Sound Atomicity Inference for Data-Centric Synchronization

Hervé Paulino[1], Ana Almeida Matos[2], Jan Cederquist[2], Marco Giunti[1], João Matos[2], and António Ravara[1]

[1]NOVA-LINCS and FCT NOVA, NOVA University Lisbon
[2]Instituto de Telecomunicações, and IST, University of Lisbon

September 12, 2023

Abstract

Data-Centric Concurrency Control (DCCC) shifts the reasoning about concurrency restrictions from control structures to data declaration. It is a high-level declarative approach that abstracts away from the actual concurrency control mechanism(s) in use. Despite its advantages, the practical use of DCCC is hindered by the fact that it may require many annotations and/or multiple implementations of the same method to cope with differently qualified parameters. Moreover, the existing DCCC solutions do not address the use of interfaces, precluding their use in most object-oriented programs. To overcome these limitations, in this paper we present AtomiS, a new DCCC model based on a rigorously defined type-sound programming language. Programming with AtomiS requires only (atomic)-qualifying types of parameters and return values in interface definitions, and of fields in class definitions. From this *atomicity specification*, a static analysis infers the atomicity constraints that are local to each method, considering valid only the method variants that are consistent with the specification, and performs code generation for all valid variants of each method. The generated code is then the target for automatic injection of concurrency control primitives, by means of the desired automatic technique and associated atomicity and deadlock-freedom guarantees, which can be plugged-into the model's pipeline. We present the foundations for the AtomiS analysis and synthesis, with formal guarantees that the generated program is well-typed and that it corresponds behaviourally to the original one. The proofs are mechanised in Coq. We also provide a Java implementation that showcases the applicability of AtomiS in real-life programs.

1 Introduction

Parallelism is omnipresent in today's computer systems, from cloud infrastructures, personal computers to handheld devices. Concurrent programming, hence, becomes a fundamental tool for software to fully harness the processing power available in the underlying hardware. However, writing efficient concurrent code remains complex, and assessing its correctness difficult, resulting in execution errors caused by concurrency issues [4,33], some constituting real threats, such as the Therac-25 [31] and the Northeast blackout [44] incidents.

The *shared memory* model is the *de facto* approach to program concurrent applications in shared memory architectures (message-passing alternatives used in languages like Erlang or Go remain underutilised). Some modern programming languages (*e.g.* Rust) offer tight control of resources preventing interference statically, but inherently concurrent data structures still require dealing with accesses to shared memory [54]. Expressing restrictions on the manipulation of shared resources usually consists of explicitly delimiting the sequences of instructions that access those

1

resources (the critical region), and coding the necessary synchronisation actions to prevent interference. This can be done via low-level primitives—*i.e.*, hardware *read–modify–write* instructions, *locks*, *semaphores*, etc.—or higher level mechanisms like Java's *synchronized* blocks, transactional memory [49], and others [10, 16].

The limitations of control-centric approaches. The mechanisms described thus far are *control-centric* approaches that require a distributed analysis of the code, making reasoning about correctness harder as the code grows. This is prone to human error and can often result in disorganised and *ad hoc* code [53]. A single missing or ill-placed locking operation, *synchronized* block, atomic region, etc. is enough to compromise an application's behaviour. To mitigate the impact of such methodology, relevant work has addressed correctness of concurrent accesses to shared resources, with focus on the absence of deadlocks [5, 10, 16, 17, 25, 36], data-races [5, 17] and atomicity violations [1, 19, 34, 48]. Despite these efforts, the study in [33] identifies that 97% of the non-deadlock errors stem from either atomicity or protocol violations.

Although there are successful static analysis tools that deal with these kinds of problems, like Infer[1] or ThreadSafe[2], as well as ongoing research [12, 46, 56], the generalised use of such solutions is still distant, as the associated effort and learning curve are not negligible. Most companies and programmers could greatly benefit from a language-based approach that would help them to reason about the code they are developing.

Data-Centric Concurrency Control (DCCC) shifts the expression of concurrency-related constraints away from control structures and into data declaration. A simple example are atomic types in C++ [28], whose scope includes only individual accesses to atomic-qualified types. By centralising all concurrency control management onto data declaration, DCCC promotes local rather than distributed reasoning, a key change to achieve simpler reasoning on interference supervision. Several proposals have adopted DCCC for either shared memory concurrent programming [9, 15, 30, 43, 51], extending the concept to groups of resources that share consistency requirements, or to distributed programming [29, 32, 55].

Atomic sets [15, 51] laid the foundations for DCCC in the shared memory context we are targeting. Although innovative, *Atomic sets* have the problem of requiring multiple keywords, with non-trivial semantics, that render the specification burdensome and even complex at times. Moreover, progress guarantees requires extra work from the programmer and is not guaranteed in all situations [35]. Resource-Centric Concurrency Control (RC³) [43] is a proposal that builds from a single keyword (**atomic**), presenting itself as a simpler solution that combines the concepts of *atomically qualified type* and *unit of work*. Units of work comprise a method's sequence of instructions from the first to the last access to a variable of atomic type and which are to be executed atomically. In DCCC approaches, the task of the programmer is to simply atomic-qualify the declaration of the variables that must be accessed atomically in a unit of work. If the annotations are consistent, a compilation process should produce code free of data-races, atomicity violations, and deadlocks.

Problem. RC³ is a promising solution. However, its practical use (and the use of type-based concurrency control solutions in general) is hindered by the fact that the code (method bodies, for instance) is not agnostic to the atomic-qualification of the local variables and parameters. Consequently, the programmer has to write several versions of the same code to account for the supported combinations of atomic-qualified and not atomic-qualified types. Consider the following example of a Java class `ArrayList` that implements interface `List`, including method `equals` that checks if the contents of the current collection are the same of another received as argument:

---

[1] https://fbinfer.com/
[2] http://www.contemplateltd.com/threadsafe-1-2

```
class ArrayList implements List {
  ...

  // Returns true if 'this' and 'other' have the same elements
  Boolean equals(List other) { ... }
}
```

In order to account for all combinations of atomic-qualified and not atomic-qualified for **this** and parameter `other`, the programmer must provide 4 implementations of the method. The number of combinations naturally doubles with each parameter whose atomicity must be considered (non-primitive types). Moreover, if the method's result is a non-primitive value that may be assigned to both an atomic or a non-atomic recipient, *e.g.*

```
Boolean b = x.equals(y) or @Atomic Boolean b = x.equals(y)
```

the return type's atomicity must also be considered, doubling the number of combinations.

Furthermore, providing atomic and regular class implementations of a given functionality would result in the same code, modulo the DCCC annotations. This goes much in the vein of what is currently done (with other constructs) in many programming languages to implement sequential and concurrent data structures[3]. The problem shares similarities with the duplication of `const` and `non-const` member functions in C++. However, unlike such problem [37,50], it cannot be efficiently solved by simply having the non-atomic version call the atomic one, since this would imply non-negligible synchronisation overhead for all accesses, independently of the atomic-qualification of the types.

In truth, from the programmer's perspective, a method's actual implementation is independent of the atomic-qualification of **this** or of the method's parameters and, hence, should be coded in way agnostic to these atomicity concerns. Conversely, the compiler-generated code is considerably dependent of those same concerns, since they are the ones guiding the generation of the required low-level synchronisation instructions. In the example above, concurrency control has to be generated to guarantee that no elements are concurrently added to atomic-qualified lists during the execution of `equals`, otherwise the result of `equals` could be erroneous.

Therefore, the challenge is to *infer and synthesise* the use of the `atomic` type qualifier in method definitions, so that they are removed from the programmer's concerns but are available for the compiler to generate the necessary synchronisation code. Our proposal provides a framework to write methods in a concurrency-agnostic way (*i.e.*, without worrying about the concurrency annotations): the programmer atomic-qualifies some class fields to implement their concurrency restrictions. In short, what differentiates our type-based DCCC from other approaches is that the concurrency control information is encoded on the *type* of the resource. Our insight is to leverage on type parameters to abstract away the atomic-qualification of fields from class implementation and move the concern to class instantiation. This approach will allow to use *the same* base class implementations for both concurrent and sequential settings.

Our proposal. We build on DCCC atomicity annotations and on the *unit of work* concepts of Atomic Sets and RC[3] to propose ATOMIS, a novel concurrency control model that only requires from the programmer a *high-level atomicity specification* – the atomic-qualification of class fields and, in each interface method signature, the enumeration of which combinations of atomic parameters and return types must be supported by the method's implementations.

The `List` interface depicted in Listing 1 features such specification by using annotation `@All` that we will present in §2. From this specifications our approach:

1. infers and synthesises the atomic-qualifications of the type of method parameters, local variables and return values;

2. allows for the atomic-qualification of class field types to be abstracted into class type parameters.

---

[3]A known example are Java's synchronized collections: https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/Collections.java.

```
1  interface List<T> {
2    void add(T element);
3    T get(int pos);
4    @All Boolean equals(@All List<T> other);
5  }
6
7  class Node<T> {
8    T value;
9    Node<T> next, prev;
10 }
11
12 class BaseList<T, N extends Node<T>>
        implements List<T> {
13   N head, tail;
14   Supplier<N> factory;
15
16   BaseList(Supplier<N> factory) {
17     this.factory = factory; }
```

```
19   void add(T element) {
20     N node = this.factory.get();
21     node.value = element;
22     node.next = null;
23     if (this.head == null) {
24       node.prev = null;
25       this.head = node;
26     } else {
27       node.prev = this.tail;
28       this.tail.next = node;
29     }
30     this.tail = node;
31   }
32
33   T get(int pos) {...}
34   Boolean equals(List<T> other) {...}
35 }
```

Listing 1: The base list. We make use of the `Supplier` functional interface[4] to overcome Java's limitations on the creation of new objects from type parameters.

```
1  class ConcurrentList<T> extends BaseList<T, @Atomic Node<T>> {
2    ConcurrentList() { super(Node<T>::new); }
3  }
4  class SeqList<T> extends BaseList<T, Node<T>> {
5    SeqList() { super(Node<T>::new); }
6  }
```

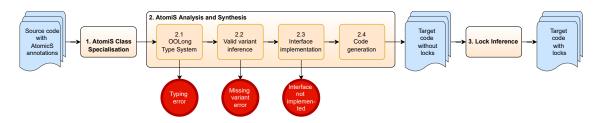Listing 2: Concurrent and sequential list class definitions.

Figure 1: ATOMIS compilation stages

ATOMIS decouples the reasoning about functionality from the reasoning about concurrency. Methods are implemented in a concurrency-agnostic way and, when convenient, the atomic-qualification of types can also be encoded in type parameters. As a result, very specific optimisations aside, with ATOMIS it is no longer needed to code concurrent and sequential versions of classes, as it is usual in programming languages, such as Java and C++. Listing 1 presents a sketch of a simple implementation of a generic list in Java. Besides the usual type parameter (T) for the elements' type, we add a new one (N) for the type of the node to be used. Listing 2 showcases how a concurrent list and a sequential may be derived from the base list. Either of these lists may be used to store collections of both atomic and non-atomic values. For instance, a sequential list of atomic bank accounts may be created as new SeqList<@Atomic Account>().

The ATOMIS compiler generates: 1. specialisations of all classes with type parameters, one version for each combination of {atomic-qualified, not atomic-qualified} for each type parameter (stage ATOMIS *Class Specialisation* in Fig. 1), 2. the type-safe atomicity-related overloaded versions of each method (stage ATOMIS *Analysis* in Fig. 1); and, from this code, 3. the low-level concurrency control (locks) necessary to guarantee key properties like thread-safety (stage *Lock Inference* in Fig. 1).

---

[4] https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html

Although we are addressing the problem in the context of AtomiS, in this paper we focus mainly on the first two of the three stages depicted in Fig. 1, where the main novelty of our work resides. In what concerns step 3, AtomiS is a high-level data-centric specification that may be encoded in virtually any base synchronisation mechanism, an advantage of data-centric synchronisation pointed out by the authors of Atomic Sets [15,51]; AtomiS ' units of work simply define a sequence of instructions whose execution has to be perceived as an atomic operation; such behaviour can be obtained via an encoding in many control-centric synchronisation mechanisms, such as mutual exclusion locks, read-write locks, transactional memory, and so on and so forth. We resort to the inference of either mutual exclusion or read-write locks.

Regarding step 2, we propose an approach based on a static analysis with three steps (2.1 to 2.3 in Fig. 1). It begins by considering all possible combinations of {atomic-qualified, not atomic-qualified} for each class type parameter, internally generating a different class for each combination. Next, within each class, it considers all possible combinations for each method parameter, return type, and the instance, what we refer to as *method variants*. For each variant, the analysis infers the atomicity of the local variables and considers *valid* only the variants that ensure the consistency of the atomicity type qualifications (atomic resources are *never* assigned to non-atomic variables, and vice-versa) – Step 2.2 in Fig. 1. Then, in Step 2.3, it checks if all AtomiS-annotated interfaces are correctly implemented, when considering the valid variants inferred for each class. Lastly, Step 2.4 launches a correct-by-construction compilation process that produces code comprising atomic versions of each type in the source code and, for each class, the set of valid method variants. The resulting method overloading allows programmers to continue implementing methods agnostic of atomicity concerns and to be unburdened with the task of atomic-qualifying method parameters and local variables.

To define a provably correct analysis and inference of the atomicity annotations, we formalise a *type qualifier inference system* for a well-defined, type sound, core multi-threaded O. O. language (OOlong [6]). We choose OOlong not only due to the minimal grammar and rigorously defined (static and dynamic) semantics, but also because type soundness (progress and subject reduction) has been mechanised in the Coq proof assistant[5], thus being a certified language to build analyses on. Therefore, following the high-level DCCC atomicity specification approach, AtomiS is a formal model that aims to provide a basis for formally guaranteeing thread-safety properties like absence of atomicity violations, data-races and deadlocks.[6]

Contributions. The contributions of this paper are thus the following:

1. AtomiS, a new DCCC language-based approach, designed to be used in the programming of real-life applications. AtomiS requires only writing atomicity specifications (presented in §2), delegating to a static analysis the inference of the atomicity concerns local to each method (in §4). Furthermore, the code generated includes the atomic versions of the source code's types and, for all classes, the code of all valid method variants (in §4.2.2 and §4.2.5).

2. A formalisation of AtomiS in a rigorously defined, type-sound, multi-threaded, core object-oriented programming language OOlong (in §4.1), illustrating the use of the language in this context with a significant example (a concurrent list).

3. Soundness results supporting the inference and code generation processes.[7] Concretely, this work provides formal guarantees (in §5) that: (1) the generated program is well-typed (in §5.1.1), with types consistent with those in the original program, being the proof of type preservation mechanised in Coq (§5.1.2); and (2) the generated code behaviourally corresponds to the original one, *i.e.*, it does not do more nor less than the original (in §5.2).

A companion paper presents this work from a practical perspective, focusing on a prototype Java implementation that showcases the applicability of AtomiS to real code [18].

---

[5]https://coq.inria.fr/distrib/current/refman/

[6]Again, to obtain formal guarantees of thread-safety is work currently under development.

[7]The Coq mechanisation can be found at *AtomiS-Coq proof of type preservation (2022)*, at https://zenodo.org/record/6346649 and https://zenodo.org/record/6382015

```
 1  interface ListAtomic {
 2    void add(@Atomic Object element);
 3    @Atomic Object get(int pos);
 4    boolean containsAll(@All ListAtomic other);
 5  }

 7  class NodeOfAtomic {
 8    @Atomic Object value;
 9    NodeOfAtomic next, prev;
10  }

11  class ConcurrentListOfAtomic implements
        ListOfAtomic {
12    @Atomic NodeOfAtomic head, tail;

14    void add(Object element) { ... }
15    ...
16  }
```

Listing 3: A concurrent list example. `add` is the same as in Listing 1 with `N` replaced by `NodeOfAtomic`

2   The AtomiS Model

ATOMIS is a generic data-centric synchronization model applicable to any concurrent language with shared state. The data-centrality comes from the addition of the **atomic** type qualifier to variables whose values are shared across multiple concurrent execution flows. Our Java implementation does this by defining annotation `@Atomic`, applicable to any type use.

