Nested Atomic Sections with Thread Escape: An Operational Semantics

Frédéric Dabrowski, Frédéric Loulergue, Thomas Pinsard
Univ Orléans, ENSI de Bourges, LIFO EA 4022, Orléans, France

October 18, 2013

Abstract

We consider a simple imperative language with fork/join parallelism and lexically scoped nested atomic sections from which threads can escape. In this context, our contribution is a formal operational semantics of this language that satisfies a specification on execution traces designed in a companion paper.

Keywords: formal methods; operational semantics; atomic section; programming language

1 Introduction

In the context of multi-core architectures, an alternative to mutexes for dealing with shared data has been proposed: atomic sections. They are often presented as simpler to use, and less error-prone [6]. Indeed contrary to locks, the user has only the responsibility to delimit the portions of code to be executed in isolation, and the compiler or the run-time system ensures the interference freedom. However, it is not clear if it can be very efficiently implemented [10]. There are several existing implementations, mostly based on transactions inspired from database management systems [17] [12]. Another approach [2] [7] is to generate a set of locks to protect the atomic sections.

These implementations generally suffer from limitations. The nesting of atomic sections, often forbidden, is however mandatory to achieve a good modularity. For example, it is interesting to be able to call a method that uses atomic sections from a portion of code itself inside an atomic section. In this scenario, there is not necessarily parallelism inside atomic sections. But modularity is again increased if the called method is allowed to fork new threads. This kind of inner parallelism is forbidden or incurs forced synchronisation in current implementations. We argue that both section nesting and inner parallelism without unnecessary synchronisation are essential for a language with atomic sections. Therefore we consider a simple imperative language with fork/join parallelism and lexically scoped nested atomic sections from which threads can escape. We name this language Atomic Fork Join or AFJ.

In this context, it is very important to be precise on the meaning of atomicity. Two kinds of atomicity can be defined: weak atomicity where sections are isolated against other sections only, and strong atomicity which offers a total isolation. Several implementations are not clear on which kind of atomicity they offer, and this can lead to surprising behaviour. One way to be explicit about it, is to propose a formal semantics: This is what we did in a companion paper [3]. Rather than to define the suitable notions of well-synchronisation and atomicity on an operational semantics of AFJ, we were more abstract and defined these notions on execution traces, i.e. sequences of events. Of course any sequence of events does not necessarily represent a meaningful program execution, for a reasonable definition of program execution. To model which sequences should be considered, we defined well-formed traces. In this way, well-formedness is a specification for formal definitions of program execution, i.e. formal operational semantics.

The contribution of this paper is the formal definition of the AFJ language: its syntax and operational semantics, and the proof that it generates only well-formed traces. The paper is organised as follows. Section 2 Then we recall the notion of well-formed traces (Section 3). We propose an operational semantics for AFJ (Section 4). In Section 5 we prove the properties of well-formedness of traces generated by AFJ operational semantics. We compare our contribution with related work (Section 6), before concluding and give future research directions (Section 7).
2 The AFJ Language

We assume disjoint countable sets of memory locations and thread names, elements of which are respectively noted \( \ell \) and \( t \), possibly with subscript. There is also one special memory location written \( \text{null} \) that could not be allocated. The set of values, elements of which are noted \( v \), possibly with subscripts, contains at least memory locations, integers, booleans and thread names.

\[
v ::= n \mid \ell \mid t \mid b \quad \text{where} \ n \in \mathbb{N}, \ell \in \mathbb{L}, t \in \mathbb{T} \quad \text{and} \ b \in \mathbb{B}
\]

In the following grammars, \( \mathcal{X} \) means the countable set of local variables, \( \vec{e} \) is a tuple of expressions and \( \text{op} \) indicates a predefined operation on values. \( d \) is the set of values that users can write as constants, for example \( 42 \) for an integer constant, or \( \text{true} \) for a boolean constant. This set excludes thread identifiers.

\[
d ::= n \mid b \mid \ell
\]

\[
e ::= d \mid x \mid \text{op}(\vec{e}) \quad \text{where} \ x \in \mathcal{X}
\]

The set of predefined operations are not detailed here, but contains usual arithmetic and boolean operations. An AFJ program is a set of methods and a sequence of main instructions. A method is defined by a name, an argument and a body which is a sequence of instructions. The grammar for programs, methods and instructions is:

\[
mth ::= m(x) \ c
\]

\[
r ::= mth \ c
\]

\[
c ::= \text{skip} \mid c \ c \mid \text{if} \ b \ \text{then} \ c \ \text{else} \ c
\]

\[
| \text{while} \ b \ \text{do} \ c \ | \ x := e \ | \ x := y[e'] \ | \ x[e'] := e''
\]

\[
x := \text{allocate}(e) \ | \ \text{dispose}(e)
\]

\[
| \text{atomic} \{e\} \ | \ x = \text{fork}(m, e) \ | \ \text{join} \ c
\]

The AFJ language has a \( \text{skip} \) instruction and usual sequential control structures: sequence, conditional and loop. A local variable can be assigned with the value of an expression, or the value contained at a given memory location. One can read from or write to a memory location. However to access such a memory location, it must have been previously allocated thanks to the \( \text{allocate} \) instruction which takes as argument the number of memory cells to allocate contiguously and returns the address of the first memory cell allocated. The deallocation of memory previously allocated is done by the \( \text{dispose} \) primitive which frees a given memory location.

The \( \text{atomic} \) primitive allows to indicate that the given sequence of instructions must be executed as atomic. The \( \text{fork} \) primitive allows to launch the execution, by a newly created thread, of a method with a given argument. The \( \text{fork} \) primitive returns the identifier of the created thread. The \( \text{join} \) primitive allows to synchronise the current thread with the thread whose identifier is given as a parameter. A very short example follows:

\[
m(x)
\]

\[
\text{atomic}\
\quad x[0] := x[0]+1;
\}
\]

\[
x := \text{allocate}(1);
\]

\[
x[0] := 1;
\]

\[
u := \text{fork}(m,x);
\]

\[
\text{atomic}\
\quad x[0] := x[0]+1;
\}
\]

\[
\text{join}(u);
\]

3 A Specification of Execution Traces

In [3] we precisely defined the notions of well-synchronisation and atomicity on partial “execution” traces without specifically referring to a language syntax and semantics. Instead we defined criteria for a trace to be considered as a valid trace, i.e. a trace that could be obtained from the execution of a program (for a reasonable definition of execution). In order to show that the operational semantics we propose in this paper indeed produces only valid traces, and that therefore the results we obtained in [3] could be applied to AFJ, we summarise the notion of well-formedness for traces.

