A Formal Semantics of Nested Atomic Sections with Thread Escape[☆]

Frédéric Dabrowski^b, Frédéric Loulergue^{a,b}, Thomas Pinsard^b

^a*Inria πr^2 , PPS, Univ Paris Diderot, CNRS, Paris, France France*

^b*Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France*

Abstract

The multi-core trend is widening the gap between programming languages and hardware. Taking parallelism into account in the programs is necessary to improve performance. Unfortunately, current mainstream programming languages fail to provide suitable abstractions to do so. The most common pattern relies on the use of mutexes to ensure mutual exclusion between concurrent accesses to a shared memory. However, this model is error-prone and scales poorly by lack of modularity. Recent research proposes atomic sections as an alternative. The user simply delimits portions of code that should be free from interference. The responsibility for ensuring interference freedom is left either to the compiler or to the run-time system.

In order to provide enough modularity, it is necessary that both atomic sections could be nested and threads could be forked inside an atomic section. In this paper we focus on the semantics of programming languages providing these features. More precisely, without being tied to a specific programming language, we consider program traces satisfying some basic well-formedness conditions. Our main contribution is the precise definition of atomicity, well-synchronisation and the proof that the latter implies the strong form of the former. A formalisation of our results in the Coq proof assistant is described.

Keywords: Atomic sections, well-synchronisation, atomicity, program traces, formal semantics, proof assistant

1. Introduction

The multi-core trend in architectures development is widening the gap between programming languages and hardware. Improving performances now comes at the price of a deep software renewal because it cannot be done without taking parallelism on board. Unfortunately, current mainstream programming languages fail to provide suitable abstractions to do so. The most common pattern relies on the use of mutexes to ensure mutual exclusion between concurrent accesses to a shared memory. It is widely accepted that this model is error-prone and scales poorly by lack of modularity. In this context, the user is responsible for preserving some sequences of operations from interference. This is typically done by mapping the target (a collection of data) of such operations to a lock to be held when executing the sequence. Different mappings correspond to different choices of granularity, e.g. during a list update one can choose to protect the whole list or simply the updated item and its neighbourhood. A set of coarse-grained locks helps to keep the code simple, but in general leads to poor performances. On the opposite, fine grain locks lead to better performance but the complexity growth is inversely proportional. Despite an important effort of the community to help users in specifying such mappings, mainstream programming languages still do not offer support for doing so mainly

[☆]Thomas Pinsard was supported by a PhD grant from the French ministry of higher education and research. This work was partly supported by ANR (France) and JST (Japan) (project PaPDAS “Parallel Program Development with Algorithmic Skeletons” JST 10102704 and ANR-2010-INTB-0205-02).

Email addresses: Frederic.Dabrowski@univ-orleans.fr (Frédéric Dabrowski), Frederic.Loulergue@univ-orleans.fr (Frédéric Loulergue), Thomas.Pinsard@univ-orleans.fr (Thomas Pinsard)

URL: <http://frederic.loulergue.eu> (Frédéric Loulergue)

because current proposals fail in handling programs in which the mapping changes dynamically. Quoting O’Hearn et al. [1], ownership is in the eye of the assenter.

Recent research proposes atomic sections as an alternative. In this context, the user simply delimits portions of code that should be free from interference; the responsibility for ensuring interference freedom is left either to the compiler or to the run-time system. Proposals for implementing atomic sections fall in two categories, depending on the choice of an optimistic or pessimistic approach to concurrency. The former relies on transactions [2, 3], a well established mechanism in database management systems. Intuitively, in this approach, sections are executed optimistically, assuming no interference, but *cancelled* if any interference occurs. For a discussion on issues raised by the implementation of transactions in a programming language, readers are referred to [4, 5]. The latter relies on lock inference [6, 7], sections are executed pessimistically, enforcing mutual exclusion between sections by means of locks. With [8] we consider that transactions are a mean, an implementation technique, to ensure *atomicity*. The same remark holds for lock inference and the two approaches could even be combined. For example, lock inference could be used to help a transaction-based system to deal with I/O.

At first glance, and independently of the underlying implementation, atomic sections seem simpler to learn compared to more classical synchronisation primitives [9]. However, it is not yet clear whether they can be efficiently implemented [10] and whether they are really simpler, considering *formal* semantics and reasoning about programs. Some systems allow the nesting of transactions [11, 12] while others do not [2], thus leading to poor modularity. When nesting is possible one needs to define precisely the meaning of spawning threads within an atomic section. In [13], two primitives for thread creation are proposed: The first one delays the creation until the end of the section (if it is created inside), and the second one forces the thread to live entirely inside the transaction. In [6], nesting is allowed but the lock inference scheme prevents an atomic section inside another one to run before the enclosing section terminates when they access the same memory location. More importantly, one needs to define precisely the meaning of atomicity in this context.

In this paper we do not consider implementation issues but focus on the semantics of atomic sections. We consider a simple imperative language with fork/join parallelism and lexically scoped atomic sections. It supports the nesting of atomic sections, inner parallelism where threads are allowed to escape from surrounding sections, meaning that there is no synchronisation between the end of a section and the termination of threads started within this section. The semantics of the language is as permissive as possible and is not tied to any particular implementation. More precisely, we consider program traces satisfying some basic well-formedness conditions and, more importantly, satisfying the weak atomicity property, i.e there is no interference between *concurrent sections*. In this context, our contribution is the precise definition of atomicity, well-synchronisation and the proof that the latter implies the strong form of the former (up-to an equivalence relation over traces). A formalisation in Coq [14, 15] of our results is available.

We first motivate our work with some examples using parallelism and nested atomic sections (Section 2). We expose the context of this work with some related work (Section 3). We then describe the semantics domains of our work including the characterisation of well-formed traces (Section 4). We present our notion of well-synchronised traces in Section 5. Section 6 is devoted to our notion of atomicity and to the proof sketch for the main result. In Section 7 we discuss the design choices for the formalisation in the Coq proof assistant and provide a matching between the definitions and results stated in the paper and their counterparts in Coq. We conclude and give future research directions in Section 8.

2. Nested Sections and Threads

To motivate the need of parallelism inside atomic sections and nesting, let us see some examples of simple programs using such constructions. We need to consider an imperative language. It has the classical set of imperative instructions (loops, conditional, ...), primitives for creation and synchronisation of threads (*fork*, *join*), and the possibility to delimit syntactically portion of code to protect with atomic section (*atomic{ }*). We expect this language to produce traces verifying the properties defined in the rest of the paper. We already defined a similar language with its operational semantics in [16] and proved a such result.

Let us see some examples of programs where parallelism inside atomic section and nesting are useful. For example, consider a shared array list that provides a resize operation. During the resize operation, the list needs to be isolated from the other threads. So an atomic section surrounds the process. However, in order that resizing is not a performance bottleneck, the copy must be done in parallel, where each thread copies a part of the array. That is an example of parallelism in atomic section.

Now suppose one wants to copy two arrays into a third one. These arrays are shared. So there must be an atomic section to protect the target array. Threads read the source arrays and write in the destination one. To read the array, threads must create an atomic section. So this section is nested in the first one.

This following program does this operation. The 5-size arrays **a1** **a2** are merged into to the 10-size array **a** thanks to the method **merge**. This method creates an atomic section in order to protect the array **a**, and spawns two threads in charge of the copy, thanks to the method **copy**. This method copies the content of **b** inside **a**, starting at the index **k** of the array **a**. These threads don't interfere with each other because each thread writes in its own portion of array **a**. However, it reads a shared array, so this reading must be protected by an atomic section.

```
int SIZE :=5;
```

```
copy(int [] a,int [] b, int k){
  int i = 0; int v;
  while(i<SIZE){
    atomic{ v = b[i] }
    a[k*SIZE+i] = v;
    i = i+1
  }
}

merge(int[] a, int[] a1, int[] a2){
  atomic{
    t1:=fork(copy(a,a1,0));
    t2:=fork(copy(a,a2,1));
    join t1;
    join t2;
  }
}

main(){
  int [] a = allocate(2*SIZE);
  int [] a1 = allocate(SIZE);
  int [] a2 = allocate(SIZE);
  merge(a,a1,a2);
}
```

3. Related Work

Atomic sections have been the subject of many implementations, mainly using transactions, starting from only hardware based [17], to software based [2, 18, 19] or even a mixed of both [20]. However in general, they did not allow parallelism (and so nesting) in atomic sections. Several papers try to improve these lacks.

Harris [21] proposed to improve composability of atomic sections by adding to them new constructions, like the possibility of specifying another atomic section in case of failure of the first one. But there is still no parallelism or nesting in atomic sections.

The need for nesting was expressed by Moss et al., in [11], by pointing out the example of library use, but also by the fact that atomic sections are less monolithic, and thus lighter. However, inner parallelism is restricted to top level atomic sections (i.e. atomic sections that are not included in another atomic section). This restriction aims to ease the conflict detection and allows to optimise implementation.

The combination of nested parallelism and nested atomic sections was proposed by Agrawal et al. in [12]. They chose structural parallelism instead of *pthread* usually used, and thus they rely on a tree representation for the parallelism and atomic sections. Each inner atomic section must be terminated before the atomic section parent completes.

Some other works focus more on the proof of serialisability. In [22, 23], they wanted to prove atomicity of transactional memory (without nesting). They are not tied to a specific language to be abstract, and their lemmas and definitions are defined on execution traces. However, it is more technical than our work, because they work with transactions and not atomic sections. Their proofs have the same structure than ours, involving the notions of equivalence and serial traces.

These works focus deal with a specific implementation of atomic sections. The next ones look over the semantics of atomic sections.

Jagannathan et al. gave operational semantics of a derived version of Featherweight Java with nested and multi-thread atomic sections, independent of any implementation in [24]. They allow nesting but each

child atomic section must terminate before its parent. Their work on the proof of atomicity presents some similarities to ours. To prove the correctness of their work, they use program traces to abstract from the operational semantics, and shows the serialisability, i.e. for any abort-free program, there must be a corresponding trace where atomic sections are executed serially.

In [13], Moore et al. pointed out common problems found in several implementations: the precise meaning of atomicity and unnecessary limitations of parallel nesting. To avoid the problem related to the unclear model of atomicity (weak or strong), their advice is to give a semantics with proofs on atomicity. They illustrate it through four languages, and one of them allows parallel nesting with two primitives for thread spawning. The first one spawns threads which must live entirely inside the parent section, and so can only create inner-sections, and the other one delays the spawning at the end of the parent section, and so these threads can only create sub-sections that live outside. We acknowledge the same fact about the unnecessary limitations, and we go further by letting threads escape from atomic sections.

