Resource-Centered Concurrency Control Mechanization of a Type Safety proof

Beatriz Abecasis Gomes Ferreira
beatrizabecasis@gmail.com

Instituto Superior Técnico, Lisboa, Portugal

May 2019

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 [24], 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 article, 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.

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 [18]. 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 them aim at assuring safety and liveness properties described earlier [17]. 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) [24]. 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 [14]. 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.

2. Background

In this Section, we analyze which concurrency control models have proved the type safety of their model. This is followed by the proofs of type safety made for Java-like languages and the theorem provers that were considered for verifying the type safety of the RC^3 model.

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 [24].

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 [2].

In order to withstand this, there are several approaches in which the target is progress and isolation [21, 10, 15, 7, 26] and others in which the

central point is atomicity and protocol compliance [25, 1].

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 paper [26], they prove it using a model that helps doing hierarchical correctness proofs of distributed algorithms [19]. In, [21, 7], they formalize the definitions, present the type system and prove type safety [1] with operational semantics like we are going to do. Concerning the papers in which the central point is atomicity and protocol compliance, one of them [1] also formalizes the definitions and present the proofs about preservation and progress to prove type safety like we'll do.

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

Nonetheless, despite this the errors remain in great quantity [18].

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 [9, 28].

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 [9], 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).

[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.

Variants. One solution to the Deadlock problem is presented in [20], 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 [16], 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 [8] 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. Mechanization of Type Safety

Java is the language chosen for the implementation of the model we are going to evaluate [24]. 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.

Featherweight: The smallest proposed candidate for a core Java calculus is probably Featherweight Java (FJ), which omits all forms of assignment and object state, focusing on a functional core of Java. In Featherweight paper [12], they use the theorem prover Isabelle/HOL to formalize the language and its type system. 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. Therefore, a new minimal core Java calculus

was made which models this: Welterweight Java (WJ) [23]. Because of its easy extensibility, WJ is suitable for extensions to Java-like languages that need imperative features and threads, just like the OOlong language. There isn't any mechanized version of the type safety proof of this language.

OOlong: A similar object calculus, to the Welterweight Java is OOlong. OOlong is more lightweight than WJ by omitting mutable variables and using a single flat stack frame rather than modeling the call stack. This is the closest Java-like language to the RC^3 model language. In this paper [6], 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 (MJ), 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 [13]. MJ was never mechanized. There is however a simplification of MJ: Lightweight Java (LJ) [27]. 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.

2.3. Theorem Provers

Theorem prover is a software tool whose purpose is to assist with the development of formal proofs. This not only make the proof easier to do, but also increase the assurance of the correctness of the proof. This involves some sort of interactive proof editor, or other interface, with which a human can guide the machine.

Agda: Agda's predecessor is ALF (A Logical Framework). Like its predecessors, Agda supports a wide range of inductive data types, pattern matching, termination checking, and comes with an interface for programming and proving by direct manipulation of proof terms [4]. It has many similarities with other proof assistants based on dependent types, like Coq. However, unlike it, has no support for tactics, and proofs are written in a functional programming style. Also, Agda is primarily being developed as a programming language and not as a proof assistant.

Isabelle/HOL: The Isabelle theorem prover [22] 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. However, it 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 [3] 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.

3. Model

As seen in Subsection 1.4, the formalization of the RC^3 language is based on OOlong. In order to mechanize the formalized language of the RC^3 model in the tool Coq, we need to analyze it. Readers wishing to know more about the OOlong language can refer to: [5], and [6].

3.1. 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.

3.2. Semantics

The semantics is based on the small-step semantics formalized for OOlong. Table 1 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

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)*

Table 1: Runtime syntax

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**.

3.3. Architecture

In order to compile a RC^3 program there are several stages that need to be performed. As we can see in figure 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 based 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.

To achieve this, we'll have several stages of the model, starting with the OOlong language and finishing with our RC^3 language.

First stage. As we can see in figure 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 classes and interfaces duplicated: one will be the regular version and the other its **atomic** counterpart.

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 2, 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.

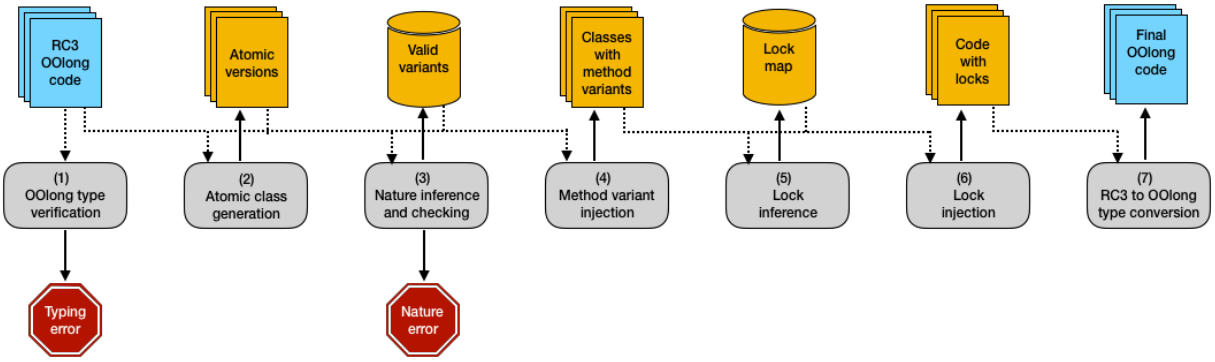

Figure 1: The compilation stages of a RC^3 program

$\eta \in \text{Nature} ::=$
atomic | **non-atomic** | $\bot$ | $\check{x}$ | $(\eta_1, \eta_2)\ ?\ \eta_3 : \eta_4$ | $t.m(\eta')$

