28 division titles in CASC since 1999: more than any other theorem prover in the history of the competition. Vampire won all the FOF (first-order formulas) divisions since 2002.

- *Twelf:* The probably highest potential of full automation among the set of solutions submitted to the PoplMark challenge is the Twelf approach [45]. Twelf is a special-purpose theorem prover for properties of logics and programming languages, based on the logical framework (LF). It provides an interactive proof mode as well as support for automated inductive theorem proving. However, encoding a type system specification and a corresponding type safety proof in Twelf requires rigorous

knowledge of logical frameworks.

2.3.2 Tools for development of sound type systems

- *Why3:* Why3 [46], a tool for deductive program verification, is based on first-order logic. This platform provides a rich language for specification and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. Why3 can also be used as a software library, through an OCaml API.

- *Dafny:* Dafny specialize on verification of programs and programming languages [47]. This verifier was designed to provide a simple introduction to formal verification and has been used widely in teaching. It follows in the lineage of many previous tools, like Why3. Such tools rely on the use of automated theorem proving unlike, for example, those based on dependent types (e.g. Agda) which require more human intervention. Dafny's program verifier works by translating a given Dafny program into the intermediate verification language Boogie, in such a way that the correctness of the Boogie program implies the correctness of the Dafny program. The Boogie tool is then used to generate first-order verification conditions that are passed to a theorem prover, in particular to the Z3 automated theorem prover.

- *Veritas:* The available tools like Agda, Isabelle/HOL, Coq and Dafny offer good automation for some simple proof tasks, but require that the developers translate their specification into the respective input language of the tool and then develop proofs using the language and concepts offered by the tool. More recently there were attempts to reduce the burden on developers shoulders and the expert knowledge needed, namely Veritas [34], a tool that simplifies the development of mechanized progress and preservation proofs.

- *OTT:* Ott [48, 49] is a tool for writing definitions of programming languages and calculi. It takes as input a definition of a language syntax and semantics, and allows a formalization of the type system and semantics. It can be run as a filter, taking a LaTeX/Coq/Isabelle/HOL source file with embedded (symbolic) terms of the defined language, parsing them and replacing them by target-system terms.

2.4 Chapter Summary

This Chapter started by explaining some of the existing control mechanisms for concurrency control: some with control-centric approach and others with the data-centric approach, and talk about their type safety proofs. Then, it introduced some basic notions about inductive definition of relations, semantics,

type system and type safety. These notions are necessary to understand the proofs made in pen and paper and the ones made in a theorem prover, that were talked afterwards. For each, one we saw the early proofs and some Java-like languages due to being the language of the RC^3 model: Featherweight, Welterweight, OOlong and Middleweight. Given that OOlong provides a mechanized version in the theorem prover Coq, of the language and the type safety proof, the RC^3 language is built on top of OOlong, and the theorem prover chosen to mechanically prove type safety of the RC^3 model is also Coq. Finally, to finish this Section, we talked about the main theorem provers and make a brief description of them.

Chapter 3

Model

As we saw in Chapter 2, the RC^3 language is based on OOlong. In this Chapter we are going to describe briefly the RC^3 language and how it differs from OOlong. We will present the syntax, semantics and type system, as well as the architecture of the model with the respective stages. Readers wishing to know more about the OOlong language can refer to: [50], and [30].

3.1 Language

3.1.1 Syntax

The language used in the RC^3 model extends OOlong with an **atomic** keyword in order to support the declaration of **atomic** resources, which may appear wherever a (non-unit) type may be used. That extra keyword allows the declaration of **atomic** variables so that we can know which data items must be protected, and will be used, in future work, to generate locks automatically. Given that the concurrency control is made solely with this, the lock and unlock operations are now only available at runtime.

Let's proceed to the description of the syntax represented in Table 3.1. The symbols *Cds* and *Ids*

Table 3.1: Syntax of RC^3 programs. Constructs with grey background are only available at runtime.

P	$::=$	$Ids\ Cds\ e$	*(Programs)*
Id	$::=$	**interface** $I\ \{Msigs\}$	*(Interfaces)*
	$\mid$	**interface** I **extends** I_1, I_2	
Cd	$::=$	**class** C **implements** $I\ \{Fds\ Mds\}$	*(Classes)*
$Msig$	$::=$	$m(x : t_1) : t_2$	*(Signatures)*
Fd	$::=$	$f : t$	*(Fields)*
Md	$::=$	**def** $Msig\{e\}$	*(Methods)*
e	$::=$	$v \mid x \mid x.f \mid x.f := e \mid x.m(e) \mid$ **let** $x := e_1$ **in** $e_2 \mid$ **new** $C \mid (t)\ e$	*(Expressions)*
	$\mid$	**finish** $\{$**async** $\{e_1\}$ **async** $\{e_2\}\}; e_3 \mid$ **lock** (x) **in** $e \mid$ **locked** $_\iota\{e\}$	
v	$::=$	**null** $\mid \iota$	*(Values)*
I	$::=$	$J \mid$ **atomic** J	*(Interface Types)*
C	$::=$	$D \mid$ **atomic** D	*(Class Types)*
t	$::=$	$I \mid C \mid$ **Unit**	*(Types)*
Γ	$::=$	$\epsilon \mid \Gamma, x : t \mid \Gamma, \iota : C$	*(Typing environment)*

Table 3.2: Runtime syntax

cfg	$::=$	$\langle H; V; T\rangle$	*(Configuration)*
H	$::=$	$\epsilon \mid H, \iota \mapsto obj$	*(Heap)*
V	$::=$	$\epsilon \mid V, x \mapsto v$	*(Variable map)*
T	$::=$	$(\mathcal{L}, l : e) \mid T_1 \,\|\|\, T_2 \rhd e \mid$ **EXN**	*(Threads)*
obj	$::=$	(C, F, L)	*(Objects)*
F	$::=$	$\epsilon \mid F, f \mapsto v$	*(Field map)*
L	$::=$	**locked** | **unlocked**	*(Lock status)*
EXN	$::=$	**NullPointerException**	*(Exceptions)*

represents lists of classes and interfaces respectively, and *Msigs*, *Fds* and *Mds* lists of signatures, fields and methods. The symbol *I* is an interface and *C* a class. Local variables are represented by *x* and *y*, *m* is for method names and *f* for field names. For simplicity we assume that all names are unique. The main difference from OOlong, is the new symbols *D* and *J*, that represent respectively the **non-atomic** classes and interfaces types, while **atomic** D and **atomic** J include their atomic counterparts. A program is a collection of interfaces and classes together with a starting expression e (referred to as main). Classes implement interfaces, and interfaces are defined either as a collection of method signatures, or as an interface that inherits from two other interfaces. Regarding the expressions, we have values (which are **null** or a location), variables, field accesses (access the field f of the object x), field assignments (assigns the field f of the object x to an expression e), method calls (calls the method m with the argument e in the object target x), **let**-bindings (assigns the variable x to the expression $e1$ as long as the expression $e2$ holds), object instantiation and type casts (change the type of e to t). The expression **finish**$\{$**async**$\{e_1\}$**async**$\{e_2\}\}$; e_3 represents the parallel threads, which wait for the completion of e_1 and e_2 that are running in parallel, and then continues with e_3. Types can either be a class or an interface name, or **Unit** (used as the type of assignments), and are assigned to fields, method return values and method parameters. The typing environment Γ maps variables into types and abstract locations into classes.

3.1.2 Semantics

The semantics is based on the small-step semantics formalized for OOlong. Table 3.2 presents $\langle H; V; T\rangle$, that is the representation of the execution of a RC^3 program (a configuration). The heap *H* is a map from locations (l) to objects, where the object is a tuple composed by its class (*C*), the map of fields into values (*F*) and lock status (*L*). In turn, the lock status can either be **locked** or **unlocked**. The stack *V* maps variables to values, and finally *T* is a collection of threads, that can be a single thread, two parallel asyncs threads or a thread in an exceptional state **EXN**.

The rules for the evaluation of expressions are the same as for OOlong, and can be checked in table 3.3.

Table 3.3: Dynamic semantics

$\boxed{cfg \hookrightarrow cfg'}$

$$\text{DYN\_EVAL\_ASYNC\_LEFT}\ \frac{\langle H;\, V;\, T_1\rangle \hookrightarrow \langle H';\, V';\, T_1'\rangle}{\langle H;\, V;\, T_1 \,||\, T_2 \rhd e\rangle \hookrightarrow \langle H';\, V';\, T_1' \,||\, T_2 \rhd e\rangle}$$

$$\text{DYN\_EVAL\_ASYNC\_RIGHT}\ \frac{\langle H;\, V;\, T_2\rangle \hookrightarrow \langle H';\, V';\, T_2'\rangle}{\langle H;\, V;\, T_1 \,||\, T_2 \rhd e\rangle \hookrightarrow \langle H';\, V';\, T_1 \,||\, T_2' \rhd e\rangle}$$

$$\text{DYN\_EVAL\_ASYNC\_JOIN}\ \frac{}{\langle H;\, V;\, (\mathcal{L}, v) \,||(\mathcal{L}', v') \rhd e\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, e)\rangle}$$

$$\text{DYN\_EVAL\_SPAWN}\ \frac{e = \texttt{finish}\ \{\ \texttt{async}\ \{\ e_1\ \}\ \texttt{async}\ \{\ e_2\ \}\ \};\, e_3}{\langle H;\, V;\, (\mathcal{L}, e)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, e_1) \,||(\emptyset, e_2) \rhd e_3\rangle}$$

$$\text{DYN\_EVAL\_SPAWN\_CONTEXT}\ \frac{\langle H;\, V;\, (\mathcal{L}, e)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, e_1) \,||(\emptyset, e_2) \rhd e_3\rangle}{\langle H;\, V;\, (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, e_1) \,||(\emptyset, e_2) \rhd E[e_3]\rangle}$$

$$\text{DYN\_EVAL\_CONTEXT}\ \frac{\langle H;\, V;\, (\mathcal{L}, e)\rangle \hookrightarrow \langle H';\, V';\, (\mathcal{L}', e')\rangle}{\langle H;\, V;\, (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H';\, V';\, (\mathcal{L}', E[e'])\rangle}$$

$$\text{DYN\_EVAL\_CAST}\ \frac{}{\langle H;\, V;\, (\mathcal{L}, (t)v)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, v)\rangle}$$

$$\text{DYN\_EVAL\_DOWNCAST}\ \frac{\begin{array}{c} H(\iota) = (C, F, L) \\ C <: t \end{array}}{\langle H;\, V;\, (\mathcal{L}, (t)\iota)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, \iota)\rangle}$$

$$\text{DYN\_EVAL\_UPCAST}\ \frac{t_2 <: t}{\langle H;\, V;\, (\mathcal{L}, (t)\iota)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, \iota)\rangle}$$

$$\text{DYN\_EVAL\_CAST\_NULL}\ \frac{}{\langle H;\, V;\, (\mathcal{L}, (t)\texttt{null})\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, \texttt{null})\rangle}$$

$$\text{DYN\_EVAL\_LET}\ \frac{\begin{array}{c} x' \textbf{ fresh} \\ V' = V[x' \mapsto v] \\ e' = e[x \mapsto x'] \end{array}}{\langle H;\, V;\, (\mathcal{L}, \texttt{let}\ x = v\ \texttt{in}\ e)\rangle \hookrightarrow \langle H;\, V';\, (\mathcal{L}, e')\rangle}$$

$$\text{DYN\_EVAL\_CALL}\ \frac{\begin{array}{c} V(x) = \iota \\ H(\iota) = (C, F, L) \\ \textbf{methods}\,(C)(m) = y : t_2 \to t, e \\ \texttt{this}' \textbf{ fresh} \\ y' \textbf{ fresh} \\ V' = V[\texttt{this}' \mapsto \iota][y' \mapsto v] \\ e' = e[\texttt{this} \mapsto \texttt{this}'][y \mapsto y'] \end{array}}{\langle H;\, V;\, (\mathcal{L}, x.m(v))\rangle \hookrightarrow \langle H;\, V';\, (\mathcal{L}, e')\rangle}$$

$$\text{DYN\_EVAL\_LOCK}\ \frac{\begin{array}{c} V(x) = \iota \\ H(\iota) = (C, F, \textbf{unlocked}) \\ \iota \notin \mathcal{L} \\ H' = H[\iota \mapsto (C, F, \textbf{locked})] \\ \mathcal{L}' \equiv \mathcal{L} \cup \{\iota\} \end{array}}{\langle H;\, V;\, (\mathcal{L}, \texttt{lock}(x)\ \texttt{in}\ e)\rangle \hookrightarrow \langle H';\, V;\, (\mathcal{L}', \texttt{locked}_\iota\{e\})\rangle}$$