In [25], Abadi et al. wanted also to face the problem associated with weak and strong atomicity, with the development of transactions based on the automatic mutual exclusion model. In their representation, only one atomic section can be executed at a time. They show that existing implementations of transactions can lead to surprising behaviour due to the weak semantics. They present the semantics of their language, and with some restrictions they can give the behaviour of strong atomicity but with a permissive implementation.

4. Semantics Domain

We consider a kernel imperative language with dynamic creation of threads communicating through shared memory. Synchronisation of concurrent accesses to this memory is ensured by means of *atomic sections* which can be nested and support inner-parallelism. These two properties are necessary to support calls to external library functions inside atomic sections without restrictions. Indeed these functions can also use atomic sections or parallelism. Threads forked within a section can *escape* from its scope, i.e. they may continue their executions after the section terminates. We argue that this asynchronous behaviour of threads, with respect to surrounding atomic sections, benefits to programs modularity. This choice contrasts with that of [13] where threads must either complete before the section may terminate, or run after termination of the section (depending on the choice of the primitive used for spawning the thread). We think that this kind of constraint is not necessary to define atomicity.

We base our study on partial program traces (Section 4.1) and abstract from program syntax by stating some well-formedness properties (Section 4.2). In particular, we assume that *weak atomicity*, i.e. non interference of *concurrent atomic sections*, is ensured by the run-time system through some mutual exclusion mechanism. Because we consider inner-parallelism and thread-escape, the definitions of interference and concurrency between atomic sections require specific care. A program running in atomic section without interference, means that it does not receive information which depends on the execution of the section. One must define precisely which threads and atomic sections should be considered as part of an atomic section.

4.1. Traces

We assume disjoint countable sets of memory locations, thread names and section names, elements of which are respectively noted ℓ , t and p , possibly with subscript. The set of values, elements of which are noted v , possibly with subscripts, contains at least memory locations, integers and thread names.

Actions, events and traces. We define the set of actions, elements of which are noted a , possibly with subscripts, as follows.

$$a ::= \begin{array}{l} \tau \\ \text{alloc } \ell \ n \mid \text{free } \ell \mid \text{read } \ell \ n \ v \mid \text{write } \ell \ n \ v \\ \text{fork } t \mid \text{join } t \mid \text{open } p \mid \text{close } p \end{array}$$

Intuitively, τ denotes an internal, non observable, action. An action $\text{alloc } \ell \ n$ denotes heap allocation of a block of size n at memory location ℓ and an action $\text{free } \ell$ removes such a block from the heap. An action

read $\ell \ n \ v$ (resp. **write** $\ell \ n \ v$) denotes a read (resp. write) access from (resp. to) the offset n from location ℓ and v is the read (resp. written) value. Actions **fork** t and **join** t respectively denote creation and join on a thread t . Finally, **open** p and **close** p respectively denote section opening and closing. Note that section names are purely decorative and have no operational contents, this will be formalised in well-formedness conditions in Section 4.2. Their sole purpose is to name occurrences of atomic sections occurring in traces. An *event* e is a pair of a thread name and an action, a *trace* s is a sequence of events.

$$(\text{events}) \quad e ::= (t, a) \quad (\text{traces}) \quad s ::= \epsilon \mid s \cdot e$$

Notations. We note $s_1 s_2$ the concatenation of partial traces (or traces for short) s_1 and s_2 where by abuse of notation e stands for $\epsilon \cdot e$. For $i \in \mathbb{N}$, we define the partial function π_i by $\pi_{es}(0) = e$ and $\pi_{es}(n+1) = \pi_s(n)$. We respectively note $\pi_s^{act}(i)$ and $\pi_s^{tid}(i)$ the first and second projections over the event $\pi_s(i)$. We note $e \in s$, if there exists i such as $\pi_s(i) = e$, and by extension $a \in s$ if $\pi_s^{act}(i) = a$.

Definitions. To define precisely what should be considered as part of an atomic section, we introduce some auxiliary definitions. Given a trace s , the relations $owner_s$ and $father_s$ respectively relate a section to its owner thread and a thread to its father. The relation $range_s$ denotes the range of a section. By convention, we state that a section p ranges up to the last position of a trace s if p is pending in s . For a *well-formed* trace s , as defined in Section 4.2, the relations $owner_s$, $father_s$ and $range_s$ will define partial functions.

$$owner_s \ p \ t \triangleq (t, \text{open } p) \in s \quad father_s \ t \ t' \triangleq (t', \text{fork } t) \in s$$

$$\frac{\pi_s^{act}(i) = \text{open } p \quad \pi_s^{act}(j) = \text{close } p}{range_s \ p \ (i, j)}$$

$$\frac{\pi_s^{act}(i) = \text{open } p \quad \text{close } p \notin s}{range_s \ p \ (i, |s| - 1)}$$

It is now possible to define precisely which threads and atomic sections should be considered as part of a section. Given a section p of a trace s , the relation $tribe_s \ p$ is defined as the least set of thread identifiers containing the owner of the section and threads forked as a side effect of executing the section (relation $tribeChildren_s$).

$$\frac{range_s \ p \ (i, j) \quad i < k \leq j \quad owner_s \ p \ t' \quad \pi_s(k) = (t', \text{fork } t)}{tribeChildren_s \ p \ t}$$

$$\frac{tribeChildren_s \ p \ t' \quad father_s \ t \ t'}{tribeChildren_s \ p \ t}$$

$$\frac{owner_s \ p \ t}{tribe_s \ p \ t} \quad \frac{tribeChildren_s \ p \ t}{tribe_s \ p \ t}$$

Intuitively, if t belongs to $tribe_s \ p$ then the thread t is part of the computation of the atomic section p and thus should not be considered as an interfering thread. Our definition of atomicity in Section 6 will be based on this notion. In the same way, we define a relation over section names stating that an atomic section is part of the computation of another. We say that p' is a *subsection* of p if $p' \subseteq_s p$, as defined below, holds.

$$\frac{range_s \ p \ (i, j) \quad i < k \leq j \quad owner_s \ p \ t \quad \pi_s(k) = (t, \text{open } p')}{p' \subseteq_s p}$$

$$\frac{tribeChildren_s \ p \ t \quad owner_s \ p' \ t}{p' \subseteq_s p} \quad \frac{}{p \subseteq_s p}$$

Figure 1: Example of threads and sections

Finally, two atomic sections are said to be concurrent if $p \smile_s p'$, as defined in (1), holds.

$$p \smile_s p' \triangleq p \not\subseteq_s p' \wedge p' \not\subseteq_s p \quad (1)$$

Example. Let us see the previous definitions applied to the example on Figure 1. The white dots denote fork actions. The atomic sections are symbolised by rounded boxes. We suppose that “w” represents write actions on the same memory location. The other actions are labelled with a letter. Let s be the trace get from this example. The thread t_2 opens the section p_0 at the action e , and it closes it at the action i . We have $owner_s p_0 t_2$ and $father_s t_0 t_2$. The thread t_0 is created by t_2 while the section p_0 is active, so it is a child of the tribe of p_0 ($tribeChildren_s p_0 t_0$). The thread t_1 is also one of them because it is created by a child of the section. So the tribe of p_0 is made of t_2 (the owner), and t_0, t_1 (the children). We have $p_2 \subseteq_s p_0$ because p_2 is created inside p_0 by the thread owner of p_0 , and $p_1 \subseteq_s p_0$ because p_1 is created by a child of the tribe of p_0 . Note here that p_1 runs outside p_0 thanks to thread escape. We also have $p_0 \smile_s p_3$.

4.2. Well-formed traces

In this section we state formally some well-formedness conditions over program traces. Those conditions can be seen as a specification for possible implementations of our language. They range from common sense conditions to design choices.

To formalise these conditions we use the following definitions: The predicate see_s which can be seen as an over-approximation of the information flow in s , and is defined as the transitive closure of (2); the relation $\prec_s$ on section names defined at (3).

$$\frac{i < j \quad \pi_s^{tid}(i) = t \quad \pi_s^{tid}(j) = t}{see_s i j} \quad (2)$$

$$\frac{i < j \quad \pi_s^{act}(i) = \text{fork } t \quad \pi_s^{tid}(j) = t}{see_s i j}$$

$$\frac{i < j \quad \pi_s^{act}(i) = \text{write } \ell \ n \ v \quad \pi_s^{act}(j) = \text{read } \ell \ n \ v}{see_s i j}$$

$$p \prec_s p' \triangleq \exists i, j. i < j \wedge \pi_s^{act}(i) = \text{close } p \wedge \pi_s^{act}(j) = \text{open } p' \quad (3)$$

A trace s is *well-formed* if it satisfies the conditions in Figure 2 which are explained below:

- Condition **(wf₁)** ensures that section and thread names respectively identify dynamic sections and threads. Conditions **(wf₂)** and **(wf₃)** state simple properties of sections names. Each close action matches a previous open action which should be performed by the same thread. Moreover, each close action of a thread matches the last opened, but not yet closed, section opened by the same thread. As far as section names are concerned, those conditions impose no restrictions over the implementation as section names are purely decorative.