Mutable values assigned to variables with atomic type (*atomic variables*) are referred to as *atomic values*. A process' memory is thus partitioned into *atomic* and *regular* (non-atomic) values. Atomic values may only be manipulated within the scope of a *unit of work* – a sequence of instructions whose execution must be perceived as an atomic operation to the remainder computation [15], which in ATOMIS translates to blocks of instructions (such as method bodies) that access at least one atomic value. So, the execution of units of work that access the same atomic values may *not* overlap in time (must occur in mutual exclusion). The goal of ATOMIS is to ensure that *all accesses to atomic values within each unit of work are, in effect, a single atomic operation.*

For such purpose, concurrency restrictions may be specified in both interface and class definitions. In the case of interfaces, each method specifies what combinations of atomicities are supported in its parameter and return types. To that end, we define the `@AtomiSSpec` annotation, which takes a string with specifications of the form

$$(\text{atomicity\_of\_parameter}_1, \ldots, \text{atomicity\_of\_parameter}_n) \rightarrow \text{atomicity\_of\_return}$$

where the values may be `atomic`, `non_atomic` or `_` (to denote *not relevant*).

Consider the following interface for lists of atomic objects. We want to specify that the methods support the insertion and retrieval of atomic values.

```
interface ListAtomic {
  @AtomiSSpec("(atomic) -> _") void add(Object element);
  @AtomiSSpec("(_) -> atomic") Object get(int pos);
  @AtomiSSpec("(non_atomic) -> _, (atomic) -> _") boolean containsAll(ListAtomic other);
}
```

Accordingly, method `add` receives an atomic parameter, method `get` receives the position of the value to retrieve (a value of an immutable primitive type that requires no concurrency control) and returns a value of atomic type and lastly, as mentioned in §1, in method `containsAll` we want to be able to compare the contents of the current list with any other, atomic or not.

Note the lack of atomicity information about the object providing the methods. Any implementation of a given interface must provide implementations for all supported atomicity combinations of all methods, regardless of the object's atomicity (see §2.2).

Atomicity annotation can be made easier with the introduction of some syntactic sugar. Parameter and return types can be left *bare* or annotated with `@Atomic` or `@All`. These three possibilities signify that the supported values are, respectively, `non_atomic`, `atomic` or *both*. The `ListOfAtomic` interface can now be rewritten as presented in Listing 3.

To express relations between the atomicity of certain parameter/return values, we introduce the `@AtomicityOf` type annotation to express that a type must have the same atomicity of a
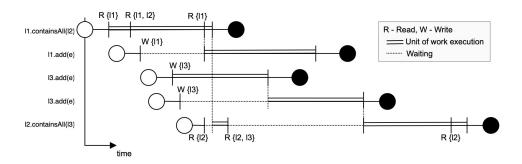
Figure 2: Execution trace of multiple units of work working upon the state of 3 lists: $l1$, $l2$ and $l3$

given variable or type. For example, an identity function that must return a value with the same atomicity as its parameter can be written as `@AtomicityOf("x") Object identity(Object x)`.

Regarding class implementations, class fields may be accessed by multiple methods which may, in turn, be executed by multiple threads. We thus require field atomicity to be specified in the source code. The implementation of `ListOfAtomic` in Listing 3 supports concurrent access to list nodes, so these are qualified as atomic. At this point we are still not dealing with type parameters, so nodes are implemented as instances of class `NodeOfAtomic`, which stores atomic objects.

Arrays specify the atomicity of both the array and its contents, so `@Atomic` may be used on the type parameter and the array itself. An array of `T` may thus be declared in one of four possible combinations: `T[]`, `@Atomic T[]` (a non-atomic array of atomic T), `T @Atomic []` (an atomic array of non-atomic T), or `@Atomic T @Atomic []` (an atomic array of atomic T).

2.1 Units of Work

Recall that we consider a *unit of work* to be a sequence of instructions (of a method) that performs an access (read or write) to an atomic value and thus must be perceived as an atomic operation to the remainder computation. So, we consider *units of work* for every method accessing atomic values, comprising the method's code from the first to the last access to an atomic value. More precisely, the boundaries of a unit of work begin at the first instruction I_1 in the method's body, not within calls, that accesses an atomic value, and end at the last instruction I_2 in the method's body, not within calls, that accesses an atomic value. Note that the unit of work includes all accesses to atomic values done in the context of calls between the execution of I_1 and I_2.

The concept bears similarities with transactional memory [49], as units of work may be interpreted as transactions over atomic values, and transactional memory may even be used to implement units of work. The main difference lies in the fact that units of work arise from the annotation of data, rather than from the explicit delimitation of transactions. Also, in ATOMIS, atomicity means that the operation is perceived to be indivisible (no intermediate state is visible), rather than to be executed in its entirety or not at all. We do not have a notion of commit and rollback, as exists in transactions. In our context, atomicity is a concept closer to isolation in ACID transactional systems.

Consider class `ConcurrentList` (Listing 1). Given that type parameter N was instantiated with `@Atomic Node<T>`, we have that method `add` features a unit of work that begins in the comparison `this.head == null`, performed in line 23, and ends at the assignment `this.tail = node`, in line 30. The same happens with methods `get` and `containsAll`, that also read fields `head` and `next`. So, the executable code of these methods will have to feature concurrency control code to ensure that the execution of their units of work does not overlap in time, when accessing the same memory positions. Figure 2 showcases the trace of the execution of multiple methods over multiple lists. A unit of work may only proceed its execution if it retains exclusive access to all the resources it needs to write and read access to all the values it needs to read.

7

Composition of units of work. The composition of units of work must be considered when:

1. a method does not access an atomic value but calls two or more methods that do. An example is a bank transfer operation that calls existing deposit and withdraw operations that manipulate an atomic account balance. The latter operations include a unit of work, but transfer does not, since its body does not access any atomic value.

2. a method includes a unit of work and calls methods that access atomic values before the start or after the end of its unit of work.

To deal with unit of work composition, the compilation process must receive a *unit of work composition policy*. Currently, two are supported: *conservative* and *standard*. The first handles the composition automatically, creating a new unit of work for the composition of all units of work. For case (1), it creates a unit of work for the method performing the calls and, for case (2), it extends the unit of work to include the first and the last methods that access atomic values. This approach is thus correct by design but may lead to extreme cases of over synchronization. To handle such scenarios, composition may be avoided for individual methods with the annotation `@NoUnitOfWork`. In turn, the *standard* policy instructs the compiler to generate new units of work only when all the calls made by the current method, outside its own unit of work, do not invoke other methods that also comprise units of work. This excludes typical implementations of *getters* and *setters*, `equals` and other similar methods. By not being exhaustive, this policy might not address composition for all scenarios sought by the programmer. To specifically determine which additional methods require composition handling, the annotation `@UnitOfWork` must be used.

These two new annotations are code- rather than data-centric. This derives from the fact that they operate upon units of work rather than data and should be placed in the code as reactions to the output given by the compiler. In our opinion, moving the annotations to data declaration would be harder to reason about because, for a given atomic value, composition is desirable in some methods and not in others. To cope with such flexibility would require the inclusion of the lists of the methods to process (or equivalent) in the data-centric annotations. As the composition of units of work may involve multiple atomic values, this approach could easily lead to complex and possibly inconsistent specifications.

In sum, synchronization in ATOMIS remains data-centric but, when aiming for better performance, the unit-of-work-directed annotations allow for a fine grained control over which methods should feature units of work. Given their availability, we permit the utilization of the annotations in any method, not only to address unit of work composition.

2.2 Atomic Types and Method Variants

Atomic-qualified (*atomic*, for short) and regular types define two different type trees, and thus are not convertible into each other. If a class C implements an interface I, then the atomic type `@Atomic C` (that we denote by **atomic** C) will implement interface **atomic** I. Equivalently, if a class C extends a class D then **atomic** C will extend **atomic** D. As a result, an object with atomic type cannot be cast into a regular type, and, by this way, give rise to uncontrollable atomicity violations.

Concerning atomic class types, a decision must be made about the atomicity of unqualified fields of the same type of the hosting class, such as field `next` from class `NodeOfAtomic` in Listing 3. The atomicity of such fields may be inherited from the class itself, preserving the type equality between the field and the hosting class, or simply remain unaltered, breaking this equality. Currently, we chose to preserve the equality, causing the atomic class type to be slightly different from the one of its regular counterpart. Ergo, in Listing 3, the type of `head.next` is **atomic** `NodeOfAtomic`, allowing for the `NodeOfAtomic` type to be used both in the implementation of both concurrent and regular lists of atomic objects. As such behavior may not always be desired, one may consider inferring the atomicity of these fields.

Contrarily to interfaces, the atomicity of method parameters and return types is not explicitly conveyed in method definitions. This allows for the use of the same definition in multiple atomicity

contexts: the same method name defines different methods, with different signatures and behaviors, depending of the atomic nature of its parameters. Thus, the atomicity of parameters types must be matched, for each method call, with the atomicity of the types of the values assigned to them. The atomicity of the types of local variables and of the return type of a method may depend on the atomicity of the types of the values that have been (or will be) assigned to a field or a parameter. A method may hence have several different signatures, what we refer to as *methods variants*, corresponding to combinations of atomicity qualifiers for the object itself (**this**, because, for a class D, instances of both D and **atomic** D may be created), the method's parameters and return type. For each variant, the atomicity of all non-qualified local variables is inferred during the compilation process, having as base the atomicity of fields and the given atomicity combination (more details in §4). Accordingly, some variants may not be type-safe. For instance, the validity of the variants of method add from Listing 3 depends on the atomic-qualification of the element parameter. If the parameter is regular it cannot be assigned to field value that is of an atomic type. It is thus the parameter that defines the validity of the variant in this particular case. This is not always the case: in method containsAll, all variants are valid because the method does not impose any atomicity constraints on either **this**, the parameter and the return value.

2.3 Leveraging Type Parameters

Method variants provide the framework for writing atomicity-agnostic methods. To take the extra step and remove most (if not all) ATOMIS annotations from class implementations, we leverage on *type parameters*. The insight is to encode the atomicity of the type as an attribute of the type parameter. Looking again at the BaseList class of Listing 1, the atomicity of type parameter N will determine if the list is of atomic or regular nodes. In turn, the atomicity of type parameter T will determine if the list stores atomic or regular values of type T.

The Java implementation supports the atomic-qualification of type parameters at type definition (class C<@Atomic T> {...}), as well as at type usage, such as on subtyping, variable declaration or instance creation (new C<@Atomic Integer>(...)). For a given object, each of its type parameters is atomic if it is atomic-qualified at either of the alternatives. As with the atomicity of the class types themselves, the atomic-qualification of type parameters only impacts directly on the atomicity of fields. The one of the parameters, return values and local variables is inferred as before.

There are cases where atomicity encompasses several fields of unrelated types. For example, in an atomic data structure based on an array, the array, as well as the accounting of the number of elements in the array, must be atomic. To avoid having to parameterize the class in several type parameters that simply convey the same atomicity information (or to have type parameters of "kind" atomicity), we, once again, make use of the @AtomicityOf annotation to express that a field must have the same atomicity of another field or type parameter. In the next example:

```
1  class ArrayList<T, A> {
2      @AtomicityOf("A") int count;
3      T @AtomicityOf("A") [] array;
4      ...
5  }
```

fields count and array have the atomicity of type parameter A, whose existence has the sole purpose of defining the atomicity of such fields. Java does not support annotations as type arguments. Accordingly, to create an instance of the class to store strings in an atomic array, one writes new ArrayList<String, @Atomic Object>() or simply new ArrayList<String, Atomic>() without the @ prefix because we want to denote the type of the Atomic annotation rather than the annotation itself. To create an instance of the class to store strings in an non-atomic array, we may use any (non-atomic-qualified) type other than Atomic. For the sake of code readability we defined the NotAtomic type: new ArrayList<String, NotAtomic>().

Table 1: Valid variants and units of work resulting from different placements of annotation `@Atomic` in the subtyping of `BaseList<T, N>` in Listing 1. For the sake of conciseness, valid variants are represented by tuples of four elements (method_identifier, atomicity qualifiers of **this**, atomicity qualifiers of the parameter, atomicity qualifiers of the return value), where the atomicity qualifiers are represented by a, for **atomic**, and n, for **non_atomic**.

Type	Variables with atomic type	Valid variants	Units of work
`SeqList<T>`	–	(add, n, n, {a, n}), (get, n, {a, n}, n) and (containsAll, n, {a, n}, {a, n})	containsAll when other is atomic or has atomic fields
`ConcurrentList<T>`	head and tail		add, get and containsAll
`SeqList<@Atomic T>`	Node.value	(add, n, a, {a, n}), (get, n, {a, n}, a) and (containsAll, n, {a, n}, {a, n})	containsAll when other is atomic or has atomic fields
`ConcurrentList<@Atomic T>`	head, tail and Node.value		add, get and containsAll
`@Atomic SeqList<T>`	instance of the class (**this**)	(add, a, n, {a, n}), (get, a, {a, n}, n) and (containsAll, a, {a, n}, {a, n}) (add, a, a, {a, n}), (get, a, {a, n}, a) and (containsAll, a, {a, n}, {a, n})	add, get and containsAll
`@Atomic ConcurrentList<T>`	**this**, head and tail		
`@Atomic SeqList<@Atomic T>`	**this** and Node.value		
`@Atomic ConcurrentList<@Atomic T>`	**this**, head, tail and Node.value		

3   Programming with AtomiS

Programming with ATOMIS requires a specification that includes annotating interfaces and class fields. We discuss herein the impact of such annotations.

Interface Annotation. The annotation of interfaces augments the original specification with new method signatures. By indicating the combinations of the types' atomicities, we only generate signatures that have to be fulfilled by the classes that implement the interface.

The use of type parameters abstracts away most of the `@Atomic` annotations, but the programmer still has to place the `@All` and `@AtomicityOf` annotations whenever, respectively, both atomic and regular types are to be supported in a parameter or return type, and/or there is a need to relate the atomicities of multiple types in a signature.

Class Implementation. Programming classes with ATOMIS requires reasoning about which memory objects are prone to data races and atomic-qualify the ones on which all accesses must be done within units of work. This is usually done by qualifying fields but, may also require qualifying local variables whenever these are shared by multiple threads, as may happen in Java.

The placement of the `@Atomic` annotation allows for a more coarse or fine grained concurrency control. In our running list example, the `@Atomic` annotation of fields head and tail depends on the annotation of the type parameters. The use of type parameters renders the code ATOMIS-annotation-free, but that does not mean that the programmer does not have to reason about which fields need to be atomic in a concurrent setting. It just means that the concurrency control may be injected through subtyping, as demonstrated in Listing 3. The `@Atomic` field annotations are replaced by the use of the type parameter.

Table 1 showcases the impact of ATOMIS-related subtyping by using the derived `SeqList` and `ConcurrentList` types of Listing 3. The use of type `ConcurrentList` (cf. second row), that atomic-qualifies fields head and tail, ensures that the list is one of atomic nodes; so, every access to these fields must ensure the atomicity of the list's operations. Therefore, units of work emerge in methods add, get and containsAll. In the latter, the enclosed unit of work also encompasses the accesses to the atomic fields of the other list. The qualification of the fields has no impact on the variant validity. Note, however, that qualifying just one of the fields would preclude the existence of valid variants, since there would not exist a combination of atomicities that would guarantee that the type of head and tail are the same.

The atomic-qualification of Node's value field, by qualifying the class' type parameter (T) (cf. third and fourth rows), makes the list one that stores objects of atomic type. Comparing with the remainder alternatives, we observe that the qualification has no impact on the units of work. This happens because the methods do not access the state of the objects they store nor modify the contents of field value. The units of work will appear in methods that access elements stored in the list. In the table, the set of valid variants of the classes with `@Atomic` T is naturally adjusted

Table 2: Syntax of AtomiS-OOlong programs.