We assume a countable set of section names, elements of which are noted \( p \), possibly with subscript.
**Actions, events and traces** We define the set of actions, elements of which are noted $a$, possibly with subscripts, as follows.

$$a ::= | e \mid \text{fork } t \mid \text{join } t \mid \text{open } p \mid \text{close } p$$

$$a ::= | \text{alloc } \ell n \mid \text{free } \ell n v \mid \text{read } \ell n v \mid \text{write } \ell n v$$

Intuitively, $e$ denotes an internal, non observable, action. An action $\text{alloc } \ell n$ denotes heap allocation of a block of size $n$ at memory location $\ell$ and an action $\text{free } \ell$ removes such a block from the heap. An action $\text{read } \ell n v$ (resp. $\text{write } \ell n v$) denotes a read (resp. write) access from (resp. to) the offset $n$ from location $\ell$ and $v$ is the read (resp. written) value. Actions $\text{fork } t$ and $\text{join } t$ respectively denote creation and join on a thread $t$. Finally, $\text{open } p$ and $\text{close } p$ 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 below. Their sole purpose is to name occurrences of atomic sections occurring in traces. An event $e$ is a pair $(t,a)$ of a thread name and an action. A trace $s$ is a sequence of events: $s ::= e \mid s \cdot e$.

**Notations** We note $s_1 s_2$ the concatenation of partial traces (or traces for short) $s_1$ and $s_2$. For a trace $s$, we define the partial function $\pi_s$ by $\pi_s(0) = e$ and $\pi_s(n+1) = \pi_s(n)$, where by abuse of notation $e$ stands for $e \cdot e$. We respectively note $\pi_s(i)^+$ and $\pi_s(i)^-$ 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 $\pi_s(i)^+(i) = a$.

**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 $\text{owner}_s$ and $\text{father}_s$ respectively relate a section to its owner thread and a thread to its father. The relation $\text{range}_s$ 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 $s$ defined below, the relations $\text{owner}_s$, $\text{father}_s$ and $\text{range}_s$ will define partial functions.

$$\text{owns}_{s,p} t \triangleq (t, \text{open } p) \in s$$

$$\text{father}_{s,t} t' \triangleq (t', \text{fork } t) \in s$$

$$\pi_s^{act}(i)^+ p_s^{act}(i) = \text{open } p \quad \pi_s^{act}(j) = \text{close } p$$

$$\text{range}_{s,p} i j \quad \pi_s^{act}(i) = \text{open } p \quad \text{close } p \notin s$$

$$\text{range}_{s,p} i |s| - 1$$

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 $\text{tribe}_s p$ 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 $\text{tribeChildren}_s$).

$$\text{range}_{s,p} i j \quad i < k \leq j$$

$$\text{owns}_{s,p} t' \quad \pi_k(s) = (t', \text{fork } t)$$

$$\text{tribeChildren}_{s,p} t$$

$$\text{tribeChildren}_{s,p} t' \quad \text{father}_{s,t} t'$$

$$\text{tribeChildren}_{s,p} t$$

Intuitively, if $t$ belongs to $\text{tribe}_s p$ then the thread $t$ is part of the computation of the atomic section $p$ and thus should not be considered as an interfering thread. 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. The notation and more precisely the order between $p$
and $p'$, can be seem counter-intuitive as set inclusion but it matches the notion of bubble defined in the next section.

$$\begin{align*}
\text{range}_{s,p} i j & \quad i < k \leq j \\
\text{owns}_{s,p} t & \quad \pi_k(s) = (t, \text{open } p') \\
\text{tribeChildren}_{s,p} t & \quad \text{owns}_{s,p'} t
\end{align*}$$

Finally, two atomic sections are said to be, written $p \preceq_s p'$ if $p \not\in_s p'$ and $p' \not\in_s p$.

**Well-formed traces** We now state formally some well-formedness conditions over program traces that can be seen as a specification for operational semantics of our language. They range from common sense conditions to design choices. To formalise these conditions we use the following definitions: The predicate $\text{see}_s$ which can be seen as an over-approximation of the information flow in $s$, is defined as the transitive closure of (1); the relation $\prec_s$ on section names is defined at (2).

\begin{align*}
\text{see}_s i j & \quad \pi_{s}^{\text{act}}(i) = t \\
\text{see}_s i j & \quad \pi_{s}^{\text{act}}(j) = t \\
\text{fork } t & \quad \pi_{s}^{\text{act}}(i) = \text{fork } t \\
\text{read } \ell n v & \quad \pi_{s}^{\text{act}}(j) = \text{read } \ell n v \\
\text{write } \ell n v & \quad \pi_{s}^{\text{act}}(j) = \text{write } \ell n v \\
\text{close } p & \quad \pi_{s}^{\text{act}}(i) = \text{close } p \\
\text{open } p' & \quad \pi_{s}^{\text{act}}(j) = \text{open } p'
\end{align*} \tag{1}

$A$ 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. Conditions (wf4) and (wf5) state usual properties of fork/join instructions.

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. Condition (wf8) states that concurrent sections do not overlap.

## 4 An Operational Semantics for AFJ

We describe now the operational semantics which satisfies the specification stated in Section [3].

### 4.1 Definitions

Each thread has its own local environment. The threads are sharing the heap. The local environment is a partial function from variables to values. The heap is a partial function taking a memory location and an offset and returning a value. The value of a non-allocated memory cell is undefined. The set of threads is a partial function from thread identifier to a pair made of the remaining code to be executed by the thread, and its local environment. In the following, $\rightarrow$ stands for partial functions. $\mathcal{V}$ and $\mathcal{C}$ respectively means the set of values and instructions.

\begin{align*}
\rho & \quad \in \ \mathcal{X} \rightarrow \mathcal{V} \quad \text{local environment} \\
\sigma & \quad \in \ \mathcal{L} \rightarrow \mathcal{N} \rightarrow \mathcal{V} \quad \text{heap} \\
\phi & \quad \in \ T \rightarrow \mathcal{C} \times \mathcal{E} \quad \text{set of threads}
\end{align*}
Every action open $p$, close $p$ and fork $t$ occurs at most once in $s$.  

$$\forall i, p. \pi_s^w(i) = close p \Rightarrow \exists j. j < i \land \pi_s^w(j) = open p \land \pi_s^w(i) = \pi_s^w(j)$$  

$$\forall i, j, i < j \Rightarrow \pi_s^w(i) = \pi_s^w(j) = close p$$  

$$\forall k, p, i < k < j \Rightarrow \pi_s^w(i) = \pi_s^w(k) \Rightarrow \pi_s^w(k) = open p' \Rightarrow$$  

$$\exists j', k < j' < j \land \pi_s^w(j') = close p'$$  

$$\forall i, t. \pi_s^w(i) = fork t \Rightarrow \forall j. \pi_s^w(j) = t \Rightarrow i < j$$  

$$\forall i, t. (\pi_s^w(i) = t \lor \pi_s^w(i) = fork t) \Rightarrow \forall j. \pi_s^w(j) = join t \Rightarrow i < j$$  

$$\forall p, p', i, j, t. range_s p i j \Rightarrow owns_s p t \Rightarrow \forall k. \pi_s^w(k) = join t \Rightarrow j < k$$  

$$\forall t, i, j. \pi_s^w(i) = fork t \Rightarrow \pi_s^w(j) = join t \Rightarrow see_s i j$$  

$$\forall p, p', open p \in s \Rightarrow open p' \in s \Rightarrow p \leadsto p' \Rightarrow p \leadsto p' \lor p' \leadsto p$$  

Figure 1: Well-Formedness Conditions

We use an array notation to manipulate the heap. To get a value of the heap we use $\sigma(\ell)[n]$ which stands for $\sigma(\ell, n)$. To modify the heap we use $\sigma \cdot [\ell + n \mapsto v]$, which means that for all $\ell', n' \neq \ell, n$, $\sigma \cdot [\ell + n \mapsto v](\ell')[n'] = \sigma(\ell')[n']$, and $\sigma \cdot [\ell + n \mapsto v](\ell)[n] = v$.