Every action **open** p , **close** p and **fork** t occurs at most once in s . (wf₁)

$\forall i, p. \pi_s^{act}(i) = \text{close } p \Rightarrow \exists j. j < i \wedge \pi_s^{act}(j) = \text{open } p \wedge \pi_s^{tid}(i) = \pi_s^{tid}(j)$ (wf₂)

$\forall p, i, j. \text{range}_s p(i, j) \Rightarrow \pi_s^{act}(j) = \text{close } p \Rightarrow$
 $\forall k, p'. i < k < j \Rightarrow \pi_s^{tid}(i) = \pi_s^{tid}(k) \Rightarrow \pi_s^{act}(k) = \text{open } p' \Rightarrow$ (wf₃)
 $\exists j', k < j' < j \wedge \pi_s^{act}(j') = \text{close } p'$

$\forall i, t. \pi_s^{act}(i) = \text{fork } t \Rightarrow \forall j. \pi_s^{tid}(j) = t \Rightarrow i < j$ (wf₄)

$\forall i, t. (\pi_s^{tid}(i) = t \vee \pi_s^{act}(i) = \text{fork } t) \Rightarrow \forall j. \pi_s^{act}(j) = \text{join } t \Rightarrow i < j$ (wf₅)

$\forall p, i, j, t. \text{range}_s p(i, j) \Rightarrow \text{owner}_s p \ t \Rightarrow \forall k. \pi_s^{act}(k) = \text{join } t \Rightarrow j < k$ (wf₆)

$\forall t, i, j. \pi_s^{act}(i) = \text{fork } t \Rightarrow \pi_s^{act}(j) = \text{join } t \Rightarrow \text{see}_s i \ j$ (wf₇)

$\forall p, p', \text{open } p \in s \Rightarrow \text{open } p' \in s \Rightarrow p \smile_s p' \Rightarrow p \prec_s p' \vee p' \prec_s p$ (wf₈)

Figure 2: Well-Formedness Conditions

- Conditions (wf₄) and (wf₅) state usual properties of fork/join instructions. The first one states that the actions done by a thread happen necessarily after the creation of the thread. The second one asserts that the creation of a thread and its actions are done before the synchronisation on this thread.
- Condition (wf₆) states that termination of a thread cannot be observed by another thread if the former has pending sections. An implementation can choose either to prevent termination of threads having pending sections or to force closing of such sections on termination. Condition (wf₇) states that it is not possible for a thread to join another thread without having explicitly received its name. These conditions ensure that external threads will not interfere with an atomic section by observing termination of inner threads. These conditions match the intuition that atomic section should appear as taking zero-time and thus termination of threads within a section should not be observable before the section is closed. External threads must not see the termination of inner threads to avoid interferences. Non-interference could also be achieved by forbidding the opposite. We choose to make the inside of a section invisible from the outside, rather than make the outside invisible from the section.
- Condition (wf₈) states that concurrent sections do not overlap.

A trivial result is that the subsection relation safely over-approximates overlapping of atomic sections in well-formed traces. Note that, in the absence of thread escape, the two notions would coincide.

Lemma 1. *For every well-formed trace s , section p ranging from i to j in s and k such that $i \leq k \leq j$ and $\pi_s^{act}(k) = \text{open } p'$, for some p' , we have $p' \in_s p$.*

Sketch of Proof By hypothesis and conditions (wf₁) and (wf₂) we have neither $p \not\prec_s p'$ nor $p' \not\prec_s p$. From (wf₈) it comes $p \in_s p'$ or $p' \in_s p$. Suppose that p and p' are opened by distinct threads t and t' (otherwise $i \leq k \leq j$ entails the result by definition of $\in_s$). We assume $p \in_s p'$ and show the contradiction. By definition of $\in_s$ it comes $\text{tribeChildren}_s p' \ t$. It is then immediate that $\pi_s^{act}(k) = \text{open } p'$ and $\pi_s^{tid}(i) = t$ imply $k < i$, thus contradicting the hypothesis.

$$\begin{array}{lcl}
\text{read } \ell \ n \ v & \bowtie & \text{write } \ell \ n \ v' \\
\text{write } \ell \ n \ v & \bowtie & \text{read } \ell \ n \ v' \\
\text{write } \ell \ n \ v & \bowtie & \text{write } \ell \ n \ v'
\end{array}$$

$$\begin{array}{c}
\frac{i < j \quad \pi_s^{tid}(i) = t \quad \pi_s^{tid}(j) = t}{sw_s \ i \ j} \quad (\mathbf{SW}_{\text{thread}}) \\
\\
\frac{i < j \quad \pi_s^{act}(i) = \mathbf{fork} \ (t) \quad \pi_s^{tid}(j) = t}{sw_s \ i \ j} \quad (\mathbf{SW}_{\text{fork}}) \\
\\
\frac{i < j \quad \pi_s^{tid}(i) = t \quad \pi_s^{act}(j) = \mathbf{join} \ (t)}{sw_s \ i \ j} \quad (\mathbf{SW}_{\text{join}}) \\
\\
\frac{i < j \quad \pi_s^{act}(i) = \mathbf{close} \ p \quad \pi_s^{act}(j) = \mathbf{open} \ p' \quad p \smile_s p'}{sw_s \ i \ j} \quad (\mathbf{SW}_{\text{section}})
\end{array}$$

Figure 3: Synchronisation

5. Well Synchronised Traces

An important aspect is to identify what kind of isolation is offered by atomic sections. Traditionally two kinds of atomicity can be distinguished [26]. In the weak form the atomic sections are protected only against other sections. It means that instructions outside sections can interfere with data accessed in atomic sections. In this case atomic sections only provide a weak form of protection. On the contrary, with strong atomicity, code inside an atomic section is totally protected both from code in other sections and code outside sections.

As stated in section 4, we rely on the run-time system to ensure some weak-atomicity property: with condition $(\mathbf{wf}_8)$ concurrent sections do not overlap. Traditionally, we define a notion of well-synchronisation which provides a sufficient condition for ensuring strong atomicity (as defined in section 6). To do so we define the notion of conflict over actions by the relation $\bowtie$ given in Figure 3 and state that a trace is *well-synchronised* if a synchronisation occurs between any two events involving conflicting actions. Synchronisation between two events is defined by sw which is the least predicate defined by the *transitive closure* of rules in Figure 3. There are four rules for sw , each requires that the two actions must be ordered. The rule $\mathbf{SW}_{\text{thread}}$ states that two events done by the same thread are synchronised. The rule $\mathbf{SW}_{\text{fork}}$ asserts that the creation of a thread is synchronised with the events done by this thread. The rule $\mathbf{SW}_{\text{join}}$ specifies the synchronisation between the actions of a thread and the action join done on this thread. Finally, the rule $\mathbf{SW}_{\text{section}}$ states the synchronisation between the closing and the opening of two concurrent sections.

Example. In the example of Figure 1, we can observe ordered synchronised events. (d, e) corresponds to two events done by the same thread. (f, a) is the fork of a thread and one of its actions. (i, j) corresponds to the closing and opening of two concurrent sections. The two events of (d, a) are synchronised by transitivity. The two write actions are in conflict, but they are both in concurrent sections, and so synchronised.

Intuitively, we consider a high-level programming language in which well-synchronisation should not be seen as an additional programming constraint. Indeed, in such languages the user is not expected to deal with non sequentially consistent executions and is responsible for writing data-race free programs. Moreover, thread names and memory locations are assumed to be values of opaque data-types. In particular, there must exist some communication (and thus some synchronisation in well-synchronised traces) between the allocation of a location (resp. the fork of a thread) and any access to that location (resp. join on that thread). Concerning threads names this is ensured by condition $(\mathbf{wf}_7)$. We do not impose an equivalent

Figure 4: Two examples of ill-formed trace

condition for memory locations here because it is useless to our purpose but in practice we will consider programs satisfying such a property.

Now we can define our property of well-synchronisation by requiring each conflicting action to be in synchronisation.

Definition 1. A trace s is well-synchronised if for any conflicting actions a and a' occurring respectively at position i and j in s such that $i < j$, we have $sw_s i j$.

The following lemma states that in well-synchronised traces information cannot flow without synchronisation.

Lemma 2. Let s be a well-synchronised trace. For all positions k and k' , $see_s k k' \Rightarrow sw_s k k'$.

Sketch of Proof Immediate by induction on the proof of $see_s k k'$. $\square$

Examples. We expect, in a trace both well-formed and well-synchronised, that a section is free from outside interference. This motivates well-formedness conditions (**wf₆**) and (**wf₇**). Without these conditions, traces of Figure 4 would be well-formed and well-synchronised traces but they have indeed a problem of interference. The black dots denote join actions, “w” and “r” respectively represent write and read operations on the same given memory location (and we assume this memory location is not used elsewhere inside section p). In both cases the red write operation interferes with the read action on this memory location inside section p . Without the join action done by thread t_0 , the two conflicting write operations would not be synchronised. Condition (**wf₇**) forbids the join in the first case, and condition (**wf₆**) forbids it in the second case.

We can now relate our notions of section and synchronisation to express a property on tribes. The following proposition states that in well-synchronised traces members of a tribe cannot synchronise with non-members while the section is active. An immediate corollary is that in well-synchronised traces actions of members of a section cannot conflict with actions of non members while the section is active.

Proposition 1. For every well-formed and well-synchronised trace s , section p ranging from i to j in s , and k, k' such that $i \leq k, k' \leq j$ and $sw_s k k'$ we have $tribe_s p t \Rightarrow tribe_s p t'$ where $\pi_s^{tid}(k) = t$ and $\pi_s^{tid}(k') = t'$.

Sketch of Proof Let p be a section ranging from i to j in s and let k' be such that $i \leq k' \leq j$ and $\pi_s^{tid}(k') = t'$ for some t' . We consider the property:

$$\begin{aligned} P(k_0) \equiv \quad & \forall k, t'. \quad i \leq k_0 \leq j \wedge i \leq k \leq j \wedge sw_s k k_0 \\ & \wedge \pi_s^{tid}(k) = t \wedge \pi_s^{tid}(k_0) = t' \wedge \\ & tribe_s p t \Rightarrow tribe_s p t' \end{aligned}$$

Now suppose that $P(k_0)$ holds for all $k_0 < k'$. We prove that $P(k')$ holds by induction on the derivation of $sw_s k k'$.

- If $\pi_s^{tid}(k) = \pi_s(k')$ or $\pi_s^{act}(k) = \mathbf{fork}(t')$ the result is immediate by definition of *tribe*.
- Suppose that $\pi_s^{act}(k') = \mathbf{join}(t)$. By **(wf₆)** we have $tribeChildren_s p t$ and then, by definition of *tribe*, we can find some k_0 and t_0 such that $i < k_0 < k'$, $tribe_s p t_0$ and $\pi_s^{act}(k_0) = \mathbf{fork}(t)$. By **(wf₇)** it comes $see_s k_0 k'$. By definition of *see*, it is easy to check that $\pi_s^{act}(k') = \mathbf{join}(t)$ implies that there exists some $k_1 < k'$ such that either $\pi_s^{tid}(k_1) = t'$ or $\pi_s^{act}(k_1) = \mathbf{fork}(t')$ and either $k_0 = k_1$ or $see_s k_0 k_1$. If $k_0 = k_1$ then the result is immediate. Otherwise, $P(k_1)$ holds by induction hypothesis and by Lemma 2 we have $sw_s k_0 k_1$. By definition of *tribe* it comes $tribe_s p t'$.
- Suppose that $\pi_s^{act}(k) = \mathbf{close}(p')$ and $\pi_s^{act}(k') = \mathbf{open}(p'')$ for some p' and p'' . The result is immediate by Lemma 1.
- The induction step is immediate by applying twice the induction hypothesis. $\square$

6. Atomicity

In this section, we prove that well-formed and well-synchronised traces satisfy the strong atomicity property. More formally, we prove that well-synchronised traces are *serialisable*, i.e. any such trace is *equivalent* to a *serial* trace. A serial trace [27], is traditionally defined as a trace obtained by a program where atomic sections are executed serially i.e. without interleaving. This definition suits well without inner-parallelism. We can delay the execution of a forked thread inside the section, to let only the thread owner of the section run and use the classical definition of serialisability. We think this is an unnecessary limitation of parallelism. An atomic section provides an isolation for a piece of program, and the threads created during the execution are a part of this program, and so they must be “allowed” to run. That is why we have to define our own notion of serialisability. To define the notion of serial trace, we must take care that sections support nesting, inner-parallelism and thread-escape.