P	::=	$Ids\ Cds\ e$	(Programs)
Id	::=	**interface** I $\{IMsigs\}$ | **interface** I **extends** I_1, I_2	(Interfaces)
$IMsig$	::=	$Msig$ | $Msig$ $[Sas]$	(Interface Signatures)
Sa	::=	$(q_1) \to q_2$	(Signature Annotations)
Cd	::=	**class** C **implements** I $\{Fds\ Mds\}$	(Classes)
Fd	::=	$f : t$ | $f :$ **atomic** t	(Fields)
Md	::=	**def** $Msig\{e\}$	(Methods)
$Msig$	::=	$m(x : t_1) : t_2$	(Signatures)
e	::=	v | x | $x.f$ | $x.f = e$ | $x.m(e)$ | **let** $x = e_1$ **in** e_2 | **new** C	(Expressions)
	|	**new atomic** C | $(t)\ e$ | **finish** $\{$**async** $\{e_1\}$ **async** $\{e_2\}\}$; e_3	
v	::=	**null**	(Values)
t	::=	C | I | **Unit**	(Base Types)
q	::=	**atomic** | **non_atomic**	(Atomicity Qualifiers)

to receive objects of atomic type.

Instead of atomic-qualifying fields of a class (C)'s implementation, one can simply create an object of type **atomic** C (cf. rows 5 to 8). Applying this approach to the list implementations produces types on which every access to the list falls within a unit of work. The resulting units of work are the same as ConcurrentList but their boundaries include the whole code of each method variant rather than only the list of instructions that encloses all accesses to atomic values. Regarding variant validity, the impact is confined to the restriction of the valid variants to the ones on which **this** is atomic, instead of being of a regular type. The impact would be larger if any of the methods related one of the their parameters with a field of the class' type, as the atomicity of the parameter would have to match the one of the class.

4 Static Analysis and Synthesis

In this section, we describe the static analysis performed by AtomiS to infer all atomic-qualifiers of the program's types, and to consistently synthesise the code for the valid variants of every method. To formally prove the soundness of the analysis we build on OOlong [6], an object-oriented multi-threaded programming calculus. It is a principled language with minimal syntax, a formal static and dynamic semantics, and a computer-aided proof of type soundness (progress and type preservation for well-typed programs) developed in Coq and publicly available (it is thus re-usable, and the mechanisation of our results in Coq build on it). Concretely, we developed a variant of OOlong, that we refer to as AtomiS-OOlong, that is a simplified version of AtomiS-Java. Crucially, the operational semantics of the language is exactly the same of OOlong, and we thus omit its presentation here.[8]

4.1 AtomiS-OOlong

AtomiS-OOlong extends OOlong to support the declaration of atomic fields and the instantiation of atomic objects, via the **atomic** keyword, and signature annotations, which become the only constructs to express concurrency restrictions. Accordingly, all lock related operations of OOlong are removed.

Table 2 presents the grammar of the language. It uses the following meta-identifiers: $C \in$ ClassName ranges over class names, $I \in$ InterfaceName over interface names, $f \in$ FieldName over field names, $m \in$ MethodName over method names, x and $y \in$ VarName over local variables. Programs consist of (lists of) interfaces Ids and classes Cds, and a starting expression e (that we refer to as main). The remaining expressions are standard, or have a restricted use for the sake of simplification, as in OOlong. Types are class or interface names, or **Unit** (used as the type of

---

[8]The interested reader can check Section 4, Figures 8 and 9, of OOlong main reference [6].

```
 1  interface Object {}                          15  class BaseList_na implements List_n{
                                                 16   head : atomic Node_n
 3  interface List_n {                           17   tail : atomic Node_n
 4   add(element : Object) : Unit [
         (non_atomic) → non_atomic,              19   def add(element : Object) : Unit {
         (non_atomic) → atomic]                  20    let node = new Node_n in
 5   get(pos : Integer) : Object [               21      node.value = element;
         (non_atomic) → non_atomic,              22      node.next = null;
         (atomic) → non_atomic]                  23      if (this.head == null) {
 6   equals(other : List) : Boolean [            24       node.prev = null;
         (non_atomic) → non_atomic,              25       this.head = node;
         (non_atomic) → atomic, (atomic) → non_atomic,  26     } else {
         (atomic) → atomic]                      27       node.prev = tail;
 7  }                                            28       tail.next = node;
                                                 29      } ; this.tail = node
 9  class Node_n implements Object {             30   }
10   value : Object
11   next : Node                                 32   def get(pos : Integer) : Object { ... }
12   prev : Node                                 33   def equals(other : List) : Boolean { ... }
13  }                                            34  }
```

Listing 4: ATOMIS-OOlong version of generated class `BaseList_na` and of its dependencies.

assignments). Types are assigned to fields, method return values and method parameters. The typing environment maps variables to types and abstract locations to class types.

The list of atomicity combinations that must be supported in the implementations of each interface method is here expressed as signature annotations, which are sequences of elements $(q_1) \to q_2$, where q_1 denotes the atomicity of the parameter and q_2 the atomicity of the return type. As in ATOMIS-Java, the programmer may also atomic-qualify the type in class instantiation to address the case of objects that are assigned to a local variable and then shared among threads. This happens in the following example, for the new `BaseList_na`:

```
let x = new atomic BaseList_na in let y = x in
 finish { async { x.add(new Object) } async { y.add(new Object) }} ; null
```

Listing 4 presents the ATOMIS-OOlong version of the generated `BaseList_na` specialisation of class `BaseList<T, N>`. For the sake of code readability, in the example we use the sequential composition operator (';', defined in [6]) to omit some of the **let** expressions. We also assume the existence of the **if** instruction, equality (denoted by ==), and classes `Boolean` and `Integer`. The code is very similar to what would be a Java version of a concurrent list of regular objects. The differences are mainly in interface `List`. Given that expressions of type **Unit** may be assigned to variables, method `add` must support the return of both **atomic** and **non_atomic** values. Also, as there are no primitive values, method `get` receives an object that may be either **atomic** or **non_atomic**.

4.2 AtomiS Analysis

The processing of an ATOMIS-OOlong program P_{orig}, written in the language defined in Table 2 (OOlong without locks and with signature and atomicity annotations), produces an OOlong program (still without locks) with safe (*i.e.*, type-safe and preventing atomicity violations) code that makes the atomicity qualification of every type in P_{orig} explicit. For each type t in P_{orig}, the generated code includes the regular and atomic versions of t. Moreover, the class types have their original methods unfolded into the corresponding valid variants, and all method calls are explicitly resolved into a valid variant.

The process consists of four stages, as illustrated in Fig. 1. The first starts by checking if, ATOMIS annotations aside, the input program is a well-typed OOlong program, and produces a program decorated with local types (prePness). It may fail if an OOlong typing error occurs. The second stage computes the valid variants for each method of the program and, for each of

these, the atomicities of its local variables. This stage may also fail if any of the variants required for the program to execute is not valid. When that happens, it returns an empty set causing the entire analysis to fail (validVariants). Then, the third stage checks if the set of valid variants of each class includes all the signatures defined in the class' ATOMIS-annotated interface (interfaceImpl). Naturally, it may also fail if the condition is not met. The first three stages guarantee the correct typing of the program, producing what we call *the solution* (Sol) that is used by the fourth stage to generate the final OOlong program (viP). Stage 1 checks if, ATOMIS annotations aside, the input program is a well-typed OOlong program, and may fail if an OOlong typing error occurs. Stage 2 computes the valid variants for each method of the program and, for each of these, the atomicities of its local variables. The stage may also fail if any of variants required for the program to execute is not valid. When that happens, validVariants returns an empty set, causing the entire analysis to fail. Lastly, Stage 3 checks if the set of valid variants of each class include all the signatures defined in the class' ATOMIS-annotated interface. Naturally, it may also fail if the condition is not met.

The first three stages guarantee the correct typing of the program, producing what we call *the solution* (Sol) that is used by Stage 4 to generate the final OOlong program. The complete analysis of an original program is a partial function, denoted AtomiSAnalysis, and defined as follows:

Definition 1 (ATOMIS Analysis). *Consider an original program P_{orig}, that is an annotated ATOMIS-OOlong program written in the language of Table 2. If*

Stage 1: ***preProcess***$(P_{orig}) = P$

Stage 2: ***validVariants***$(P) = Sol \neq \varnothing$

Stage 3: ***interfaceImpl***$(Sol, P) = true$

Stage 4: ***viP***$(Sol, P) = P^+$

then **AtomiSAnalysis**$(P_{orig}) = (P, Sol, P^+)$.

The final program is guaranteed to be well-typed and behaviourally equivalent to the program resulting of Stage 1 (see §5). The remainder of this section explains each of the 4 stages.

4.2.1 Stage 1 - The OOlong Type System

The static analysis is only applied to programs that, ATOMIS annotations aside, are well-typed OOlong programs. This property is guaranteed by a pre-processing of the original program:

$$\mathsf{preProcess}(P_{\mathrm{orig}}) = \begin{cases} \mathsf{dress}(P_{\mathrm{orig}}, \mathsf{OOlongTS}(\mathsf{strip}(P_{\mathrm{orig}}))) & \text{if } \mathsf{OOlongTS}(\mathsf{strip}(P_{\mathrm{orig}})) \text{ succeeds} \\ \bot & \text{otherwise.} \end{cases}$$

The first step is to strip[9] the code from all **atomic** and interface signature annotations. The resulting program is then submitted to OOlongTS, an instrumented version of the original OOlong type system [6]: the original verification includes program well-formedness and the matching of the types of the various components of the program; the added instrumentation incorporates in the type system rules a code decoration process that assigns types to local variables. So, in ATOMIS-OOlong, the syntax of the **let** construct is actually **let** $x : t = e_1$ **in** e_2. This decoration has no impact on the semantics of the **let** instruction – it can be trivially shown that OOlong's *Progress and Preservation* results still hold for the decorated language. The resulting program is lastly *redressed* with the original ATOMIS annotations to produce the program that will be passed to the subsequent stages of the analysis.[9]

4.2.2 Stage 2 - Valid Method Variants

The purpose of the second stage is to compute the valid variants of each method and, for each of these, obtain the atomicity of its local variables. The process is defined in function validVariants that receives the program returned by function preProcess and outputs the *solution*

---

[9]Function defined in Appendix B.

Table 3: Atomicity Inference Output

$\nu \in$ AtomicityValue	$=$	❄ $\mid$ ❅
$\check{x}, \check{y} \in$ AtomicityVar		
$\omega \in$ ValidityValue	$=$	**valid** $\mid$ **invalid**
$\mu \in$ VariantID	$::=$	$\nu_1.t.m.\nu_2.\nu_3$
Sol $\in$ Acs Solution	$=$	(AtomicityVar $\rightarrow$ AtomicityValue) $\cup$ (VariantID $\rightarrow$ ValidityValue)
$\alpha \in$ Atomicity	$::=$	$\nu \mid \check{x} \mid \check{x}@\mu$
$\eta \in$ SolExpr	$::=$	$\alpha_1 = \alpha_2 \mid (\check{\mu}, \omega) \mid \eta_1 \vee \eta_2 \mid \eta_1 \wedge \eta_2 \mid (\check{\mu}, \omega) \rightarrow \eta \mid (\eta)$
Acs $\in$ Constraint Systems	$=$	$\wp(\text{SolExpr})$
VariantVar	$::=$	$\check{x}_1.t.m.\check{x}_2.\check{x}_3 \mid \check{x}_1.t.m.\check{x}_2.\check{x}_3@\mu$
Vns $\in$ Method Call Systems	$=$	$\wp(\text{VariantVar})$
Mcs $\in$ Method Constraint Systems	$=$	VariantVar $\rightarrow$ Constraint Systems $\times$ Method Call Systems
$\check{\mu} \in$ ValidityVar		

(Sol $\in$ Acs Solution, syntax is defined in Table 3) – a map from variant identifiers to validity values and from atomicity variables to atomicity values:

$$\text{validVariants}(P) = \begin{cases} \text{Sol} & \text{if solve(Acs)} = (\textbf{true}, \text{Sol}) \\ \varnothing & \text{otherwise.} \end{cases} \quad \text{where:} \quad \begin{array}{l} \vdash P \rhd \text{Mcs and} \\ \text{variantConstraints(Mcs)} = \text{Acs} \end{array}$$

The validVariants function is defined by the composition of three other (partial) functions. The first ($\vdash$) performs a type-based generation of atomicity-related information for each method in the input program. The second (variantConstraints) uses this information to build a constraint system with the restrictions that have to be met, in order for a given variant to be valid. The constraint system is then passed to solve that makes use of a SAT solver to obtain the solution for both variant validity and atomicity of local variables. It returns a pair with the information of whether the given system has a solution, and if so, the solution itself; if there is no solution (the 'otherwise' case in the definition), solve returns a pair with **false** in the first position.

The first two steps are performed independently for each method and use only local information, allowing for separate compilation. The last requires the information collected for the entire program and hence must be done during the linking stage.

4.2.3 Type-Based Generation of Atomicity-Related Information

The generated atomicity-related information is a map that, for each method of the source program, provides: (a) the set of constraints imposed on the atomicity of the method's local variables; and (b) the set of the method variants that the method will call in its execution. As defined in Table 3, atomicity values $\nu \in$ Atomicity, as used in the analysis, consist of ❄ (atomic) and ❅ (non-atomic). Many times it is not possible to infer an expression's atomicity information in a small step analysis, as the atomicity of local variables is inherited from the expression assigned to them, which may be a method parameter or the result of a method call. To refer to such atomicity in subsequent expressions we use atomicity variables $\check{x}, \check{y} \in$ AtomicityVar. The same holds for variants, which are identified by five components: atomicity of the object itself (**this**), name of the class, name of the method, atomicity of the method's parameter, and atomicity of the method's return value. The latter two atomicities, in particular, are not always possible to infer in a small step analysis. Thus, to refer to method variants in method calls and to use them as keys to the generated atomicity-related information, we leverage on atomicity variables to set up variant variables of the form $\check{x}_1.t.m.\check{x}_2.\check{x}_3 \in$ VariantVar.

The rules of the type system are presented in Table 4. The variant identifiers $\check{x}_1.t.m.\check{x}_2.\check{x}_3$ used as keys in rule [AI_METHOD] have their $\check{x}_1$ and $\check{x}_2$ components constructed from program variable names, respectively, **this** (in rule [AI_CLASS]) and the method parameter, by using function **natvar** that generates an atomicity variable from program variable. The identifier $\check{x}_3$ is a fresh variable but could also be constructed from a reserved identifier, such as **return**. Rule [AI_PROGRAM] unites the information gathered for each class with the one generated for the *main* expression, which is also stored in the table under a special key.

Table 4: Generation of a program's atomicity information.

