Lock Analysis for an Asynchronous Object Calculus

Elena Giachino and Tudor A. Lascu

FOCUS Research Team INRIA / Dipartimento di Scienze dell'Informazione, Università di Bologna
[giachino | lascu]@cs.unibo.it

1 Introduction

The model. The problem we are interested in is discovering deadlocks and livelocks in an object-oriented setting, where method calls are asynchronous, meaning that after a method is invoked the caller does not wait for the returned value, instead it continues its activity until the result is strictly necessary. In this model [?,?] objects have multiple tasks in execution, spawned by method invocations, and there is at most one active task per object at each point in time. The active task may explicitly return the control in order to let another task of the same object progress. The decoupling of method invocation and returned value is realized by means of *future variables*, which are pointers to values that may be not available yet. Clearly, the access to values of future variables may require waiting for the value to be returned. In order to program in this model we introduce a language called *Featherweight Java with futures* (`FJf`) [?], inspired by `FJ` [?], that features two primitives for dealing with *future*s and control release. The `get` operation is used to retrieve a return value, keeping object's lock while waiting for it. The `await` operation, instead, is used to wait for the availability of a computation's result. If the result is not ready, the task suspends, by first releasing the lock of the object. If the result is ready, `await` is non-blocking but in order to retrieve the actual value it waited for, the task still needs to perform a `get`.

Deadlocks. In the considered model a typical deadlock occurs when one or more tasks are waiting for each other's termination to return a value. Let us consider a simple scenario with two objects `a` and `b` as in Fig. 1. Object `a` contains a task related to some method `m` which in turn invokes a method `n` on object `b` and explicitly blocks waiting for the result. This invocation triggers a task at `b` responsible of executing `n`'s body, which again performs an invocation on `a` waiting for the result. This last task inside `a` is created but it never obtains the lock (held by `m`) for executing its own code. The computation is deadlocked. Black arrows between objects correspond to *object dependencies*, introduced by `get` operations. A circular dependency means that a deadlock is encountered.

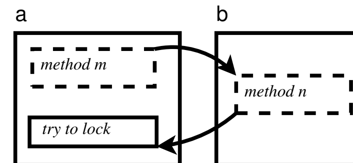

Fig. 1. A simple deadlock.

Livelocks. Let us consider a slightly different example, where method `n` at `b` performs an `await`. Thus, while waiting it releases the lock instead of keeping it. The semantics of `await` requires the task to compete again for the lock with other tasks in the same object, and then try again for the result. If it is available the computation proceeds, otherwise the lock is released again and so on. Releasing the lock in `b` does not change the fact that tasks in `a` are blocked. The circular dependency still holds, however the system is not completely blocked but the task of method `n` is caught in an infinite loop of getting and releasing the lock (a *livelock*).

A non-trivial dangerous pattern. We now discuss a program that can manifest a subtle deadlock. In `FJf` a program is a collection of class definitions plus an expression to evaluate, just as in `FJ`. A simple definition in `FJf` is the class `C` in Fig. 2 that defines two methods: `m` to build a new object and `r` that we discuss below. Class `D` extends `C` with a method, `q`, which returns

```
class C {
    C m() { return new C() ;}
    C r(C x) { return x!m().get ;}
}
class D extends C {
    Fut(C) q(D y) {
      return (y!r(this); this!r(y));
    }
}
```