This is exactly the purpose of the notion of tribe which captures the set of threads that should be allowed to run while a section is active.

Definition 2. A trace s is serial if for every section p and for all positions i, j and k such that $range_s p (i, j)$ and $i \leq k \leq j$ we have $tribe_s p t$ where $\pi_s^{tid}(k) = t$.

Definition 3. A trace s is serialisable if there exists a serial trace s' that is equivalent to s .

We now need to define formally what it means for two traces to be equivalent. Usually, traces are defined to be equivalent up to re-orderings preserving the order of conflicting actions, for a broader definition of conflicting actions than the one we use. For technical purpose, we consider a stronger definition of equivalence.

Definition 4. Two traces s and s' are equivalent, noted $s \equiv s'$, if there exists a bijection γ between positions of s and s' such that

$$\pi_s(i) = \pi_{s'}(\gamma(i)) \quad \text{for all } i < |s| \quad (\mathbf{e}_1)$$

$$sw_s i j \Leftrightarrow sw_{s'} \gamma(i) \gamma(j) \quad \text{for all } i, j < |s| \quad (\mathbf{e}_2)$$

Proposition 2. For every well-formed and well-synchronised trace s , and for every trace s' , if $s \equiv s'$ then s' is well-formed and well-synchronised.

Sketch of Proof The proof of well-synchronisation is done by examining the order between positions in the two traces and exploiting the properties of the bijection to show that these positions are still in synchronisation. The proof of well-formedness is done for each of the eight conditions, using properties of the bijection. $\square$

Theorem 1. Every well-formed and well-synchronised trace is serialisable.

Sketch of Proof The proof is by structural induction on s . The result is obvious for the empty trace. Suppose $s \cdot (t, a)$ is a well-formed and well-synchronised trace. Then s is both well-formed and well-synchronised because these properties are prefix-closed. By induction hypothesis, there exists s' such that $s' \equiv s$ and s' is serial.

Then we prove that $s' \cdot (t, a) \equiv s \cdot (t, a)$. The bijection used (noted γ) is the same as in $s \equiv s'$ for all positions $< |s|$ and $\gamma(|s|) = |s'|$. The proof of condition **(e₁)** of equivalence is trivial. The trickiest part of condition **(e₂)** is when we have $sw_{s \cdot (t, a)} i |s|$ so we have to prove $sw_{s' \cdot (t, a)} \gamma(i) \gamma(|s|)$. We know that $\pi_s(i) = \pi_{s'}(\gamma(i))$ and $\pi_{s \cdot (t, a)}(|s|) = \pi_{s' \cdot (t, a)}(\gamma(|s|)) = (t, a)$. By an induction on $sw_{s \cdot (t, a)} i |s|$ we can conclude.

By Proposition 2, $s' \cdot (t, a)$ is well-formed and well-synchronised.

Now we show that $s' \cdot (t, a)$ is serialisable.

We note $excludes_{s, p} t$ when p is pending in s and $\neg tribe_s p t$. Assume p is the left-most section such that $excludes_{s', p} t$.

- Suppose that no such section exists. Let the positions i, j, k , the thread name t' and the section name p' be such that $range_{s' \cdot (t, a)} p' (i, j)$, $\pi_{s' \cdot (t, a)}^{tid}(k) = t'$ and $i \leq k \leq j$. We want to prove that $tribe_{s' \cdot (t, a)} p' t'$. If $k < |s'|$ the result is immediate by the serialisability of s' . We suppose now that $k = |s'|$ (and then $t = t'$). If $\pi_{s' \cdot (t, a)}^{act}(k) = \text{open } p'$ then $k = i$ by **(wf₁)**, and then as the owner of the section p' , t' is in the tribe of p' . If $\pi_{s' \cdot (t, a)}^{act}(k) \neq \text{open } p'$ then necessarily p' is pending in s' . By assumption we have thus necessarily that $tribe_{s'} p' t'$ holds. As $tribe$ is preserved by trace concatenation, $tribe_{s' \cdot (t, a)} p' t'$: The trace $s' \cdot (t, a)$ is serial.
- If section p exists, let i be such that $\pi_{s'}^{act}(i) = \text{open } p$. We will reason using insertion which is useful to represent the shifting of an element in a trace. Given a non-empty trace $s \cdot e$, we note $s \upharpoonright_{i_0} e$ the trace obtained by inserting e in s at position i_0 . A bijection γ is induced by the insertion:

$$\begin{aligned} \gamma(k) &= k \text{ if } k < i & \gamma(k) &= k + 1 \text{ if } i \leq k < |s| \\ \gamma(|s|) &= i_0 \end{aligned}$$

Note γ keeps the relative order of positions lower than $|s|$, and its inverse for positions different from i_0 . There are two important properties of insertion we use:

For every well-formed and well-synchronised trace $s \cdot (t, a)$,

for every section name p and position i such that

- $excludes_{s, p} t$ and $\pi_s^{act}(i) = \text{open } p$ **(ins_{eq})**
- $\forall k. i \leq k < |s| \Rightarrow \neg sw_{s \cdot (t, a)} k |s|$

we have $s \cdot (t, a) \equiv s \upharpoonright_i (t, a)$

For every well-formed trace $s \cdot (t_0, a)$ and position i such that,

- there exists a section name p_0 such that $excludes_{s, p_0} t_0$
- $\forall k. i \leq k < |s| \Rightarrow \neg sw_{s \cdot (t_0, a)} k |s|$ **(ins_{tribe})**

then for all p and t , $tribe_{s \cdot (t_0, a)} p t \Rightarrow tribe_{s \upharpoonright_i (t_0, a)} p t$

Let us consider the trace $s'' = s' \upharpoonright_i (t, a)$. We now prove that s'' is equivalent to $s \cdot (t, a)$ and serial. To do so we need to use the two properties of insertion, so first we prove: $\forall k. i \leq k < |s'| \Rightarrow \neg sw_{s' \cdot (t, a)} k |s'|$. Suppose $sw_{s' \cdot (t, a)} k |s'|$ for a k such that $i \leq k < |s'|$. By serialisability of s' we know that $tribe_{s'} p t_1$, where $\pi_s^{tid}(k) = t_1$, and $tribe_{s' \cdot (t, a)} p t_1$ by preservation of $tribe$ by trace concatenation. By Proposition 1, we can conclude that $tribe_{s' \cdot (t, a)} p t$. Because $s' \cdot (t, a)$ is well-formed we know that $a \neq \text{open } p$ and $a \neq \text{fork } t$. By definition of $tribe$ we obtain that $tribe_{s'} p t$ that contradicts assumption $\neg tribe_{s'} p t$.

By **(ins_{eq})** and transitivity, we conclude $s'' \equiv s \cdot (t, a)$.

For proving that s'' is serial, let i', j', k be positions, p' a section name, t' a thread such that $\text{range}_{s''} p' (i', j')$, $i' \leq k \leq j'$ and $\pi_{s'}^{tid}(k) = t'$. To prove $\text{tribe}_{s''} p' t'$ we first prove $\text{tribe}_{s'} p' t'$ by transposing the $\text{range}_{s''} p' (i', j')$ on s' and use the serialisability of s' to conclude. To do so prove that the relative positions of i', k and j' are preserved by the inverse of the induced bijection γ , i.e. consider whether each position is equal to i or not: First prove that $i' \neq i$ and conclude for the four remaining cases using various well-formedness conditions. So $\text{tribe}_{s'} p' t'$ holds.

As tribe is preserved by trace concatenation, we have $\text{tribe}_{s'.(t,a)} p' t'$ and by $(\text{ins}_{\text{tribe}})$, we conclude $\text{tribe}_{s''} p' t'$. Therefore s'' is serial. $\square$

7. Formalisation in Coq

Pen and paper proofs for complex semantics are error-prone. This is thus safer to use a proof assistant (also called interactive prover) to formalise and prove the properties of such semantics. In this work we used Coq [14]. As its core it is based on the Curry-Howard correspondence that relates terms of a typed λ -calculus with proof trees of a logical system in natural deduction form. A type could be the statement of a property and a program could be a proof of a property. The calculus behind Coq is the calculus of (co)-inductive constructions, an extension of the initial Calculus of Constructions [28].

From a more practical side, Coq can be seen as a functional programming language, close to OCaml or Haskell but with a richer type system that allows to express logical properties. As a matter of fact, Coq is often used as follows: programs are developed in Coq and their properties are also proved in Coq. But it is also used to model semantics of programming languages and to prove their properties. Coq has also been used to prove mathematical theorems, for example the four-color theorem [29].

Writing proofs with an interactive theorem prover is longer because no routine or folklore results can be omitted in the proofs. However, confidence in the results is greater as there are no omissions and the proofs are checked by the prover. In the case of Coq, the only component of the system to be trusted is the type checker: a proof being a program and a logical statement being a type, checking that a proof is indeed a correct proof is type checking it against the statement/type. Coq being a rich language, there are always many ways to formalise a concept. In the development we try to find a balance between elegance of the modelling and simplicity of its use for further steps including proofs.

In this section we explain the choices we made when modelling the trace semantics and proving its properties in Coq. We give all the elements needed to read the statement of the main theorem. Then we explain the proof of this main theorem, using a set of tactics, designed to make the proof understandable even without running the proof script. Finally we also briefly explain how the development is organised and relate the Coq material with the definitions and results of the previous sections.

To understand this section, the reader must have basic notions of Coq. We refer to [30, 31] for introductions to Coq and [15, 32] for reference books.