$\vdash P \triangleright \mathsf{Mcs}$	AI_PROGRAM	$\dfrac{\forall\, Cd \in Cds.\ \vdash Cd \triangleright \mathsf{Mcs}_{Cd} \quad \vdash_{\check{x}_3} e \triangleright (\mathsf{Acs}, \mathsf{Vns}) \quad \check{x}_1, \check{x}_2, \check{x}_3\ \textit{fresh}}{\vdash Ids\ Cds\ e \triangleright \bigcup_{Cd \in Cds}.\mathsf{Mcs}_{Cd} \cup \{\check{x}_1.\mathbf{Unit}.\mathrm{main}.\check{x}_2.\check{x}_3 \mapsto (\mathsf{Acs}, \mathsf{Vns})\}}$

$\vdash Cd \triangleright \mathsf{Mcs}$	AI_CLASS	$\dfrac{\forall\, Md \in Mds.\ \mathbf{this}:\check{x}.C \vdash Md \triangleright \mathsf{Mcs}_{Md} \quad \check{x} = \mathbf{natvar}(\mathbf{this})}{\vdash \mathbf{class}\ C\ \mathbf{implements}\ I\ \{\ Fds\ Mds\ \} \triangleright \bigcup_{Md \in Mds} \mathsf{Mcs}_{Md}}$

$\Upsilon \vdash Md \triangleright \mathsf{Mcs}$	AI_METHOD	$\dfrac{\mathbf{this}:\check{x}_1.C, x:\check{x}_2.t_1 \vdash_{\check{x}_3} e \triangleright (\mathsf{Acs}, \mathsf{Vns}) \quad \check{x}_2 = \mathbf{natvar}(x) \quad \check{x}_3\ \textit{fresh}}{\mathbf{this}:\check{x}_1.C \vdash \mathbf{def}\ m(x:t_1):t_2\ \{\ e\ \} \triangleright \{\check{x}_1.C.m.\check{x}_2.\check{x}_3 \mapsto (\mathsf{Acs}, \mathsf{Vns})\}}$

$\Upsilon \vdash_{\check{y}} e \triangleright (\mathsf{Acs}, \mathsf{Vns})$	AI_VAR	$\dfrac{\Upsilon(x) = \check{x}.t}{\Upsilon \vdash_{\check{y}} x \triangleright (\{\check{x} = \check{y}\}, \varnothing)}$	AI_NULL	$\dfrac{}{\Upsilon \vdash_{\check{y}} \mathbf{null} \triangleright (\{\check{y} = \check{y}\}, \varnothing)}$

AI_LET $\dfrac{\check{x} = \mathbf{natvar}(x) \quad \Upsilon \vdash_{\check{x}} e_1 \triangleright (\mathsf{Acs}_1, \mathsf{Vns}_1) \quad \Upsilon, x:\check{x}.t \vdash_{\check{y}} e_2 \triangleright (\mathsf{Acs}_2, \mathsf{Vns}_2)}{\Upsilon \vdash_{\check{y}} \mathbf{let}\ x:t = e_1\ \mathbf{in}\ e_2 \triangleright (\mathsf{Acs}_1 \cup \mathsf{Acs}_2, \mathsf{Vns}_1 \cup \mathsf{Vns}_2)}$

AI_CALL $\dfrac{\Upsilon(x) = \check{x}_1.t \quad \check{x}_2 = \mathrm{nextvar}() \quad \check{x}_3 = \mathrm{nextvar}() \quad \Upsilon \vdash_{\check{x}_2} e \triangleright (\mathsf{Acs}, \mathsf{Vns})}{\Upsilon \vdash_{\check{y}} x.m(e) \triangleright (\mathsf{Acs} \cup \{\check{x}_3 = \check{y}\}, \mathsf{Vns} \cup \{\check{x}_1.t.m.\check{x}_2.\check{x}_3\})}$
AI_NEW $\dfrac{\check{x} = \mathrm{nextvar}()}{\Upsilon \vdash_{\check{y}} \mathbf{new}\ C \triangleright (\{\check{x} = \check{y}\}, \varnothing)}$

AI_NEWAT $\dfrac{}{\Upsilon \vdash_{\check{y}} \mathbf{new\,atomic}\ C \triangleright (\{\check{y} = \circledast\}, \varnothing)}$
AI_CAST $\dfrac{\Upsilon \vdash_{\check{y}} e \triangleright (\mathsf{Acs}, \mathsf{Vns}) \quad \check{x} = \mathrm{nextvar}()}{\Upsilon \vdash_{\check{y}} (t)\ e \triangleright (\mathsf{Acs} \cup \{\check{x} = \check{y}\}, \mathsf{Vns})}$

AI_UPDATE $\dfrac{\Upsilon(x) = \check{x}.C \quad \check{x}_1\ \textit{fresh} \quad \Upsilon \vdash_{\check{x}_1} e \triangleright (\mathsf{Acs}, \mathsf{Vns})}{\Upsilon \vdash_{\check{y}} x.f = e \triangleright (\mathsf{Acs} \cup \{\check{y} = \check{y}, (\check{x} = \circledast \wedge \check{x}_1 = \mathsf{fatom}(C, \circledast, f)) \vee (\check{x} = \circleddash \wedge \check{x}_1 = \mathsf{fatom}(C, \circleddash, f))\}, \mathsf{Vns})}$

AI_SELECT $\dfrac{\Upsilon(x) = \check{x}.C}{\Upsilon \vdash_{\check{y}} x.f \triangleright (\{(\check{x} = \circledast \wedge \check{y} = \mathsf{fatom}(C, \circledast, f)) \vee (\check{x} = \circleddash \wedge \check{y} = \mathsf{fatom}(C, \circleddash, f))\}, \varnothing)}$

AI_FJ $\dfrac{\Upsilon \vdash_{\check{x}_1} e_1 \triangleright (\mathsf{Acs}_1, \mathsf{Vns}_1) \quad \Upsilon \vdash_{\check{x}_2} e_2 \triangleright (\mathsf{Acs}_2, \mathsf{Vns}_2) \check{x}_1, \check{x}_2\ \textit{fresh} \quad \Upsilon \vdash_{\check{y}} e \triangleright (\mathsf{Acs}, \mathsf{Vns})}{\Upsilon \vdash_{\check{y}} \mathbf{finish}\ \{\ \mathbf{async}\ \{\ e_1\ \}\ \mathbf{async}\ \{\ e_2\ \}\ \};e \triangleright (\mathsf{Acs}_1 \cup \mathsf{Acs}_2 \cup \mathsf{Acs}, \mathsf{Vns}_1 \cup \mathsf{Vns}_2 \cup \mathsf{Vns})}$

$\mathsf{fatom}(t, \nu, f) = \begin{cases} \circledast & \mathsf{fields}(t)(f) = \mathbf{atomic}\ t \vee (\mathsf{fields}(t)(f) = t \wedge \nu = \circledast) \\ \circleddash & \text{otherwise} \end{cases}$

Function $\mathsf{fields}(C)(f)$ is defined as in [6] and returns the possibly qualified type of field f of class C.
Function $\mathsf{interfaceOf}(C)$ consults the code to return the interface implemented by class C.

Rules for expressions receive: (a) a typing environment with elements of the form $x:\check{x}.t$, where $\check{x}$ and t denote, respectively, the atomicity and type of variable x; and (b) an atomicity variable ($\check{y}$) that carries the atomicity of the expression evaluation's recipient, be it a variable, a field or a method call argument. The output is a pair, whose first element (Acs) is a set of expressions that impose constraints on the values of atomicity variables, while the second (Vns) is the aforementioned set of method variants that the expression calls.

The atomicity of a local variable is given by the expression assigned to it. To that end, rule [AI_LET] creates a new atomicity variable ($\check{x}$) to represent the atomicity of variable x and uses it in the typing of e_1 (as the recipient's atomicity) and e_2 (in the typing environment). The *recipient's atomicity* variable is actively used in several other rules. An example is rule [AI_VAR], where it is used to ensure that the atomicity of the recipient is the same of the variable. In rules where it is not relevant, such as [AI_NULL] – **null** does not have a defined atomicity and, hence, may be passed to any recipient – we simply add the constraint $\check{y} = \check{y}$ to ensure that all atomicity variables are represented in the constraint set. A simple example of the application of these initial rules follows, where $\check{x}_2 = \mathbf{natvar}(x)$, $\check{x}_3$ *fresh*, $\hat{y} = \mathbf{natvar}(y)$, and $\hat{z} = \mathbf{natvar}(z)$; the generated constraints ensure that the atomicity of the parameter, of the return type, and of variable y bound by the first let-expression are all equal:

$$\mathbf{this}:\check{x}_1.C \vdash \mathbf{def}\ m(x:t):t\ \{\mathbf{let}\ y:t = x\ \mathbf{in}\ \mathbf{let}\ z:t = \mathbf{null}\ \mathbf{in}\ y\} \triangleright$$
$$\{\check{x}_1.C.m.\check{x}_2.\check{x}_3 \mapsto (\{\check{y} = \check{x}_2, \check{z} = \check{z}, \check{y} = \check{x}_3\}, \mathsf{Vns})\}$$

When calling a method, several variants may be available. The choice of the one to call is determined by the atomicity of the target object, of the expression passed as argument, and of the recipient. Accordingly, to compose the identifier of the variant to add to the set of calls, rule [AI_CALL] obtains the first three components ($\check{x}_1$, t and m) from the expression and the typing environment. The atomicity of the parameter ($\check{x}_2$) and of the return value ($\check{x}_3$) require fresh variables, which are used to match the referred atomicities with the one of the expression passed

as argument and with the one of the recipient of the call's return value. We will need to refer to variables $\check{x}_2$ and $\check{x}_3$ in subsequent stages of our analysis and hence need to know which identifiers have been generated. The usual approach is to store this information on a data structure, along with context information, and propagate it along the analysis. For the sake of simplicity and readability, in this presentation we opt for a different solution. We make use of the nextvar() deterministic variable generator that, once reset, always generates the same sequence of variables. Given that we will visit the source program always by the same order, if every rule generates the same amount of variables, we will always obtain the same identifier at the same points of the analysis. We make use of this generator every time we need (to remember) an atomicity variable that cannot be constructed from the code nor is reflected in the variant identifier, such as in rule [AI_NEW], where it used to represent the atomicity of the class being instantiated, or in rule [AI_CAST], where it is used to guarantee that the atomicity of the type used is the same of the one of the receiver and of the expression being cast.

The constraints imposed by rules [AI_SELECT] and [AI_UPDATE] are based on the atomicity of the field under analysis, which depends on the atomicity of the target object. When this object is not atomic ($\check{x} = \circledast$), the atomicity of the field is the one coded in the program. Otherwise, as explained in §2.2 and defined in function fatom, the field also becomes atomic if it is of the same type of the enclosing class. The application of the rules to a simplified version of method add from Listing 2 follows:

$$
\begin{aligned}
&\mathbf{this} : \check{x}_1.\mathsf{BaseList\_na} \vdash \mathbf{def}\ \mathsf{add}(\mathsf{element} : \mathsf{Object}) : \mathbf{Unit}\ \{\mathbf{let}\ \mathsf{node}\ :\ \mathsf{Node} = \mathbf{new}\ \mathsf{Node}\ \mathbf{in} \\
&\quad \mathbf{let}\ \_ : \mathbf{Unit} = \mathsf{node.value} = \mathsf{element}\ \mathbf{in}\ (\ldots)\ \mathbf{this}.\mathsf{tail} = \mathsf{node}\} \rhd \\
&\{\check{x}_1.\mathsf{BaseList\_na}.\mathsf{add}.\check{x}_2.\check{x}_3 \mapsto (\{
\end{aligned}
$$

$\check{\mathsf{node}} = \check{x}_4,$	Added by $[\text{AI\_NEW}],$ for $\check{x}_4 = \mathsf{nextvar}()$
	and $\check{\mathsf{node}} = \mathbf{natvar}(\mathsf{node})$
$\check{x}_6 = \check{x}_6, (\check{\mathsf{node}} = \circledast \wedge \check{x}_5 = \cancel{\circledast}) \vee (\check{\mathsf{node}} = \cancel{\circledast} \wedge \check{x}_5 = \cancel{\circledast}),$	Added by $[\text{AI\_UPDATE}],$
	for $\check{x}_5$ fresh and $\check{x}_6 = \mathbf{natvar}(\_)$
$\check{x}_5 = \check{x}_2,$	Added by $[\text{AI\_VAR}]$
$\check{x}_3 = \check{x}_3, (\check{x}_1 = \circledast \wedge \check{x}_7 = \circledast) \vee (\check{x}_1 = \cancel{\circledast} \wedge \check{x}_7 = \circledast),$	Added by $[\text{AI\_UPDATE}],$ for $\check{x}_7$ fresh
$\check{x}_7 = \check{\mathsf{node}}$	Added by $[\text{AI\_VAR}]$

$\},\varnothing)\}$ (1)

We thus have that the atomicity of local variable node must be the same as of the type instantiated in **new** Node ($\check{x}_4$) and of **this**.tail ($\check{x}_7$), whose atomicity is always $\circledast$, independently of the atomicity of **this** ($\check{x}_1$). Also the atomicity of parameter element ($\check{x}_2$) must be the same of node.value ($\check{x}_5$), which is always $\cancel{\circledast}$, independently of the atomicity of node. Given that the method's return type is **Unit**, no constraints are imposed on its atomicity ($\check{x}_3$).

Variant Constraints. Function variantConstraints takes the atomicity-information previously generated for each method (collected in Mcs) and builds a single (global) constraint system, comprising a variant validity constraint per every possible variant of each method. Validity of each variants is represented by a unique validity variable, used in the validity constraints so as to be solved.

$\mathsf{variantConstraints} : \mathsf{Method\ Constraint\ Systems} \to \mathsf{Constraint\ Systems}$
$$\mathsf{variantConstraints}(\mathsf{Mcs}) = \mathsf{map}(\lambda x.\mathsf{vvMCS}(x), \mathsf{Mcs})$$

$\mathsf{vvMCS}(\check{x}_1.\mathbf{Unit}.\mathsf{main}.\check{x}_2.\check{x}_3 \mapsto (\mathsf{Acs}, \mathsf{Vns})) = (\mathbf{valvar}(\mathsf{mainV}) = \mathbf{valid} \wedge \mathsf{Acs}@\mathsf{mainV} \wedge \mathsf{bindCall}(\mathsf{Vns}, \mathsf{mainV}))$
$$\text{where } \mathsf{mainV} = \cancel{\circledast}.\mathbf{Unit}.\mathsf{main}.\cancel{\circledast}.\cancel{\circledast}$$

$$\mathsf{vvMCS}(\check{x}_1.C.m.\check{x}_2.\check{x}_3 \mapsto (\mathsf{Acs}, \mathsf{Vns})) = \bigwedge_{\substack{\nu_1,\nu_2,\nu_3 \in \{\circledast, \cancel{\circledast}\} \\ \mu = \nu.C.m.\nu_1.\nu_2}} (\mathbf{valvar}(\mu) = \mathbf{valid}) \to \left(\begin{array}{l} (\mathsf{Acs} \cup \{\check{x}_1 = \nu_1, \check{x}_2 = \nu_2, \check{x}_3 = \nu_3\})@\mu \\ \wedge\ \mathsf{bindCall}(\mathsf{Vns}, \mu) \end{array} \right)$$

$$\mathsf{bindCall}(\mathsf{Vns}, \mu) = \bigwedge_{\check{x}_1.C.m.\check{x}_2.\check{x}_3 \in \mathsf{Vns}} \Big(\bigvee_{\nu_1,\nu_2,\nu_3 \in \{\circledast, \circledast\!\!\!/\}} \big(\mathbf{valvar}(\nu_1.C.m.\nu_2.\nu_3) = \mathbf{valid} \wedge \check{x}_1@\mu = \nu_1 \wedge \check{x}_2@\mu = \nu_2 \wedge \check{x}_3@\mu = \nu_3 \big) \Big)$$

$\mathbf{valvar}(\mu)$ generates a validity variable $\check{\mu}$ from variant identifier μ.

$\mathsf{Acs}@\mu$ defined in Appendix B, qualifies all occurrences of unqualified atomicity

variables ($\check{x}$) with the given variant identifier μ: $\check{x}@\mu$.

The *main* expression has a single variant, as it does not belong to any class, has no input parameter and the result of its evaluation may simply be discarded. We represent this variant with special identifier $\circledast\!\!\!/.\mathbf{Unit}.\mathsf{main}.\circledast\!\!\!/.\circledast\!\!\!/$, shortened to mainV. The validity of mainV is required to ensure that the program has an initial expression, meaning that all the constraints (Acs) imposed by the program's original initial expression must be satisfied, and the variants called must be valid. For global uniqueness, the local variables in Acs are qualified with the variant's id.[10] The constraints needed to guarantee that, for each method to call, there is a valid variant that satisfies the atomicity restrictions imposed by the caller are generated by function $\mathsf{bindCall}$. For each variant variable received ($\check{x}_1.C.m.\check{x}_2.\check{x}_3 \in \mathsf{Vns}$), the function adds a constraint to ensure that at least one variant (with identifier $\nu_1.C.m.\nu_2.\nu_3$ for ν_1, ν_2 and $\nu_3 \in \{\circledast, \circledast\!\!\!/\}$) of the target method is both valid and matches the atomicity restrictions for the object, the parameter and the return value. The resulting constraints are conjugated to ensure satisfiability of all calls.