$$\text{DYN\_EVAL\_LOCK\_REENTRANT}\ \frac{\begin{array}{c} V(x) = \iota \\ H(\iota) = (C, F, \textbf{locked}) \\ \iota \in \mathcal{L} \end{array}}{\langle H;\, V;\, (\mathcal{L}, \texttt{lock}(x)\ \texttt{in}\ e)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, e)\rangle}$$

$$\text{DYN\_EVAL\_LOCK\_RELEASE}\ \frac{\begin{array}{c} H(\iota) = (C, F, \textbf{locked}) \\ \mathcal{L}' \equiv \mathcal{L} \backslash \{\iota\} \\ H' = H[\iota \mapsto (C, F, \textbf{unlocked})] \end{array}}{\langle H;\, V;\, (\mathcal{L}, \texttt{locked}_\iota\{v\})\rangle \hookrightarrow \langle H';\, V;\, (\mathcal{L}', v)\rangle}$$

$$\text{DYN\_EVAL\_VAR}\ \frac{V(x) = v}{\langle H;\, V;\, (\mathcal{L}, x)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, v)\rangle}$$

$$\text{DYN\_EVAL\_CONSUME}\ \frac{\begin{array}{c} V(x) = v \\ V' = V[x \mapsto \texttt{null}] \end{array}}{\langle H;\, V;\, (\mathcal{L}, x)\rangle \hookrightarrow \langle H;\, V';\, (\mathcal{L}, v)\rangle}$$

$$\text{DYN\_EVAL\_SELECT}\ \frac{\begin{array}{c} V(x) = \iota \\ H(\iota) = (C, F, L) \\ \textbf{fields}\,(C)(f) = t \\ F(f) = v \end{array}}{\langle H;\, V;\, (\mathcal{L}, x.f)\rangle \hookrightarrow \langle H;\, V;\, (\mathcal{L}, v)\rangle}$$

$$\text{DYN\_EVAL\_UPDATE}\ \frac{\begin{array}{c} V(x) = \iota \\ H(\iota) = (C, F, L) \\ \textbf{fields}\,(C)(f) = t' \\ H' = H[\iota \mapsto (C, F[f \mapsto v], L)] \end{array}}{\langle H;\, V;\, (\mathcal{L}, x.f = v)\rangle \hookrightarrow \langle H';\, V;\, (\mathcal{L}, \texttt{null})\rangle}$$

$$\text{DYN\_EVAL\_NEW}\ \frac{\begin{array}{c} \textbf{fields}\,(C) \equiv f_1 : t_1, \ldots, f_n : t_n \\ F \equiv f_1 \mapsto \texttt{null}, \ldots, f_n \mapsto \texttt{null} \\ \iota \textbf{ fresh} \\ H' = H[\iota \mapsto (C, F, \textbf{unlocked})] \end{array}}{\langle H;\, V;\, (\mathcal{L}, \texttt{new}\ C)\rangle \hookrightarrow \langle H';\, V;\, (\mathcal{L}, \iota)\rangle}$$

$$\text{DYN\_NPE\_SELECT}\ \frac{V(x) = \texttt{null}}{\langle H;\, V;\, (\mathcal{L}, x.f)\rangle \hookrightarrow \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\text{DYN\_NPE\_UPDATE}\ \frac{V(x) = \texttt{null}}{\langle H;\, V;\, (\mathcal{L}, x.f = v)\rangle \hookrightarrow \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\text{DYN\_NPE\_CALL}\ \frac{V(x) = \texttt{null}}{\langle H;\, V;\, (\mathcal{L}, x.m(v))\rangle \hookrightarrow \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\text{DYN\_NPE\_LOCK}\ \frac{V(x) = \texttt{null}}{\langle H;\, V;\, (\mathcal{L}, \texttt{lock}(x)\ \texttt{in}\ e)\rangle \hookrightarrow \langle H;\, V;\, \textbf{NPE}\rangle}$$

$$\text{DYN\_EXCEPTION\_CONTEXT}\ \frac{\langle H;\, V;\, (\mathcal{L}, e)\rangle \hookrightarrow \langle H';\, V';\, \textbf{EXN}\rangle}{\langle H;\, V;\, (\mathcal{L}, E[e])\rangle \hookrightarrow \langle H';\, V';\, \textbf{EXN}\rangle}$$

$$\text{DYN\_EXCEPTION\_ASYNC\_LEFT}\ \frac{}{\langle H;\, V;\, \textbf{EXN} \,||\, T_2 \rhd e\rangle \hookrightarrow \langle H;\, V;\, \textbf{EXN}\rangle}$$

$$\text{DYN\_EXCEPTION\_ASYNC\_RIGHT}\ \frac{}{\langle H;\, V;\, T_1 \,||\, \textbf{EXN} \rhd e\rangle \hookrightarrow \langle H;\, V;\, \textbf{EXN}\rangle}$$

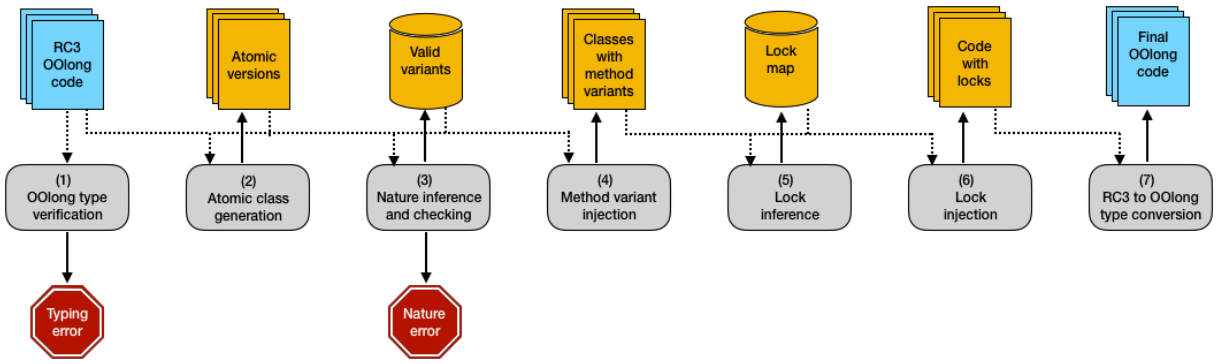

Figure 3.1: The compilation stages of a RC^3 program

3.2 Overview

In order to compile a RC^3 program there are several stages that need to be performed. As we can see in figure 3.1 in brief, we start by checking the program for typing errors. Afterwards we add **atomic** annotations to the classes and interfaces. What follows is an analysis of the variables in terms of whether they are **atomic** or not. One of the last steps involves the creation of two method variants: an atomic version and a regular one. The last three stages can be summarized into assuring that the **atomic** resources are exclusively accessed.

The idea of the RC^3 model is that the resources are the ones that are being annotated with the **atomic** keyword. Thanks to the fact that this model can infer which variables need to be **atomic**, the programmer doesn't have to insert all the **atomic** annotations in his program, but just in some methods and fields of that program. Because RC^3 is built on OOlong, the input program is an OOlong program with atomic annotations in both the fields and types of methods. Then, if the programmer doesn't make a mistake with the program's language, its variable's type or put an **atomic** annotation in an incorrect place (they must be consistent), the RC^3 model's type system can infer which other variables are supposed to be **atomic** or **non-atomic**, including classes and interfaces. If the programmer makes a mistake, then the RC^3 model's type system will output an error. However, it can be the case that the model's type system rejects a program even with correct atomic annotations, due to the analysis of the model being static.

First stage

As we can see in figure 3.1, the first thing to do is to just verify if the program introduced by the programmer without the **atomic** annotations is indeed an OOlong program. So all the **atomic** annotations inserted are removed using a function called `strip`. The resulting program is analyzed. If it isn't an OOlong program, then a type error is generated.

Second stage

In the second stage, provided that the program passed the first stage, the **atomic** counterparts are generated for all classes and interfaces in the source code. This means that, the program will have all the

Table 3.4: The atomic class generation function

$$
\begin{aligned}
\textbf{atogen}(Ids\ Cds\ e) &= Ids\ \textbf{atogen}(Ids)\ Cds\ \textbf{atogen}(Cds)\ e \\
\textbf{atogen}(\textbf{interface}\ J\ \{Msigs\}\ Ids) &= \textbf{interface atomic}\ J\ \{Msigs\}\ \textbf{atogen}(Ids) \\
\textbf{atogen}(\textbf{interface}\ J\ \textbf{extends}\ J_1, J_2\ Ids) &= \textbf{interface atomic}\ J\ \textbf{extends atomic}\ J_1, \textbf{atomic}\ J_2\ \textbf{atogen}(Ids) \\
\textbf{atogen}(\textbf{class}\ D\ \textbf{implements}\ J\ \{Fds\ Mds\}\ Cds) &= \textbf{class atomic}\ D\ \textbf{implements atomic}\ J \\
&\quad \{\textbf{atogenfield}(D, J, Fds)\ Mds\}\ \textbf{atogen}(Cds) \\
\textbf{atogen}(\epsilon) &= \epsilon \\
\textbf{atogenfield}(D, J, f : D\ Fds) &= f : \textbf{atomic}\ D\ \textbf{atogenfield}(D, J, Fds) \\
\textbf{atogenfield}(D, J, f : J\ Fds) &= f : \textbf{atomic}\ J\ \textbf{atogenfield}(D, J, Fds) \\
\textbf{atogenfield}(D, J, f : t\ Fds) &= f : t\ \textbf{atogenfield}(D, J, Fds) \ \text{ for } t \neq D \text{ and } t \neq J \\
\textbf{atogenfield}(D, J, \epsilon) &= \epsilon
\end{aligned}
$$

classes and interfaces duplicated: one will be the regular version and the other its **atomic** counterpart. The function that does this is called **atogen**, and it's defined in table 3.4.

Third stage

The third stage is called *Nature Inference*, and here the Type System verifies that the program can be given a certain type and nature, and produces as output a set of restrictions. After having not only some methods and fields, but also all classes and interfaces with **atomic** annotations, we also need to infer and extend the atomicity qualification to variables. We mustn't forget that our objective is to turn **atomic** the resources that are concurrently accessed. Therefore, a new notion is going to be introduced: the atomicity nature notion, which can either be **atomic** or **non-atomic**. Furthermore, a value with **atomic** nature is an **atomic** resource. In order to isolate the values that must be atomically manipulated in concurrent accesses, **non-atomic** resources cannot be assigned to **atomic** variables and an **atomic** resource can only be assigned to a variable with **non-atomic** nature if it does not become accessible to more than one thread (because then there won't be a concurrency problem). Now we have two definitions: the atomicity, which is the annotations that have been explicitly made with the **atomic** keyword, and the nature notion referring to an implicit atomicity that must be inferred.

The syntax of natures is defined in table 3.5, and besides the base nature values previously mentioned **atomic** and **non-atomic**, a nature can also be undefined ($\perp$), and is used when we don't now the nature yet. Because the nature of local variables is inherited from the expression assigned to them, it is possible that we can't infer information about the expression's nature in a small-step analysis, thus leading us to use nature variables ($\check{x}$) instead of local variables. These nature variables will be generated from the variables *x* in the code and are to be matched with other natures. We also have conditional natures represented by $(\eta_1, \eta_2)\ ?\ \eta_3\ :\ \eta_4$, that means that if there is a match between η_1 and η_2, the value taken is η_3, if not, then η_4. The last option that a nature can be is a call to a method denoted by $t.m(\eta')$, where *t* is the type of the target object, *m* the method's name and η' the nature of the value passed as argument. This table shows the output of the type system, that is a set of method constraints. The method constraints map the method name into nature constraint systems, which in turn is a set of

Table 3.5: Type System Output

$$
\begin{array}{rcl}
\eta \in \text{Nature} & ::= & \textbf{atomic} \mid \textbf{non-atomic} \mid \bot \mid \check{x} \mid (\eta_1, \eta_2)\ ?\ \eta_3 : \eta_4 \mid t.m(\eta') \\
\\
\text{Nature Constraint} & = & \text{Nature} \times \text{Nature} \\
\text{Ncs} \in \text{Nature Constraint System} & \subset & \wp(\text{Nature Constraint}) \\
\\
\text{Method Constraints} & = & \text{MethodName} \mapsto \text{Nature Constraint System} \\
\text{Mcs} \in \text{Method Constraint System} & = & \wp(\text{Method Constraints}) \\
\\
\text{Type System Output} & \subset & \text{Mcs}
\end{array}
$$

nature-related constraints that have tuples of two natures.

Fourth stage

As for the fourth stage, it creates several variants of the methods, corresponding to whether the nature of the actual arguments is atomic or not. This way we resolve an overloading of the method name where the same method name defines different methods, with different signatures and behaviors. The choice of which variants of a method to call will depend on the nature of the input arguments, and typing those method calls requires checking if the nature of the expressions assigned to the parameters are compatible with the nature of the mentioned parameters. Each method call is replaced with a call to the right method definition variant. Furthermore, when the result of a method call is assigned to a variable, the nature of both must be compatible. The resulting code will then not contain ambiguities respecting the nature of a variable.