7.1. Design Choices

Section 4 begins to define memory locations, section names, thread identifiers and values. The three first elements are defined as disjoint countable sets. To model that in Coq we define the following module type :

Module Type DecidableInfiniteSet.

Parameter $t : \text{Set}$.

Axiom eq_dec : $\forall (x y : t), \{ x = y \} + \{ x <> y \}$.

Axiom infinite: infinite t .

End DecidableInfiniteSet.

where **Definition** infinite ($A : \text{Type}$) : **Type** := $\{ \text{bij} : (A \rightarrow \text{nat}) * (\text{nat} \rightarrow A) \mid \forall (n : \text{nat}), (\text{fst } \text{bij})((\text{snd } \text{bij}) n) = n \}$.

In the remaining of the formalisation we assume having modules Ad (for memory addresses), TId (for thread identifiers) and SN (for section names) whose type is DecidableInfiniteSet. The property eq_dec is necessary to be able to have reasonings of the form: either these two values are equal or they are different. Indeed Coq is based on a constructive logic and we cannot reason using excluded middle. Instead of decidable

equality we could have added the excluded middle axiom. This axiom could be added safely to Coq's logic. However as certain combinations of axioms make the logic incoherent, using an axiom that is not absolutely necessary hinders the modular nature of a development. That is why we prefer to prove that equality is decidable on the data structures we manipulate, or that a property is decidable when we need to reason by case on whether an instance of the property holds or not.

The infinite property is necessary to generate fresh names (for threads and sections), and un-allocated memory addresses using the following lemma:

Lemma `infiniteNotFinite`: $\forall A, \text{infinite } A \rightarrow \forall (l:\text{list } A), \exists (a:A), \text{not}(\text{In } a \text{ } l)$.

In the previous part, we assumed that values contain at least memory locations, numbers (data), and thread identifiers. In order to formalise that in Coq, we first define types (of the object language) that are associated to the values (of the object language), and we add a type for Boolean values. We relate these types of the object language to types of Coq by a function `typeOf` that allows us to define a type for representing the values of our object language. All these definitions are given in several modules, for the sake of simplicity, we assume here we have only one module:

Module `Va`.

Inductive `_type_` := | `Location` | `Number` | `Thread` | `Boolean`.

Definition `typeOf` (`t:_type_`) : `Set` :=

match `t` **with**
 | `Thread` $\Rightarrow$ `TId.t`
 | `Number` $\Rightarrow$ `Z`
 | `Boolean` $\Rightarrow$ `bool`
 | `Location` $\Rightarrow$ `Ad.t`
end.

Inductive `t` := `Value` : $\forall (ty:_type_)(v:\text{typeOf } ty), t$.

End `Va`.

The next step is to defined actions, events and traces. The action type is an inductive type, where each constructor represents an action. It uses the address `Ad.t`, thread identifier `TId.t`, section name `SN.t`, and value `Va.t` types:

Inductive <code>action</code> : <code>Type</code> :=	<code>Write</code> : <code>Ad.t</code> $\rightarrow$ <code>nat</code> $\rightarrow$ <code>Va.t</code> $\rightarrow$ <code>action</code>
<code>Silent</code> : <code>action</code>	<code>Fork</code> : <code>TId.t</code> $\rightarrow$ <code>action</code>
<code>Allocate</code> : <code>Ad.t</code> $\rightarrow$ <code>nat</code> $\rightarrow$ <code>action</code>	<code>Join</code> : <code>TId.t</code> $\rightarrow$ <code>action</code>
<code>Free</code> : <code>Ad.t</code> $\rightarrow$ <code>action</code>	<code>Open</code> : <code>SN.t</code> $\rightarrow$ <code>action</code>
<code>Read</code> : <code>Ad.t</code> $\rightarrow$ <code>nat</code> $\rightarrow$ <code>Va.t</code> $\rightarrow$ <code>action</code>	<code>Close</code> : <code>SN.t</code> $\rightarrow$ <code>action</code> .

From that, the type of events is defined as : **Definition** `t` := `TId.t` * `action` in a module `Event`. The type for traces is then **Definition** `tr` := `list Event.t`. We must define an additional induction principle because the automatically generated principle is the one that matches the constructors: New elements are added to the beginning of the list. In the `tr_ind` induction principle, the inductive case considers that new elements are added at the end of the list.

In order to define the notion of well-formedness and well-synchronisation, it is first necessary to define various predicates on the data-types just defined.

Ownership and fatherhood are defined as:

Inductive `owns` (`s` : `Tr`) : `SN.t` $\rightarrow$ `TId.t` $\rightarrow$ `Prop` :=

| `owns_cons` :
 $\forall t \text{ } p,$
`threadId_of` (`pi i s`) == `t` $\rightarrow$
`action_of` (`pi i s`) == `Open p` $\rightarrow$
`owns s p t`.

Definition `father` (`s` : `Tr`) (`t` : `TId.t`) : `TId.t` $\rightarrow$ `Prop` :=

fun `t'` $\Rightarrow$
 $\exists i,$
`threadId_of` (`pi i s`) == `t'` $\wedge$
`action_of` (`pi i s`) == (`Fork t`).

With respect to the definitions given page 5 we choose to state the membership of an event to a trace in the form $\exists i, \pi_s(i) = e$ rather than $e \in s$. And actually we use $\pi_s^{tid}(i) = t$ and $\pi_s^{act}(i) = a$ rather

than $\pi_s(i) = (t, a)$. The Coq counterparts of $\pi_s^{tid}(i)$ and $\pi_s^{act}(i)$ are respectively `threadId_of (pi i s)` and `action_of (pi i s)`. It was possible to use the `In` predicate of Coq to have definitions similar to the ones given in the first part of the paper. However, this would imply additional steps in the proofs to obtain statements such as $\pi_s^{act}(i) = a$. There is therefore a trade-off between conciseness of predicates definitions and ease of use in proofs. A last detail on the functions used in these definitions: they are partial functions, because `i` may be an index out of the bounds of the list. Thus they return values of type `option` defined as:

Inductive `option (A : Type) : Type := Some : A → option A | None : option A`

and the notation `x == y` stands for `x = Some y`.

The definition of `range`, `tribeChildren` and `tribe` are very close to their counterparts of in Section 4. Most of the time, inferences systems are modelled in Coq as inductive definitions:

Inductive `range (s : Tr) (p : SN.t) : nat → nat → Prop :=`
`| range_closed : ∀ i (Hi : action_of (pi i s) == Open p) j (Hj : action_of (pi j s) == (Close p)), range s p i j`
`| range_opened : ∀ i (Hi : action_of (pi i s) == (Open p)) (HNotClosed : ¬occursIn s (Close p)), range s p i (length s - 1).`

Inductive `tribeChildren (s : Tr) (p : SN.t) : nat → Prop :=`
`| tribeChildren_direct :`
`∀ i j t, range s p i j → owns s p t →`
`∀ k t', i < k ≤ j → threadId_of (pi k s) == t → action_of (pi k s) == Fork t' → tribeChildren s p t'`
`| tribeChildren_indirect : ∀ t t', tribeChildren s p t → father s t' t → tribeChildren s p t'.`

Definition `tribe (s : Tr) (p : SN.t) (t : TId.t) : Prop := owns s p t ∨ tribeChildren s p t.`

Note the use of `occursIn s a` which simply states that there exists an event `e` containing `a` in the trace `s`.

Well-formedness is a predicate on traces defined as:

Inductive `wellFormed (s : Tr) :=`
`wellFormed_cons :`
`∀ (WF1 : wf_occurrences s) (WF2 : wf_open_close s) (WF3 : wf_seq_order s) (WF4 : wf_fork s)`
`(WF5 : wf_join s) (WF6 : wf_join_all_closed s) (WF7 : wf_join_see_fork s) (WF8 : wf_mutualExclusion s),`
`wellFormed s.`

It is basically a conjunction of the basic properties needed to insure well-formedness, but put in an inductive definition to ease its use in proofs. We do not detail all the properties but the first that is given as an English sentence in the first part, and `wf_fork (wf4)` to show that the Coq versions are very close to the mathematical definitions of page 7:

<p>Inductive <code>singleAction : action → Prop :=</code> <code> singleAction_fork : ∀ t, singleAction (Fork t)</code> <code> singleAction_open : ∀ p, singleAction (Open p)</code> <code> singleAction_close : ∀ p, singleAction (Close p).</code></p>	<p>Definition <code>occursAtMostOnce (s : Tr) (a : action) :=</code> <code> ∀ i j, action_of (pi i s) == a →</code> <code> action_of (pi j s) == a →</code> <code> i = j.</code></p>
--	---

Definition `wf_occurrences (s : Tr) := ∀ a, singleAction a → occursAtMostOnce s a.`

Definition `wf_fork (s : Tr) := ∀ i t, action_of(pi i s) == Fork t → ∀ j, threadId_of(pi j s) == t → i < j.`

The definition of well synchronised traces follows closely the definitions of the first part, $\bowtie$ is modelled by `conflict`, the rules of figure 3 by `sw`, `sw` (defined as the transitive closure of the relation defined by the rules) by `synchronizeWith`, and Definition 1 by `wellSynchronized`:

Inductive `conflict : action → action → Prop :=`
`| conflict_rw : ∀ l n v v', conflict (Read l n v) (Write l n v')`
`| conflict_wr : ∀ l n v v', conflict (Write l n v) (Read l n v')`
`| conflict_ww : ∀ l n v v', conflict (Write l n v) (Write l n v').`

Inductive `sw : Tr → nat → nat → Prop :=`

```

| swSeq :  $\forall s \ i \ j \ t, i < j \rightarrow \text{threadId\_of}(\pi i \ s) == t \rightarrow \text{threadId\_of}(\pi j \ s) == t \rightarrow \text{sw } s \ i \ j$ 
| swFork :  $\forall s \ i \ j \ t, i < j \rightarrow \text{action\_of}(\pi i \ s) == \text{Fork } t \rightarrow \text{threadId\_of}(\pi j \ s) == t \rightarrow \text{sw } s \ i \ j$ 
| swJoin :  $\forall s \ i \ j \ t, i < j \rightarrow \text{threadId\_of}(\pi i \ s) == t \rightarrow \text{action\_of}(\pi j \ s) == \text{Join } t \rightarrow \text{sw } s \ i \ j$ 
| swAtomic :  $\forall s \ i \ j \ p \ p', i < j \rightarrow \text{concurrent } s \ p \ p' \rightarrow \text{action\_of}(\pi i \ s) == \text{Close } p \rightarrow$ 
                $\text{action\_of}(\pi j \ s) == \text{Open } p' \rightarrow \text{sw } s \ i \ j.$ 