The case for methods is similar, having its differences grounded on the fact that methods have 2^n possible variants (for $n = 3$ in ATOMIS-OOlong) and not all of these have to be valid. The validity of a variant implies the satisfiability of the restrictions imposed by the method's body, complemented by the constraints imposed by the variant on the atomicity of the object, parameter and return value, as well as the satisfiability of the restrictions imposed by $\mathsf{bindCall}$.

Solving the Constraint System. The global constraint system only makes use of equality, implication, conjunction and disjunction operations over two sets of binary values. It can thus easily be transformed into a Boolean satisfiability problem by mapping values $\circledast$ and $\mathbf{valid}$ into $\mathbf{true}$ and, $\circledast\!\!\!/$ and $\mathbf{invalid}$ into $\mathbf{false}$. The model resulting from the system's satisfiability assigns Boolean values to all validity and atomicity variables, producing what we call the *solution*. For the example of Eq. 1, independently of the variant, we have that $\mathsf{node} = \check{x}_4 = \check{x}_7 = \circledast$ and that $\check{x}_5 = \check{x}_2 = \circledast\!\!\!/$. So, only variants on which $\check{x}_2 = \circledast\!\!\!/$ are valid. For those we have (omitting auxiliary variables):

$\{ \circledast.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast = \mathbf{valid}, \circledast.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast.\mathsf{node} = \circledast, \circledast.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast.\check{x}_4 = \circledast,$

$\circledast\!\!\!/.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast = \mathbf{valid}, \circledast\!\!\!/.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast.\mathsf{node} = \circledast, \circledast\!\!\!/.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast.\check{x}_4 = \circledast,$

$\circledast.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast\!\!\!/ = \mathbf{valid}, \circledast.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast\!\!\!/.\mathsf{node} = \circledast, \circledast.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast\!\!\!/.\check{x}_4 = \circledast,$

$\circledast\!\!\!/.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast\!\!\!/ = \mathbf{valid}, \circledast\!\!\!/.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast\!\!\!/.\mathsf{node} = \circledast, \circledast\!\!\!/.\mathsf{BaseList\_na.add}.\circledast\!\!\!/.\circledast\!\!\!/.\check{x}_4 = \circledast \}$

We show the values for $\check{x}_4$, because it is the variable that will be used in the code generation to known the type to instantiate in the **new** Node expression, which in this case will be **atomic** Node.

4.2.4 Stage 3 - Interface Implementation

An ATOMIS-OOlong interface may feature signature annotations that enable the programmer to explicitly convey the atomicity of the methods parameter and return type. The third stage of the ATOMIS analysis has the goal of guaranteeing that the set of valid variants computed for each class C includes the signatures that result from the parsing of the interface I implemented by C. To that end, we define function $\mathsf{interfaceImpl}$ that receives the solution obtained from $\mathsf{validVariants}$ and the source code. The solution is used to retrieve the signatures of the valid variants of the class' methods, while the code is used the retrieve the signatures originally defined in the program's classes and interfaces. These signatures are represented by triples of the form $(\nu_1, \nu_2, m(x:t_1):t_2)$, where ν_1 and ν_2 denote, respectively, the atomicity of the parameter and of the return value, and $m(x:t_1):t_2$ denotes the method's original non-annotated signature.

---

[10]This is not important in the case of mainV, because there is only one variant, but is needed for methods in general.

$$\mathsf{interfaceImpl}(\mathsf{Sol}, Ids\ Cds\ e) = \forall \mathbf{class}\ C\ \mathbf{implements}\ I\ \{Mds\} \in Cds\ .\ \forall \nu \in \mathsf{AtomicityValue}\ .$$
$$\bigcup_{IMsig \in \mathsf{msigs}(I)} \mathsf{interfaceMsig}(IMsig) \subseteq \bigcup_{Msig \in \mathsf{msigs}(C)} \mathsf{variantMsigs}(\mathsf{Sol}, \nu, C, Msig)$$

$$\mathsf{interfaceMsig}(m(x:t_1):t_2\ Sas) = \{(\mathsf{atom}(q_1), \mathsf{atom}(q_2), m(x:t_1):t_2) \mid (q_1) \to q_2 \in Sas\}$$

$$\mathsf{atom}(\mathbf{atomic}) = \circledast \qquad \mathsf{atom}(\mathbf{non\_atomic}) = \circledast$$

$$\mathsf{variantMsigs}(\mathsf{Sol}, \nu, C, m(x:t_1):t_2) = \{(\nu_1, \nu_2, m(x_1:t_1):t_2) \mid \mathsf{Sol}(\mathbf{valvar}(\nu.C.m.\nu_1.\nu_2)) = \mathbf{valid}\}$$

The set of signatures defined by interface `List` of our running example is:

$$\{\ (\circledast, \circledast, \mathsf{add}(x:\mathtt{Object}):\mathbf{Unit}), (\circledast, \circledast, \mathsf{add}(x:\mathtt{Object}):\mathbf{Unit}), (\circledast, \circledast, \mathsf{get}(x:\mathtt{Integer}):\mathtt{Object}),$$
$$(\circledast, \circledast, \mathsf{get}(x:\mathtt{Integer}):\mathtt{Object}), (\circledast, \circledast, \mathsf{equals}(x:\mathtt{List}):\mathtt{Boolean}), (\circledast, \circledast, \mathsf{equals}(x:\mathtt{List}):\mathtt{Boolean}),$$
$$(\circledast, \circledast, \mathsf{equals}(x:\mathtt{List}):\mathtt{Boolean}), (\circledast, \circledast, \mathsf{equals}(x:\mathtt{List}):\mathtt{Boolean})\ \}$$

On the other hand, in agreement with the discussion about valid variants in §2.2, we have that the set of signatures for methods `add`, `get` and `equald` deemed valid for either `BaseList_na` and **atomic** `BaseList_na` is also the one above. Ergo, given that the latter set is included in the former, the `Collection` interface is correctly implemented by both `BaseList_na` and **atomic** `BaseList_na`.

4.2.5   Stage 4 - Code Generation

The purpose of the final code generation stage is five-fold: 1 - replace original method signatures in interfaces by the ones resulting from the parsing of the atomicity annotations; 2 - replace the method definitions in classes by the variants considered valid in the solution generated by Stage 2 (validVariants); 3 - replace the method names by the right variant in all method calls; and 4 - types and qualifiers by new types from a set that is implicitly partitioned into atomic and no-atomically qualified types. The resulting code will be cleared from all ambiguities with regard to variable atomicity.

The viP code generation function is presented in Table 5. It is parametric on the solution produced by Stage 2 and on the source program obtain in Stage 1, and makes use of functions ootype, oosig and oomn, defined in Appendix B. These functions generate OOlong identifiers for the types and method names to include in the final program, namely ootype generates type identifiers for pairs (atomicity, type), oosig generates method signature for triples (atomicity, atomicity, method signature) and oomn generates method names for triples (atomicity, atomicity, method name).

The viP function processes the definitions of every type t (interface or class) from the input program, and generates the two correspondent types $\mathsf{ootype}(\circledast, t)$ and $\mathsf{ootype}(\circledast, t)$. In the case of interfaces, the signatures that compose the type are obtained from function interfaceMsig that parses the programmers signature annotations. Concerning classes, the atomicity of fields is defined by function fatom (see §4.2.2) and the method definition list is obtained by consulting the valid variants of the class' methods in Sol.

The generation of expressions, for the methods' body and the *main* expression, is given by function viE. The case for **let** requires the retrieval of the atomicity of type t in the expression. For that, we query the solution for the atomicity value assigned to atomicity variable $\tilde{x}$, obtained (as in Table 4) from the **natvar** generator. The cases for method calls, **new** and casts also require information stored in the solution. In these cases (also as in Table 4) the atomicity variables are obtained from the nextvar() generator. In the particular case of method calls, the identifier of the method to call is replaced by the one of the correct variant, i.e. the one with the signature compatible with the atomicities of the parameter and of the return value. This identifier is generated from the atomicity values computed in the solution for the atomicity variables created in rule [AI_CALL] from Table 4. The remainder rules are straightforward code translations.

We note that this final stage does not fail due to the fact that, when passing the solution Sol to viE, we know that Sol is defined on all the variables generated by natVar and nextVar and tagged with a valid variant $\nu.D.m.v_1.v_2$ (*cf.* viMd).

Table 5: Code generation

$$\mathsf{viP}(\mathsf{Sol}, Ids\ Cds\ e) = \biguplus_{Id \in Ids} \mathsf{viI}(Id)\ \uplus \biguplus_{Cd \in Cds} \mathsf{viC}(\mathsf{Sol}, Cd)\ \uplus \mathsf{viE}(\mathsf{Sol}, \mathsf{mainV}, e)$$

$$\mathsf{viI}(\textbf{interface}\ I\ \{IMsigs\}) = \biguplus_{\nu \in \{\circledast, \nparallel\}} \textbf{interface}\ \mathsf{ootype}(\nu, I)\ \{ \biguplus_{IMsig \in IMsigs} \mathsf{oosig}(\mathsf{interfaceMsig}(IMsig))\}$$

$$\mathsf{viI}(\textbf{interface}\ I\ \textbf{extends}\ I_1, I_2) = \biguplus_{\nu \in \{\circledast, \nparallel\}} \textbf{interface}\ \mathsf{ootype}(\nu, I)\ \textbf{extends}\ \mathsf{ootype}(\nu, I_1), \mathsf{ootype}(\nu, I_2)$$

$$\mathsf{viC}(\mathsf{Sol}, \textbf{class}\ C\ \textbf{implements}\ I\ \{Fds\ Mds\}) = \biguplus_{\nu \in \{\circledast, \nparallel\}} \textbf{class}\ \mathsf{ootype}(\nu, C)\ \textbf{implements}\ \mathsf{ootype}(\nu, I)\ \Big\{$$

$$\biguplus_{Fd \in Fds} \mathsf{viField}(\nu, Fd)\ \uplus \biguplus_{Md \in Mds} \mathsf{viMd}(\mathsf{Sol}, \nu, C, Md)\Big\}$$

$$\mathsf{viField}(C, \nu, f : t) = \begin{cases} f : \mathsf{ootype}(\circledast, t) & \text{if } \mathsf{fatom}(C, \nu, f) = \circledast \\ f : \mathsf{ootype}(\nparallel, t) & \text{otherwise} \end{cases}$$

$$\mathsf{viMd}(\mathsf{Sol}, \nu, D, \textbf{def}\ m(x : t_1) : t_2\{e\}) = \biguplus_{\mathsf{Sol}(\textbf{valvar}(\nu.D.m.\nu_1.\nu_2)) = \textbf{valid}} \textbf{def}\ Msig'\ \{\ \mathsf{viE}(\mathsf{Sol}, \nu.D.m.\nu_1.\nu_2, e)\ \}$$

$$\text{where } Msig' = \mathsf{oosig}(\nu_1, \nu_2, m(x : t_1) : t_2)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, \textbf{let}\ x : t = e_1\ \textbf{in}\ e_2) = \textbf{let}\ x : \mathsf{ootype}(\nu_1, t) = \mathsf{viE}(\mathsf{Sol}, \mu, e_1)\ \textbf{in}\ \mathsf{viE}(\mathsf{Sol}, \mu, e_2)$$

$$\text{where } \check{x} = \textbf{natvar}(x)\ \text{and}\ \nu_1 = \mathsf{Sol}(\check{x}@\mu)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, x.m(e)) = x.m'(\mathsf{viE}(\mathsf{Sol}, \mu, e))$$

$$\text{where } \check{x}_1 = \mathsf{nextvar}(); \check{x}_2 = \mathsf{nextvar}(), \nu_1 = \mathsf{Sol}(\check{x}_1@\mu), \nu_2 = \mathsf{Sol}(\check{x}_2@\mu)$$

$$\text{and } m' = \mathsf{oomn}(m, \nu_1, \nu_2)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, \textbf{new}\ C) = \textbf{new}\ C'\quad \text{where } \check{x} = \mathsf{nextvar}(), \nu = \mathsf{Sol}(\check{x}@\mu)\ \text{and}\ C' = \mathsf{ootype}(\nu, C)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, \textbf{new atomic}\ C) = \textbf{new}\ C'\quad \text{where } C' = \mathsf{ootype}(\circledast, C)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, x.f = e) = x.f = \mathsf{viE}(\mathsf{Sol}, \mu, e)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, (t)\ e) = (t')\ \mathsf{viE}(\mathsf{Sol}, \mu, e)$$

$$\text{where } \check{x} = \mathsf{nextvar}(), \nu = \mathsf{Sol}(\check{x}@\mu)\ \text{and}\ t' = \mathsf{ootype}(\nu, t)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, \textbf{finish}\{\textbf{async}\{e_1'\}\ \textbf{async}\{e_2'\}\}; e)$$

$$= \textbf{finish}\ \{\textbf{async}\{\mathsf{viE}(\mathsf{Sol}, \mu, e_1')\}\ \textbf{async}\{\mathsf{viE}(\mathsf{Sol}, \mu, e_2')\}\};\ \mathsf{viE}(\mathsf{Sol}, \mu, e)$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, x.f) = x.f$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, x) = x$$

$$\mathsf{viE}(\mathsf{Sol}, \mu, v) = v$$

5 Soundness

The algorithm presented in the previous section is elaborate enough to require a rigorous statement, and proof, of soundness. In this section we show that the AtomiSAnalysis process guarantees type and behavioural soundness, *i.e.*, that (when successful) it preserves typeability and atomicity annotations, and moreover, it produces a program behaviourally equivalent to the original one.

The type soundness of the automatically generated program has been proved in Coq. The definitions and proofs weight about 18 kloc, for a total of 297 definitions, 310 lemmas, one theorem (type preservation for programs), and six axioms about unimplemented aspects; the axiomatisation of the constraint solver relies on standard assumptions relating the input to the generated solution (see file README in the mechanisation[11] artefact for more details). The proof uses standard tactics from Coq's library.

The following definitions and resultsare stated for programs P_{orig}, *i.e.*, annotated Atomis-OOlong programs written in the language of Table 2, for which the analysis succeeds and produces a final OOlong program, according to Definition 1.

---

[11] AtomiS-Coq proof of type preservation (2022), URL: https://zenodo.org/record/6346649 and https://zenodo.org/record/6382015

5.1 Type Soundness and Mechanisation

We start by formalising how programs that are generated by the analysis are typeable by the OOlong type system, while attributing the same base types to fields and methods, and atomicities are consistent with those annotated in the original code. To lighten the notation, in the following we say that type $\mathsf{ootype}(\nu, t)$ is a compound of atomicity ν and type t.

5.1.1 Type Soundness

The Preservation of Base Types theorem states that the ATOMIS analysis transforms programs that, when stripped of atomicity annotations, are typable with a certain type, into (OOlong) programs that are also typeable with a type that is a compound of the former. It thus assumes that AtomiSAnalysis succeeds, and typability respects the OOlong type system, which we denote by $\vdash_{Ool}$.

Theorem 2 (Preservation of Base Types). *Consider an original program P_{orig} such that* **AtomiSAnalysis**$(P_{orig}) = (P, \mathsf{Sol}, P^+)$. *If* **strip**$(P_{orig}) = S$ *is typeable with type t, then the final (OOlong) program P^+ is also typeable with a type that is a compound of t: for all t such that $\vdash_{Ool} S : t$, there exists ν, such that $\vdash_{Ool} P^+ : $* **ootype**$(\nu, t)$.