Last stages

Relatively to the last stages, lock inference and lock injection, they will inject locks automatically in order to a assure that the **atomic** resources are exclusively accessed. However this is still under development.

3.3 Type System

3.3.1 Well-Formed Program

In OOlong's paper [30] we have the definitions for well-formed OOlong program, that consists on well-formed classes and interfaces, as well as a well-typed starting expression. In order for an interface to be well-formed, its method signatures need to mention only well-formed types, and in case of an inheriting interface, the interfaces that it extends need to be well-formed. A class is well-formed if it implements all the methods in its interface, and if all fields and methods are well-formed. In turn, a field is well-formed if its type is well-formed, and a method is well-formed if, under an environment containing the single parameter the type of the current this, its body has the type specified as the method's return type.

Before we analyze the differences between OOlong and RC^3, let us first define the typing judgments:

Table 3.6: Well-formed programs in RC[3].

$$\boxed{\vdash P : t \triangleright \text{Mcs}} \qquad \text{WF\_PROGRAM}\ \frac{\forall\, Cd \in Cds. \vdash Cd \triangleright \text{Mcs}_{Cd} \quad \epsilon \vdash e : (t, \eta) \triangleright \text{Ncs}}{\vdash Ids\ Cds\ e : t \triangleright \bigcup_{Cd \in Cds} . \text{Mcs}_{Cd} \cup \{\text{main} : (\text{Ncs}, \eta)\}}$$

$$\boxed{\vdash Id} \qquad \text{WF\_INTERFACE}\ \frac{\forall\, m(x : t) : t' \in Msigs. \vdash t \wedge \vdash t'}{\vdash \texttt{interface}\ I\ \{\ Msigs\ \}}$$

$$\text{WF\_INTERFACE\_EXTENDS}\ \frac{\begin{array}{c}\vdash I_1 \\ \vdash I_2\end{array}}{\vdash \texttt{interface}\ I\ \texttt{extends}\ I_1, I_2}$$

$$\boxed{\vdash Cd \triangleright \text{Mcs}} \qquad \text{WF\_CLASS}\ \frac{\forall\, Md \in Mds. \texttt{this} : C \vdash Md \triangleright \text{Mcs}_{Md}}{\vdash \texttt{class}\ C\ \texttt{implements}\ I\ \{\ Fds\ Mds\ \} \triangleright \bigcup_{Md \in Mds} \text{Mcs}_{Md}}$$

$$\boxed{\vdash Fd} \qquad \text{WF\_FIELD}\ \frac{\vdash t}{\vdash f : t}$$

$$\boxed{\Gamma \vdash Md \triangleright \text{Mcs}}$$

$$\text{WF\_METHOD}\ \frac{\begin{array}{c}\texttt{this} : C, x : t \vdash e : (t', \eta') \triangleright \text{Ncs}' \\ if\ \textbf{natureof}(t') = \textbf{atomic}\ then\ \text{Ncs} = \{(\eta', \textbf{atomic})\}\ else\ \text{Ncs} = \emptyset\end{array}}{\texttt{this} : C \vdash \texttt{def}\ m(x : t) : t'\ \{\ e\ \} \triangleright \{m : \text{Ncs}' \cup \text{Ncs} \cup \{(\textbf{this}, \textbf{natureof}(C)), (\textbf{return}, \eta')\}, \}}$$

$$\Gamma \vdash e : (t, \eta) \triangleright \text{Ncs}$$

Here, the Γ is the typing environment, *t* is the type, η the nature of expression *e* and Ncs is the output of the typing judgment, that is the Nature Constraint System produced from the analysis of *e*.

Now we can see that the well-formed definitions that differ from OOlong are the well-formed program, well-formed method and well-formed class, that define the generation of the previously mentioned in Section 3.2 constraints. Those rules are presented in table 3.6, along with the rules that are common to OOlong. Besides the new typing judgments regarding expressions, now the rules WF_PROGRAM, WF_CLASS and WF_METHOD output a set of Method Constraints (Mcs). Each of these sets maps the method's name into a set of constraints that has the set with the constraints that came from the method's implementation, the information that the variable `this` has the same nature that the one declared as the class type and that the nature of `return` has the same nature of the method's implementation. If this one is atomic, then the nature's return will also be so. The output of WF_CLASS has all the constraints of the class' methods, and the output of WF_PROGRAM includes those for all of the program's classes, along with the nature and nature constraints of the program's starting expression.

3.3.2 Well-formed typing and subtyping

The rules for well-formed types and subtyping, typing environment and the frame rule are equal to OOlong, as we can see in table 3.7. So from top to bottom, each class or interface in the program corresponds to a well-formed type. The subtyping relation between two types is transitive and reflexive,

and is defined by the interface hierarchy of the program. An environment Γ is well-formed if it maps variables to well-formed types and locations to valid class types. Last but not least, the frame rule is the combination of disjoint typing environments (which may share locations ι). This is used to prevent new threads from sharing variables.

3.3.3 Typing Expressions

In order to adapt the expressions' rules from OOlong to be well-formed in RC^3, a lot of changes were made. The new rules are presented in Table 3.8. Notwithstanding, they are all based on the changes of the typing judgment. Before, the variables were looked up in the well-formed environment (WF-VAR) and introduced using let bindings (WF-LET). Now, we also have to add the natures and the nature constraint system to the new typing judgments definitions. The variables looked up in the environment generate a nature variable, which is the nature of the variable because we can't infer the nature yet, as seen in Section 3.2, and so the nature constraint system is empty (WF-VAR). When introducing the variables using let-bindings, this rule assures that the nature of the variable x must have the same nature (η_1) as the expression that is assigned to it (e_1), and outputs the union of that constraint with the two expressions nature constraint system (WF-LET). This happens because when the nature of an object isn't marked as atomic, it is only sort out when assigned to a variable with a nature. As such, upon creation an object's nature is open (and atomic if marked as atomic) (WF-NEW). Before, this rule only had the restriction that any class in the program could be instantiated. Likewise, the primitive values null don't have a defined nature, and just as they can be given any well-formed type, they can also be assigned to any variable (WF-NULL). In cast rule (WF-CAST) there is no surprise: it continues to only support upcasts, but the nature of the expression is preserved and the rule generates the same nature constraint system as the expression.

Before proceeding, let's just define some auxiliary functions that are presented in table 3.9: **atomicV**, **discardatomic** and **natureof**. The **atomicV** and **discardatomic** return, respectively, the atomic and regular versions of a given type. The **natureof** just informs what is the nature of a given type.

When selecting a method or a field from an object, in order to select the right class: original or atomic version, we need to know the nature of such object. However this may not be known in a small-step analysis. As such, the rules for well-formed update, select and call output conditional natures that depend on the target object's nature. Regarding the rules for update and select, fields are looked up with the helper function `fields`, and may only be looked up in class types (as interfaces do not define fields). In rule well-formed update, in addition to field updates having the Unit type, and outputting the conditional nature as previously mentioned, it also outputs the nature constraint system of the expression and its nature. Because the result of a field update is **null**, the nature is undefined (WF-UPDATE). The select rule is very similar, but instead of having the Unit type it has the field's type, the output is now empty and the conditional nature is in its nature, resulting in it being stored (WF-SELECT). The well-formed call rule is a little different: in OOlong, method calls require that the argument matches exactly the parameter type of the method signature. Additionally, we have the new output with the Nature Constraints System

Table 3.7: Well-formed typing and subtyping.

$\boxed{\vdash t}$ *(Well-formed types)*

$$\text{T\_WF\_CLASS}\ \frac{\texttt{class}\ C\ \texttt{implements}\ I\ \texttt{\{ \_ \}} \in P}{\vdash C}$$

$$\text{T\_WF\_INTERFACE}\ \frac{\texttt{interface}\ I\ \texttt{\{ \_ \}} \in P}{\vdash I}$$

$$\text{T\_WF\_INTERFACE\_EXTENDS}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, I_2 \in P}{\vdash I} \qquad \text{T\_WF\_UNIT}\ \frac{}{\vdash \texttt{Unit}}$$

$\boxed{t_1 <: t_2}$ *(Subtyping)*

$$\text{T\_SUB\_CLASS}\ \frac{\texttt{class}\ C\ \texttt{implements}\ I\ \texttt{\{ \_ \}} \in P}{C <: I}$$

$$\text{T\_SUB\_INTERFACE\_LEFT}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, I_2 \in P}{I <: I_1}$$

$$\text{T\_SUB\_INTERFACE\_RIGHT}\ \frac{\texttt{interface}\ I\ \texttt{extends}\ I_1, I_2 \in P}{I <: I_2} \qquad \text{T\_SUB\_TRANS}\ \frac{t_1 <: t_2 \quad t_2 <: t_3}{t_1 <: t_3}$$

$$\text{T\_SUB\_EQ}\ \frac{\vdash t}{t <: t}$$

$\boxed{\vdash \Gamma}$ *(Well-formed environment)*

$$\text{WF\_ENV}\ \frac{\forall\, x : t \in \Gamma. \vdash t \quad \forall\, \iota : C \in \Gamma. \vdash C}{\vdash \Gamma}$$

$\boxed{\Gamma_1 = \Gamma_2 + \Gamma_3}$ *(Frame Rule)*

$$\text{WF\_FRAME}\ \frac{\forall\, \gamma : t \in \Gamma_2. \Gamma_1(\gamma) = t \quad \forall\, \gamma : t \in \Gamma_3. \Gamma_1(\gamma) = t \quad (\textbf{vardom}\,(\Gamma_2) \cap \textbf{vardom}\,(\Gamma_3)) \equiv \emptyset}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

Table 3.8: Typing of RC^3 expressions

$\boxed{\Gamma \vdash e : (t, \eta) \triangleright \ \mathsf{Ncs}}$ *(RC^3 Typing Expressions)*

$$\text{WF\_VAR}\ \frac{\begin{array}{c}\vdash \Gamma \\ \Gamma(x) = t\end{array}}{\Gamma \vdash x : (t, \textbf{natvar}(x)) \triangleright \ \emptyset}$$

$$\text{WF\_LET}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : (t_1, \eta_1) \triangleright \ \mathsf{Ncs}_1 \\ \Gamma, x : t_1 \vdash e : (t, \eta) \triangleright \ \mathsf{Ncs}\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e : (t, \eta) \triangleright \ \mathsf{Ncs}_1 \cup \mathsf{Ncs} \cup \{(\textbf{natvar}(x), \eta_1)\}}$$

$$\text{WF\_CALL}\ \frac{\begin{array}{l}\Gamma(x) = t1 \\ \textbf{msigs}\,(t1)(m) = y : t2 \rightarrow t \\ \Gamma \vdash e : (t2, \eta) \triangleright \ \mathsf{Ncs} \\ \eta' = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \textbf{atomicV discardatomic}(t1).m(\eta) : \textbf{discardatomic}(t1)).m(\eta) \\ \eta'' = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \textbf{atomicV discardatomic}(t1).m.\eta.ret : \textbf{discardatomic}(t1)).m.\eta.ret\end{array}}{\Gamma \vdash x.m(e) : (t, \eta'') \triangleright \ \mathsf{Ncs} \cup \{(\eta'', \eta')\}}$$

$$\text{WF\_CAST}\ \frac{\begin{array}{c}\Gamma \vdash e : (t', \eta) \triangleright \ \mathsf{Ncs} \\ t' <: t\end{array}}{\Gamma \vdash (t)e : (t, \eta) \triangleright \ \mathsf{Ncs}}$$

$$\text{WF\_SELECT}\ \frac{\begin{array}{l}\Gamma \vdash x : (C, \eta') \triangleright \ \emptyset \\ \textbf{fields}\,(C)(f) = t \\ \textbf{natureof}(\textbf{fields}(\textbf{atomicV discardatomic}(t))(f)) = \eta_1 \\ \textbf{natureof}(\textbf{fields}(\textbf{discardatomic}(t))(f)) = \eta_2 \\ \eta = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \eta_1 : \eta_2\end{array}}{\Gamma \vdash x.f : (t, \eta) \triangleright \ \emptyset}$$

$$\text{WF\_LOC}\ \frac{\begin{array}{c}\vdash \Gamma \\ \Gamma(\iota) = C \\ C <: t\end{array}}{\Gamma \vdash \iota : (t, \textbf{natureof}(C)) \triangleright \ \emptyset}$$

$$\text{WF\_NULL}\ \frac{\begin{array}{c}\vdash \Gamma \\ \vdash t\end{array}}{\Gamma \vdash \texttt{null} : (t, \bot) \triangleright \ \emptyset}$$

$$\text{WF\_UPDATE}\ \frac{\begin{array}{l}\Gamma \vdash x : (C, \eta') \triangleright \ \emptyset \\ \Gamma \vdash e : (t'', \eta'') \triangleright \ \mathsf{Ncs} \\ \textbf{fields}\,(C)(f) = t \\ \textbf{natureof}(\textbf{fields}(\textbf{atomicV discardatomic}(t))(f)) = \eta_1 \\ \textbf{natureof}(\textbf{fields}(\textbf{discardatomic}(t))(f)) = \eta_2 \\ \eta = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \eta_1 : \eta_2\end{array}}{\Gamma \vdash x.f = e : (\texttt{Unit}, \bot) \triangleright \ \mathsf{Ncs} \cup \{(\eta, \eta'')\}}$$

$$\text{WF\_NEW}\ \frac{\begin{array}{l}\vdash \Gamma \\ \vdash C \\ \eta = (\textbf{natureof}(C), \textbf{atomic})\ ?\ \textbf{atomic} : \bot\end{array}}{\Gamma \vdash \texttt{new}\ C : (C, \eta) \triangleright \ \emptyset}$$

$$\text{WF\_FJ}\ \frac{\begin{array}{c}\Gamma = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e_1 : (t_1, \eta_1) \triangleright \ \mathsf{Ncs}_1 \\ \Gamma_2 \vdash e_2 : (t_2, \eta_2) \triangleright \ \mathsf{Ncs}_2 \\ \Gamma \vdash e : (t, \eta) \triangleright \ \mathsf{Ncs}\end{array}}{\Gamma \vdash \texttt{finish}\ \{\ \texttt{async}\ \{\ e_1\ \}\ \texttt{async}\ \{\ e_2\ \}\ \}\,;\, e : (t, \eta) \triangleright \ \mathsf{Ncs}_1 \cup \mathsf{Ncs}_2 \cup \mathsf{Ncs}}$$