```

Definition synchronizeWith $(s : \text{Tr}) := \text{clos_trans } _ (\text{sw } s).$

Definition wellSynchronized $(s : \text{Tr}) :=$

$\forall i \ j \ a \ a', i < j \rightarrow \text{action_of}(\pi i \ s) == a \rightarrow \text{action_of}(\pi j \ s) == a' \rightarrow \text{conflict } a \ a' \rightarrow \text{synchronizeWith } s \ i \ j.$

Finally, we need the definition of equivalence between two traces and the definition of atomicity for a trace. The latter is simple and is very close to Definition 2:

Definition atomic $(s : \text{Tr}) : \text{Prop} :=$

$\forall p \ i \ j, \text{range } s \ p \ i \ j \rightarrow \forall k, i \leq k \leq j \rightarrow \forall t, \text{threadId_of}(\pi k \ s) == t \rightarrow \text{tribe } s \ p \ t.$

Two traces are considered equivalent if there is a compatibility relation between them, i.e. a bijective relation which has the properties stated in Definition 4. We also use a notation in the Coq development:

Definition compatible_by $R \ (s1 \ s2 : \text{Tr}) :=$

$\text{bijective}(\text{fun } k \Rightarrow k < \text{length } s1) (\text{fun } k \Rightarrow k < \text{length } s2) \ R$
 $\wedge (\forall i \ j, R \ i \ j \rightarrow \pi i \ s1 = \pi j \ s2)$
 $\wedge (\forall i \ j \ i' \ j', R \ i \ j \rightarrow R \ i' \ j' \rightarrow (\text{synchronizeWith } s1 \ i \ i' \leftrightarrow \text{synchronizeWith } s2 \ j \ j')).$

Definition compatible $(s1 \ s2 : \text{Tr}) := \exists R, \text{compatible_by } R \ s1 \ s2.$

Infix "≡" := compatible (at level 70) : trace_scope.

We now have all the elements needed to state the main theorem:

Theorem wsync_atomic :

$\forall s,$
 $\text{wellFormed } s \rightarrow \text{wellSynchronized } s \rightarrow$
 $\exists s', s \equiv s' \wedge \text{atomic } s'.$

7.2. Proof of the main theorem

In interactive proving systems, there are roughly two kinds of style for writing proofs: using a procedural language to write proof scripts, where tactics are used to transform the proof state (the goals, i.e. the statements remaining to be proved, their hypotheses, and the partially built proof terms), or using a declarative language that is very close to usual mathematical texts. It is easier to write proof scripts in the procedural style, and user specific tactics (usually the procedural language allows to write procedures that define new tactics) are very useful for automation. However proof scripts in this style are difficult to read, even if there are good practices that can be used within this style to improve readability, and almost impossible to understand without replaying the proof script in order to see the goals. Declarative proofs are much easier to read and understand even without executing them in a proof assistant. However they are longer to write, and often more verbose. There are proposals to mix the two styles [33], and Coq, among some proof assistants, offers both styles [34], even if the declarative proof style is not mature yet and has few users.

We also think that mixing the two styles allows to have more readable proofs still allowing to have automation. However, we take a very lightweight approach: we rely on Coq's procedural proof language features that help to structure proof scripts, and we design some tactics and tactic notations to write proofs in a declarative spirit. Moreover we use the usual procedural style whenever the proof details would have been omitted in a mathematical text.

We develop new tactics and tactic notations using only the Ltac language of Coq. It would be possible to improved these tactics, and provide additional ones, by developing a plugin (implemented in OCaml) for

Coq. However this would no longer be a lightweight approach. We also are not advocating to use this style systematically, and in the whole development, we commit lightly to this style, mostly to help us remember quickly the main steps of a proof, but playing the proof script may still be needed to grasp the important details.

To show the advantages of the style, we wrote a version of the main theorem that strongly commits to it. We think the obtained proof script could be read on its own, without running it, provided our specific tactics, and some important Coq tactics, are informally explained. We begin with Coq’s standard tactics:

- While writing a proof in the procedural style, a tactic could generate several sub-goals from one goal. In the proof style we advocate, such a tactics should be *always* followed by “bullets”. Basically these could be understood as: for each sub-goal we start the proof by a symbol, such that the proof script displays a kind of itemised list of proofs. For example if in a proof, a tactic **tac** generates two sub-goals, the proof script should have the form:

tac.

- (* *proof script for first case* *)
- (* *proof script for second case* *)

Bullet symbols include –, + and *. Without thinking of the Coq goals, to read such a proof script we only need to know that **tac** is followed by a reasoning by case.

- **induction t** is a tactic that starts a proof by induction on the term **t**. It generates a sub-goal for each possible form of the inductive term **t**. When necessary, the sub-goal contains in hypothesis, the induction hypothesis. Optionally an alternative induction scheme **ind** could be specified **induction t using ind**. In our style **induction** should always be followed by bullets.
- **case t** is a tactic that is similar to **induction** but without generation of the induction hypothesis. This is just a reasoning by case. In our style **case** should always be followed by bullets.
- **apply H** is a tactic that tries to match the conclusion of the proposition named **H** and the goal. It generates sub-goals for each of the premises of **H**. It can be followed by **with t₁ ... t_n**, to provide explicitly the *n* first arguments of **H**.

In our style, **apply** is often use in combination with the tactical **now tac** that applies automatically simple tactics to solve the sub-goals that are generated by the tactic **tac**.

There are some cases that are not very complicated but not handled by **now**. That is why we provide the tactical **conclude by tac** which does the same that **now** but using more powerful automation.

In our style, we make sure to provide in the context all the hypotheses (or hypotheses that quite straightforwardly imply these hypotheses) needed to apply a lemma **H**.

- In order to do so, we heavily rely on the **assert** tactic of Coq. With **assert t as H** we state that we want to prove **t** and add it in the context under the name **H**. It can be optionally followed by **by** tactic which gives the proof script necessary to prove **t**.

In our style we use main forms :

- **assert t as H by** tactics when tactics is short,
- **assert t as H. { tactic₁. ... tactic_n. }** when the proof is longer. The { } is also a standard Coq way of structuring proofs. It indicates that what is inside is a proof of the first goal generated by the tactic that precedes it. In our style we only use it when there is only one goal.

- In usual Coq proof scripts, the tactic **destruct t as ...** is often used. While being powerful, it does not make proof scripts easy to read. In the proof scripts that follows our style, we use it in only one case: when we want to decompose a value of a singleton inductive type into its components. As this tactics allows to name the components (after **as**) but not explicitly remind how this components are combined together, we use comments to explain that. Therefore in the proof script of the main theorem we have:

destruct e as [t a]. (* e = (t,a) *)

Having a tactic such as **decompose** e as (t,a) for example, would require to write a Coq plugin.

- **inversion** H can be used when H is the name of a term in the context whose type is an inductive type. For each constructor of the type (thus it could produce several sub-goals), it generates, if possible, the arguments for this constructor that are used to build H.

In our proof style we mainly use it to prove a statement that is included as an argument of a constructor in an hypothesis present in the context.

- $\exists$ is used when the goal is an existential quantification, it provides the witness t.
- **contradiction** concludes when there are both a proposition and its negation in the context.

One major drawback of Coq's tactics, is that most of the tactics uses names of terms in the context rather than the terms themselves. One exception is **assert**, and it is one of the reasons we use it a lot. There is also no standard tactic to remind what is the goal. Thus we provide:

- **prove** t this tactic succeeds when the goal to prove is t. It makes the proof script much more readable, in particular we systematically use it in each of the bullets after an **induction** or **case**. When we want also to remind the hypotheses used to prove this goal we provide the form:

assuming t₁ as H₁ and ... and t_n as H_n **prove** t".

This form is in particular used to explicitly state the induction hypothesis in the inductive cases of a proof by **induction**.

Note that this form could also be used when the goal has the form t₁ → ... → t_n → t. In this case the terms t_i are introduced in the context, with the names specified by **as**.

prove t should always be followed by a block { } that contains the proof of t.

- **have** t is used to remind one of the assumptions present in the context. t is the statement of the assumption, not its name. Optionally, the assumption may be renamed using the form **have** t as H.
- In usual Coq proof scripts, we mentioned **destruct** is often used. In particular when there is a hypothesis H: $\exists x:T.P$ in the context, to extract a witness w that as the property P that we name Hw we write: **destruct** H as [w Hw]. It is the same thing than decomposing a pair into its components. That is sensible since in Coq, H is a dependent pair: the first component is w, and the second a proof that w has property P. However, as the structure of H is not made visible in the proof script, this kind of tactic is difficult to read.

Therefore we provide the tactic: **witness** t from H to extract a witness from an assumption, and often we use it in conjunction with **have** it order to remind the property that holds for t. For example the proof of the main theorem contains:

witness s' from Hs'. **have**(insertion i s' (t, a) == s').

Sometimes we do not use **have** but instead we assert derived properties that are easily proved from the property of the witness.

In the following proof scripts, there are some tactics that have not been explained. The convention that if **by** is followed by unexplained tactics, then it corresponds to omission of details in a mathematical proof. The statements of the lemmas (together with some definitions) applied in the proof are given in Section AppendixA.

Theorem `wsync_atomic` : $\forall s, \text{wellFormed } s \rightarrow \text{wellSynchronized } s \rightarrow \exists s', s \equiv s' \wedge \text{atomic } s'$.

Proof.

induction `s` **using** `tr_ind`.

