4.2 Advanced Manipulation with Locks

The original version of Atomer does not distinguish *different lock instances* in a program. ~~Only~~ calls of locks/unlocks are identified, ~~and~~ the parameters of these calls — *lock objects* — are not considered. Thus, if there are several lock objects used, the analysis does not work correctly.

In order to consider lock objects, ~~it was proposed~~ to distinguish between them using Facebook Infer's built-in mechanism called *access paths* [22], explained below. The analyser does not perform a classical *alias analysis*, i.e., it does not perform a precise analysis for saying when arbitrary pairs of accesses to lock objects may alias (such an analysis is considered too expensive).

Access Paths ~~The~~ *syntactic access paths* [22] represent *heap locations* via the paths used to access them, i.e., they have the form of an expression consisting of a base variable followed by a sequence of fields. More formally, let Var be a set of all variables that can occur in a given program. Let $Field$ be a set of all possible field names that can be used in the program (e.g., structure fields). An access path π from the set Π of all access paths is then defined as follows:

$$\pi \in \Pi ::= Var \times Field^*$$

Access paths are already implemented in Facebook Infer. For instance, the principle of using access paths is used in ~~an existing~~ analyser in Facebook Infer — RacerD [3, 4, 14] — for data race detection. ~~In general,~~ no sufficiently precise *alias analysis* works *compositionally* and *at scale*. That is the motivation for using access paths in ~~Facebook Infer~~.

Given a pair of accesses to lock objects, to determine whether these locks are equal, it is needed to answer the following question: "Can the accesses touch the same address?". Remarkably, according to the authors of [3], access paths alone *almost* convey enough semantic information to answer the above question on their own. If two access paths are syntactically equal, it is almost (but not quite) true that they must refer to the same address. Syntactically identical paths can refer to different addresses if (i) they refer to different instances of the same object, or (ii) a prefix of the path is reassigned along one execution trace but not the other. These conditions ~~cannot~~ do not hold if an access path is *stable*, i.e., if none of its proper prefixes appears in assignments during a given execution trace, then it touches the same memory as all other stable accesses to the syntactic path. Therefore, the access paths' syntactic equality is a reasonably efficient way to say (in an *under-approximate fashion*) that heap access ~~touches~~ the same address. Also, by using access paths, RacerD detected many errors in real-world programs, proving that the use of access paths can reveal real errors. This is why it was decided to use this principle to represent locks in Atomer.

During the analysis performed by Atomer (in both phases), each atomic section is identified by an access path of the lock that guards the section; see Sections 4.2.1, 4.2.2. Because *syntactically identical access paths* are used as the means for distinguishing atomic sections, some atomicity violations could be missed (or some false alarms could be reported) due to distinct access paths that refer to the same memory. However, the analysis's precision is still significantly improved this way while preserving its *scalability*, and the stress is anyway put on finding likely violations, not on being *sound*.

Another limitation of Atomer in its basic version is that it does not count with *re-entrant locks* when a process can lock the same object multiple times without blocking itself, and then it should unlock the lock object the same number of times. This approach is, in fact, widespread, e.g., in Java, where so-called *synchronised blocks* are used, as demonstrates Listing 4.2. These blocks are re-entrant by default. To consider re-entrant locks in the analysis, the number of locks of individual lock objects is tracked in the abstract states of both phases of the analysis. A lock is unlocked as soon as this number decreases to 0. Also, an input parameter $t \in \mathbb{N}$ was proposed to limit the *upper bound* to which the analysis tracks precisely the number of times a given lock is locked. When this bound is reached, the *widening operator* is used to abstract the number to any value bigger than the bound. This is to ensure *termination* of the analysis. The idea of this upper bound limit comes from the approach used in RacerD.

```
1 public synchronized void f() {
2     x();
3     synchronized (this) {
4         a();
5         synchronized (this) { b(); c(); }
6         d();
7     }
8     y();
9 }
```

Listing 4.2: An example of *re-entrant locks* in Java using the `synchronized` keyword, which is implemented as a *monitor*. In the example, there are three locks used simultaneously over the same object. The entire method `f` is synchronised, which implicitly uses `this` as a *lock object*. Furthermore, the two *synchronised blocks* explicitly use the lock object `this`.