The evaluation of expressions, noted $\llbracket c \rrbracket_p$, is defined classically, as for example in [15]. Local reduction contexts, noted $C$, allow to make explicit which instruction is evaluated by a thread. They are defined as follows: $C : = \emptyset | C \cup c$.

To ensure the atomicity in the operational semantics, we use a recursive structure, called bubble and noted $B$. A program always starts in a initial bubble and its main instruction is run by an initial thread, both are not explicitly created in the program. The formal definition of a bubble is:

$$B : = \emptyset \cup \emptyset ; B^o \mid B$$  

$$B^o : = \emptyset \mid B$$  

where $p$ is the name of the section, $t$ is the thread that started the section, i.e. the owner of the section, $C$ is the local context, allowing to resume the execution after the end of the section.

For example, let us consider a program containing the instruction: $C_0[atomic \{C_1[atomic \{c\}]\}]$. Then an execution of this program may reach a state where the bubble is

$$\llbracket \emptyset \cup \emptyset ; (c, p)^{t_1} \cup \emptyset \rrbracket_{C_0}^{p_1, t_0}$$  

just after the program started to execute the instruction atomic $\{c\}$. If in $C_1$, we have a forked thread that also opens an atomic section, it can only be opened when the bubble $p_1$ is closed. It is only possible to have one of several concurrent sections opened at a time. Intuitively this ensures atomicity.

In the bubble structure there is often several possible threads to execute, possibly at different sub-bubbles. Thus to express the semantics, we need a notion of bubble context. A bubble context is noted $K$ and defined as follow:

$$K : = \emptyset \mid \emptyset \llbracket \emptyset \rrbracket_{C_0}^{p_1, t_0}$$

We use the following notation $t \in B$ if $B = \llbracket \emptyset \cup \emptyset ; B_0 \rrbracket$ and $\exists c. \phi(t) = c$ or $t \in B_0$. By an abuse notation we use it for bubble context and $B^o$. We also use it for the presence in a pool of threads, $t \in \phi$ if $\exists c. \phi(t) = c$. We say that $p \in B$ if $B = \llbracket \emptyset \cup \emptyset ; B_0 \rrbracket$ and $p = p_1 \lor p \in B_0$.

4.2 The Rules

The operational semantics is defined by a set of relations. The first one is a relation on program states consisting of a bubble, a heap and a set of used section names:

$$\vdash_r B, \sigma, P \xrightarrow{\sigma, \pi} B', \sigma', P'$$
It is defined in Figure 4 by rule \( (\text{main}) \) and simply selects the sub-bubble on which to operate, i.e. the bubble context. The actual execution is formalised by the next relation. The other atomic section rules have the form:

\[
K \vdash_r B,\sigma,P \xrightarrow{a} t, B',\sigma',P'
\]

and are defined in Figure 4. They are used to open/close a section or to execute instructions not related to sections using a relation that defines inter-threads behaviour. This relation (Figure 3) has the form:

\[
K, B^o, t_0 \vdash_r \phi,\sigma \xrightarrow{a} \phi',\sigma'
\]

where \( t_0 \) (resp. \( B^o \)) is the owner of the section (resp. the direct sub-bubble of the bubble) in which thread \( t \) is executing. It is important to have this information when generating fresh threads names or checking the termination of threads.

The last relation used in the previous one, only deals with “sequential” instructions. It is defined in Figure 2 and has the form:

\[
c, \rho, \sigma \xrightarrow{a} c', \rho', \sigma'
\]

The rule \( (\text{open}) \) describes the creation of a sub-section. First of all, there must not be already a sub-section. In the bubble representing the current section a sub-bubble is added. The name of the new bubble must be unique. This is ensured by the predicate \( \text{fresh} \) which uses the set of names already taken. This new name is then added to this set. The local context is saved to resume the thread’s execution in the future. The new sub-bubble contains only one thread, which is the thread owner and its instructions are those protected by the section.

The rule \( (\text{close}) \) describes the closing of a section: It can be closed when the owner thread is terminated. We can note here that there can be other running threads. This set of threads are added to the set of the parent section. The thread owner of the closing section can continue its execution thanks to the saved local context. The last reduction \( (\text{inter}) \) modifies only the set of threads and the heap.

The rule \( (\text{fork}) \) creates a new thread. The predicate \( \text{fresh} \) ensures that the new thread identifier is not in the current state and as used thread names remain in the state, has never been used. The new thread is associated to the body of the method given as parameter in the \( \text{fork} \) instruction, and its local store to the value of the parameter.

The rule \( (\text{join}) \) allows to wait for a terminated thread. The predicate \( \text{term} \), ensures that the thread is terminated, i.e. the statement associated must be \( \text{skip} \) and it must not be the owner of any section.

The rule \( (\text{intra}) \) makes a step for one thread and is described in Figure 2 with more classical instructions (conditional, loop, ...) . Note in the rule \( (\text{alloc}) \), that the new cells have the default value 0, and we verify that \( \ell \) is free in the heap thanks to the condition \( \ell \notin \text{dom}(\sigma) \).

Each reduction rule is labelled with an action (described in 3) and a thread. These two elements allow us to create an event, and several reduction steps of a program \( r \) starting from the initial state \( \Sigma_0(r) \) create a trace. The initial state is the state from where the programs begin. It is parametrised by the program, because it must load the main instructions. Its is formally defined as

\[
\Sigma_0(\text{main} c) = (\ll (c,\emptyset)^{t_\ell}; \circ \prod_{p_\ell}^{t_\ell}; \emptyset, \{p_\ell\}).
\]

This initial state is composed of the initial bubble with the initial section name \( p_\ell \) and the initial thread name \( t_\ell \). The thread \( t_\ell \) is associated to the main command \( c \) of the program. The initial heap contains nothing. The initial set of used section names contains only the initial section name \( p_\ell \).

An example of a sequence of reduction from the example program of the previous section is shown in Figure 5.

4.3 Possible Implementations

A possible implementation can use locks. Each level of a bubble is associated with a lock. To open a new bubble the lock must be free. The lock is held while the bubble is active, and release when closed. With this implementation, only one bubble can be opened at each level.