The Consistency of Types and Atomicity Inference theorem states that the program transformation performed by the AtomiS analysis produces programs whose field and signature types are consistent with those of the original ATOMIS annotations. More specifically, the field types of the final program are a compound of the atomicity and types of the fields given by the original program to fields with the same name in corresponding classes, and the signature types of methods in the final program are a compound of the atomicity and types of the methods given by the original program to methods with the corresponding name in corresponding classes.

Theorem 3 (Consistency of Types and Atomicity Inference). *Consider an original program P_{orig}, that is annotated according to the ATOMIS model such that* **AtomiSAnalysis**$(P_{orig}) = (P, \mathsf{Sol}, P^+)$. *The types of the final program P^+, of its signatures and of its fields are generated from a compound version of those in P that is consistent with P_{orig}'s atomicity annotations, i.e.:*

1. *for all C^+, f, t^+ such that* **fields**$_{P^+}(C^+)(f) = t^+$, *there exist C, ν, t such that $C^+ = $* **ootype**$(\nu, C)$, *and either*

   (a) **fields**$_{P_{orig}}(C)(f) = t$ *and $t^+ = $* **ootype**$(\circledast, t)$, *or*

   (b) **fields**$_{P_{orig}}(C)(f) = $ **atomic** t *and $t^+ = $* **ootype**$(\circledast, t)$;

2. *for all t^+, m^+, t_1^+, t_2^+ s.t.* **msigs**$_{P^+}(t^+)(m^+) = x : t_1^+ \to t_2^+$, *there exist $t, m, q_1, t_1, q_2, t_2, \nu$ s.t.* **msigs**$_{P_{orig}}(t)(m) = x : q_1\ t_1 \to q_2\ t_2$ *and $t^+ = $* **ootype**$(\nu, t)$, *with $t_1^+ = $* **ootype**(**atom**$(q_1), t_1)$, *and $t_2^+ = $* **ootype**(**atom**$(q_2), t_2)$ *and also $m^+ = $* **oomn**$(m, $**atom**$(q_1), $**atom**$(q_2))$.

Note that the Progress and Preservation results that hold for the OOlong language and type system ensure that *the output of the analysis never goes wrong* in what regards both base types and atomicities.

5.1.2 Proof Mechanisation

The proofs are based on a refinement of the OOlong type system, that represents an internal step of the analysis, where types are formalised as pairs of atomicity qualifiers and base types. The definition of the source language and typing system largely reuses code in [6], while we narrowed programs in order to avoid repetitions:

```
Definition program := ( { cds : list classDecl | NoDup (map pclassName cds) ∧ Forall NoDupMethods cds } *
    { ids : list interfaceDecl | NoDup (map pinterfaceName ids) } * expr) %
```

The constraint generator in Table 4 has been implemented as an inductive type, denoted `programConstraints`, mimicking a partial function: the definition relates the program and the variables received in input to a domain of Variant IDs and a partial map produced in output; the map, or *ID environment* (denoted type `muvarEnv`), associates the domain's IDs to pairs $(\mathsf{Acs}, \mathsf{Vns})$, while variables are used to implement the "freshness" mechanism required by `nextvar()`:

```
Check programConstraints: program →  list hatVar →  list nat →  MethodVarSystems →  muvarEnv →  Prop
```

The domain and the ID environment are passed to `variantConstraints` in order to produce the constraint system to be passed to the solver, which in turn relies on `vvMcs` to analyse all possible atomicity combinations (v_1, v_2, v_3) and generate entries of the form $\mathbf{valvar}(v_1.t.m.v_2.v_3) = \mathbf{valid} \rightarrow \eta$, where t and m are the (cast of the) type and the method name of `muv`, respectively, and η is a conjunction of pairs qualified with μ:

```
Definition variantConstraints (valVar : valVarT) (domain : MethodVarSystems) (mue : muvarEnv) : list
    NatureConstraintSystems := map (fun muv ⇒  match mue muv with | Some (acs, vns) ⇒  vvMcs valVar
    muv acs vns | _ ⇒  nil end) domain.
```

The Coq formulation of type preservation is stated below, where for short we omit some (minor) context hypotheses. It ensures that if P has type t (that is the type of the main expression), then the generated program P' has type (corresponding to the non-atomic version of) t, where `valVar` is a partial map from IDs to variables, and `fqmMain` is the ID reserved for the main:

```
Check T_preserve: ∀ ..., wfProgram P t →  ...→  programConstraints P fv sn domain mue →  makeV P = (domV,
    valVar) →  variantConstraints valVar domain mue = acs →  solver (concat acs) = sol →  valVar fqmMain
    = Some valvar →  sol valvar = valid →  viP P sol valVar fv = Some P' →  wfProgramOO P' (setNonAtomic
    (castT t))
```

5.2 Behavioural Soundness

We have seen that our approach performs a program transformation that consistently fleshes out the atomicity qualifiers of every type in the program, while unfolding classes and methods according to their determined qualified types. It remains to assert that the transformation does not affect the original behaviour, *i.e.*, that the final program does everything the original one does, and nothing more, according to a notion of indistinguishability. Intuitively, we wish to consider the correspondence between types and method names occurring in the final program, and those from which they originated in the original code.

Bisimulations are often used to relate pairs of concurrent programs that exhibit the same behaviour according to some criteria, step-by-step, by establishing a full correspondence between the possible outcomes of both programs in such a way that they will also simulate each other. In our case we need a relation between thread collections that is based on the notion of heap correspondence. To establish it, we define *syntactic correspondences* between *thread collections* and between *program contexts*, understood as the list of interfaces and classes of a program. These correspondences are defined as pairs of mappings between types and between method names that are used in programs and configurations:

Definition 4 (Expression and Thread Collection Correspondence). *Given a pair*
$\boldsymbol{MAP} = (\boldsymbol{methodMap}, \boldsymbol{typeMap})$ *of maps, where*
$\boldsymbol{methodMap}$: *MethodName* $\rightarrow$ *MethodName and* $\boldsymbol{typeMap}$: *Type* $\rightarrow$ *Type, and two program contexts* PC_1 *and* PC_2, *we say that there is a syntactic correspondence between:*

1. *expressions* e_1 *and* e_2, *well-formed with respect to* PC_1 *and* PC_2 *respectively, if* $\vdash^{PC_1, PC_2}_{MAP} e_1 \propto e_2$

2. *thread collections* T_1 *and* T_2, *well-formed with respect to* PC_1 *and* PC_2 *respectively, if* $\vdash^{PC_1, PC_2}_{MAP} T_1 \propto T_2$

according to the rules in Table 6.

Table 6: Syntactic Correspondence for Expressions and Thread Collections

$\boxed{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2 \text{ and } \vdash^{PC_1,PC_2}_{\text{MAP}} T_1 \propto T_2}$

$\text{SC\_VAR} \dfrac{}{\vdash^{PC_1,PC_2}_{\text{MAP}} x \propto x} \qquad\qquad \text{SC\_LET} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2 \qquad \vdash^{PC_1,PC_2}_{\text{MAP}} e'_1 \propto e'_2}{\vdash^{PC_1,PC_2}_{\text{MAP}} \text{let } x : \text{typeMap}(t_2) = e_1 \text{ in } e'_1 \propto \text{let } x : t_2 = e_2 \text{ in } e'_2}$

$\text{SC\_CALL} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2}{\vdash^{PC_1,PC_2}_{\text{MAP}} x.\text{methodMap}(m_2)(e_1) \propto x.m_2(e_2)} \qquad \text{SC\_CAST} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2}{\vdash^{PC_1,PC_2}_{\text{MAP}} (\text{typeMap}(t_2))e_1 \propto (t_2)e_2}$

$\text{SC\_SELECT} \dfrac{}{\vdash^{PC_1,PC_2}_{\text{MAP}} x.f \propto x.f} \qquad \text{SC\_LOC} \dfrac{}{\Gamma \vdash^{PC_1,PC_2}_{\text{MAP}} \iota \propto \iota} \qquad \text{SC\_NULL} \dfrac{}{\vdash^{PC_1,PC_2}_{\text{MAP}} \text{null} \propto \text{null}}$

$\text{SC\_UPDATE} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2}{\Gamma \vdash^{PC_1,PC_2}_{\text{MAP}} x.f = e_1 \propto x.f = e_2} \qquad \text{SC\_NEW} \dfrac{}{\vdash^{PC_1,PC_2}_{\text{MAP}} \text{new typeMap}(C_2) \propto \text{new } C_2}$

$\text{SC\_SPAWN} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1{}' \propto e_2{}' \qquad \vdash^{PC_1,PC_2}_{\text{MAP}} e_1{}'' \propto e_2{}'' \qquad \vdash e_1 \propto e_2}{\vdash^{PC_1,PC_2}_{\text{MAP}} \text{finish } \{ \text{ async } \{ e_1{}' \} \text{ async } \{ e_1{}'' \} \}; e_1 \propto \text{finish } \{ \text{ async } \{ e_2{}' \} \text{ async } \{ e_2{}'' \} \}; e_2}$

$\text{SC\_ASYNC} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} T_1 \propto T_2 \qquad \vdash^{PC_1,PC_2}_{\text{MAP}} T_1{}' \propto T_2{}' \qquad \vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2}{\vdash^{PC_1,PC_2}_{\text{MAP}} T_1 \parallel T_1{}' \triangleright e_1 \propto T_2 \parallel T_2{}' \triangleright e_2}$

$\text{SC\_EXN} \dfrac{}{\vdash^{PC_1,PC_2}_{\text{MAP}} \textbf{EXN} \propto \textbf{EXN}} \qquad \text{SC\_THREAD} \dfrac{\vdash^{PC_1,PC_2}_{\text{MAP}} e_1 \propto e_2}{\vdash^{PC_1,PC_2}_{\text{MAP}} (\mathcal{L}, e_1) \propto (\mathcal{L}, e_2)}$

A Program Context Correspondence is defined as a pair of maps, the first relating method names, and the second relating types, which establishes a correspondence from a program context (which can be thought of as the final one) into another (which can be thought of as the original one) in such a way that: the types given to fields of each class in the original program correspond to those given to fields in corresponding classes of the final program; and the name, types and method bodies of methods in the original program correspond to those given to methods in corresponding classes of the final program.

Definition 5 (Program Context Correspondence). *A pair of maps*
$MAP = (methodMap, typeMap)$, *where* $methodMap : MethodName \to MethodName$ *and* $typeMap :$
$Type \to Type$, *is said to establish a correspondence from program context* PC_2 *(and implicit* $fields_2$,
$methods_2$*) to program context* PC_1 *(and implicit* $fields_1$, $methods_1$*), written* $PC_1 \leq_{MAP} PC_2$,
if:

1. *for all* C_2, f, *we have that*
   $fields_1(typeMap(C_2))(f) = typeMap(fields_2(C_2)(f))$;

2. *for all* t_2, m_2 *such that* $methods_2(t_2)(m_2) = x : t'_2 \to t''_2, e_2$, *for some* t'_2, t''_2, *we have that*
   $methods_1(typeMap(t_2))(methodMap(m_2)) = x : typeMap(t'_2) \to typeMap(t''_2), e_1$ *and*
   $\vdash^{PC_1,PC_2}_{MAP} e_1 \propto e_2$.

We consider herein the dynamic semantics of OOlong, and its run-time constructs, which we assume well-formed (Figure 6 and 7 in [6]). Oolong's *configurations* $\langle H; V; T \rangle$ include heaps H mapping abstract locations to objects, variable maps V, and collections of threads T.

A program context correspondence is further said to define a correspondence from a heap and its implicit program context (which can be thought of as the final one) to another heap and its implicit program context (which can be thought of as the original one) if the two heaps have the same domain, and the heaps assign corresponding classes, the same field map and lock status to all locations.

Definition 6 (Heap Correspondence). *A program context correspondence* MAP *from* PC_2 *to* PC_1
is said to define a correspondence from heap H_2 *(and implicit program context* PC_2*) to heap* H_1
(and implicit program context PC_1*), written* $H_1 \leq^{PC_1,PC_2}_{MAP} H_2$, *iff:*

1. $\boldsymbol{dom}(H_1) = \boldsymbol{dom}(H_2)$, *and*

2. *for all $\iota \in \boldsymbol{dom}(H_1)$: $H_1(\iota) = (\boldsymbol{typeMap}(C_2), F, L)$ iff $H_2(\iota) = (C_2, F, L)$.*

This means that, if **MAP** = (**methodMap**, **typeMap**) is a program context correspondence, then **typeMap** is homomorphic with respect to **fields**, and **methodMap** is homomorphic with respect to **msigs**. Furthermore, if **MAP** is a heap correspondence, then it is homomorphic with respect to the structure of the heap.

The following bisimulation relation is designed to relate thread collections that derive from programs between which there is a program context correspondence, and which preserve a heap correspondence throughout all possible execution paths. It is inspired by the notion of abstraction homomorphism, which relates processes with structurally simpler but semantically equivalent ones [7]. The configuration steps in the definition below derive from the small-step dynamic semantics of OOlong (*cf.* Figures 8 and 9 in [6]).

Definition 7 (Bisimulation). *Given a program context correspondence $\boldsymbol{MAP}$ from PC_2 to PC_1, a $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation is a binary relation $\mathcal{B}$ on well-formed thread collections, that satisfies, for all T_1, T_2, H_1, H_2, V:*

- *if $T_1 \; \mathcal{B} \; T_2$ and $H_1 \leq_{\boldsymbol{MAP}}^{PC_1, PC_2} H_2$ then:*

  1. *$\langle H_1; V; T_1 \rangle \rightarrow \langle H_1'; V'; T_1' \rangle$ implies that exist H_2', T_2' such that $\langle H_2; V; T_2 \rangle \rightarrow \langle H_2'; V'; T_2' \rangle$ and $H_1' \leq_{\boldsymbol{MAP}}^{PC_1, PC_2} H_2'$ and $T_1' \; \mathcal{B} \; T_2'$;*

  2. *$\langle H_2; V; T_2 \rangle \rightarrow \langle H_2'; V'; T_2' \rangle$ implies that exist H_1', T_1' such that $\langle H_1; V; T_1 \rangle \rightarrow \langle H_1'; V'; T_1' \rangle$ and $H_1' \leq_{\boldsymbol{MAP}}^{PC_1, PC_2} H_2'$ and $T_1' \; \mathcal{B} \; T_2'$.*

The set of pairs of values is a $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation. Furthermore, the union of a family of $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulations is a $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation. Consequently, the union of all $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulations is a $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation, the largest $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation, which we denote by $\simeq_{\boldsymbol{MAP}}^{PC_1, PC_2}$.

The final result states that when a program undergoes the AtomiSAnalysis, the initial and final programs simulate each other by means of a $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation as defined above. To prove it we will define a concrete bisimulation and use an auxiliary Replacement lemma.

Lemma 8 (Replacement w.r.t. $\propto$).

1. *If $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} eE_1 \propto E_2[e_2]$, then exist $E_1[\bullet], e_1$ such that $eE_1 = E_1[e_1]$ and $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} e_1 \propto e_2$. Furthermore, if $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} \hat{e}_1 \propto \hat{e}_2$, then $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} E_1[\hat{e}_1] \propto E_2[\hat{e}_2]$.*

2. *If $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} E_1[e_1] \propto eE_2$, then exist $E_2[\bullet], e_2$ such that $eE_2 = E_2[e_2]$ and $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} e_1 \propto e_2$. Furthermore, if $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} \hat{e}_1 \propto \hat{e}_2$, then $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} E_1[\hat{e}_1] \propto E_2[\hat{e}_2]$.*

Proof. By induction on the structure of E_2 and E_1, respectively. $\square$

We prove that, for typable programs, $\propto$ is a Bisimulation:

Lemma 9 ($\propto$ is a Bisimulation). *Given a program context correspondence $\boldsymbol{MAP}$ from PC_1 to PC_2, $\propto$, when restricted to typable programs, is a $(\boldsymbol{MAP}, PC_1, PC_2)$-bisimulation.*

Proof. Since typability is preserved by reduction, it is enough t prove that if $PC_1 \leq_{\boldsymbol{MAP}} PC_2$, and $H_1 \leq_{\boldsymbol{MAP}}^{PC_1, PC_2} H_2$, and $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} T1 \propto T2$, then:

1. $\langle H_1; V; T_1 \rangle \rightarrow \langle H_1'; V; T_1' \rangle$ implies
   $\langle H_2; V; T_2 \rangle \rightarrow \langle H_2'; V; T_2' \rangle$ and $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} T1' \propto T2'$ and $H_1' \leq_{\boldsymbol{MAP}}^{PC_1, PC_2} H_2'$.