4.2.1 Advanced Manipulation with Locks in Phase 1

Recall that the *detection of sets of calls to be executed atomically* is based on generating the pairs $(A, B) \in 2^{\Sigma} \times 2^{\Sigma}$. Now, these pairs are to be extended to store the access paths and the number of locks of lock objects that guard calls executed atomically, i.e., the B sets. Therefore, the (A, B) pairs are extended to tuples $(A, B, \pi, l) \in 2^{\Sigma} \times 2^{\Sigma} \times \Pi \times \mathbb{N}^{\top}$ where π is an access path that identifies a lock object that locks the atomic section that contains the calls from B, and l is the number of locks of the lock identified by π. For the clarity of the below description, let π be just a base variable, i.e., $\pi \in \Pi ::= Var \cup \{\varepsilon\}$. Note that π could also be ε, which is a special case when there is no lock associated with the (A, B) pair so far, i.e., B is empty and a lock was not called yet. $\mathbb{N}^{\top}$ denotes $\mathbb{N} \cup \{\top\}$, where $\top$ represents a number larger than t. Thus, the *abstract states* are elements of the set $2^{2^{\Sigma} \times 2^{\Sigma} \times \Pi \times \mathbb{N}^{\top}}$.

The analysis works as follows. When a function is called, it is appended to the A set of the element without an associated lock, i.e., the element where $\pi = \varepsilon$. Besides, the function is appended to the B sets of all the elements that have associated some lock that is currently locked, i.e., $\pi \neq \varepsilon$ and $l > 0$. When a lock is called, its identifier π_i is associated with the element without any lock associated with it, and the counter l of this element is set to 1. Then, l is incremented in all the elements where $\pi = \pi_i$ and $l > 0$. Moreover, it is created a new empty element without a lock. Finally, when an unlock with the identifier π_i is called,

l is decremented in all the elements where $\pi = \pi_i$ and $l > 0$. To formalise this process, let $inc_\top : \mathbb{N}^\top \to \mathbb{N}^\top$ and $dec_\top : \mathbb{N}^\top \to \mathbb{N}^\top$ be functions for incrementing and decrementing the number of locks of some lock objects w.r.t. the upper bound $\top$, respectively. These functions are defined as follows:

$$inc_\top(l) = \begin{cases} l+1 & \text{for } l \neq \top \text{ and } l+1 < t \\ \top & \text{otherwise} \end{cases} \qquad dec_\top(l) = \begin{cases} l-1 & \text{for } l \neq \top \text{ and } l-1 \geq 0 \\ 0 & \text{for } l = 0 \\ t-1 & \text{otherwise, i.e., } l = \top \end{cases}$$

The *initial abstract state* of a function is changed to $s_{init} = \{\{(\emptyset, \emptyset, \varepsilon, 0)\}\}$. During the analysis of a function $\mathtt{g}$ with an abstract state $s_\mathtt{g}$, when a leaf function $\mathtt{f}$ is called, the abstract state's transformation is changed as follows:

$$s'_\mathtt{g} = \{p' \in 2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top} \mid \exists p \in s_\mathtt{g} : p' = \{(A', B', \pi', l') \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top \mid \exists (A, B, \pi, l) \in p : [\pi = \varepsilon \wedge l = 0 \wedge (A', B', \pi', l') = (A \cup \{\mathtt{f}\}, B, \pi, l)] \vee [\pi \neq \varepsilon \wedge ([l > 0 \wedge (A', B', \pi', l') = (A, B \cup \{\mathtt{f}\}, \pi, l)] \vee [l = 0 \wedge (A', B', \pi', l') = (A, B, \pi, l)])]\}\}$$