Using static analysis, we could improve this implementation by trying to detect if two concurrent bubbles share memory. If not, the two bubbles can be run simultaneously, i.e. using two different locks.
\[
x := e, \rho, \sigma \xrightarrow{\epsilon} \text{skip, } \rho[x \mapsto v], \sigma \quad \text{if } [\epsilon]_{\rho} = v \quad \text{(assign)}
\]

\[
x := \text{allocate}(e), \rho, \sigma \xrightarrow{\text{alloc } \ell} \text{skip, } \rho[x \mapsto \ell], \sigma \cdot [\ell \mapsto [0]^{n}] \quad \text{if } [\epsilon]_{\rho} = n, n > 0, \ell \neq \text{null, and } \ell \notin \text{dom}(\sigma) \quad \text{(alloc)}
\]

\[
\text{dispose}(e), \rho, \sigma \cdot [\ell \mapsto v] \xrightarrow{\text{free } \ell} \text{skip, } \rho, \sigma \quad \text{if } [\epsilon]_{\rho} = \ell \text{ and } \ell \notin \text{dom}(\sigma) \quad \text{(dispose)}
\]

\[
x := y[e], \rho, \sigma \xrightarrow{\text{read } \ell} \text{skip, } \rho[x \mapsto v], \sigma \quad \text{if } \rho(y) = \ell, [e]_{\rho} = n, \rho(\ell)[n] = v \quad \text{(get)}
\]

\[
x[e_1] := e_2, \rho, \sigma \xrightarrow{\text{write } \ell} \text{skip, } \rho, \sigma \cdot [\ell + n \mapsto v] \quad \text{if } \rho(x) = \ell, [e_1]_{\rho} = n, [e_2]_{\rho} = v \text{ and } \ell + n \in \text{dom}(\sigma) \quad \text{(put)}
\]

\[
\text{skip; } s, \rho, \sigma \xrightarrow{\epsilon} s, \rho, \sigma \quad \text{(sequence)}
\]

\[
\text{while } b \text{ do } s, \rho, \sigma \xrightarrow{\epsilon} s; \text{while } b \text{ do } s, \rho, \sigma \quad \text{if } [b]_{\rho} = \text{true} \quad \text{(loop)}
\]

\[
\text{while } b \text{ do } s, \rho, \sigma \xrightarrow{\epsilon} \text{skip, } \rho, \sigma \quad \text{if } [b]_{\rho} = \text{false} \quad \text{(loop)}
\]

\[
\text{if } b \text{ then } s_1 \text{ else } s_2, \rho, \sigma \xrightarrow{\epsilon} s_1, \rho, \sigma \quad \text{if } [b]_{\rho} = \text{true} \quad \text{(cond)}
\]

\[
\text{if } b \text{ then } s_1 \text{ else } s_2, \rho, \sigma \xrightarrow{\epsilon} s_2, \rho, \sigma \quad \text{if } [b]_{\rho} = \text{false} \quad \text{(cond)}
\]

Figure 2: Intra-thread

\[
K, B^r, t \vdash_r \phi \cdot (C[s], \rho)^{t_0}, \sigma \xrightarrow{\alpha} t_0 \phi \cdot (C[s'], \rho')^{t_0}, \sigma' \quad \text{if } s, \rho, \sigma \xrightarrow{\alpha} s', \rho', \sigma' \quad \text{(intra)}
\]

\[
K, B^r, t \vdash_r \phi \cdot (C[y := \text{fork}(m, e)], \rho)^{t_1}, \sigma \xrightarrow{\text{fork } t_2} t_1 \phi \cdot (C[\text{skip}], \rho[y \mapsto t_2])^{t_1} \cdot (c, [x \mapsto v])^{t_2}, \sigma \quad \text{if } [\epsilon]_{\rho} = v, m(x) \ c \text{ is defined in } r, \text{ and } \text{fresh}_{K, r}(C[y := \text{fork}(m, e)], \rho)^{t_1}, B^{r}(t_2) \quad \text{(fork)}
\]

\[
K, B^r, t \vdash_r \phi \cdot (C[\text{join } e], \rho)^{t_1}, \sigma \xrightarrow{\text{join } t_2} t_1 \phi \cdot (C[\text{skip}], \rho)^{t_1}, \sigma \quad \text{if } [\epsilon]_{\rho} = t_2 \text{ and } \text{term}_{K, r}(C[\text{join } e], \rho)^{t_1}, B^{r}(t_2) \quad \text{(join)}
\]

Figure 3: Inter-thread
5 Properties of the Semantics

We expressed some well-formed properties on traces in Section 3. These properties can be viewed as a specification for operational semantics and implementations. We want to show that this specification is satisfied by the operational semantics we proposed. Our main theorem states that every trace produced by a program verifies the well-formedness conditions. Let us begin with additional definitions.

We say that a bubble $B$ is reachable by a program $r$ starting from the initial state $Σ_i(r)$ and generating a trace $s$, if there is a sequence of reduction where the sequence of events is equal to $s$, starting from $Σ_i(r)$ which ends to a state where $B$ is the first component. It is noted $\lceil B \rceil_s$. We define the bubble reachable in $i$ as the bubble reached after the $i^{th}$ event of a trace $s$ produced by a program $r$ starting from $Σ_i(r)$. It is noted $\lceil B \rceil_i$. If there is no specification of the starting state, it is the initial state. If the context is clear we will not specify the program, and sometimes not the trace but simply the position and write $B'$.

We define the characteristic sequence of a bubble $B$, noted $cs(B)$, as the list of pairs (thread owner, section name) of each level except the top-level one. We excluded the top-level because it has some specificity (there is no open for the section or fork for the thread).

\[
\begin{align*}
\text{cs}(\phi; B_0) &= \epsilon \\
\text{cs}(\phi; B_0 \circ \cdot t) &= cs(B_0)
\end{align*}
\]

Figure 4: Atomic section

For the proof of the conditions, we need the following lemmas:

**Lemma 1** For all program $r$, trace $s$, positions $i, j$, section name $p$ such as $s$ is the trace generated by the execution of the program $r$ starting from the initial state, and range $p$, $i$, $j$, we have for all position $k$ such as $i < k < j$ if it is a closed-range, or $i < k \leq j$ if it is a opened-range, that the characterised sequence $cs(\lceil B \rceil_k)$ is a prefix of $cs(\lceil B \rceil_j)$.

**Sketch of Proof:** The proof is done by induction on $k$. For the base case where $k = 0$, if $i = 0$ then $cs(\lceil B \rceil_j) = cs(\lceil B \rceil_0)$ and so the relation prefix holds, else if $i \neq 0$ then the hypothesis $i \leq k$ is contradict.

For the inductive case, if $k = i$ then $cs(B^i) = cs(B^k)$ and so the prefix relation holds. Otherwise, since $i < k < j$, we have $i \leq (k - 1) < j$, and so the induction hypothesis can be used on $(k - 1)$.