(for Type Safety)

$$\text{WF\_LOCK}\ \frac{\begin{array}{c}\Gamma \vdash x : (t_2, \eta_2) \triangleright \ \mathsf{Ncs}_2 \\ \Gamma \vdash e : (t, \eta) \triangleright \ \mathsf{Ncs}\end{array}}{\Gamma \vdash \texttt{lock}(x)\ \texttt{in}\ e : (t, \eta) \triangleright \ \mathsf{Ncs}}$$

$$\text{WF\_LOCKED}\ \frac{\begin{array}{c}\Gamma \vdash e : (t, \eta) \triangleright \ \mathsf{Ncs} \\ \Gamma(\iota) = t_2\end{array}}{\Gamma \vdash \texttt{locked}_\iota\{e\} : (t, \eta) \triangleright \ \mathsf{Ncs}}$$

Table 3.9: Auxiliary functions

$$\textbf{natureof}(t) = \begin{cases} \textbf{atomic} & \text{if } t = \textbf{atomic } D \\ \textbf{atomic} & \text{if } t = \textbf{atomic } J \\ \textbf{non-atomic} & \text{otherwise} \end{cases}$$

$$\textbf{discardatomic}(t) = \begin{cases} D & \text{if } t = D \text{ or } t = \textbf{atomic } D \\ J & \text{if } t = J \text{ or } t = \textbf{atomic } J \end{cases}$$

$$\textbf{atomicV}(t) = \begin{cases} \textbf{atomic } D & \text{if } t = D \text{ or } t = \textbf{atomic } D \\ \textbf{atomic } J & \text{if } t = J \text{ or } t = \textbf{atomic } J \end{cases}$$

for the expression that is the argument of the method call and with the set that checks if the nature that is going to be returned by the method call is the same as the nature of the method's body when called with the current arguments (WF-CALL).

Locations can be given any super type of their class type given in the environment (WF-LOC), and in the RC^3 version, the nature is whatever the class type's nature and there aren't any constraints. Forking new threads requires that the accessed variables are disjoint (WF-FJ). When adopting this rule to the RC^3 model, the changes are what we expected: the nature of the forking is the same as the last expression, and the nature constraint system is the union of all the expressions' Ncs. Locks are only going to be needed in order to prove Type Safety, given that the concurrency control is made solely with the atomic keyword.

3.3.4 Typing Configurations

First, let us see the OOlong [30] definitions and then we can see what are the differences between OOlong and RC^3. An OOlong configuration is well-formed if its heap H, stack V, the collection of threads T and the current lock situation in the system are well-formed. The heap is well-formed under a Γ if all locations in Γ maps to objects in H, the fields of all objects are well-formed under Γ, and all objects in the heap have an entry in Γ. The fields of an object of class C are well-formed if each one of the field's name maps to a value of the corresponding type. A stack V is well-formed under Γ if each variable in Γ maps to a value of the corresponding type in V, and each variable in V has an entry in Γ. Regarding the collection of threads, they are well-formed if all the sub-threads and expressions are well-formed, or, in case of an exceptional state, any type is well-formed. About the current lock situation, we have to consider three cases: a thread, an async of threads and the exceptional state. If it is well-formed for a thread, then all locations in its set of held locks L correspond to objects whose lock status is locked. Moreover, e must have at most one lock and for each **locked** $_\iota$ in the current expression, ι must be in L. The parallel case propagates these properties. Additionally, it requires that the two parallel threads do not hold the same locks in their respective set of held locks. Given that the one that will continue execution after the threads join is the first thread of the async, this needs to hold the locks that are held in the continuation e. In the case of an exceptional state, they are always well-formed.

Now, as we can see in table 3.10, the rules for well-formed heap and well-formed locking are the

Table 3.10: Typing of RC^3 configurations

$\boxed{\Gamma \vdash \langle H; V; T\rangle : t}$ *(Well-formed configuration)*

$$\text{WF\_CFG}\ \frac{\begin{array}{l}\Gamma \vdash H\\ \Gamma \vdash V\\ \Gamma \vdash T : (t,\eta) \rhd \ \mathsf{Ncs}\\ H \vdash_{\mathsf{lock}} T\end{array}}{\Gamma \vdash \langle H; V; T\rangle : (t,\eta) \rhd \ \mathsf{Ncs}} \qquad \text{WF\_HEAP}\ \frac{\begin{array}{l}\forall\, \iota : C \in \Gamma.H(\iota) = (C,F,L) \wedge \Gamma; C \vdash F\\ \forall\, \iota \in \textbf{dom}\,(H).\iota \in \textbf{dom}(\Gamma)\\ \vdash \Gamma\end{array}}{\Gamma \vdash H}$$

$$\text{WF\_FIELDS}\ \frac{\begin{array}{l}\textbf{fields}\,(C) \ \equiv\ f_1 : t_1,\ ..,f_n : t_n\\ \Gamma \vdash v_1 : t_1,\ ..,\Gamma \vdash v_n : t_n\end{array}}{\Gamma; C \vdash f_1 \mapsto v_1,\ ..,f_n \mapsto v_n}$$

$$\text{WF\_VARS}\ \frac{\begin{array}{l}\forall\, x : t \in \Gamma.V(x) = v \wedge \Gamma \vdash v : t\\ \forall\, x \in \textbf{dom}\,(V).x \in \textbf{dom}(\Gamma)\\ \vdash \Gamma\end{array}}{\Gamma \vdash V}$$

$$\text{WF\_T\_ASYNC}\ \frac{\begin{array}{l}\Gamma \vdash T_1 : (t_1,\eta_1) \rhd \ \mathsf{Ncs}_1\\ \Gamma \vdash T_2 : (t_2,\eta_2) \rhd \ \mathsf{Ncs}_2\\ \Gamma \vdash e : (t,\eta) \rhd \ \mathsf{Ncs}\end{array}}{\Gamma \vdash T_1 \,||\, T_2 \rhd e : (t,\eta) \rhd \ \mathsf{Ncs}_1 \cup \mathsf{Ncs}_2 \cup \mathsf{Ncs}}$$

$$\text{WF\_T\_THREAD}\ \frac{\Gamma \vdash e : (t,\eta) \rhd \ \mathsf{Ncs}}{\Gamma \vdash (\mathcal{L}, e) : (t,\eta) \rhd \ \mathsf{Ncs}} \qquad \text{WF\_T\_EXN}\ \frac{\begin{array}{l}\vdash t\\ \vdash \Gamma\end{array}}{\Gamma \vdash \textbf{EXN} : (t,\bot) \rhd \ \emptyset}$$

$$\text{WF\_L\_ASYNC}\ \frac{\begin{array}{l}\textbf{heldLocks}\,(T_1) \cap \textbf{heldLocks}\,(T_2) \ \equiv\ \emptyset\\ \forall\, \iota \in \textbf{locks}\,(e).\iota \in \textbf{heldLocks}\,(T_1)\\ distinctLocks(e)\\ H \vdash_{\mathsf{lock}} T_1\\ H \vdash_{\mathsf{lock}} T_2\end{array}}{H \vdash_{\mathsf{lock}} T_1 \,||\, T_2 \rhd e}$$

$$\text{WF\_L\_THREAD}\ \frac{\begin{array}{l}\forall\, \iota \in \mathcal{L}.H(\iota) = (C,F,\textbf{locked})\\ distinctLocks(e)\\ \forall\, \iota \in \textbf{locks}\,(e).\iota \in \mathcal{L}\end{array}}{H \vdash_{\mathsf{lock}} (\mathcal{L}, e)} \qquad \text{WF\_L\_EXN}\ \frac{}{H \vdash_{\mathsf{lock}} \textbf{EXN}}$$

same. However, we must notice that now the rule for well-formed configuration is different: it outputs the same Ncs and has the same type and nature as the collection of threads. As for the well-formed fields and vars, the difference is not that interesting. The only additional thing is new typing judgment of the values. Finally, regarding the well-formed collection of threads, a thread also needs to have the same typing judgment as the expression, as well as the async of threads. In this last case, the output is the union of the Ncs of each thread and the Ncs that comes from the expression. The exceptional state is not so interesting: the constraints that it outputs are empty and the nature in undefined.

3.4 Results

Lastly, the final objective is to prove the type safety of the model RC^3. As seen before in Chapter 2, Section 2.2, in order to prove type safety it is necessary to prove the lemmas progress and preservation. The progress theorem seen in Theorem 1 is formulated in almost exactly the same way as in as OOlong, but the well-formed configuration is adapted to the RC^3 language, so it will have the natures and output the nature constraint system.

Theorem 1 (Progress)**.** *A well-formed configuration is either done, has thrown an exception, has deadlocked, or can take one additional step.*

$$\begin{aligned}
&\forall \Gamma,\ H,\ V,\ T,\ t\ .\ \Gamma \vdash \langle H; V; T\rangle : (t,\eta) \triangleright \textit{Ncs} \implies \\
&\quad T = (\mathcal{L}, v) \vee T = \textbf{\textit{EXN}} \vee Blocked(\langle H; V; T\rangle) \vee \\
&\quad \exists cfg', \langle H; V; T\rangle \hookrightarrow cfg'
\end{aligned}$$

RC^3 preservation seen in Theorem 2 is also similar to the OOlong. It also needs the same subsumption relation $\Gamma_1 \subseteq \Gamma_2$ between environments, that says that the all the variable mappings in Γ_1 are also in Γ_2. Nevertheless, in RC^3 the program generates the nature constraint system.

Theorem 2 (Preservation)**.** *If $\langle H; V; T\rangle$ types to (t,η) and produces a nature constraint system Ncs under some environment Γ, and $\langle H; V; T\rangle$ steps to some $\langle H'; V'; T'\rangle$, there exists an environment subsuming Γ which types $\langle H'; V'; T'\rangle$ to (t,η') and produces Ncs$'$, such that η can instantiate to η' and Ncs can instantiate to Ncs$'$.*

$$\begin{aligned}
&\forall \Gamma,\ H,\ H',\ V,\ V',\ T,\ T',\ t. \\
&\quad \Gamma \vdash \langle H; V; T\rangle : (t,\eta) \triangleright \textit{Ncs} \wedge \langle H; V; T\rangle \hookrightarrow \langle H'; V'; T'\rangle \implies \\
&\quad\quad \exists \Gamma', \textit{Ncs}', \eta'.\Gamma' \vdash \langle H'; V'; T'\rangle : (t,\eta') \triangleright \textit{Ncs}' \wedge \Gamma \subseteq \Gamma'
\end{aligned}$$

The above results focus on the type information that is treated by the type system, and do not makes assertions about the natures and nature constraints systems that is generated. Formal results about the correctness of the nature related information are under development and depend on other stages of the analysis that are not studied here. They are therefore outside the scope of the type safety proof. Nevertheless, the information about natures has been formalized, which provides some guarantees about the formal consistency of their definitions.