2. $\langle H_2; V; T_2 \rangle \rightarrow \langle H_2'; V; T_2' \rangle$ implies
   $\langle H_1; V; T_1 \rangle \rightarrow \langle H_1'; V; T_1' \rangle$ and $\vdash_{\boldsymbol{MAP}}^{PC_1, PC_2} T1' \propto T2'$ and $H_1' \leq_{\boldsymbol{MAP}}^{PC_1, PC_2} H_2'$.

For (2) We distinguish the following cases (the cases for (1) are analogous):

1. If $T_2 = (\mathcal{L}, e_2)$, then exist $E_2[\bullet], \hat{e}_2, \hat{e}_2'$ such that $e_2 = E_2[\hat{e}_2]$ and $\langle H_2; V; (\mathcal{L}, \hat{e}_2) \rangle \to \langle H_2'; V'; \hat{T}_2' \rangle$, and:

   (a) $\hat{T}_2' = (\mathcal{L}, \hat{e}_2')$, and $T_2' = (\mathcal{L}, E_2[\hat{e}_2'])$;

   (b) $\hat{e}_2 = \texttt{finish}\ \{\ \texttt{async}\ \{\ \hat{e}_2^1\ \}\ \texttt{async}\ \{\ \hat{e}_2^2\ \}\ \};\hat{e}_2^3$, and $\hat{T}_2' = (\mathcal{L}, \hat{e}_2^{\,1}) \| (\mathcal{L}, \hat{e}_2^{\,2}) \rhd \hat{e}_2^3$, and $T_2' = \hat{e}_2^1 \| \hat{e}_2^2 \rhd E_2[\hat{e}_2^3]$.

   Assuming, without loss of generality, that $\hat{e}_2$ is the smallest in the sense that it cannot be written in terms of a non-empty context, we prove these cases using the Replacement Lemma 8, and by case analysis on the transitions $\langle H_2; V; \hat{e}_2 \rangle \to \langle H_2'; V'; \hat{e}_2' \rangle$.

2. If $T_2 = \hat{T}_2^{\,1} \| \hat{T}_2^{\,2} \rhd \hat{e}_2$, and

   (a) $\langle H_2; V; \hat{T}_2^{\,1} \rangle \to \langle H_2'; V; \hat{T}_2^{\,1}{}' \rangle$ and $T_2' = \hat{T}_2^{\,1}{}' \| \hat{T}_2^{\,2} \rhd \hat{e}_2$

   (b) $\langle H_2; V; \hat{T}_2^{\,2} \rangle \to \langle H_2'; V; \hat{T}_2^{\,2}{}' \rangle$ and $T_2' = \hat{T}_2^{\,1} \| \hat{T}_2^{\,2}{}' \rhd \hat{e}_2$

   (c) $\hat{T}_2^{\,1} = (\mathcal{L}, \hat{v}_2^{\,1})$, $\hat{T}_2^{\,2} = (\mathcal{L}', \hat{v}_2^{\,2})$ and $T_2' = (\mathcal{L}, \hat{e}_2)$.

   These cases are easy to prove, using SC_ASYNC and DYN_EVAL_ASYNC_*.

   $\square$

Finally, to formulate the Behavioural Correspondence theorem we denote by $P = PC[e]$ a program P that is composed of a program context PC, *i.e.*, its list of interfaces and classes, and a main expression e.

Theorem 10 (Behavioural Correspondence). *Consider an original program P_{orig}, that is annotated according to the ATOMIS model such that* **AtomiSAnalysis**$(P_{orig}) = (P, Sol, P^+)$, *where there are program contexts PC and PC^+, and main expressions e and e^+, such that $P = PC[e]$ and $P^+ = PC^+[e^+]$. Then, for* **MAP** $= ($**methodMap**, **typeMap**$)$, *where* **methodMap** $=$ **fst** $\circ$ **oomn**$^{-1}$ *and* **typeMap** $=$ **snd** $\circ$ **ootype**$^{-1}$, *we have that $PC \leq_{\textbf{MAP}} PC^+$ and $e \simeq_{\textbf{MAP}}^{PC,PC^+} e^+$.*

Proof. By Lemma 9, we have that the relation

$$B = \{(T_1, T_2) | \vdash_{\textbf{MAP}}^{PC,PC^+} T_1 \propto T_2\}$$

is a $(\textbf{MAP}, PC, PC^+)$-bisimulation. We then show that $(e, e^+) \in B$. According to Stage 4 of the analysis, $P^+ = \mathsf{viP}(\mathsf{Sol}, P)$, with $e^+ = \mathsf{viE}(\mathsf{Sol}, \mathsf{mainV}, e)$. We show by induction on the definition of viE that for all $\mathsf{Sol}, \mathsf{mainV}, e$, we have that $\vdash_{\textbf{MAP}}^{PC,PC^+} e \propto \mathsf{viE}(\mathsf{Sol}, \mathsf{mainV}, e)$. $\square$

6 The Java Implementation

In this section we overview how the AtomiSAnalysis has been implemented in ATOMIS-Java. The ATOMIS-Java static analysis is built on top of the Eclipse Java Development Tools (JDT), but are independent of the Eclipse IDE, being easily incorporated in any Java project via the Gradle Build Tool. The compilation receives Java source code with ATOMIS annotations and generates Java bytecode with the @Atomic annotations forwarded, which will be important for the subsequent lock inference analysis that is performed over the Java bytecode.

To implement AtomiSAnalysis in Java we need to accommodate primitive types that do not exist in OOlong. Their impact on the atomicity inference rules is minimal. Atomic fields of any type have to be accessed in units of work to avoid data races and atomicity violations. However, fields of primitive types store values that are immutable and, hence, the *atomicity property* of such fields does not have to be propagated to parameters, return values or local variables. For convenience, we associate a new **do_not_care** atomicity value to primitive types, but these are filtered out when added to a constraint system.

To implement the solve function presented in §4.2.2, we resorted to the Z3 solver [13]. Both atomicity values and validity values are mapped to Booleans: ✳ and **valid** to **true**, and ✴ and **invalid** to **false**. For atomicity values, we favour the ✴ value to reduce the overhead of the concurrency control mechanisms in the execution of the generated code. In turn, for validity values, we favour the **valid** value to have as much valid variants as possible. Accordingly, we want to obtain a model that is optimal relatively to the number of valid variants and the number of variables with atomicity value ✴. For that purpose, we use Z3's optimising solver over a set of boolean formulas and added soft constraints for every validity and atomicity variable.

The challenge of solving a program's constraint system is dealing with scale, as the number and complexity of classes increases. When moving from theory to practice, we designed our approach to support separate compilation. Each class is therefore processed separately, producing a set of constraints that indicate the validity value computed for each variant of its non-private methods. The process features two main steps. The **first** step (*Locally Valid*) guarantees that only variants that are *locally valid* make their way to the class' constraint system. Remembering the definition of the vvMCS function in §4.2.2, we have that

$$\big(\textbf{valvar}(\mu) = \textbf{valid}\big) \rightarrow \Big(\big(\mathsf{Acs} \cup \big\{\check{x}_1 = \nu_1, \check{x}_2 = \nu_2, \check{x}_3 = \nu_3\big\}\big)@\mu \,\wedge \mathsf{bindCall}\big(\mathsf{Vns}, \mu\big)\Big)$$

We say that a variant is *locally valid* if and only if the left term of the conjunction has a solution. The existence of this solution depends only on the constraints collected from method's code, being completely independent of any method calls. Accordingly, all non *locally valid* variants are removed from the set of variants to consider from such point forward.

The **second** step (*BindCalls*) creates and solves the class' constraint system. However, to improve the compilation time, instead of building a single system and solving it all at once (like in §4.2.2), we build and solve this system incrementally, by making use of Z3's ability to organise constraints as a stack. We first generate the class' call graph and from there guarantee that, before being processed, every method only progresses if all its dependencies have already been processed, down to the nodes with out-degree 0. Naturally, precautions have to be made to deal with recursion.

The constraint system is initially populated with equations on the validity of the variants with out-degree 0 (computed on the *Locally Valid* step). From then onward, the incremental solving works as follows: we select a method (with out-degree > 1) from the call-graph and, having all its dependencies processed, we push the set of constraints for the validity of each of its variants onto the constraint system (stack).[12] We then solve the system and, from the resulting model (if any), retrieve the valid variants and the corresponding atomicity values for their local variables. Lastly, we pop the constraints previously pushed and complement the constraint system with validity of the variants freshly processed. Hence, at any point, the constraints system is composed of the constraints concerning the validity of the variants of the methods that have already been processed, plus the constraints for the validity of the variants of the method being processed. The algorithm stops when there are no more methods to process.

The result of the analysis of each class is cached on disk, in a user-configurable folder. Prior to the analyses of any class, a lookup in the cache is made. If information is present and its date is more recent than the class' file date, it is retrieved and no new analysis is performed. This means that the analysis' time only focuses on types that have been modified since the previous compilation.

7 Related Work

We review approaches to static analysis of concurrent object calculi/languages to prevent atomicity violations. We concentrate on those analysing source code and either requiring or inferring annotations/types to control the critical resources. Therefore, we briefly present two of the main

---

[12]The incremental solving is done at method level, meaning that we solve the system once per method. It could also be done at variant level, but the granularity is too small, yielding longer compilation times.

approaches: access permissions and behavioural types; we review briefly reference works on semantic correctness criteria for language encodings; the main part is dedicated to closely related work on data-centric concurrency control.

DCCC in shared memory programming. *Atomic Sets* (aka AJ) [15,51,52] is one of the reference works in the area. Variables holding values that share consistency properties have to be placed in an *atomic* set by prefixing the variable's declaration with `atomic(a)`, where `a` denotes the set's identifier. Sets may have multiple units of work associated, which can be explicitly augment to, for instance, account for multiple method parameters. Likewise, alias annotations enable the union of sets from distinct classes at object creation. Although a seminal work, Atomic Sets propose several annotations that hampers reasoning and is error-prone, as some annotations may be easily forgotten. Moreover, progress is not guaranteed. To avoid deadlocks, the programmer may have to intervene to explicitly define a partial order between sets when the static analysis is not able to infer it [35].

AWJS [30] combines Atomic Sets with work-stealing-based task parallelism in the Java language. Other works have addressed the automatic inference of atomic sets. AJ-lite [26] is a lighter version that assumes a single atomic set per Java class and is only applicable to libraries and not entire programs. The number of annotations is reduced to three, the ones needed to relate the class' atomic set to the ones of its fields. In [14], the authors propose to automatically infer atomic sets from patterns recognised in execution traces. Although being able to infer most of the required annotations, the quality of the result is sensitive to the quality of the input traces, and may generate more annotations than necessary.

Ceze *at al.* [9] associate variables sharing consistency proprieties to a colour, defining a consistency domain. However, concurrency control is not centralised on data declaration. Code annotations have to be added to the methods' implementations to signal when a thread concludes its work on a domain. Moreover, atomic-region-like control-centric concurrency control is needed to handle *high-level data races* [3] in composite operations. In [8], some of the same authors proposed hardware support for data-centric synchronisation.

RC^3 [43] proposes a DCCC model that uses a single keyword (**atomic**), being a source of inspiration for ATOMIS. Most of the concerns delegated on the programmer in AJ are shifted to static analyses. However, as mentioned in §1, the methods' implementations are atomicity-aware, requiring the duplication of code to account for the atomicities of the parameters. Also, as in all other DCCC solutions, there is no support for interfaces.

Concurrency control via access permissions and protocol compliance. A popular approach to provide static control of data-races is the use of explicit or implicit access permissions. The key ideas were adopted by the Rust programming language which ensures memory-safety via its type system. Sadiq *et al.* provide a comprehensive survey of the state-of-the-art [47]. The idea is to either declare, or have an implicit policy about, when resources are readable and/or writable. To write on a resource, one needs exclusive access; reads may be shared but forbid writes. Access permissions do not avoid the use of explicit concurrency control. They are a way of ensuring safe units of work. Our approach is "declarative" and aims to generate locks in a correct-by-construction manner.

Static verification of protocol compliance avoids atomicity violations by combining tight aliasing control and enforcing specific sequences of method calls. It uses behavioural types to declare object usage protocols and force any sequence of calls to be one of the allowed paths; it uses access permissions to control aliasing and guarantee protocol compliance and completion (see [2,27] for comprehensive surveys of the state-of-the-art; a book on tools is also available [22]). In object-oriented languages and frameworks supporting behavioural types, one needs to provide protocol specifications for each class and define contracts for methods to protect resources. The annotation burden is significant and might not be easy to get right, though the correctness guarantees go beyond those provided by our approach.

Type qualifier inference and Type-directed code synthesis. Generic frameworks and tools for type qualifier inference have been proposed, *e.g.* CQUAL [20, 21] and CLARITY [11] for C, and JQual [24] for Java, and their usage has been experimented for different purposes (see the later reference for a review). The ATOMIS model performs a field-based, context-sensitive, flow-insensitive, analysis, as does Greenfieldboyce and Foster's CS/FS JQual, but for specifically inferring atomicity type qualifiers to achieve strong concurrency control guarantees. Imposing program-wide atomicity constraints on how data is manipulated, in a flow-insensitive approach in the presence of alias appears to be incompatible with enabling subtyping between the atomic and non-atomic qualifiers. Indeed, the same object should not be treated simultaneously as atomic and non-atomic by different parts of the program. Automatic generation of method variants by the ATOMIS model provides the possible flexibility that is compatible with the methods' code, up to the conservative nature of the constraint generation.

Our code generation approach bears connections with Osera's [41] constraint-based type-directed program synthesis technique for synthesising polymorphic code from types and *examples*. In their work, a type inference system generates constraints between types; these constraints are solved, in order to infer types and ultimately synthesise code. The main differences are that [41] targets a functional programming language, and that ATOMIS's code generation is based on an original program.

Correctness criteria for language encodings. When performing code transformation, *behavioural soundness*, *i.e.*, the preservation of the original program's functional behaviour, is a key property. To define this criteria rigorously, a standard approach is to define a notion of mutual simulation between the source and the target programs, for proving that the transformation in question *preserves and reflects observable behaviour* (original and generated code simulate each other, being indistinguishable for an external observer) [39, 45]. Bisimulations are standard relations to capture indistinguishably of two systems' observable behaviour [38, 42] and are used extensively in process calculi to reason about language encodings and expressiveness [23].

8 Conclusions

We propose herein a sound new model of data-centric concurrency control, that only requires the annotation of interfaces and of atomic class fields. The approach has two phases — an analysis, which infers missing annotations and produces type-safe atomicity-related overloaded versions of each method; followed by a lock inference phase, to manage concurrency and prevent interferences. In this paper we rigorously define and prove correct the first phase for a foundational object calculus—OOlong—, showing that the generated code is type-safe and behaviourally corresponds to the original one. Moreover, we apply the algorithms developed for OOlong also to a Java implementation, to show its viability in real-life code.