We need to reason on the type of the action done on $k$. If the action is not an open or a close, the characterised sequence is not modified, so prefix still holds.

- For the rule open, there exists a pair $r$ such as $cs(B^k) = cs(B^{k-1}) \cdot r$. So $cs(B^{k-1})$ is a prefix of $cs(B^k)$ and then by transitivity $cs(B^i)$ is a prefix of $cs(B^k)$.
- Now we have $\pi_{\text{act}}(k) =$ close $p'$. We examine if $cs(B^i)$ is a prefix strict of $cs(B^{k-1})$ or not.
\[ ((x := allocate(1); x[0] := 1; u := fork(m, x); atomic \{ x[0] := x[0] + 1; join u \}, \emptyset)^t; o \}_{\ell}^{r,t}, \emptyset, \{ r \}) \]

\[ alloc \ell 1 \]

\[ ((x[0] := x[0] + 1; u := fork(m, x); atomic \{ x[0] := x[0] + 1; join u \}, [x \mapsto \ell])^t; o \}_{\ell}^{r,t}, [\ell \mapsto 0], \{ r \}) \]

\[ write \ell 0 1 \]

\[ ((u := fork(m, x); atomic \{ x[0] := x[0] + 1; join u \}, [x \mapsto \ell])^t; o \}_{\ell}^{r,t}, \sigma : [\ell \mapsto 1], \{ r \}) \]

\[ fork t2 \]

\[ (atomic \{ x[0] := x[0] + 1; join u \}, [u \mapsto t2, x \mapsto \ell]^t \cdot (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; o \}_{\ell}^{r,t}, [\ell \mapsto 1], \{ r \}) \]

\[ open r2 \]

\[ (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; \quad (atomic \{ x[0] := x[0] + 1, [u \mapsto t2, x \mapsto \ell] \})^t; o \}_{\ell}^{r,t}, \ell \mapsto 1, \{ r, r2 \}) \]

\[ read \ell 0 1 \]

\[ (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; \quad (atomic \{ x[0] := x[0] + 1, [u \mapsto t2, x \mapsto \ell] \})^t; o \}_{\ell}^{r,t}, \ell \mapsto 1, \{ r, r2 \}) \]

\[ write \ell 0 2 \]

\[ (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; \quad (atomic \{ x[0] := x[0] + 1, [u \mapsto t2, x \mapsto \ell] \})^t; o \}_{\ell}^{r,t}, \ell \mapsto 2, \{ r, r2 \}) \]

\[ open r2 \]

\[ (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; o \}_{\ell}^{r,t}, \ell \mapsto 2, \{ r, r2 \}) \]

\[ close r2 \]

\[ (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; o \}_{\ell}^{r,t}, \ell \mapsto 2, \{ r, r2, r3 \}) \]

\[ read \ell 0 2 \]

\[ (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (atomic \{ x[0] := x[0] + 1, [x \mapsto \ell] \}^{t2}; o \}_{\ell}^{r,t}, \sigma : [\ell \mapsto 2], \{ r, r2, r3 \}) \]

\[ close r2 \]

\[ (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t; o \}_{\ell}^{r,t}, \ell \mapsto 2, \{ r, r2, r3 \}) \]

\[ write \ell 0 3 \]

\[ (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t; o \}_{\ell}^{r,t}, \ell \mapsto 3, \{ r, r2, r3 \}) \]

\[ close r3 \]

\[ (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t \cdot (join u, [u \mapsto t2, x \mapsto \ell])^t; o \}_{\ell}^{r,t}, \ell \mapsto 3, \{ r, r2, r3 \}) \]

\[ join t2 \]

\[ (skip, [u \mapsto \ell])^t \cdot (skip, [x \mapsto \ell])^t; o \}_{\ell}^{r,t}, \ell \mapsto 3, \{ r, r2, r3 \}) \]

\[ join t2 \]

\[ (skip, [u \mapsto \ell])^t \cdot (skip, [x \mapsto \ell])^t; o \}_{\ell}^{r,t}, \ell \mapsto 3, \{ r, r2, r3 \}) \]

Figure 5: Example of reduction
contradiction. Let the reduction rule $K a$ now that $s$ trivial. We have to prove the condition on the trace $s$, as follows:

- $cs(B^i) = cs(B^{k-1})$: in this case, we have by (close) that $p' = p$. Now let’s see which range we have:
  - closed-range: since $k \neq j$ we have two close $p$, this is in contradiction with (wf).
  - opened-range: $\pi^w(s) = close p$ is in contradiction with the definition of an opened-range.
- $cs(B^i)$ is a strict prefix of $cs(B^{k-1})$, and thanks to (close) we know that $cs(B^k)$ is equal to the sequence of $cs(B^{k-1})$ where the last element is removed, so $cs(B^i)$ is a prefix of $cs(B^k)$.

\[ \square \]

Lemma 2 For all section name $p$, thread $t$, and a bubble $B$ reachable by a program $r$ producing a trace $s$ starting from initial state, such as $(t, p) \in cs(B)$, then we have $owns_{s, p} t$.

Sketch of Proof: The proof is done by induction on trace $s$. The base case is trivial. Then we have our induction hypothesis for $B_s$, we have to prove it for $B_{s(t_1, a_1)}$. We examine the action $a_1$. If it is not a close or an open, then $B_s = B_{s(t_1, a_1)}$ and we can conclude. For the other cases:

- $a_1 = close p'$: thanks to the rule (close) we have $cs(B_{s(t_1, a_1)}) \cdot (t_1, p') = cs(B_s)$, and so we can use our induction hypothesis to conclude.
- $a_1 = open p'$: We conclude thanks to the rule (open) that $owns_{s(t_1, a_1), p'} t_1$ and for the other pair in $cs(B)$ we use the induction hypothesis.

\[ \square \]

Lemma 3 For all program $r$, trace $s$, bubble $B$ such as $B = \tau B_s$, and for all sequence $l$ such as $l$ is a prefix of $cs(B)$, then for all $(t, p) \in l$ and for all $(t', p') \in cs(B)$ and not in $l$, we have $tribe_{s, p} t'$.

Sketch of Proof: The proof is done by induction on the trace $s$. The base case is trivial. Then we have our induction hypothesis for $B_s$, we have to prove it for $B_{s(t_1, a_1)}$. We examine the action $a_1$. If it is not a close or an open, $B_s = B_{s(t_1, a_1)}$, and we can conclude. Let examine the other cases:

- $a_1 = close p'$: we have thanks to the rule (close), $cs(B_{s(t_1, a_1)}) \cdot b = cs(B_s)$, and so every list which is prefix of $cs(B_{s(t_1, a_1)})$ is also a prefix of $cs(B_s)$, so we can use our induction hypothesis to conclude.
- $a_1 = open p'$: let the last element of $cs(B_s)$ be $(t_0, p_0)$ (if not such pair exists, then $cs(B_{s(t_1, a_1)}) = (t_1, a_1)$ and it’s easy to conclude). Then thanks to (open) and by definition of tribe we have $tribe_{s(t_1, a_1), p_0} t_1$. So we can conclude thanks to induction hypothesis, and by transitivity of tribe.

\[ \square \]

Proposition 1 Every trace $s$ produced by a program $r$ starting from the initial state satisfies condition (wf), i.e. that every action open $p$, close $p$ and fork $t$ occurs at most once in $s$.