3.5 Chapter Summary

This Chapter started by presenting the Language of the RC^3 model, by describing the old syntax and semantics for OOlong, and showing the new adapted ones for RC^3. Then, it explained that in order to formalize the RC^3 language, we need to change the OOlong language step by step until we reach the RC^3, and presented the stages that make that happen in a figure. After that, the Chapter introduces the type system adapted from OOlong, explaining each well-formed rule of the OOlong and the changes made. This is always followed up by a table for the reader to visualize the formalization of the RC^3 model. Finally, this Chapter ends with the formalization of theorems that prove type safety: progress and preservation.

Chapter 4

Implementation

In order to formalize the RC^3 language in Coq and mechanically prove progress and preservation, we are going to modify the OOlong language step by step until we reach RC^3 language, like we saw in Chapter 3. In this Chapter, we are going to see the OOlong language formalized using the theorem prover Coq without going into detail. Then, we are going to see how the language of RC^3 model was implemented in Coq, and some of the decisions made. We are also going to see some problems that occurred and the respective solutions. Regarding the Coq files for RC^3 model, they are in this repository [51] that the reader can check to follow the implementation.

4.1 Oolong

The OOlong source code is in its github repository: [50]. The reader can check it anytime to understand more about this Section. By following the link, one can see that there are three similar directories with Coq sources of OOlong: vanilla, assert and region. As they say in paper [30], vanilla is the original version of the OOlong's semantics, and the other two versions are the two examples of the extensibility of OOlong. The only directory that matter to us is the vanilla one.

Proof: $\begin{cases} Soundness.v \\ Progress.v \\ Preservation.v \end{cases}$

Auxiliary: $\begin{cases} LibTactics.v \\ CpdtTactics.v \\ Shared.v \\ ListExtras.v \\ Locking.v \end{cases}$

Syntax, semantics and type system: $\begin{cases} Meta.vandMetaProp.v \\ Syntax.vandSyntaxProp.v \\ Types.vandTypesProp.v \\ Dynamic.vandDynamicProp.v \\ Wellformedness.vandWellformednessProp.v \end{cases}$

We have various Coq files in the proof of Type Safety. The file `Soundness.v` (remember that Type Safety can also be called Type Soundness or just soundness if it only refers to preservation and progress), proved using progress and preservation. The files `Progress.v` and `Preservation.v` which contains the proofs of progress and preservation, respectively. `LibTactics.v` and `CpdtTactics.v` contains the LibTactics library from Software foundations and the crush tactic by Adam Chlipala, respectively, which assist the proof of soundness. `Shared.v` also assists the proof, and contains general tactics including the Case library. This Case tactic structures the proofs making them more general (easier to change). `ListExtras.v` contains auxiliary lemmas about lists, and `Locking.v`, lemmas about locking behavior.

For each of the following files, there is a "Prop" file that contains lemmas (with proofs) about properties of the constructs in the files. These files are needed to mechanize the language semantics. Starting with `Meta.v`, it defines all the ID's for variables and the functionality for partial maps. These variables will be used in `Syntax.v`, that defines the syntax of OOlong (Table 3.1without the **atomic** keyword). The static semantics of types and subtyping in table 3.7, and also expressions, are defined in `Types.v`, and dynamic semantics in table 3.3 are defined in `Dynamic.v`. Table 3.6 (without the adaptations for RC^3) of well-formedness rules for static constructs and table 3.10 with the ones for dynamic constructs, are defined in `Wellformedness.v`.

Concerning the RC^3 source code, it is organized in three directories: one for each of the implemented stage. These directories follow the same structure as the OOlong source code, and are going to be explained in each of the following Sections, with the respective list of modified files.

4.2 First stage - strip

In this Section we will add the atomic keyword to the OOlong language. However, in the first stage, atomic annotations are ignored. In this directory, we modified the files presented in table 4.1, along with minimal consistency changes in other files.

Table 4.1: Modified files for stage 1

$\begin{cases} Meta.v \\ Syntax.v \end{cases}$

Like we saw in the previous Chapter, in Section 3.2, the first stage of the model is to verify whether the program, without the atomic annotations, is indeed an OOlong program. First of all, we need to add

the **atomic** keyword to the OOlong language. We do that by adding a new definition (see listing 1) to the `Meta.v` file: the atomicity inductive definition. The set `atomicity` has two constructors: *Atomic* and *NonAtomic*.

```
Inductive atomicity : Set :=
  | Atomic : atomicity
  | NonAtomic : atomicity.
```

Listing 1: Atomicity

Then, we changed the inductive definition of `ty` in file `Syntax.v` (see listing 2), where `class_id` and `interface_id` are just the ID's of classes and interfaces and are represented by a natural number. We changed the constructors `TClass` and `TInterface` from naturals into tuples where the second element is the `atomicity`. As the name implies, `TClass` is the type class and `TInterface` the type interface. We also have TUnit that is the type **Unit**.

```
Inductive ty : Type :=
  | TClass : class_id -> atomicity -> ty
  | TInterface : interface_id -> atomicity -> ty
  | TUnit : ty.
```

Listing 2: Atomicity in types

However, this implied that we needed to change all the definitions that receives a `TClass` or `TInterface` in order to accept the new definition. So we thought that it would be smarter to change `class_id` and `interface_id` into tuples where the second element is the `atomicity`, instead of the type's definition, because in this way we don't have to do all the changes that were needed the other way 3.

```
Definition class_id := prod nat atomicity.
Definition interface_id := prod nat atomicity.
```

Listing 3: Atomicity in ID's

The next step of this stage is to ignore the **atomic** annotations. This can be done by completely removing the `atomicity`, transforming the tuples `class_id` and `interface_id` back again to only natural numbers, or forcing the atomicity of every one of those tuples to be **non-atomic**. We first tried the first option by adding the two definitions in listing 4. However, even though it would be easy to implement, it could cause trouble later on when we would like to test some programs, because all the code is treating the `class_id` (and `interface_id`) as a tuple and not as a variable like the `stripClass` function is changing it to.

```
Definition stripClass (x: class_id) : nat := fst(x).
Definition stripInterface (x: interface_id) := fst(x).
```

Listing 4: Removing atomicity of classes and interfaces id

That is why we tried the second option, and defined the two strip functions in listing 5 by forcing the second element to be `NonAtomic`. Afterwards, we also have to define functions that strip atomicity for

```
Definition stripClass (x: class_id) := (fst x, NonAtomic).
Definition stripInterface (x: interface_id) := (fst x, NonAtomic).
```

Listing 5: Transforming classes and interfaces id into non-atomic

the list of methods, methods signatures and fields. We can see the definition for the list of methods in listing 6, and the other two functions follow the same reasoning.

```
Fixpoint stripListMethod(mds : list methodDecl) : list methodDecl :=
  match mds with
    | nil => nil
    | h :: t => match h with
                          | Method m (sv,t1) t2 e => Method m (sv, (stripType t1)) (stripType t2) e
                end :: stripListMethod t
  end.
```

Listing 6: Transforming list of methods into non-atomic

This function receives a list of methods, and removes atomicity recursively for each method, by applying the *stripType* function to the types that define method. The *stripType* function checks which type it receives and if it is a class or an interface, it applies `stripClass` and `stripInterface`. Finally, we have to apply these functions to the program. Considering that the program is defined by the list of classes and interfaces, and an expression,

```
Definition program := (list classDecl * list interfaceDecl * expr)%type.
```

we have to define three new recursive definitions. One receives the list of classes, scrolls through it and for each class (that is defined by the class id, interface id and the lists of fields and methods, like we saw in the table of the syntax 3.1) applies `stripClass`, `stripInterface` and *stripListMethod*. Another does the same but for the list of interfaces (that are defined by the interface id and the list of methods signatures, or by three interfaces id in case of the inheriting interface). Finally, the last one receives an expression, checks if it is an expression NEW or CAST, and if that is the case, the function strips its atomicity. This way, we removed all the possible **atomic** keywords (by transforming them into **non-atomic**) that the programmer could have added.

The only thing that remains to adapt is the equality for the `class_id`. Now that we have a tuple, we have to introduce new definitions for comparing atomicities. Nevertheless, an interested reader can check those definitions on the RC^3 Coq files repository [51].

This implements the first stage of the model, that upon receiving a program, strips the atomic annotations and checks if it satisfies the OOlong model.

4.3 Second stage - duplicate

In this directory, we modify the files presented in table 4.2, along with minimal consistency changes in other files.

Table 4.2: Modified files for stage 2

$$\begin{cases} Meta.v \\ Syntax.v \end{cases}$$

Regarding the second stage, we have to generate the **atomic** counterparts for all classes and interfaces in the source code, leaving the program with duplicate classes and interfaces. We saw in Section 3.2 that the **atogen** function that will be doing this is the one defined in table 3.4. The objective of this stage is therefore to introduce each of the definitions represented in that table 3.4 in the theorem prover Coq. We are going to have one **atogen** for each input: an **atogen** for programs presented in listing 7, one for the list of interfaces showed in listing 8 and another for the list of classes in listing 9, and lastly one for the the list of fields in listing 10.

```
Definition atogenProg(P : program) :=
  match P with
    | (cs, ids, e) => ((cs ++ atogenCl cs), (ids ++ atogenIn ids), e)
  end.
```

Listing 7: Atogen that receives a program

As we can see, the definition of **atogen** that receives a program is straightforward: it receives a program and duplicates the list of classes and interfaces. So we have the regular list of classes and interfaces, and now we add the **atomic** counterpart of each list. The `atogenProg` in listing 7 needs to call the `atogenIn` in listing 8 and `atogenCl` in listing 9, that are defined next.

```
Fixpoint atogenIn(ids : list interfaceDecl) : list interfaceDecl :=
 match ids with
    | nil => nil
    | h :: t => match h with
                  | Interface i l => Interface (atogenInId i) l
                  | ExtInterface i i0 i1 =>
                                  ExtInterface (atogenInId i) (atogenInId i0) (atogenInId i1)
                end :: atogenIn t
  end.
```

Listing 8: Atogen that receives a list of interfaces

The `atogenIn` in listing 8, is a recursive function that upon receiving a list of interfaces, if the list isn't null, it takes the first element, and sees if it corresponds to a normal interface or an inheriting interface. Then, it transforms all the existing `interface_id` (that we mustn't forget that they are tuples of the natural id number and the atomicity) into **atomic** tuples, and calls again the `atogenIn` function for the rest of the elements of the list. The same goes for `atogenCl` 9. The definitions that they both call: `atogenInId` and `atogenClId`, are very similar to the strip functions in listing 5 of the previous Section 4.2. However, instead of transforming the second element of the tuple into **non-atomic**, these **atogen** functions transforms the second element in **atomic**. The **atogen** that receives classes and the one that receives interfaces in table 3.4 are also already defined. The empty case is represented on each of the **atogen** functions that we are defining, like we can see. We are left with the **atogen** definitions that receive the list of fields.

```
Fixpoint atogenCl(cds : list classDecl) : list classDecl :=
  match cds with
    | nil => nil
    | h :: t => match h with
                  | Cls c i f l0 =>
                       Cls (atogenClId c) (atogenInId i) (atogenfield c i f) l0
                end :: atogenCl t
  end.
```

Listing 9: Atogen that receives a list of classes

As we can see in table 3.4, we must consider 4 cases where the type of the field is: the same as the class received, the same as the interface received, empty, or neither of these cases. The `atogenfield` in listing 10 is also defined as a recursive function, that checks if the first element of the list of fields is a field with type `TClass`, `TInterface` or `TUnit`. Because we don't have that option on the definition of types, the type can't be empty. So, we just check if the list is empty. If the type is `TUnit`, then we don't make any changes, because it fits in the last case of the table 3.4. If it is `TClass` or `TInterface`, we have to check with `eqb_class_id` and `eqb_interface_id` respectively if the `class_id` (or `interface_id`) of the field is the same of the one received. If it is indeed the same one, we apply `atogenClId` (or `atogenInId`) in order to transform the atomicity into **atomic**, if not, it remains the same.

```
Fixpoint atogenfield(c : class_id)(i : interface_id)(fs : list fieldDecl) :=
 match fs with
    | nil => nil
    | h :: t => match h with
                | Field f (TClass c') => match eqb_class_id c c' with
                                          | true => Field f (TClass (atogenClId c))
                                       | false =>  Field f (TClass c')
                                          end
                | Field f (TInterface i') => match eqb_interface_id i i' with
                                          | true => Field f (TInterface (atogenInId i))
                                          | false =>  Field f (TInterface i')
                                             end
                | Field f TUnit => Field f TUnit
                 end :: atogenfield c i t
  end.
```

Listing 10: **Atogen** that receives a list of fields

As result, all the cases of the function **atogen** are covered, and if one introduces an OOlong program, then the model will duplicate all classes and interfaces in order to have a regular and an **atomic** version of each one.

4.4 Third stage - Nature Inference

In this directory, we modified the files presented in table 4.3, along with minimal consistency changes in other files.