– **prove** $(\exists s' : \text{Tr}, \text{nil} \equiv s' \wedge \text{atomic } s')$.

```
{
  assert(nil ≡ nil) by apply compatible_refl.
  assert(atomic nil) by apply nil_atomic.
  conclude by ∃nil.
}
```

– **assuming** $(\text{wellFormed } s \rightarrow \text{wellSynchronized } s \rightarrow \exists s' : \text{Tr}, s \equiv s' \wedge \text{atomic } s')$

prove $(\text{wellFormed } (s \cdot e) \rightarrow \text{wellSynchronized } (s \cdot e) \rightarrow \exists s' : \text{Tr}, s \cdot e \equiv s' \wedge \text{atomic } s')$.

```
{
  assuming (wellFormed (s · e)) as Hwf_se and (wellSynchronized (s · e)) as Hws_se
  prove (∃ s' : Tr, s · e ≡ s' ∧ atomic s').
  {
    assert (∃ s', compatible s s' ∧ atomic s') as Hs'.
    {
      have (wellFormed s → wellSynchronized s → ∃ s'' : Tr, s ≡ s'' ∧ atomic s'') as He.
      assert (wellFormed s) by now apply wellFormed_se.s with e.
      assert (wellSynchronized s).
      {
        assert (wf_open_close (s · e)) by now inversion Hwf_se.
        assert (wellSynchronized (s · e)) by now inversion Hwf_se.
        assert (wf_occurrences (s · e)) by now inversion Hwf_se.
        now apply wellSynchronized_se.s with e.
      }
      now apply He.
    }
    witness s' from Hs'.
    assert(s ≡ s') by intuition.
    assert(atomic s') by intuition.
  }
}
```

assert $(s \cdot e \equiv s' \cdot e)$ **by now apply** `compatible_S`.

assert $(\text{wellFormed } (s' \cdot e))$ **as** `Hwf_s'e` **by now apply** `compatible_wellFormed` **with** $(s \cdot e)$.

assert $(\text{wellSynchronized } (s' \cdot e))$ **by now apply** `compatible_wellSynchronized` **with** $(s \cdot e)$.

destruct `e` **as** $[t \ a]$. $(* \ e = (t, a) *)$

case $(\text{exclude_dec } s' (\text{wellFormed_se_s } s' (t, a) \text{Hwf_s'e}) \ t)$.

+ **assuming** $(\forall p : \text{SN.t}, \neg \text{exclude } s' \ p \ t)$ **as** `h_noExclude`

prove $(\exists s'0 : \text{Tr}, s \cdot (t, a) \equiv s'0 \wedge \text{atomic } s'0)$.

```
{
  have (s · (t, a) ≡ s' · (t, a)).
  assert (atomic (s' · (t, a))).
  {
    assert (conflicts s' t → False).
    {
      assuming (conflicts s' t) as h_conflicts prove False.
      {
        witness p from h_conflicts. have (exclude s' p t).
        assert(¬exclude s' p t) by now apply h_noExclude.
        contradiction.
      }
    }
    now apply noExclude_atomic.
  }
  conclude by ∃(s' · (t, a)).
}
```

```

}
+ assuming ( $\exists p : \text{SN.t}, \text{outerExclude } s' \ p \ t$ ) as  $h\_outerExclude$ 
prove ( $\exists s'0 : \text{Tr}, s \cdot (t, a) \equiv s'0 \wedge \text{atomic } s'0$ ).
{
  witness  $p$  from  $h\_outerExclude$ .
  assert ( $\exists i, \text{action\_of } (pi \ i \ s') == \text{Open } p$ ) as  $Hi$  by now apply  $\text{outerExclude\_open}$  with  $t$ .
  witness  $i$  from  $Hi$ . have ( $\text{action\_of } (pi \ i \ s') == \text{Open } p$ ).

  assert ( $\exists s'', \text{insertion } i \ s' \ (t, a) == s''$ ) as  $Hs''$ .
  {
    assert ( $i \leq \text{length } s'$ ) as  $Hv$  by (assert ( $i < \text{length } s'$ ) by (eauto with nth_error; intuition); intuition).
    pose ( $Hins\_def := (\text{insertion\_defined } i \ s' \ (t, a) \ Hv)$ ).
    witness  $s''$  from  $Hins\_def$ .
    now  $\exists s''$ .
  }
  witness  $s''$  from  $Hs''$ . have( $\text{insertion } i \ s' \ (t, a) == s''$ ).

  assert ( $s \cdot (t, a) \equiv s''$ ).
  {
    assert ( $s' \cdot (t, a) \equiv s''$ ).
    {
      assert ( $\text{exclude } s' \ p \ t$ ) by now inversion  $h\_outerExclude$ .
      assert ( $\forall k : \text{Tld.t}, i \leq k < \text{length } s' \rightarrow \neg \text{synchronizeWith } (s' \cdot (t, a)) \ k \ (\text{length } s')$ )
        by (inversion  $h\_outerExclude$ ; now apply  $\text{exclude\_noSync with } p$ ).
      now apply  $\text{insertion\_compatible with } i \ p$ .
    }
    now apply  $\text{compatible\_trans with } (s' \cdot (t, a))$ .
  }
  assert ( $\text{atomic } s''$ ).
  {
    assert ( $\forall k : \text{Tld.t}, i \leq k < \text{length } s' \rightarrow \neg \text{synchronizeWith } (s' \cdot (t, a)) \ k \ (\text{length } s')$ )
      by (inversion  $h\_outerExclude$ ; now apply  $\text{exclude\_noSync with } p$ ).
    now apply  $\text{outerExclude\_insertion\_atomic with } s' \ p \ t \ a \ i$ .
  }
  conclude by  $\exists s''$ .
}
}
}
Qed.

```

7.3. Organisation

The formal development in Coq is available at:

<https://traclifo.univ-orleans.fr/PaPDAS/wiki/TransactionsInCoq>

In order to improve the readability and the reutilisability of definitions, Coq has a module system. There are two types of module. **Module Type** acts as signature, and defines what is visible for the rest of the program. **Module** defines the implementation. There are two ways for a module to instantiate a module type. A module M can instantiate a signature S , $M:S$, and then all the definitions that are in M and not in S will not be available. A module M can be compatible with a signature S , noted $M<:S$ and the extra definitions will be visible for the rest of the program.

In the tables of Figures 5 and 6, we give the matching between the definitions and lemmas in the paper and their counterparts in Coq. For each one we indicate its Coq name and the Coq file. All the files are in the **Transactions** directory of the archive.

Name	Coq	File
Lemma 1	wellFormed_prec_tribe	Trace_Theory.v
Lemma 2	see_synchronizeWith	Synchronisation.v
Proposition 1	sw_in_tribe	Synchronisation.v
Proposition 2	compatible_wellSynchronised compatible_wellFormed	EquivalenceTheory.v
ins_{eq}	insertion_compatible	SyncInsertion.v
ins_{tribe}	insertion_tribe	SyncInsertion.v
Theorem 1	wsync_atomic	AtomicityFinal.v

Figure 5: Lemmas, propositions and theorems

There is about 12,000 lines of code including definitions, lemmas and their proofs. Proofs count for about 70% of the total size. In order to make this development more navigable, it has been divided in directories and files according to the following rules:

- Definitions are grouped together in a file naming with respect to the main notion these definitions concern. For example traces and the main predicates on them are defined in the file **Trace.v**.
- Basics and auxiliary lemmas are in a file suffixed with **_Basics**, and in case of a large number of this lemmas, it could be divided into several files, for example **Trace_Basics_tribe.v** is about results on the tribes of a trace.
- The main lemmas and theorems are in a file suffixed with **_Theory**. For example Lemma 1 is in the file **Trace_Theory.v**.

As we consider a concrete language for nested atomic sections and thread escape, its operational semantics [16] and its compilation towards a language without atomic sections but with locks, each main part of this overall development is located in a directory. These additional parts are not yet in the online version, only three directories are available:

- **Support** contains some very general definitions and results, not specific to this development but nonetheless useful,
- **Common** contains material shared by the different parts of the full development such as the definition of types and values,
- **Transactions** contains the Coq formalisation of the previous sections. Inside this directory the notions having their specific set of files (and their dependencies) follows:

Trace $\leftarrow$ **Synchronisation** $\leftarrow$ **Equivalence** $\leftarrow$ **Insertion** $\leftarrow$ **Atomicity**

8. Conclusion and Future Work

In our study, we consider program traces of imperative languages with fork-join parallelism and lexically scoped atomic section which supports nesting and parallelism to provide a good modularity. This parallelism does not impose unnecessary synchronisation, thus threads can escape from surrounding sections. We design a semantics independent of any implementation. We precisely define well-formedness, that could be seen as a specification for the operational semantics of imperative languages with nested atomic sections and inner parallelism, as well as well-synchronisation and serialisability. We prove that all well-formed and well-synchronised traces are serialisable. We use the interactive theorem prover Coq to express our definitions and lemmas, and to check our proofs. In another paper [16], we design an operational semantics for a

Name	Coq	File
section name	t (in module Permission)	Permission.v
actions	action	Trace.v
events	t (in module Event)	Trace.v
traces	Tr	GenericTrace.v
$\subseteq$	sec_order	Trace.v
$\smile$	#	Trace.v
owns	owns	Trace.v
father	father	Trace.v
$\prec$	precedes	Trace.v
see	see	Trace.v
range	range	Trace.v
tribeChildren	tribeChildren	Trace.v
tribe	tribe	Trace.v
excludes	exclude	Trace.v
insertion	insertion	Insertion.v
Definition 1	wellSynchronized	Synchronisation.v
Definition 2	atomic	Atomicity.v
Definition 3	atomic, compatible	
Definition 4	compatible	Equivalence.v
wf₁	wf_occurences	Trace.v
wf₂	wf_open_close	Trace.v
wf₃	wf_seq_order	Trace.v
wf₄	wf_fork	Trace.v
wf₅	wf_join	Trace.v
wf₆	wf_join_all_closed	Trace.v
wf₇	wf_join_see_fork	Trace.v
wf₈	wf_mutual_exclusion	Trace.v

Figure 6: Definitions

simple imperative language that produces only well-formed traces. This is evidence that the well-formedness conditions are indeed specification for language design.

This work is a part of a larger plan: we are building a verification compilation pass for our language with atomic sections towards a language without atomic sections but with locks. Our source language is the simple imperative language cited previously. Its semantics uses an abstract representation of nested atomic sections, a recursive structure where each level represents an atomic section. The target language has a concrete realisation of this structure, using records and locks. The compilation replaces atomic sections calls, and adds extra code to maintain this data structure.

An additional pass would be to use static analysis (like in [35]) to improve our compilation.

Acknowledgements

Thomas Pinsard was supported by a PhD grant from the French ministry of higher education and research. This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS “Parallel Program Development with Algorithmic Skeletons” JST 10102704 and ANR-2010-INTB-0205-02).