Sketch of Proof: The proof is done by induction on the trace $s$. The case where the trace is empty is trivial. We have to prove the condition on the trace $s \cdot (t, a)$, with the induction hypothesis on the trace $s$. If $a$ is not an open $p$, close $p$ or fork $t$ or doesn’t occur in $s$, the proof is immediate. We suppose now that $a$ is an open $p$, close $p$ or fork $t$ and happens in the trace $s$. We want to show that there is a contradiction. Let the reduction rule $K \vdash B_0, \sigma_0, P_0 \xrightarrow{a} B_1, \sigma_1, P_1$ which produces this action and let the last reduction rule be $K' \vdash B_0', \sigma_0', P_0' \xrightarrow{a} B_1', \sigma_1', P_1'$. We proceed by case on $a$:

- $a = fork t'$: from the reduction rule (fork) applied to our last reduction, we have the hypothesis $fresh_{K', \phi(0), B_0', t_0}(t')$ (where $B_0' = (\phi_0; B_0 \xrightarrow{a} t_0)$). We know that $fresh_{K, \phi_1, B_1, t_0}(t')$ (where $B_1 = (\phi_1; B_1 \xrightarrow{a} t_0)$) doesn’t hold, since $t'$ is added to the set of threads. Threads are never removed from their set, any of them which is not fresh at some state, will stay not fresh to any further state. As $B_0'$ can be reached from $B_1$, and $t'$ is not fresh in $B_1$, we can conclude that $t'$ is not fresh in $B_0'$. This is in contradiction with the rule (fork) applied to the last reduction.
• $a = \text{open } p$: from the rule $\text{open}$ we have $\text{fresh}_{P_0}(p)$ and $\neg \text{fresh}_{P_1}(p)$. The set of bubble’s section name never shrinks during the reduction. It means that if a section name is not fresh at some state, it will still be not fresh to any further state. So $p$ must not be fresh in $P'_0$ since we can reach it from $P_1$. This is a contradiction with the rule $\text{open}$ applied to the last reduction.

• $a = \text{close } p$: as the section has just been closed, we have $p \not\in B_1$. For all bubble $B$ reachable from $B_1$ we have $p \not\in B$ because section names are unique. So we have $p \not\in B'_0$ since $B'_0$ can be reached from $B_1$. However the rule $\text{close}$ applied to last reduction allows us to see that $p \in B'_0$, that leads us to a contradiction.

Proposition 2 Every trace $s$ produced by a program $r$ starting from the initial state satisfies condition $\text{wf}_2$:

$$\forall i, p. \pi^s(i) = \text{close } p \Rightarrow \exists j. j < i \land \pi^s(j) = \text{open } p \land \pi^s(i) = \pi^s(j).$$

Sketch of Proof: Let the section name $p$ and the position $i$ such as $\pi^s(i) = \text{close } p$. We have to find a position $j$ such as $j < i$, $\pi^s(j) = \text{open } p$ and $\pi^s(j) = \pi^s(i)$.

We proceed by induction on the trace $s$. The base case is trivial. If we have $i < |s|$, we can use our induction hypothesis to conclude. Now we suppose that the position $i = |s|$. So we have the last reduction rule $K \vdash B_0, \sigma_0, P_0 \xrightarrow{\text{open } p} B_1, \sigma_1, P_1$, and we can state that the section name $p \in B_0$.

For all bubble $B$, program $r'$ and trace $s'$ such as $^{B,s'}$, and for all section name $p'$ such as $p' \not\in B$, we prove thanks to an induction on the trace $s$ that there exists a position $j$ which introduce this section name by an $\text{open }$ reduction.

So there exists a position $j$ which introduces this section name by a reduction $K' \vdash B'_0, \sigma'_0, P'_0 \xrightarrow{\text{open } p} B'_1, \sigma'_1, P'_1$. We still have to prove that $t' = t$. For each bubble, we can associate a section name to the thread owner, and this association never change. So we have $p$ associated to $t'$ in $B'_1$ and to $t$ in $B_0$, and thus we have $t' = t$.

Proposition 3 Every trace $s$ produced by a program $r$ starting from the initial state satisfies condition $\text{wf}_3$:

$$\forall p, i, j. \text{range}_{s,p} i j \Rightarrow \pi^s(i) = \text{open } p \Rightarrow \pi^s(j) = \text{close } p \Rightarrow \forall k, p'. i < k < j \Rightarrow \pi^s(i) = \pi^s(k) \Rightarrow \pi^s(k) = \text{open } p' \Rightarrow \exists j', k < j' < j \land \pi^s(j') = \text{close } p'.$$

Sketch of Proof: Let the positions $i, i', j, j'$ and the section names $p, p'$ such as $\pi^s(i) = \text{open } p$, $\pi^s(i') = \text{open } p'$, $\pi^s(i) = \pi^s(i')$, $i < i' < j$ and $\pi^s(j) = \text{close } p$. We have to prove that there exists a position $j'$ such as $\pi^s(j') = \text{close } p'$ and $i < j < j'$.

We suppose that there exists a position $j'$ such as $\text{range}_{s, p'} i' j'$ and $j < j'$, and prove there is a contradiction.

We know thanks to Lemma 1 that $\text{cs}(B^i)$ is a prefix of $\text{cs}(B^{j-1})$ and $\text{cs}(B^i)$ is a prefix of $\text{cs}(B^j)$.

Then prove that $\text{cs}(B^i) = \text{cs}(B^{j-1})$ with $\text{open}$ and $\text{close}$. So we have that $\text{cs}(B^i)$ is a prefix of $\text{cs}(B^j)$. We know that $p \not= p'$ thanks to $\text{wf}_1$, and so $\text{cs}(B^i) \neq \text{cs}(B^j)$. Thus $\text{cs}(B^j)$ could not be a prefix of $\text{cs}(B^j)$ and $\text{cs}(B^j)$ be a prefix of $\text{cs}(B^j)$.

Proposition 4 Every trace $s$ produced by a program $r$ starting from the initial state satisfies condition $\text{wf}_4$:

$$\forall i, t. \pi^s(i) = \text{fork } t \Rightarrow \forall j. \pi^s(j) = t \Rightarrow i < j.$$
Sketch of Proof:
Let the thread identifier \( t \) and the positions \( i \) and \( j \) such as \( \pi_s^\pi(i) = \text{fork} \ t \) and \( \pi_s^\pi(j) = t \). We have to prove that \( i < j \). The proof is done by induction on the trace \( s \). The case where the trace is empty is trivial. Let the reduction rules \( K, B_0^0, t_0 \vdash \phi_0, \sigma_0 \xrightarrow{\text{fork}} \phi_1, \sigma_1 \) and \( K', B_0^0, t_0 \vdash \phi_0', \sigma_0' \xrightarrow{t} \phi_1', \sigma_1' \) producing respectively the events at position \( i \) and \( j \).