This is the last stage that is contemplated by this Thesis. Regarding this stage, there is a lot of things

Table 4.3: Modified files for stage 3

$$\begin{cases} Syntax.v \\ Type.v \\ WellFormedness.v \\ Soundness.v \\ Progress.v \\ Preservation.v \\ TypesProp.v \\ WellformednessProp.v \\ Locking.v \end{cases}$$

to be done here. So we are going to separate it in different Subsections to be easier.

4.4.1 Type System Output

In the previous Chapter 3, we saw that the new notion `nature` is needed in order to save the variables' nature to compare them later with the atomicity and make a decision about what is the nature of each resource. This new notion is implemented in file `Syntax.v` like we can see in listing 11, along with the nature constraints and the method constraints. Comparing to table 3.5, we have an inductive definition of `nature` where `NAtomic` has a keyword different from `atomic` in order to distinguish the atomic nature from **atomic** atomicity. The same goes for `NNonAtomic`. `NVar` is a constructor that receives a variable and transforms its type into a nature, so its the representation of nature variables.

```
Inductive nature : Set :=
  | NAtomic : nature
  | NNonAtomic : nature
  | Undefined : nature
  | NVar : var -> nature
  | ConditionalNature : nature -> nature -> nature -> nature -> nature
  | Call : ty -> method_id -> nature -> nature.

Definition NatureConstraints := (nature * nature)%type.

Definition MethodConstraints := (method_id * (NatureConstraint -> Prop))%type.
```

Listing 11: Type System Output

Nature constraints are composed by tuples of two natures, just as represented in the definition `NatureConstraints` in listing 11. The Nature Constraint System is going to be represented by "NCS" and is the set of all the nature constraint's subsets: the power set of the nature constraint. The type of this power set is `Ensemble` of Nature Constraint (where `Ensemble` is a function that maps something of type `Type` into a `Prop`, which in turn is a proposition that can be provable or unprovable), therefore we need to call the `Ensemble`'s library.

The type of Method Constraint System is similar: `Ensemble` of Method Constraints, and is going to be represented by "MCS". Given that this last definition is the map of the method name into the Nature Constraint System, it is implemented in Coq with a tuple where the first element is the ID of the method

(therefore its name), and the second element has the type `NatureConstraint -> Prop` that is the same as `Ensemble` of Nature Constraint. The Mcs is going to be the output of the well-formed program, that we are going to implement later in 4.4.3

4.4.2 Typing Expressions

To implement the new rules for well-formed expressions, it is first necessary to change the definition of typing judgments. We do that by adding to the OOlong's typing judgments notation, the nature and the Nature Constraint. The notation here in listing 12 is a little different from the one seen in Chapter 3. Instead of "(type, nature) ▷ Ncs", we have "type # nature ▷ Ncs", and this is the definition that we are going to use in the rest of the implementation.

```
Reserved Notation " P ';' Gamma '|-' expr '\in' ty "  (*OOlong*)

Reserved Notation " P ';' Gamma '|-' expr '\in' ty # nature '|>' NCS" (*RC3*)
```

Listing 12: Typing Judgments for OOlong (first) and RC^3 (second)

```
Inductive hasType (P : program) (Gamma : env) :
                        expr -> ty -> nature -> (NatureConstraint -> Prop) -> Prop :=
```

Listing 13: Beginning of the inductive definition `hasType`

The rules for well-formed expressions are defined in the inductive definition called `hasType`, that is in the file `Type.v` (as well as the typing judgments definitions), and its beginning can be seen in the listing 13. In the following listings, we have the code for each respective rule of the table of 3.8 typing expressions. Most of them are pretty similar as we can see in the following figures 4.1 and 4.2 and listings 14 and 15. So we are going to see just some examples. The differences between our files and the Coq files for OOlong are in the typing judgments definitions and in some new variables, that we use **let**-bindings to define them.

```
| T_Loc :
   forall l t c,
    wfEnv P Gamma ->
    Gamma (env_loc l) = Some (TClass c) ->
    subtypeOf P (TClass c) t ->
    P ; Gamma |- (EVal (VLoc l)) \\in
    t # NatureOf(TClass c) |> (Empty_set NatureConstraint)
```

$$\text{WF\_LOC}\ \frac{\begin{array}{c}\vdash \Gamma \\ \Gamma(\iota) = C \\ C <: t\end{array}}{\Gamma \vdash \iota : (t, \textbf{natureof}(C)) \triangleright \emptyset}$$

Figure 4.1: Well-formed Locations

```
| T_Lock :
  forall x e t t2 n ncs n2 ncs2,
   P ; Gamma |- (EVar x) \in t2 # n2 |> ncs2 ->
   P ; Gamma |- e \in t # n |> ncs ->
   no_locks e ->
   P ; Gamma |- ELock x e \in t # n |> ncs
```

$$\text{WF\_LOCK}\ \frac{\Gamma \vdash x : (t_2, \eta_2) \triangleright \text{Ncs}_2 \qquad \Gamma \vdash e : (t, \eta) \triangleright \text{Ncs}}{\Gamma \vdash \texttt{lock}(x)\ \texttt{in}\ e : (t, \eta) \triangleright \text{Ncs}}$$

Figure 4.2: Well-formed Lock

The function `no_locks` in the figure 4.2 comes from OOlong assures that the expression that it receives has no locks.

```
| T_Call :
  forall x m e t t1 t2 n msigs y ncs,
   Gamma (env_var x) = Some t1 ->
   methodSigs P t1 msigs ->
   methodSigLookup msigs m = Some (MethodSig m (y, t2) t) ->
   P ; Gamma |- e \in t2 # n |> ncs ->
   let n1 := NatureOf(addAtomic (discardAtomic t1)) in
    let n3 := NatureOf(addAtomic (discardAtomic t)) in
     let n' := ConditionalNature (NVar x) NAtomic
                    n1 (NatureOf(discardAtomic t1)) in
      let n'' := ConditionalNature (NVar x) NAtomic
                         n3 (NatureOf(discardAtomic t)) in
       P ; Gamma |- ECall x m e \in
                 t # n'' |> Union NatureConstraint ncs (Singleton NatureConstraint (NN n' n))
```

$$\text{WF\_CALL}\ \frac{\begin{array}{l}\Gamma(x) = t1 \\ \textbf{msigs}\,(t1)(m) = y : t2 \rightarrow t \\ \Gamma \vdash e : (t2, \eta) \triangleright \text{Ncs} \\ \eta' = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \textbf{atomicV discardatomic}(t1).m(\eta) : \textbf{discardatomic}(t1)).m(\eta) \\ \eta'' = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \textbf{atomicV discardatomic}(t1).m.\eta.ret : \textbf{discardatomic}(t1)).m.\eta.ret\end{array}}{\Gamma \vdash x.m(e) : (t, \eta'') \triangleright \text{Ncs} \cup \{(\eta'', \eta')\}}$$

Listing 14: Well-formed Call

The functions implemented in Coq `discardAtomic`, `addAtomic` and `NatureOf` correspond to the functions **discardatomic**, **atomicV** and **natureof**, respectively, and they are defined in `Syntax.v`. `discardAtomic`, `addAtomic` and `NatureOf` are similar to each other and are implemented exactly like presented in table 3.9, with the `match`, as we can see in the example in the listing 16 for the `NatureOf` function. Here, if the variable `t` that is a type, has type `TClass` with atomicity **atomic**, it will return `NAtomic`. If it has type `TInterface` with atomicity **atomic**, it will also return `NAtomic`. If it doesn't match any of this descriptions, then it will return `NNonAtomic`. We can see this functions being used in the examples in the listing 14 and listing 15.

```
| T_Select :
  forall x f c t fs nclass,
   P ; Gamma |- (EVar x) \in TClass c # nclass |> (Empty_set NatureConstraint) ->
   fields P (TClass c) = Some fs ->
   fieldLookup fs f = Some (Field f t) ->
    let n1 := NatureOf(addAtomic (discardAtomic (FieldsType (Field f t))))in
     let n2 := NatureOf(discardAtomic (FieldsType (Field f t))) in
      let n := ConditionalNature (NVar x) NAtomic n1 n2 in
       P ; Gamma |- ESelect x f \in t # n |> (Empty_set NatureConstraint)
```

$$\text{WF\_SELECT}\ \frac{\begin{array}{l}\Gamma \vdash x : (C, \eta') \triangleright \emptyset \\ \textbf{fields}\,(C)(f) = t \\ \textbf{natureof}(\textbf{fields}(\textbf{atomicV discardatomic}(t))(f)) = \eta_1 \\ \textbf{natureof}(\textbf{fields}(\textbf{discardatomic}(t))(f)) = \eta_2 \\ \eta = (\textbf{natvar}(x), \textbf{atomic})\ ?\ \eta_1 : \eta_2\end{array}}{\Gamma \vdash x.f : (t, \eta) \triangleright \emptyset}$$

Listing 15: Well-formed Select

```
Definition NatureOf (t : ty) : nature :=
  match t with
    | TClass (c, Atomic) => NAtomic
    | TInterface (i, Atomic) => NAtomic
    | _ => NNonAtomic
  end.
```

Listing 16: Definition of **natureof**

4.4.3 Well-Formed Program

Here we implement the rules for the well-formed program, presented in table 3.6. We changed the rules wfMethodDecl, wfClassDecl and wfProgram, that are in the file `WellFormedness.v`. The difference from OOlong is the type system output that is new and was already mentioned in Section 4.4.1, and of course the typing judgments. In listing 17, we had to add the type of the type system output, that is $(MethodConstraints- > Prop)$ and the rest are the exact modifications as we saw in table 3.6.

```
Inductive wfProgram : program -> ty -> (MethodConstraints -> Prop) -> Prop :=
  | WF_Program :
      forall cds ids e t n ncs mcs,
      forall m_id : method_id,
        Forall (wfClassDecl (cds, ids, e) mcs) cds ->
        Forall (wfInterface (cds, ids, e)) ids ->
        (cds, ids, e); empty |- e \in t # n |> ncs ->
        wfProgram (cds, ids, e) t (Union MethodConstraints mcs
                                        (Singleton MethodConstraints (m_id, ncs))).
```

Listing 17: Definition of well-formed Program

The same goes for the well-formed class in listing 18, with the slightly change that the Mcs has to go before the `classDecl` for the simple reason that is going to benefit us when proving a lemma.

Regarding the well-formed method in listing 19, this may be a little hard to understand. Here, instead of the `if` in one of the premises, we have the match technique with **let**-bindings, that works the same way as the `if`. This is because the `if` in Coq must have something of type Prop in its guard and we want

```
Inductive wfClassDecl (P : program) :
                                (MethodConstraints -> Prop) -> classDecl -> Prop :=
  | WF_Class :
      forall c i fs ms msigs mcs_c mcs_m,
        methodSigs P (TInterface i) msigs ->
        extractSigs ms = msigs ->
        Forall (wfFieldDecl P) fs ->
        Forall (wfMethodDecl P c mcs_m) ms ->
        wfClassDecl P mcs_c (Cls c i fs ms).
```

Listing 18: Definition of well-formed Class

to have a function to see if the result is `NAtomic` or not. The output has the same order change as the well-formed class, thus having the `methodDecl` in last. The variable `this` in the output of WF-METHOD in table 3.6 is actually the nature of `this`.

```
Inductive wfMethodDecl (P : program) (c : class_id) :
                                    (MethodConstraints -> Prop) -> methodDecl -> Prop :=
  | WF_Method :
      forall m x t t' e n' ncs' Nreturn ncs,
(...)
         P ; Gamma |- e \in t' # n' |> ncs') ->
        match NatureOf(t') with
        | NAtomic => let ncs := Singleton NatureConstraint (NN n' NAtomic) in NatureConstraint
        | _ => let ncs := Empty_set NatureConstraint in NatureConstraint
        end ->
        wfMethodDecl P c
          (Singleton MethodConstraints (m,
             Union NatureConstraint (Union NatureConstraint ncs' ncs)
              (Ensembles.Add NatureConstraint
                        (Singleton NatureConstraint (NN (NVar (SV this)) (NatureOf (TClass c))))
                        (NN (NVar Nreturn) n')
              )
                                    )
          )
                (Method m (x, t) t' e).
```

Listing 19: Definition of well-formed Method

4.4.4 Well-Formed Configurations

Continuing in the file `WellFormedness.v`, this also has the definitions for the well-formed configurations in table 3.10. The changes from OOlong are minimal here. In listing 20, the only modification is the new typing judgment for the value that the field's name is going to be mapped into. All the field's name of the list of the fields is represented in Coq by the `forall f`.

```
Inductive wfFields (P : program) (Gamma : env) (c : class_id) (F : dyn_fields) : Prop :=
  WF_Fields :
    forall fs,
      fields P (TClass c) = Some fs ->
      (forall f t, fieldLookup fs f = Some (Field f t) ->
                   exists v n ncs, F f = Some v /\ P; Gamma |- EVal v \in t # n |> ncs) ->
      wfFields P Gamma c F.
```