We reached a milestone of our approach: we provide a computer-verified mathematical proof that the resulting generated program is type-safe. This is a fundamental prerequisite to assess the validity of the analysis technique and in turn to propose its use in mainstream languages. We achieved that by relying on a minimal, yet expressive Object Oriented language, and by mechanising the program analysis and code generation phases in the Coq proof assistant.

In future work, we plan to extend the setting to deal with generics. While the computation of units of work and lock inference has been implemented, we are working on its formalisation, and aim to prove its soundness. We also plan to evaluate the use of AtomiS in the simplification of writing concurrent code, as well as its impact in teaching concurrent programming. We are currently developing a mechanised proof of ATOMIS's behavioural soundness, as described in section 5.2.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, 2011. `doi:10.1145/1889997.1889999`.

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. `doi:10.1561/2500000031`.

[3] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In Pedro T. Isaías, Florence Sèdes, Juan Carlos Augusto, and Ulrich Ultes-Nitsche, editors, *New Technologies for Information Systems, Proceedings of the 3rd International Workshop on New Developments in Digital Libraries, NDDL 2003, and the 1st International Workshop on Validation and Verification of Software for Enterprise Information Systems, VVEIS 2003, In conjunction with ICEIS 2003, Angers, France, April 2003*, pages 82–93. ICEIS Press, 2003.

[4] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *J. Internet Serv. Appl.*, 8(1):4:1–4:15, 2017. `doi:10.1186/s13174-017-0055-2`.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, pages 211–230. ACM, 2002. `doi:10.1145/582419.582440`.

[6] Elias Castegren and Tobias Wrigstad. Oolong: A concurrent object calculus for extensibility and reuse. *SIGAPP Appl. Comput. Rev.*, 18(4):47–60, January 2019. `doi:10.1145/3307624.3307629`.

[7] Ilaria Castellani. Bisimulations and abstraction homomorphisms. *J. Comput. Syst. Sci.*, 34(2/3):210–235, 1987. `doi:10.1016/0022-0000(87)90025-0`.

[8] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 133–144. IEEE Computer Society, 2007. URL: `https://ieeexplore.ieee.org/xpl/conhome/4147635/proceeding`, `doi:10.1109/HPCA.2007.346192`.

[9] Luis Ceze, Christoph von Praun, Calin Cascaval, Pablo Montesinos, and Josep Torrellas. Concurrency control with data coloring. In Emery D. Berger and Brad Chen, editors, *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*, pages 6–10. ACM, 2008. `doi:10.1145/1353522.1353525`.

[10] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 304–315. ACM, 2008. `doi:10.1145/1375581.1375619`.

[11] Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *IN PROC. ESOP*, pages 264–278, 2006.

[12] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. REPT: reverse debugging of failures in deployed software. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 17–32. USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/weidong.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[14] Peter Dinges, Minas Charalambides, and Gul Agha. Automated inference of atomic sets for safe concurrent execution. In Stephen N. Freund and Corina S. Pasareanu, editors, *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '13, Seattle, WA, USA, June 20, 2013*, pages 1–8. ACM, 2013. URL: http://dl.acm.org/citation.cfm?id=2462029, doi:10.1145/2462029.2462030.

[15] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4:1–4:48, 2012. doi:10.1145/2160910.2160913.

[16] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 291–296. ACM, 2007. URL: http://dl.acm.org/citation.cfm?id=1190216, doi:10.1145/1190216.1190260.

[17] Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 237–252. ACM, 2003. doi:10.1145/945445.945468.

[18] Hervé Paulino et al. Atomis: Data-centric synchronization made practical. In *OOPSLA*. ACM, 2023. doi:10.1145/3622801.

[19] Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 256–267. ACM, 2004. doi:10.1145/964001.964023.

[20] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, pages 192–203. ACM, 1999. doi:10.1145/301618.301665.

[21] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006. doi:10.1145/1186635.

[22] Simon Gay and António Ravara. *Behavioural Types: from Theory to Tools*. River Publishers, 2017. doi:doi.org/10.13052/rp-9788793519817.

[23] Daniele Gorla and Uwe Nestmann. Full abstraction for expressiveness: history, myths and facts. *Math. Struct. Comput. Sci.*, 26(4):639–654, 2016. doi:10.1017/S0960129514000279.

[24] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for java. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 321–336. ACM, 2007. doi:10.1145/1297027.1297051.

[25] Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06)*. ACM, 2006.

[26] Wei Huang and Ana Milanova. Inferring aj types for concurrent libraries. 19th International Workshop on Foundations of Object-Oriented Languages, FOOL 2012, Tucson, AZ, USA; October 22, 2012, 2012.

[27] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

[28] C++ 11 standard. https://www.iso.org/standard/50372.html, 2011. Section 6.7.2.4 - Atomic type specifiers.

[29] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, 2009. doi:10.14778/1687627.1687657.

[30] Vivek Kumar, Julian Dolby, and Stephen M. Blackburn. Integrating asynchronous task parallelism and data-centric atomicity. In Walter Binder and Petr Tuma, editors, *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, pages 7:1–7:10. ACM, 2016. doi:10.1145/2972206.2972214.

[31] Nancy G. Leveson and Clark Savage Turner. Investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. doi:10.1109/MC.1993.274940.

[32] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 281–292. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

[33] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339. ACM, 2008. doi:10.1145/1346281.1346323.

[34] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 37–48. ACM, 2006. doi:10.1145/1168857.1168864.

[35] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. Detecting deadlock in programs with data-centric synchronization. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 322–331. IEEE

Computer Society, 2013. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6596173, doi:10.1109/ICSE.2013.6606578.

[36] Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: synchronization inference for atomic sections. In Morrisett and Jones [40], pages 346–358. URL: http://dl.acm.org/citation.cfm?id=1111037, doi:10.1145/1111037.1111068.

[37] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[38] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.

[39] John C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996.

[40] J. Gregory Morrisett and Simon L. Peyton Jones, editors. *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM, 2006. URL: http://dl.acm.org/citation.cfm?id=1111037.

[41] Peter-Michael Osera. Constraint-based type-directed program synthesis. In David Darais and Jeremy Gibbons, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019*, pages 64–76. ACM, 2019. doi:10.1145/3331554.3342608.

[42] David M. R. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981. doi:10.1007/BFb0017309.

[43] Hervé Paulino, Daniel Parreira, Nuno Delgado, António Ravara, and Ana Gualdina Almeida Matos. From atomic variables to data-centric concurrency control. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1806–1811. ACM, 2016. doi:10.1145/2851613.2851734.

[44] Kevin Poulsen. Tracking the blackout bug, 2004. http://www.securityfocus.com/news/8412.

[45] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[46] Jake Roemer, Kaan Genç, and Michael D. Bond. Smarttrack: efficient predictive race detection. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 747–762. ACM, 2020. doi:10.1145/3385412.3385993.

[47] Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. A survey on the use of access permission-based specifications for program verification. *J. Syst. Softw.*, 159, 2020. doi:10.1016/j.jss.2019.110450.

[48] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 181–194. ACM, 2008. doi:10.1145/1449764.1449779.

[49] Nir Shavit and Dan Touitou. Software transactional memory. In James H. Anderson, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213. ACM, 1995. doi:10.1145/224964.224987.

[50] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[51] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In Morrisett and Jones [40], pages 334–345. URL: http://dl.acm.org/citation.cfm?id=1111037, doi:10.1145/1111037.1111067.

[52] Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 304–328. Springer, 2010. doi:10.1007/978-3-642-14107-2_15.

[53] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 163–176. USENIX Association, 2010. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Xiong.pdf.

[54] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. Ghostcell: separating permissions from data in rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473597.

[55] Nosheen Zaza and Nathaniel Nystrom. Data-centric consistency policies: A programming model for distributed applications with tunable consistency. In *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016*, page 3. ACM, 2016. doi:10.1145/2957319.2957377.

[56] Tong Zhang, Changhee Jung, and Dongyoon Lee. Prorace: Practical data race detection for production use. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 149–162. ACM, 2017. doi:10.1145/3037697.3037708.

A   AtomiS Specialisation of the Classes of Listing 1

```
1  interface List_n<T> {
2   void add(T element);
3   T get(int pos);
4   @All Boolean equals(@All List_n<T> other);
5  }

7  interface List_a<T>  {
8   void add(@Atomic T element);
9   @Atomic T get(int pos);
10  @All Boolean equals(@All List_a<T> other);
11 }

13 class Node_n<T> {
14  T value;
15  Node_n<T> next, prev;
16 }

18 class Node_a<T> {
19  @Atomic T value;
20  Node_a<T> next, prev;
21 }

23 class BaseList_nn<T, N extends Node_n<T>>
       implements List_n<T> {
24  N head, tail;
25  Supplier<N> factory;

27  BaseList_nn(Supplier<N> factory) {...}
28  void add(T element) {...}
29  T get(int pos) {...}
30  Boolean equals(List_n<T> other) {...}
31 }
```

```
34 class BaseList_an<T, N extends Node_a<T>>
       implements List_a<T> {
35  N head, tail;
36  Supplier<N> factory;

38  BaseList_an(Supplier<N> factory) {...}
39  void add(T element) {...}
40  T get(int pos) {...}
41  Boolean equals(List_a<T> other) {...}
42 }

44 class BaseList_na<T, N extends Node_n<T>>
       implements List_n<T> {
45  @Atomic N head, tail;
46  Supplier<N> factory;

48  BaseList_na(Supplier<N> factory) {...}
49  void add(T element) {...}
50  T get(int pos) {...}
51  Boolean equals(List_n<T> other) {...}
52 }

54 class BaseList_aa<T, N extends Node_a<T>>
       implements List_a<T> {
55  @Atomic N head, tail;
56  Supplier<N> factory;

58  BaseList_aa(Supplier<N> factory) {...}
59  void add(T element) {...}
60  T get(int pos) {...}
61  Boolean equals(List_a<T> other) {...}
62 }
```

B   Auxiliary functions

$$\text{strip} : \text{AtomiS-OOlong Program} \rightarrow \text{OOlong Program}$$
$$\text{strip}(Ids\ Cds\ e) = \text{strip}(Ids)\ \text{strip}(Cds)\ \text{strip}(e)$$
$$\text{strip}(\textbf{interface } I\{IMsigs\}) = \textbf{interface } I\{\text{strip}(IMsigs)\}$$
$$\text{strip}(Msig) = Msig$$
$$\text{strip}(Msig\ [Sas]) = Msig$$
$$\text{strip}(\textbf{class } C \textbf{ implements } I\ \{Fds\ Mds\}) = \textbf{class } C \textbf{ implements } I\ \{\text{strip}(Fds)\ \text{strip}(Mds)\}$$
$$\text{strip}(f : \textbf{atomic } t) = f : t$$
$$\text{strip}(\textbf{def } Msig\{e\}) = \textbf{def } Msig\{\text{strip}(e)\})$$
$$\text{strip}(x.f{=}e) = x.f{=}\text{strip}(e)$$
$$\text{strip}(x.m(e)) = x.m(\text{strip}(e))$$
$$\text{strip}(\textbf{let } x{=}e_1 \textbf{ in } e_2) = \textbf{let } x{=}\text{strip}(e_1) \textbf{ in } \text{strip}(e_2)$$
$$\text{strip}(\textbf{new atomic } C) = \textbf{new } C$$
$$\text{strip}((t)\ e) = (t)\ \text{strip}(e)$$
$$\text{strip}(\textbf{finish}\{\textbf{async}\{e_1\}\ \textbf{async}\{e_2\}\};\ e_3) = \textbf{finish}\{\textbf{async}\{\text{strip}(e_1)\}\ \textbf{async}\{\text{strip}(e_2)\}\};\ \text{strip}(e_3)$$
$$\text{strip}(e) = e \quad \text{for all other cases}$$

dress : AᴛᴏᴍɪS-OOlong Program × OOlong Program → AᴛᴏᴍɪS-OOlong Program

$$\text{dress}(Ids\ Cds\ e, Ids'\ Cds'\ e') = \text{dress}(Ids, Ids')\ \text{dress}(Cds, Cds')\ \text{dress}(e, e')$$

$$\text{dress}(\textbf{interface}\ I\{IMsigs\}, \textbf{interface}\ I\{IMsigs'\}) = \textbf{interface}\ I\{\text{dress}(IMsigs, IMsigs')\}$$

$$\text{dress}(Msig\ Sas, Msig) = Msig\ Sas$$

$$\text{dress}(Msig\ , Msig) = Msig$$

$$\text{dress}(\textbf{class}\ C\ \textbf{implements}\ I\ \{Fds\ Mds\}, \textbf{class}\ C\ \textbf{implements}\ I\ \{Fds'\ Mds'\}) =$$
$$= \ \textbf{class}\ C\ \textbf{implements}\ I$$
$$\{\text{dress}(Fds, Fds')\ \text{dress}(Mds, Mds')\}$$

$$\text{dress}(\textbf{def}\ Msig\{e\}, \textbf{def}\ Msig\{e'\}) = \textbf{def}\ Msig\{\text{dress}(e, e')\}$$

$$\text{dress}(f : \textbf{atomic}\ t, f : t) = f : \textbf{atomic}\ t$$

$$\text{dress}(f : t, f : t) = f : t$$

$$\text{dress}(x.f = e, x.f = e') = x.f = \text{dress}(e, e')$$

$$\text{dress}(x.m(e), x.m(e')) = x.m(\text{dress}(e, e'))$$

$$\text{dress}(\textbf{let}\ x{=}e_1\ \textbf{in}\ e_2, \textbf{let}\ x : t{=}e_1'\ \textbf{in}\ e_2') = \textbf{let}\ x : t{=}\text{dress}(e_1, e_1')\ \textbf{in}\ \text{dress}(e_2, e_2')$$

$$\text{dress}(\textbf{new atomic}\ C, \textbf{new}\ C) = \textbf{new atomic}\ C$$

$$\text{dress}((t)e, (t)e') = (t)\text{dress}(e, e')$$

$$\text{dress}(\textbf{finish}\{\textbf{async}\{e_1\}\ \textbf{async}\{e_2\}\};\ e_3),$$
$$\textbf{finish}\{\textbf{async}\{e_1'\}\ \textbf{async}\{e_2'\}\};\ e_3') = \textbf{finish}\{\textbf{async}\{\text{dress}(e_1, e_1')\}\ \textbf{async}\{\text{dress}(e_2, e_2')\}\};$$
$$\text{dress}(e_3, e_3')$$

$$\text{dress}(e, e) = e \qquad \text{for all other cases}$$

$$Set@\mu = \bigwedge_{x \in Set} x@\mu$$
$$(\eta_1 \vee \eta_2)@\mu = (\eta_1@\mu \vee \eta_2@\mu)$$
$$(\eta_1 \wedge \eta_2)@\mu = (\eta_1@\mu \wedge \eta_2@\mu)$$
$$(\eta)@\mu = (\eta@\mu)$$
$$(\alpha_1 = \alpha_2)@\mu = (\alpha_1@\mu = \alpha_2@\mu)$$
$$\alpha@\mu = \alpha \ \text{if}\ \alpha \neq \check{x}$$
$$\check{x}@\mu = \check{x}@\mu(\in \textsf{Atomicity},\ \text{overloaded operation})$$

$$\textsf{ootype} : \textsf{AtomicityValue} \times \textsf{Type} \to \textsf{Type}$$
$$\textsf{ootype}(\nu, s) = \nu\_s$$
$$\textsf{oomn} : \textsf{MethodName} \times \textsf{AtomicityValue} \times \textsf{AtomicityValue} \to \textsf{MethodName}$$
$$\textsf{oomn}(m, \nu_1, \nu_2) = m\_\nu_1\_\nu_2$$
$$\textsf{oosig} : \textsf{AtomicityValue} \times \textsf{AtomicityValue} \times Msig \to Msig$$

$$\textsf{oosig}(\nu_1, \nu_2, m(x : s_1) : s_2) = \textsf{oomn}(m, \nu_1, \nu_2)\ (x : \textsf{ootype}(\nu_1, s_1)) : \textsf{ootype}(\nu_2, s_2)$$