Further, when a lock identified by an access path π_i is called, the abstract state changes as follows:

$$s'_\mathtt{g} = \{p' \in 2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top} \mid \exists p \in s_\mathtt{g} : p' = \{(A, B, \pi, l) \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top \mid (A, B, \pi, l) = (\emptyset, \emptyset, \varepsilon, 0) \vee [(A, B, \varepsilon, 0) \in p \wedge \pi = \pi_i \wedge l = inc_\top(0)] \vee [(A, B, \pi, l') \in p \wedge \pi \neq \varepsilon \wedge [((l' = 0 \vee \pi \neq \pi_i) \wedge l = l') \vee (l' > 0 \wedge \pi = \pi_i \wedge l = inc_\top(l'))]]\}\}$$

Furthermore, when an unlock identified by the access path π_i is called, the abstract state changes as follows:

$$s'_\mathtt{g} = \{p' \in 2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top} \mid \exists p \in s_\mathtt{g} : p' = \{(A, B, \pi, l) \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top \mid (A, B, \pi, l) = (\emptyset, \emptyset, \varepsilon, 0) \vee [(A, B, \pi, l') \in p \wedge \pi \neq \varepsilon \wedge [((l' = 0 \vee \pi \neq \pi_i) \wedge l = l') \vee (l' > 0 \wedge \pi = \pi_i \wedge l = dec_\top(l'))]]\}\}$$

Other algorithms (e.g., calling an already analysed *nested* function) are changed analogically.

A *summary* $\chi_\mathtt{f} \in 2^{2^\Sigma} \times 2^\Sigma$ of a function $\mathtt{f}$ is the same as earlier. Only access paths and lock counters from abstract states are ignored. This is, $\chi_\mathtt{f} = (\mathcal{B}, AB)$, where:

- $\mathcal{B} = \{B' \in 2^\Sigma \mid \exists p \in s_\mathtt{f} : \exists (A, B, \pi, l) \in p : B \neq \emptyset \wedge B' = B\}$, where $s_\mathtt{f}$ is the abstract state at the end of the abstract interpretation of $\mathtt{f}$.
- $AB = \bigcup_{ab \in AB'} ab$, where $AB' = \{ab \in 2^\Sigma \mid \exists p \in s_\mathtt{f} : \exists (A, B, \pi, l) \in p : ab = A \cup B\}$.

Example 4.2.1. Consider the function `f` from Listing 4.3. There are two lock objects `L1` and `L2`, which are used simultaneously. Moreover, `L2` is locked several times without unlocking in between. Further, assume that `a`, `b`, `c` are leaf nodes of the call graph. After the extension described above, the produced summary is as follows: $\chi_\mathtt{f} = (\{\{\mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. Without the extension, the summary would be $\chi'_\mathtt{f} = (\{\{\mathtt{a}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. The reason is that only the first locks/unlocks were detected. Other locks inside atomic sections and other unlocks outside atomic sections were ignored. Moreover, the abstract state after the execution of line 7 is as follows: $s_{\mathtt{f}_7} = \{\{(\emptyset, \{\mathtt{a}, \mathtt{b}\}, \mathtt{L1}, 1), (\{\mathtt{a}\}, \{\mathtt{b}\}, \mathtt{L2}, 2), (\{\mathtt{b}\}, \emptyset, \varepsilon, 0)\}\}$.

```
1  void f()
2  {
3      lock(&L1); // {a, b, c}
4      a();
5      lock(&L2); lock(&L2); // {b}
6      lock(&L2); unlock(&L2);
7      b();
8      unlock(&L2); unlock(&L2);
9      c();
10     unlock(&L1);
11 }
```

Listing 4.3: A code snippet used to illustrate the *advanced manipulation with locks* during the first phase of the analysis

4.2.2 Advanced Manipulation with Locks in Phase 2

The pairs Ω of functions that should be called atomically are computed the same way as earlier during the *detection of atomicity violations* in Phase 2. However, dealing with access paths and re-entrant locks must, of course, be reflected in the second phase of the analysis as well. For that, while looking for *atomicity violations of pairs of function calls*, from now, the analysis stores (in addition to pairs of the most recent function calls $(\mathtt{x}, \mathtt{y})$ and the set δ of pairs that have so far been identified as violating atomicity) all the most recent pairs of function calls locked under individual locks. Hence, the *abstract state* element gets the form $(\mathtt{x}, \mathtt{y}, \delta, \lambda) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}} \times 2^{\Sigma \times \Sigma \times \Pi \times \mathbb{N}^\top}$, where $(\mathtt{x}, \mathtt{y}), \delta$ are as before, and λ is the set of the most recent function calls with their lock access paths and the number of locks of lock objects of these locks. Thus, the abstract states are elements of the set $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}} \times 2^{\Sigma \times \Sigma \times \Pi \times \mathbb{N}^\top}}$.