Listing 20: Definition of well-formed Fields

For the rule of well-formed vars in listing 21, the reasoning is the same: the only change is the typing judgment. The premises that each variable x in V has an entry in Γ is translated into the line (forall x, V x <> None − > Gamma (env_var (DV x)) <> None) that assures that the variable x in V is not empty and the same happens in Γ.

```
Inductive wfVars (P : program) (Gamma : env) (n : nat) (V : dvar_map) : Prop :=
  WF_Vars :
      wfEnv P Gamma ->
      (forall x t, Gamma (env_var (DV x)) = Some t -> exists v n' ncs',
                          V x = Some v /\ P; Gamma |- (EVal v) \in t # n' |> ncs') ->
      (forall x, V x <> None -> Gamma (env_var (DV x)) <> None) ->
      freshVars n V ->
      wfVars P Gamma n V.
```

Listing 21: Definition of well-formed Vars

As for the rule `wfThreads`, we changed the output in listing 22, by adding nature − > (NatureConstraint − > Prop). However the modifications are straightforward based on the table 3.10.

```
Inductive wfThreads (P : program) (Gamma : env) :
                        threads -> ty -> nature -> (NatureConstraint -> Prop) -> Prop :=
  | WF_Thread :
      forall Ls e t n ncs,
        freeVars e = nil ->
        P; Gamma |- e \in t # n |> ncs ->
        wfThreads P Gamma (T_Thread Ls e) t n ncs
  | WF_Async :
      forall T1 T2 e t t1 t2 n n1 n2 ncs ncs1 ncs2,
        freeVars e = nil ->
        P; Gamma |- e \in t # n |> ncs ->
        wfThreads P Gamma T1 t1 n1 ncs1 ->
        wfThreads P Gamma T2 t2 n2 ncs2 ->
        wfThreads P Gamma (T_Async T1 T2 e) t n
                        (Union NatureConstraint (Union NatureConstraint ncs1 ncs2) ncs)
  | WF_EXN :
      forall t Ls,
        wfType P t ->
        wfEnv P Gamma ->
        wfThreads P Gamma (T_EXN Ls) t Undefined (Empty_set NatureConstraint).
```

Listing 22: Definition of well-formed Threads

Finally, in listing 23 we just need to change the output of `wfThreads`, given that we changed it in listing 22, and the output of `wfConfiguration`, in order to have the same one as `wfThreads`.

```
Inductive wfConfiguration (P : program) (Gamma : env) :
                        configuration -> ty -> nature -> (NatureConstraint -> Prop) -> Prop :=
  | WF_Cfg :
      forall H V n T t n' ncs',
        0 < n ->
        wfHeap P Gamma H ->
        wfVars P Gamma n V ->
        wfThreads P Gamma T t n' ncs' ->
        wfLocking H T ->
        wfConfiguration P Gamma (H, V, n, T) t n' ncs'.
```

Listing 23: Definition of well-formed Configuration

4.4.5 Proof of Type Safety

The proof of type safety is made by proving progress and preservation. Both of these lemmas follows the same reasoning as the one made for OOlong.

Progress

```
Theorem progress :
  forall P t' Gamma cfg t mcs n ncs,
    wfProgram P t' mcs ->
    wfConfiguration P Gamma cfg t n ncs ->
    cfg_exn cfg \/ cfg_done cfg \/ cfg_blocked cfg \/
    exists cfg', P / cfg ==> cfg'.
Proof with eauto.
  introv wfP wfCfg.
  inverts wfCfg as Hfresh wfH wfV wfT wfL.
  gen t n ncs.
  induction T; intros; simpl...
  + Case "T = Thread".
    right. eapply single_threaded_progress...
  + Case "T = Async T1 T2 e".
    inverts wfT as Hfree hasType wfT1 wfT2.
    inverts wfL as wfL HL Hdisj wfL1 wfL2.
    right. right.
    pose proof (IHT1 wfL1 t1 n2 ncs1 wfT1) as IH1.
    pose proof (IHT2 wfL2 t2 n3 ncs2 wfT2) as IH2.
    destruct IH1 as [T1EXN|[T1Done|[T1Blocked|T1Steps]]]...
    - SCase "T1 done".
      unfolds in T1Done. unfold threads_done in T1Done.
      destruct T1; try(solve[inv T1Done]).
      destruct IH2 as [T2EXN|[T2Done|[T2Blocked|T2Steps]]]...
      * SSCase "T2 steps".
        destruct T2Steps as [[[[H' V'] n'] T2']].
        right. eexists; eapply EvalAsyncRight...
    - SCase "T1 blocked".
      destruct IH2 as [T2EXN|[T2Done|[T2Blocked|T2Steps]]]...
      * SSCase "T2 steps".
        destruct T2Steps as [[[[H' V'] n'] T2']].
        right. eexists; eapply EvalAsyncRight...
    - SCase "T1 steps".
      destruct T1Steps as [[[[H' V'] n'] T2']].
      right. eexists; eapply EvalAsyncLeft...
Qed.
```

Listing 24: Proof of theorem Progress

The proof in listing 24 is made by induction over the thread structure `T`, thus leading us to the three cases: either `T` is a single thread, two parallel asynchronous threads or has thrown an exception. For each one of this cases, we have to proof that a well-formed configuration is either done, has thrown an exception, has deadlocked, or can take one additional step, replacing `T` by the respective case. The proof mainly resorts to `Dynamic.v` due to this having the rules for concurrency, exceptions and blocking property of a configuration.

- If we are in the case that the thread is in an exceptional state **EXN**, then it's trivial, since it is true that the configuration had indeed thrown an exception.

- If we are in the two parallel asynchronous threads case, then either the configuration has deadlocked or can take one additional step. The induction hypothesis says that the progress proof holds for each of the parallel's threads `T1` and `T2`, so we prove our goal by checking each configuration case for `T1` and `T2`. However we know that if any of the threads has thrown an exception (or is deadlocked), then we just have to use the rule for `dyn-exception-async-` (or `blocked-`) to prove it. If both of them are done, them we use the rule `dyn-eval-async-join`. Lastly, if one of the threads can take one additional step, then we prove with the rule `dyn-eval-async-` .

- The case when `T` is a single thread, is proved by resorting to the auxiliary lemma `single_threaded_progress`, that is also in the file `Progress.v`. This lemma is proved also by induction, but over the current expression. So we have to prove for all the possible expressions one by one, and this is done by applying the rules for the evaluation of the expressions. This lemma also uses other lemmas of files `TypesProp.v` and `WellformednessProp.v` that just like this one were adapted in order to work with natures.

To use the proof of progress, we adapted it to the new rule of well-formed configuration and well-formed program, introducing the nature, Nature Constraint System and Method Constraint System.

Preservation

The reasoning for the proof of preservation in listing 25 is similar to the proof for progress seen in the previous Section 4.4.5. It is also going to be proved by induction over the thread structure `T`. However, here, instead of using the Dynamic semantics' rules 3.3 it uses a number of lemmas in `TypesProp.v` and `Wellformedness.v`, and also lemmas regarding locking in `Locking.v`.

- again, the case that the thread is in an exceptional state **EXN** is trivial since that a configuration with this thread doesn't step;

- in the case of having two parallel asynchronous threads `T1` and `T2`, we consider all the possible cases for the collection of threads `T'` of the configuration `cfg'` that `cfg` steps into: either one of the parallel asynchronous threads takes one additional step, either one of them is in an exceptional state, or we are in the case that parallel asynchronous threads are done. To solve this, if no thread steps, Γ still types the `cfg`, and if one of the threads step, IH applies.

- The case when `T` is a single thread, is similar to progress and is proved by resorting to the auxiliary `lemma single_threaded_preservation`, that is also in the file `Preservation.v`. Again, this lemma is proved by induction over he typing relation over the current expression.

To adapt the preservation lemma, although the type is preserved in the step, we say that exists a nature and a Nature Constraint System that the well-formed configuration `cfg'` outputs.

4.4.6 Auxiliary Lemmas

In order to adapt all the auxiliary lemmas to the new syntax with natures, almost all of them had to undergo changes. A common change was the addition of the natures and the constraint systems, as we can see in listing 4.3. This listing presents the lemma `hasType_wfEnv`, which says that an environment is well-formed if the program is also well-formed and the starting expression is typable. This lemma is useful to prove that the environment is well-formed in an easier way than using the definition, and is used in other lemmas and in the proof of preservation. We modified it by adding the Method Constraint System to the new definition of well-formed program, and the natures and Nature Constraint System to the new typing judgments definition.

```
Lemma hasType_wfEnv :
  forall P t' Gamma e t,
    wfProgram P t' ->
    P; Gamma |- e \in t ->
    wfEnv P Gamma.
```

```
Lemma hasType_wfEnv :
  forall P t' Gamma e t mcs n ncs,
    wfProgram P t' mcs->
          P; Gamma |- e \in t # n |> ncs ->
          wfEnv P Gamma.
```

Figure 4.3: Lemma `hasType_wfEnv` in OOlong and RC[3]

Since there is a very high number of the changed lemmas, we won't talk about all of them. So let us see the most changed lemmas. In particular, there are two lemmas in the file `TypesProp.v` that had a lot of changes: the lemmas `hasType_subst` in listing 26 and `hasType_subst_fresh` in listing 27. The first challenge was to write the right definition of the lemmas, because there is a detail that unlike the type, the natures aren't preserved with the replacement of the variables in the expression. Then, due to the fact that the variables were introduced in the form of `exists` instead of in the `forall`, we had to do many changes, for example the one in we see in the figure 4.4. In OOlong, we could use directly the induction hypothesis of the lemma `hasType_subst_fresh`. However, because we don't have the exists in our goal at that moment of the proof, we have to use the `pose proof` tactic in order to introduce that lemma in the hypothesis, as we can see in figure 4.4. That way we can manipulate the lemma in order to take off the exists with the destruct tactic and use it in our proof.

```
apply IHe1...
```

```
- pose proof hasType_subst_fresh as Lemma.
edestruct Lemma with
        (P) (drop Gamma2 (env_var (SV x))) (e2) (t2) (x) (y) (n2) (ncs2).
 inv H2. apply H8 in HeqG1...
        eapply hasType_subsumption with (Gamma := Gamma2)...
 econstructor... introv Henv. case_drop...
 unfold fresh. case_drop...
inv H.
```

Figure 4.4: Part of the proof of the Lemma `hasType_subst` in OOlong and RC[3]

Therefore, we have to be careful and check if there is another lemma that is incorrect because of the addition of the new variables. We can also see in lemma `wfProgram_wfMethodDecl` presented in listing 4.5, that the Method Constraint System of the well-formed method is different from the one in well-formed program. This variable is introduced in the `exists`, because if the program is well-formed, then the method that is in the program is also well-formed for some Method Constraint System, and not all the Method Constraint Systems. In the preservation lemma, we also have the nature and Nature Constraint System in the `exists`. Regarding the proofs with this detail, we have to remove the `exists` in the goal at the right moment so that there aren't any variables outside the scope. If we remove the `exists` too early, it can happen that the new variables that appear in the hypothesis when we do some tactic are outside the scope of the variable that was in the `exists`.

```
Lemma wfProgram_wfMethodDecl :
  forall P t' c ms m mtd,
    wfProgram P t' ->
    methods P (TClass c) = Some ms ->
    methodLookup ms m = Some mtd ->
    wfMethodDecl P c mtd.
```

```
Lemma wfProgram_wfMethodDecl :
  forall P t' c ms m mtd mcs_p,
    wfProgram P t' mcs_p ->
    methods P (TClass c) = Some ms ->
    methodLookup ms m = Some mtd ->
exists mcs_m,
    wfMethodDecl P c mcs_m mtd.
```

Figure 4.5: Lemma `wfProgram_wfMethodDecl` in OOlong and RC^3

Another lemma that suffered a lot of changes was the `single_threaded_preservation` lemma, particularly when the expression is `Call`. Here we also had to use the `pose proof` tactic in order to prove the last subgoal. We had to apply it twice, because the expression inside the first `subst` had also a `subst`, and to solve this `subst` we need to use the lemma `hasType_subst`. Therefore, it was more complicated than the lemma `hasType_subst_fresh`. Moreover, in the OOlong proof they included the well-formed class in the hypothesis with the tactic `assert`. We also need this assert in order to prove the preservation for the `Call` case. However, because our definition of the well-formed class needs the Method Constraint System, we have to first invert the definition of well-formed program in order to get that variable and add it to the assert tactic. We can see this change in the listing 28. After this point we had to change all the `P` variables to `(cds, ids, e)`, and adapt all proofs for the `assert` tactic and whenever the `eauto` tactic (an automatic tactic that replaces a sequence of tactics by applying lemmas and assumptions) didn't work.