We proceed by examining the position of \( i, j \) in the trace. If \( i < |s| \) we use our induction hypothesis. Now let us focus when \( i = |s| \), and show that it cannot hold:

- \( j < |s| \): from the rule \( \text{fork} \), we know that \( \text{fresh}_{K, \phi_0, B_0^0, t_0}(t) \) holds and \( \text{fresh}_{K', \phi_0', B_0^0, t_0}(t) \) doesn’t. A thread which is not fresh at some state will stay not fresh for the next states. Since the state of \( B_0^0 \) reachable from the state of \( B_0^0' \) there is a contradiction.

- \( j = |s| \): the last event is \((t, \text{fork} \ t)\), a thread cannot fork a thread with the same identifier, it is not allowed by the predicate \( \text{fresh} \).

\[\square\]

**Proposition 5**
Every trace \( s \) produced by a program \( r \) starting from the initial state satisfies condition \( \text{wf}_k \):

\[
\forall i, t. (\pi_s^\pi(i) = t \lor \pi_s^\pi(i) = \text{fork} \ t) \Rightarrow \\
\forall j. \pi_s^\pi(j) = \text{join} \ t \Rightarrow i < j.
\]

**Sketch of Proof:**
Let the thread identifier \( t \) and the positions \( i \) and \( j \) such as \( \pi_s^\pi(i) = t \) or \( \pi_s^\pi(i) = \text{fork} \ t \), and \( \pi_s^\pi(j) = \text{join} \ t \). We need to prove that \( i < j \). Let the reduction rules \( K \vdash B_0, \sigma_0, P_0 \xrightarrow{at} \sigma_1, P_1 \) where \( B_0 \) is reachable from the state of \( B_0' \) there is a contradiction.

- \( i = j \): we need to distinguish two cases according to the hypothesis we have on \( i \):
  - \( \pi_s^\pi(i) = t \): We can state that \( B_0 = B_0' \) and \( B_1 = B_1' \). The reduction rule \( \text{join} \) asserts that the thread \( t \) must verify \( \text{term}_{K', \phi_0', B_0', t_0} \) (where \( B_0 = \{ \phi_0; B_0' \} \) is), and here it is clearly not the case, since it is the thread \( t \) which does the action.
  - \( \pi_s^\pi(i) = \text{fork} \ t \): there is a contradiction on the action done.

- \( i > j \): from the rule \( \text{join} \) applied to the reduction associated to the position \( j \), we can state that \( \text{term}_{K', \phi_0', B_0', t_0} \) (where \( B_0 = \{ \phi_0; B_0' \} \) is), and so no other action realised by this thread \( t \) can be done later. Furthermore, \( t \) cannot be fresh for the next bubbles. This is in contradiction with the hypotheses on the position \( i \).

\[\square\]

**Proposition 6**
Every trace \( s \) produced by a program \( r \) starting from the initial state satisfies condition \( \text{wf}_k \):

\[
\forall p, i, j, t. \text{range}_{s, p} i j \Rightarrow \\
\text{owns}_{s, p} t \Rightarrow \\
\forall k. \pi_s^\pi(k) = \text{join} \ t \Rightarrow j < k.
\]

**Sketch of Proof:**
Let the section name \( p \), the positions \( i, j, k \) and the thread identifier \( t \) such as \( \text{range}_{s, p} i j \), \( \text{owns}_{s, p} t \) and \( \pi_s^\pi(k) = \text{join} \ t \). We need to prove that \( j < k \).

Let the reduction rule \( K_1, B_1, t_1 \vdash \phi_1, \sigma_1 \xrightarrow{\text{join} \ t} \phi_2, \sigma_2 \) producing the event at position \( k \). We proceed by case on the form of \( \text{range}_{s, p} i j \).

- opened range: so \( j = |s| \), and then \( j < k \) is impossible. We need to prove that we have a contradiction. To do so, we examine the relative position between \( k \) and \( i \):
\[ i < k: \] for all bubble \( B \) reachable from \( i \), we know that \( p \in B \) since there is no \texttt{close} \( p \), and this section name is associated to the thread identifier \( t \) because we have \texttt{owns}_{s,p} t. From the rule \( \texttt{join} \), we deduce that \( t \) owns no section. That leads us to a contradiction.

\[ i > k: \] from the rule \( \texttt{join} \), we know that the statement associated to \( t \) is \texttt{skip}. This association will never change. However we also have \( \pi_s(i) = (t, \text{open } p) \), so \( t \) is associated with \texttt{atomic} statement. That leads us to a contradiction.

- closed range: let suppose that \( k < j \), and we prove that we can deduce two conclusions in contradiction, from the two rules:

  - from the rule \( \texttt{join} \), we know that \( \text{term}_{K_i, \phi_1, B_1, t_1}(t) \) (where \( B_1 = \{ \phi_1; B_1 \} \)), and so that \( t \) is not the owner of any section at this point. As \( t \) is associated to the command \texttt{skip}, \( t \) will not be the owner of any section in the future.

  - from the rule \( \texttt{close} \), we know that \( t \) is the owner of the section \( p \).

These two conclusions are in contradiction, so \( j < k \).

\[ \Box \]

**Proposition 7** Every trace \( s \) produced by a program \( r \) starting from the initial state satisfies condition \texttt{w1}:

\[ \forall t, i, j. \pi^\text{out}(i) = \text{fork } t \Rightarrow \pi^\text{out}(j) = \text{join } t \Rightarrow \text{see}_{s} i j. \]

**Sketch of Proof:** Let the positions \( i, j \) and the thread identifier \( t \) such as \( \pi^\text{out}(i) = \text{fork } t \) and \( \pi^\text{out}(j) = \text{join } t \). We have to prove that \( \text{see}_{s} i j \). Let the thread identifier \( t_0 \) such as \( \pi^\text{out}(j) = (t_0, \text{join } t_0) \).

We note \( t' \in B \) if \( t \in B \) and \( \exists x \cdot \sigma(x) = t' \) where \( \sigma \) is the local store of \( t \). We also note \( \text{seeFork}_{s, t} t' j \) if there exists a position \( i \) such as \( \pi^\text{out}(i) = \text{fork } t' \), \( \pi^\text{out}(j) = t \vee \pi^\text{out}(j) = \text{fork } t \) and \( \text{see}_{s} i j \).

First we prove that for all bubble \( B \) reachable by the execution of a program which generate a trace \( s \), and for all threads \( t, t' \) such as \( t' \in B \), there exists a position \( j \) such as \( \text{seeFork}_{s, t} t' j \). We prove it by a generalised induction on the trace \( s \). The base case is trivial.