The analysis works as follows. When a function `f` is called on some path that led to an abstract state $(\mathtt{x}, \mathtt{y}, \delta, \lambda)$, a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls is created from the previous pair $(\mathtt{x}, \mathtt{y})$ such that $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$. This pair is also stored in the locked pairs λ if there are any locks currently locked. Further, it is checked whether the new pair (or just the last call) violates the atomicity, and at the same time, the pair is not locked by any of the stored locks (i.e., $((\mathtt{x}', \mathtt{y}') \in \Omega \wedge \nexists\, (\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \lambda : (\mathtt{x}_\pi, \mathtt{y}_\pi) = (\mathtt{x}', \mathtt{y}')) \vee ((\varepsilon, \mathtt{y}') \in \Omega \wedge \nexists\, (\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \lambda : (\mathtt{x}_\pi, \mathtt{y}_\pi) = (\varepsilon, \mathtt{y}'))$). When the condition holds, the pair is added to the set δ of pairs that violate atomicity. When a lock with an identifier π_i is called, it is created a new empty element of λ with this identifier, and the lock counter l of this element is set to 1. Furthermore, the lock counter l of an element from λ with the access path π_i is incremented/decremented when a lock/unlock with the identifier π_i is called, respectively.

More formally, the *initial abstract state* of a function is defined as $s_{init} = \{\{(\varepsilon, \varepsilon, \emptyset, \emptyset)\}\}$. To formalise the analysis of a function, let `f` be a called leaf function on a line c. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called.

Poznámky ke 4.2.1

- Tam už není odlišení aktivních / současných n-tic? Nevede to k nějaké velké nadaproximaci?

  Např. u

  lock;
  f_1; f_2;
  unlock;
  f_3.
  lock;
  f_4; f_5
  unlock

  bude v $B = \{f_1, f_2, f_3, f_4\}$?

  ↳ Pokud možno nějak vysvětlit.

  [Nebo, pokud tam current má být, tak dodat]

- Proč se při každém lock / unlock vyhází $(\emptyset, \emptyset, \varepsilon, 0)$? Měla by tam být nějaká rubrice, absolutně to nechápu! (*) ↩

- U $dec_T(l)$ po $l = T$ by měly být dva případy: $t-1$ a T, jinak podaproximace. Nebo je to úmyslné? Vysvětlit!

4.2

– Proč je u formule popisující volání leaf f-ce jen případ pro $l=0$? Jak to může nastat?

– U lock formule absolutně nechápu člen $(l'=0 \vee \pi \neq \pi_i) \wedge l=l'$ – co to má dělat? Opravit/Vysvětlit!

– Podobně u unlock nechápu člen $(l'=0 \vee \pi \neq \pi_i) \wedge l=l'$.

– Jak se u unlock/lock řeší případy, kdy je zamknuto ale jiným zámkem?

(+) – Nový prvek $(0,0,\varepsilon,0)$ bych pochopil u lock, pokud tam je alespoň 1 n-tice, která nepoužívá daný zámek

– U unlock bych pochopil novou n-tici při úplném odemknutí a při rozlišování aktivních a neaktivních n-tic.

⇝ V současném stavu nechápu: VYSVĚTLIT!

(+) Nemělo by teď v tom $2^{2^{2^{x\ldots}}}$ být o jednu úroveň méně? Nejsou-li aktivní/neaktivní mn-y, jsou opravdu všechny úrovně využity? Maybe because of creating the new tuples $(0,0,\varepsilon,0)$?