$\text{Nature Constraint} = \text{Nature} \times \text{Nature}$

$\text{Ncs} \in \text{Nature Constraint System} \subset \wp(\text{Nature Constraint})$

$\text{Method Constraints} = \text{MethodName} \mapsto \text{Nature Constraint System}$

$\text{Mcs} \in \text{Method Constraint System} = \wp(\text{Method Constraints})$

$\text{Type System Output} \subset \text{Mcs}$

Table 2: Type System Output

The method constraints map the method name into nature constraint systems, which in turn is a set of 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 will have 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.

3.4. Results

The final objective is to prove the type safety of the model RC^3. In order to prove it, it's 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.*

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

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$'$.*

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

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.

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 the RC^3 language. The OOlong source code is in its github repository: [5]. Regarding the Coq files for RC^3 model, they are in this repository [11] that the reader can check to follow the implementation.

4.1. First stage - strip

This stage implements the first stage of the model, that upon receiving a program, strips the atomic annotations and checks if it satisfies the OOlong model. 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): 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 `class_id` and `interface_id` into tuples where the second element is the `atomicity`, as we can see in listing 2, 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.

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

Listing 2: Atomicity in ID's

Finally, we defined the two strip functions in listing 3 by forcing the second element to be `NonAtomic`. We also defined functions that strip atomicity for the list of methods (called *stripListMethod*), methods signatures and fields. Afterwards, we had to apply these functions to the program, so, we had to define three new recursive definitions. One receives the list of classes, scrolls through it and for each class applies `stripClass`, `stripInterface` and *stripListMethod*. Another does the same but for the list of interfaces. 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.

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

Listing 3: Transforming into non-atomic

4.2. Second stage - duplicate

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. There will exist one function called **atogen** for each input: one **atogen** function for programs, presented in listing 4, one for the list of interfaces, another for the list of classes and lastly one for the the list of fields.

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

Listing 4: Atogen that receives a program

As we can see, the definition of an **atogen** that receives a program is straightforward: it receives a program and duplicates the list of classes and interfaces. As a consequence, besides having the regular list of classes and interfaces, we now have added the **atomic** counterpart of each list. The `atogenProg` in listing 4 needs to call the `atogenIn` and `atogenCl` recursive functions, which scroll through each of the respective lists and if these aren't null, they are checked in terms of its classes and interfaces ID's in order to turn them into **atomic**. The `atogenCl` function also calls the `atogenfield` function, which, in the case of the type of the field being a class with the same ID as the class's ID or an interface with the same criteria, turns the field's type ID into atomic.

4.3. Third stage - Nature Inference

In the previous Section 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.

In this stage, there were a lot of modifications in order to adapt the type system to the new notion of nature. This new notion is implemented in listing 5, along with the nature constraints and the method constraints. Comparing to table 2, 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 5: Type System Output

Nature constraints are composed by tuples of two natures, just as represented in the definition `NatureConstraints` in listing 5. The Nature Constraint System 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. 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. The reader can check all the new rules for well-formed expressions, well-formed program and well-formed configurations in the repository [11].

At the end of this stage there is the proof of type safety, that is made by proving progress and preservation. Both of these lemmas follow the same reasoning as the one made for OOlong and are proved by induction over the thread structure `T`, thus leading us to the three cases: either `T` is a single thread, two parallel asyncs threads or has thrown an exception. Also, both of them use various lemmas to help in the proofs, which suffered many modifications in order to work with the new definitions. In almost all of them we added the natures and the constraint systems, as we can see in listing 6. 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.
```

Listing 6: Lemma `hasType_wfEnv` in OOlong and RC^3

We also had to be careful and check if these variables are added in the `forall`, like in the listing 6, or if the lemma is not true for all the variables but just for some. If this is the case, we have to introduce them in the form of `exists`, as we can see in listing 7. Then we have to remove the `exists` in the proof at the right moment, so that the new variables that appear in the hypothesis when we do some tactic aren't outside the scope of the variable that was in the `exists`.

```
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 7: Lemma `hasType_subst_fresh` in OOlong and RC^3

The other modifications include adapting all

proofs whenever the `eauto` tactic (an automatic tactic that replaces a sequence of tactics by applying lemmas and assumptions) didn't work.

5. Conclusions

Mechanical proofs provide stronger guarantees than their pen and paper counterparts. We are coming towards a world where every paper on programming languages is accompanied by an electronic appendix with machine checked proofs. In this work, 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 work 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.

Acknowledgements

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.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] J. N. Foster and D. Vytiniotis. A theory of featherweight java in isabelle/hol. *Archive of Formal Proofs*, 2006, 2006.

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

[14] 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.

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

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

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

[18] 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.

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

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] 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.

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

[28] 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.