Let the bubble \( B' \) reached by the trace \( s \). Then if \( t' \in B' \) holds and we can apply the induction hypothesis. Otherwise we have to reason by case on the last action. Four rules can modify the local environment. The rule \( \text{assign} \) is excluded, because the expression cannot be evaluated to new thread, due to the restriction on the user’s constants. The rule \( \text{alloc} \) is not possible because it only adds new locations. So the last action is either done by rule \( \text{get} \) or rule \( \text{fork} \). More formally there exists \( t, n \) such as the last event is \( (t, \text{read } t \ n \ t') \) or a there exists \( t_0 \) such as the last event is \( (t_0, \text{fork } t) \) and \( t' \in B \).

- \( (t, \text{read } t \ n \ t') \): if a value is read, it has been necessarily written before, so there exists \( k_1, t_3 \) such as \( \pi^\text{out}((t, \text{read } t \ n \ t')) = (t_1, \text{write } t \ n \ t') \). Thanks the induction hypothesis, there exists a position \( j_0 \) such as \( \text{seeFork}_{s, (t, \text{read } t \ n \ t')}(t_1, t_0) \). Let \( i_0 \) such as \( \pi^\text{out}((t, \text{read } t \ n \ t')) = (t_0, \text{fork } t') \).

Then we want to prove that \( \text{see}_{s, (t, \text{read } t \ n \ t')}(j_0) \). We have by definition \( \text{see}_{s, (t, \text{read } t \ n \ t')}(j_0) \) and \( \text{see}_{s, (t, \text{read } t \ n \ t')}(k_1) \). We conclude by transitivity.

We conclude with the positions \( |s| \) and \( i_0 \).

- \( (t_0, \text{fork } t) \): thanks to the induction hypothesis there exists positions \( j_0, i_0 \) such as \( \text{seeFork}_{s, (t_0, \text{fork } t)} t_0 \) \( j_0 \) and \( \pi^\text{out}((t_0, \text{fork } t)) = (t_0, \text{fork } t') \). So we conclude with \( |s| \) and \( i_0 \).

We know that we have \( t \in B \) thanks to \( \text{join} \), so there exists a position \( k \) such as \( \text{seeFork}_{s, t} t_0 k \). We have \( \text{see}_{s} i k \) from this last result and \( \text{see}_{s} k j \) by definition. We conclude by transitivity.

\[ \Box \]

**Proposition 8** Every trace \( s \) produced by a program \( r \) starting from the initial state, satisfies condition \texttt{w1}:

\[ \forall p, p'. \text{open } p \in s \Rightarrow \text{open } p' \Rightarrow \text{see}_{s} p p' \Rightarrow \text{see}_{s} p p'. \]
These two cases being symmetric, we examine only the first one. So we suppose that $i < j$, otherwise $p = p'$, which is in contradiction with $p \prec_s p'$. We also know that $i \leq j$ and $i' \leq j'$ thanks to (wf₂) and (wf₁).

If $j < i'$ or $j' < i$ we can conclude. The problematic cases occur when $i < i' \leq j$ or $i' < i \leq j'$. These two cases being symmetric, we examine only the first one. So we suppose that $i < i' \leq j$, and we prove that it leads to a contradiction. We have that $cs(B')$ is a prefix of $cs(B')$ by Lemma [3]. Thanks to (open) we prove that the last element of $cs(B')$ is $(t, p)$ and the last of $cs(B')$ is $(t', p')$. Thus we have $tribe_{s, p} t'$ by Lemma [3]. Then we prove that we have $p \in_s p'$ by definition of $\in$. This last conclusion is in contradiction with our hypothesis $p \sim_s p'$.

\[\square\]

**Theorem 1** Every trace produced by a program verifies the well-formedness conditions.

**Sketch of Proof:** By Proposition [1] to [8].

From [3], we know that well-formed and well-synchronised traces are equivalent to a serial trace. As usual a trace is well-synchronised when conflicting actions are synchronised. A serial trace is a trace where for all section, all the running threads in the range of the section belongs to the tribe of the section. Therefore as AFJ operational semantics produces only well-formed traces, we know that all executions of a well-synchronised AFJ program are equivalent to a serial trace.

## 6 Related Work

Several implementations of atomic section have been developed, mainly using transactions. The first ones used hardware [9], then becoming purely software [17, 10], or a mixed of both [4]. However these systems are often limited by forbidding or restraining the inner parallelism and so the nesting. The implementations of transactions can use logs [5] to save the modifications (which cells have been updated), or can use transactional objects [10] which are copies of the real objects where the modifications are done. The detection of interference can be done at the end of the transaction or sooner, at each modification. If the transaction ends without encounter any interference, the modifications are reported to the real heap/store/object.

Harris et al. [8] propose to improve the composability by adding new constructions like the possibility to specify another atomic section to execute in case of failure of the first one. But there is neither inner parallelism nor nesting of atomic sections.

Moss et al. [14] state like us that nesting is necessary in atomic sections, by pointing out the example of library use. They also show that with nesting, atomic sections are easier to handle, because they are lighter: if an interference occurs at a sub-transaction, only this one is cancelled and not the whole transaction. However, the parallelism is restricted: It is allowed only for the top-level transaction. This restriction is done for implementation purpose.

Agrawal et al. [1] propose an implementation based on parallel composition instead of using threads. They thus deal with a more structured form of parallelism. Their representation of processes and atomic sections can be viewed as a tree, presenting some similarities with our bubble structure. They combine parallelism and atomic section, but each child section must terminate before the surrounding section.

In [11] Jagannathan et al. show an operational semantics of a derived version of Featherweight Java which supports atomic nesting and inner-parallelism. Like us in [3] they use program traces to prove serialisability i.e. for any abort-free program, there must be a corresponding trace where atomic sections are executed serially. However in their work, each child section must live inside the parent section.

In [13], Moore et al. introduce a language with four variations with their operational semantics to focus on the meaning of atomicity. Different fork instructions have different behaviours with respect to atomic sections. Our proposal is more permissive as more general behaviour is allowed for only one fork primitive.
7 Conclusion and Future Work

We designed an operational semantics for an imperative language with fork/join parallelism and lexically scoped atomic sections. This language supports nesting of atomic sections and thread escape. The design of this semantics was guided by well-formedness conditions on program traces that we defined to clarify the meaning of atomicity in such a context. We formally proved that the traces generated by the operational semantics indeed satisfy these well-formedness conditions.

We used the interactive theorem prover Coq to define our language and our semantics. We also use it to (partially) prove the matching between our semantics and the specifications. We plan to write a compiler toward a language where the atomic sections are replaced with locks, and prove a form of semantics preservation.

Acknowledgements

Thomas Pinsard is 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 ANR-2010-INTB-0205-02 and JST 10102704).

References