4.5 Chapter Summary

In this Chapter we saw how the mechanization of the RC^3 language was implemented, as well as its type safety proof. We started by seeing how the source code in Coq for the OOlong's language

was structured, and analyzed about each file. Then, we explained our implementation structure, and observed that it was divided in three directories, each one for each stage of the RC^3 model. In each directory, we checked which files of the source code suffered modifications and the most important changes made in order to adapt the OOlong's code to the goal of each stage. The third stage was the one with the most changes, which resulted in us having to divide this Section into Subsections, finalizing with the proof of type safety.

```
Theorem preservation :
  forall P t' Gamma cfg cfg' t mcs n_nature ncs,
    wfProgram P t' mcs ->
    wfConfiguration P Gamma cfg t n_nature ncs ->
    P / cfg ==> cfg' ->
    exists Gamma' n_nature' ncs',
      wfConfiguration P Gamma' cfg' t n_nature' ncs'/\
      wfSubsumption Gamma Gamma'.
Proof with eauto using wfConfiguration.
  introv wfP wfCfg Hstep.
  inverts wfCfg as Hfresh wfH wfV wfT wfL.
  gen t cfg' wfL wfT. gen n_nature ncs.
  induction T; intros...
  + Case "T = EXN".
    (* EXN does not step *)
    inv Hstep.
  + Case "T = T_Thread e".
    eapply single_threaded_preservation...
  + Case "T = T_Async T1 T2 e".
    inverts wfT as Hfree hasType wfT1 wfT2.
    inverts wfL as wfWl wfRl Hdisj wfL1 wfL2.
    destruct cfg' as [[[H' V'] n'] T'].
    assert (Hmono: n <= n')
      by eauto using step_n_monotonic.

    assert(wfLocking H' T').
     simple eapply wfLocking_preservation.
     exact wfP.
     simple apply WF_Cfg.
     exact Hfresh.
     exact wfH.
     exact wfV.
     eapply WF_Async.
     exact Hfree.
     exact hasType.
     exact wfT1.
     exact wfT2.
     (*external*) (wfLocking_context_tactic 1).
     exact Hstep.

    inv Hstep;
    try(
        solve
          [
            (* When no thread steps, Gamma still types the cfg *)
            exists Gamma; exists; split; eauto with env
          |
            (* When one of the threads step, IH applies *)
            match goal with
              | [IH: forall n_nature ncs t cfg', _ / (_, _, _, ?T) ==> cfg' -> _,
                   Hstep: _ / (_, _, _, ?T) ==> _ |- _]
                => eapply IH in Hstep as [Gamma' [natureH [ncsH [wfCfg' wfSub]]]];
                  eauto; inverts wfCfg'; exists Gamma'; exists
            end;
            split; eauto;
            econstructor; eauto with arith;
            econstructor;
            eauto using hasType_subsumption,
                        wfThreads_subsumption
          ]).
Qed.
```

Listing 25: Proof of theorem Preservation

```
Lemma hasType_subst :
   forall P Gamma e t t' t'' x y,
     wfProgram P t'' ->
     wfEnv P Gamma ->
     P; extend Gamma (env_var (SV x)) t' |- e \in t ->
     fresh Gamma (env_var (DV y)) ->
     P; extend Gamma (env_var (DV y)) t' |- subst x y e \in t.
```

```
Lemma hasType_subst :
  forall P Gamma e t t' t'' x y mcs n ncs,
    wfProgram P t'' mcs ->
    wfEnv P Gamma ->
    P; extend Gamma (env_var (SV x)) t' |- e \in t # n |> ncs ->
    fresh Gamma (env_var (DV y)) ->
   exists n' ncs',
    P; extend Gamma (env_var (DV y)) t' |- subst x y e \in t # n' |> ncs'.
```

Listing 26: Lemma `hasType_subst` in OOlong and RC^3

```
Lemma hasType_subst_fresh :
  forall P Gamma e t x y,
    P; Gamma |- e \in t ->
    fresh Gamma (env_var (SV x)) ->
    P; Gamma |- subst x y e \in t.
```

```
Lemma hasType_subst_fresh :
  forall P Gamma e t x y n ncs,
    P; Gamma |- e \in t # n |> ncs ->
    fresh Gamma (env_var (SV x)) ->
exists (n': nature) (ncs' : NatureConstraint -> Prop),
    P; Gamma |- subst x y e \in t # n' |> ncs'.
```

Listing 27: Lemma `hasType_subst_fresh` in OOlong and RC^3

```
(...)
apply classLookup_not_none in cLookup as (i & fs & ms & cLookup).
assert (wfCls: wfClassDecl P (Cls c i fs ms))
        by (inv wfP; lookup_forall as wfCls; eauto)...
assert (mtds = ms)
   by (simpls; destruct classLookup; eauto; inv_eq; inv_eq).
(...)
```

```
(...)
 apply classLookup_not_none in cLookup as (i & fs & ms & cLookup).
 inv wfP.
 assert (wfCls: wfClassDecl (cds, ids, e) mcs0 (Cls c i fs ms))
  by (lookup_forall as wfCls; eauto)...
 assert (mtds = ms)
                by (unfold methods in H17; destruct classLookup; eauto; inv_eq; inv_eq).
(...)
```

Listing 28: Part of the proof of the Lemma `single_threaded_preservation` in OOlong and RC^3, for the `Call` case

Chapter 5

Conclusions

Mechanical proofs provide stronger guarantees than their pen and paper counterparts. Like we saw in the POPLmark challenge [40], we are coming towards a world where every paper on programming languages is accompanied by an electronic appendix with machine checked proofs. In this Thesis, we mechanically verify the type safety proof of the concurrency control model RC^3, as well as mechanically formalize all its syntax, semantics and type system.

We used as a starting point the corresponding proofs for the Oolong language, and extended the definitions with the functions that implement the first transformation stages of the technique. Furthermore, we changed the type system and corresponding proofs, by introducing extra parameters and output, in order to infer restrictions regarding the nature of program variables.

This Thesis contemplates the first three stages of the RC^3 model:

- the first stage just adds the **atomic** keyword to the language, and verifies if the program inserted is an OOlong program (due to the fact that the model's basis is the OOlong's language);
- the second one adds for each class and interface its atomic counterpart;
- and finally the third part adds **atomic** to the variables, and adapts all the OOlong's type system to the new notion of natures.

The main contribution of this work is to guarantee that the type system that is central to the RC^3 model is type safe.

We leave for future work the formalization of the following stages of the mechanism, as well as the consistency results regarding the inferred natures, whose definition is still work in progress. Furthermore, the formalized definitions of the type system are still being adjusted, and for that reason the proofs presented here might require further adjustments as well.

Bibliography

[1] H. Paulino, D. Parreira, N. Delgado, A. Ravara, and A. Matos. From atomic variables to data-centric concurrency control. April 2016.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346323. URL `http://doi.acm.org/10.1145/1346281.1346323`.

[3] E. Kindler. Safety and liveness properties: A survey.

[4] J. Harrison, J. Urban, and F. Wiedijk. History of interactive theorem proving. In J. Siekmann, editor, *Handbook of the History of Logic vol. 9 (Computational Logic)*, pages 135–214. Elsevier, 2014.

[5] C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4): 207–227, 2003.

[6] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 346–358, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111068. URL `http://doi.acm.org/10.1145/1111037.1111068`.

[7] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 291–296, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190260. URL `http://doi.acm.org/10.1145/1190216.1190260`.

[8] M. Hicks and et al. Lock inference for atomic sections, 2006.

[9] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375619. URL `http://doi.acm.org/10.1145/1375581.1375619`.

[10] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224987. URL `http://doi.acm.org/10.1145/224964.224987`.

[11] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. *SIGPLAN Not.*, 43(10):181–194, Oct. 2008. ISSN 0362-1340. doi: 10.1145/1449955.1449779. URL `http://doi.acm.org/10.1145/1449955.1449779`.

[12] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, Jan. 2011. ISSN 0164-0925. doi: 10.1145/1889997.1889999. URL `http://doi.acm.org/10.1145/1889997.1889999`.

[13] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical report, Cambridge, MA, USA, 1987.

[14] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4:1–4:48, May 2012. ISSN 0164-0925. doi: 10.1145/2160910.2160913. URL `http://doi.acm.org/10.1145/2160910.2160913`.

[15] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111067. URL `http://doi.acm.org/10.1145/1111037.1111067`.

[16] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 322–331, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486831`.

[17] W. Huang and A. M. et al. Inferring aj types for concurrent libraries. 2012.

[18] P. Dinges, M. Charalambides, and G. Agha. Automated inference of atomic sets for safe concurrent execution. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 1–8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2128-0. doi: 10.1145/2462029.2462030. URL `http://doi.acm.org/10.1145/2462029.2462030`.

[19] G. Winskel. *Discrete Mathematics II: Set Theory for Computer Science*. Part IA Comp. Sci. Lecture Notes, 2012.

[20] D. Rémy. Type Systems for Programming Languages. Course notes, available electronically, 2015.

[21] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992. ISBN 0-471-92980-8.

[22] G. Rosu. Cs522 - programming language semantics, Spring 2011.

[23] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.

[24] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

[25] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[26] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1093. URL `http://dx.doi.org/10.1006/inco.1994.1093`.

[27] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1987.

[28] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL `http://doi.acm.org/10.1145/503502.503505`.

[29] J. Östlund and T. Wrigstad. Welterweight java. In J. Vitek, editor, *Objects, Models, Components, Patterns*, pages 97–116, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[30] E. Castegren and T. Wrigstad. Oolong: An extensible concurrent object calculus. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1022–1029, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5191-1. doi: 10.1145/3167132.3167243. URL `http://doi.acm.org/10.1145/3167132.3167243`.

[31] A. M. P. G. M. Bierman, M. J. Parkinson. Mj: An imperative core calculus for java and java with effects. 2003.

[32] A. Bove. A machine-assisted proof of the subject reduction property for a small typed functional language. Master's thesis, Department of Computer Science, Universidad de la Replública, Uruguay, November 1995. Technical Report INCO-95-06.

[33] C. Dubois. Proving ml type soundness within coq. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 126–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44659-0.

[34] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 137–150, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3688-8. doi: 10.1145/2814228.2814239. URL `http://doi.acm.org/10.1145/2814228.2814239`.

[35] S. Grewe, S. Erdweg, and M. Mezini. Automating proof steps of progress proofs: Comparing vampire and dafny. In L. Kovacs and A. Voronkov, editors, *Vampire 2016. Proceedings of the 3rd Vampire Workshop*, volume 44 of *EPiC Series in Computing*, pages 33–45. EasyChair, 2017. doi: 10.29007/5zjp. URL `https://easychair.org/publications/paper/hfP`.

[36] T. Nipkow, D. von Oheimb, and C. Pusch. Java: Embedding a programming language in a theorem prover. In *Foundations of Secure Computation, volume 175 of NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[37] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146811. URL `http://doi.acm.org/10.1145/1146809.1146811`.

[38] J. N. Foster and D. Vytiniotis. A theory of featherweight java in isabelle/hol. *Archive of Formal Proofs*, 2006, 2006. URL `https://www.isa-afp.org/entries/FeatherweightJava.shtml`.

[39] R. Strnisa. *Formalising, improving, and reusing the Java module system.* PhD thesis, University of Cambridge, UK, 2010. British Library, EThOS.

[40] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28372-2, 978-3-540-28372-0. doi: 10.1007/11541868_4. URL `http://dx.doi.org/10.1007/11541868_4`.

[41] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda — a functional language with dependent types. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_6. URL `http://dx.doi.org/10.1007/978-3-642-03359-9_6`.

[42] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

[43] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642058809, 9783642058806.

[44] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 1–35, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_1. URL `http://dx.doi.org/10.1007/978-3-642-39799-8_1`.

[45] F. Pfenning and C. Schürmann. *Twelf User's Guide*, 1.2 edition, Sept. 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.

[46] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 125–128, 2013. doi: 10.1007/978-3-642-37036-6_8. URL `https://doi.org/10.1007/978-3-642-37036-6_8`.

[47] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7. URL `http://dl.acm.org/citation.cfm?id=1939141.1939161`.

[48] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9):1–12, Oct. 2007. ISSN 0362-1340. doi: 10.1145/1291220.1291155. URL `http://doi.acm.org/10.1145/1291220.1291155`.

[49] P. SEWELL, F. Z. NARDELLI, S. OWENS, G. PESKINE, T. RIDGE, S. SARKAR, and R. STRNIŠA. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1): 71–122, 2010. doi: 10.1017/S0956796809990293.

[50] E. Castegren. Coq and ott sources for oolong. `https://github.com/EliasC/oolong`, 2017.

[51] B. Ferreira. Coq and ott sources for RC^3. `https://github.com/beatrizagf/RC3-Coq`, 2019.