References

- [1] P. W. O’Hearn, H. Yang, J. C. Reynolds, Separation and information hiding, in: POPL, ACM, 2004, pp. 268–280. doi:10.1145/964001.964024.
- [2] N. Shavit, D. Touitou, Software transactional memory, in: PODC, ACM, 1995, pp. 204–213. doi:10.1145/224964.224987.
- [3] J. Larus, C. Kozyrakis, Transactional memory, Commun. ACM 51 (7) (2008) 80–88. doi:10.1145/1364782.1364800.
- [4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transactional memory: Why is it only a research toy?, Commun. ACM 51 (11) (2008) 40–46. doi:10.1145/1400214.1400228.
- [5] A. Dragojević, P. Felber, V. Gramoli, R. Guerraoui, Why STM can be more than a research toy?, Commun. ACM 54 (2011) 70–77. doi:10.1145/1924421.1924440.
- [6] M. Hicks, J. S. Foster, P. Pratikakis, Lock inference for atomic sections, in: Transact, Ottawa, Canada, 2006.
- [7] S. Cherem, T. Chilimbi, S. Gulwani, Inferring locks for atomic sections, in: PLDI, ACM, 2008, pp. 304–315. doi:10.1145/1375581.1375619.
- [8] H.-J. Boehm, Transactional memory should be an implementation technique, not a programming interface, in: HotPar, USENIX Association, 2009, pp. 15–15.
- [9] U. Drepper, Parallel programming with transactional memory, Commun. ACM 52 (2009) 38–43. doi:10.1145/1461928.1461943.
- [10] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, M. Valero, The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment, in: 5th Conference on Computing Frontiers (CF’08), ACM, 2008, pp. 67–78. doi:10.1145/1366230.1366241.
- [11] J. E. B. Moss, A. L. Hosking, Nested transactional memory: model and architecture sketches, Sci. Comput. Program. 63 (2006) 186–201. doi:10.1016/j.scico.2006.05.010.
- [12] K. Agrawal, J. T. Fineman, J. Sukha, Nested parallelism in transactional memory, in: PPOPP, ACM, 2008, pp. 163–174. doi:10.1145/1345206.1345232.
- [13] K. F. Moore, D. Grossman, High-level small-step operational semantics for transactions, in: POPL, ACM, 2008, pp. 51–62. doi:10.1145/1328438.1328448.
- [14] The Coq Development Team, The Coq Proof Assistant, <http://coq.inria.fr>.
- [15] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development, Springer, 2004.
- [16] F. Dabrowski, F. Loulergue, T. Pinsard, Nested Atomic Sections with Thread Escape: An Operational Semantics, in: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), IEEE, Taiwan, 2013, pp. 29–35. doi:10.1109/PDCAT.2013.12.
- [17] M. Herlihy, J. E. B. Moss, Transactional memory: architectural support for lock-free data structures, SIGARCH Comput. Archit. News 21 (2) (1993) 289–300. doi:10.1145/173682.165164.
- [18] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, III, Software transactional memory for dynamic-sized data structures, in: PODC, ACM, 2003, pp. 92–101. doi:10.1145/872035.872048.
- [19] M. Herlihy, V. Luchangco, M. Moir, A flexible framework for implementing software transactional memory, in: OOPSLA, ACM, 2006, pp. 253–262. doi:10.1145/1167473.1167495.
- [20] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, D. Nussbaum, Hybrid transactional memory, in: ASPLOS XII, ACM, 2006, pp. 336–346. doi:10.1145/1168857.1168900.
- [21] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: PPOPP, ACM, 2005, pp. 48–60. doi:10.1145/1065944.1065952.
- [22] A. Bieniusa, P. Thiemann, Proving isolation properties for software transactional memory, in: G. Barthe (Ed.), ESOP, Vol. 6602 of LNCS, Springer, 2011, pp. 38–56. doi:10.1007/978-3-642-19718-5_3.

- [23] E. Koskinen, M. Parkinson, M. Herlihy, Coarse-grained transactions, in: POPL, ACM, 2010, pp. 19–30. doi:10.1145/1706299.1706304.
- [24] S. Jagannathan, J. Vitek, A. Welc, A. Hosking, A transactional object calculus, Sci. Comput. Program. 57 (2005) 164–186. doi:10.1016/j.scico.2005.03.001.
- [25] M. Abadi, A. Birrell, T. Harris, M. Isard, Semantics of transactional memory and automatic mutual exclusion, in: POPL, ACM, 2008, pp. 63–74. doi:10.1145/1328438.1328449.
- [26] M. Martin, C. Blundell, E. Lewis, Subtleties of transactional memory atomicity semantics, IEEE Comput. Archit. Lett. 5 (2) (2006) 17–17. doi:10.1109/L-CA.2006.18.
- [27] C. H. Papadimitriou, The serializability of concurrent database updates, Journal of the ACM 26 (4) (1979) 631–653. doi:10.1145/322154.322158.
- [28] T. Coquand, G. Huet, The Calculus of Constructions, Information and Computation 37 (2-3).
- [29] G. Gonthier, Formal Proof – The Four-Color Theorem, Notices of the American Mathematical Society 55 (11) (2008) 1382–1393.
- [30] Y. Bertot, Coq in a hurry, <http://hal.inria.fr/inria-00001173> (2006).
- [31] A. Chlipala, An Introduction to Programming and Proving with Dependent Types in Coq, Journal of Formalized Reasoning 3 (2). doi:10.6092/issn.1972-5787/1978.
- [32] A. Chlipala, Certified Programming with Dependent Types, MIT Press, 2014.
- [33] F. Wiedijk, A Synthesis of the Procedural and Declarative Styles of Interactive Theorem Proving, Logical Methods in Computer Science 8 (1). doi:10.2168/LMCS-8(1:30)2012.
- [34] P. Corbineau, A Declarative Language for the Coq Proof Assistant, in: M. Miculan, I. Scagnetto, F. Honsell (Eds.), Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers, Vol. 4941 of LNCS 4941, Springer, 2008, pp. 69–84. doi:10.1007/978-3-540-68103-8_5.
- [35] Y. Afek, G. Korland, A. Zilberstein, Lowering STM overhead with static analysis, in: K. Cooper, J. Mellor-Crummey, V. Sarkar (Eds.), Languages and Compilers for Parallel Computing, Vol. 6548 of LNCS, Springer, 2011, pp. 31–45. doi:10.1007/978-3-642-19595-2_3.

AppendixA. Additional Lemmas and Definitions in Coq

Lemma nil_atomic : atomic nil.

Lemma wellFormed_se_s : $\forall (s:Tr) (e:Event.t), wellFormed (s \cdot e) \rightarrow wellFormed s$.

Lemma wellSynchronized_se_s :

$\forall (s:Tr) (e:Event.t), wf_occurrences (s \cdot e) \rightarrow wf_open_close (s \cdot e) \rightarrow wellSynchronized (s \cdot e) \rightarrow wellSynchronized s$.

Lemma compatible_trans : Transitive compatible.

Lemma compatible_S : $\forall (s1 s2:Tr) (e:Event.t), wellFormed (s1 \cdot e) \rightarrow s1 \equiv s2 \rightarrow s1 \cdot e \equiv s2 \cdot e$.

Lemma compatible_wellFormed : $\forall (s s':Tr), compatible s s' \rightarrow wellFormed s \rightarrow wellSynchronized s \rightarrow wellFormed s'$.

Lemma compatible_wellSynchronized : $\forall (s s':Tr), compatible s s' \rightarrow wellSynchronized s \rightarrow wellSynchronized s'$.

Definition pending (s:Tr) (p:SN.t) : Prop := $(\exists i, range s p i (length s - 1)) \wedge \neg occursIn s (Close p)$.

Definition exclude (s:Tr) (p:SN.t) (t:TId.t) : Prop := pending s p $\wedge \neg tribe s p t$.

Definition conflicts (s : Tr) (t : TId.t) := $\exists p, exclude s p t$.

Inductive outerExclude (s:Tr)(p:SN.t)(t:TId.t) : Prop :=

outerExclude.cons:

$\forall (H0:exclude s p t) (H1:\forall p', exclude s p' t \rightarrow p <> p' \rightarrow occursAfter s (Open p) (Open p')),$
outerExclude s p t.

Lemma noExclude_atomic :

$\forall (s:Tr) (H:atomic s) (t:TId.t)(a:Action) (HWF:wellFormed (s \cdot (t,a))), \neg conflicts s t \rightarrow atomic (s \cdot (t,a))$.

Lemma `exclude_dec` :

$\forall (s:\text{Tr}) (\text{HWF}:\text{wellFormed } s) (t:\text{TId.t}),$
 $(\forall p, \neg \text{exclude } s \ p \ t) \vee (\exists p, \text{outerExclude } s \ p \ t).$

Lemma `outerExclude_open` :

$\forall (s:\text{Tr}) (p:\text{SN.t}) (t:\text{TId.t}), \text{outerExclude } s \ p \ t \rightarrow \exists i, \text{action_of } (pi \ i \ s) == \text{Open } p.$

Lemma `insertion_defined` $\{A:\text{Set}\}$: $\forall (m:\text{nat}) (l:\text{list } A) (e:A), m \leq \text{length } l \rightarrow \exists l', \text{insertion } m \ l \ e == l'.$

Lemma `exclude_noSync`:

$\forall (s:\text{Tr}),$
 $\text{atomic } s \rightarrow$
 $\forall (p:\text{SN.t}) (t:\text{TId.t}) (a:\text{action}) (i:\text{nat}),$
 $\text{wellFormed } (s \cdot (t,a)) \rightarrow \text{wellSynchronized } (s \cdot (t,a)) \rightarrow \text{exclude } s \ p \ t \rightarrow \text{action_of } (pi \ i \ s) == \text{Open } p \rightarrow$
 $(\forall k, i \leq k < \text{length } s \rightarrow \neg \text{synchronizeWith } (s \cdot (t,a)) \ k \ (\text{length } s)).$

Lemma `insertion_compatible`:

$\forall (i:\text{nat}) (p:\text{SN.t}) (s:\text{Tr}) (t:\text{TId.t}) (a:\text{action}) (s':\text{Tr}),$
 $\text{wellFormed } (s \cdot (t,a)) \rightarrow$
 $\text{wellSynchronized } (s \cdot (t,a)) \rightarrow$
 $\text{exclude } s \ p \ t \rightarrow$
 $(\forall k, i \leq k < \text{length } s \rightarrow \neg \text{synchronizeWith } (s \cdot (t,a)) \ k \ (\text{length } s)) \rightarrow$
 $\text{insertion } i \ s \ (t,a) = \text{Some } s' \rightarrow$
 $\text{action_of } (pi \ i \ s) == \text{Open } p \rightarrow$
 $\text{compatible } (s \cdot (t,a)) \ s'.$