Fig. 2. Simple classes in `FJf`

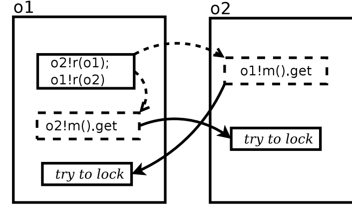

Fig. 3. Deadlock in `FJf`

a value of type `Fut(C)`, the type of a method invocation returning an object of type `C`. Let's consider the expression `new D()!q(new D())`, i.e. an asynchronous call of method `q`. Two objects, o_1 and o_2, of class `D` are created and then the method `q` is invoked on o_1 passing o_2 as a parameter. The body of `q` contains a sequence of two invocations of `r`. Each invocation spawns a new task and returns immediately the control to the caller. These two tasks are hence spawned, inside o_1 and o_2, respectively (see Fig. 3 where the method invocations are indicated by the dotted lines). The task executing `q` terminates correctly. The program, however, continues its execution with the spawned tasks. Among the possible interleavings there is one that gives a deadlock. In particular the dotted task in o_2 invokes `m` on o_1, where a new task is created, and waits for the result returned by a task inside o_1, keeping the lock of o_2 (since it's a `get` operation). Analogously, the dotted task in o_1 gets to run and it hangs waiting for a method in o_2 to return, holding o_1's lock. The two tasks shown with a plain line will never be able to execute since the lock of each object is kept by the dotted tasks. Moreover the two objects o_1 and o_2 are both indefinitely blocked and hence the program is stuck. Notice that this deadlock depends on the scheduler: had the task in o_1, on which the dotted task in o_2 depends, gotten into execution before the dotted task in o_1 no problem would have arisen.

Contracts. In order to capture circular dependencies and therefore statically detect dangerous configurations as the ones described above, we need to trace all the object dependencies, in the form of pairs of object names (graphically represented by the arrows in Fig. 1). Since we want to do this statically, we need a way to identify and track every object in the program. To this aim, we define a technique that associates to each `new` operation (i.e. the one that is responsible of creating a new object) a fresh object name, picked from a countable set of object names o_1, o_2, ... Then a type system computes the object name associated to each expression of the program. For instance, to a method invocation is assigned (a reference to) the object name returned by the method, to a field selection the object stored in the field, and so on. The type entity conceived to represent this kind of information is the *future record*. Moreover, as we said, we are interested only in detecting object pairs, which are the result of `get` and `await` operations, and we collect those pairs by following all the chains of method invocations. Therefore other local computation terms different from `get`, `await`, and method invocations are not relevant to the analysis. The type system, besides computing object identities of expressions, is responsible of extracting from the program abstract descriptions, called *contracts*, containing only relevant information. More precisely, the typing judgments have the form $\Gamma \vdash_a \texttt{e} : (\texttt{T}, \mathbb{r})\,,\ \mathbb{c}$, where Γ is the environment, a is the name of the object `this`, `e` is a `FJf` expression, `T` is its (class or future) type, $\mathbb{r}$ is a future record, and the $\mathbb{c}$ is the contract. A future record of the form $a[\bar{\texttt{f}} : \bar{b}]$, when associated to an expression `e`, means that a is the object associated to `e`, and the fields of a are assigned the objects $\bar{b}$. A future record of the form $a \rightsquigarrow \mathbb{r}$ corresponds to a future reference to the object $\mathbb{r}$, being produced by a method invoked on an object a. (In order to retrieve the actual value, a `get` operation must be perfomed on the expression assigned this record.) Future records can also be not fully specified, as $c[\texttt{f} : X]$, allowing us to avoid infinite future records. The syntax of contracts associated to expressions is the following:

$$\mathbb{c} ::= \mathbb{0} \mid \texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' \mid \texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' \,\blacksquare\, (a, a') \mid \texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' \,\blacksquare\, (a, a')^{\mathtt{a}} \mid (a, a') \mid (a, a')^{\mathtt{a}} \mid \mathbb{c} \fatsemi \mathbb{c}$$

where $\texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'$ corresponds to the invocation of `m` in class `C` on a object $\mathbb{r}$, passing objects $\bar{\mathbb{r}}$ as parameters, and object $\mathbb{r}'$ is the returned value. Object pairs (a, a') and $(a, a')^{\mathtt{a}}$ are intro-

duced by `get` and `await` operation, respectively. They can be isolated, meaning the operation has been performed on a method parameter or field, or they can be associated to a method invocation. Finally, $\mathbb{c} \fatsemi \mathbb{c}$ is used for sequential composition.
The behavior of a method is described by the contract of its body, which is wrapped around by an interface specifying the binders, i.e. the receiver's and parameters' names, for the names occurring inside the contract, and the returned value. All free occurrences correspond to new objects, and are therefore fresh names in the system. This whole description, called *method contract*, has the form $\mathbb{r}(\bar{\mathbb{r}})\{\mathbb{c}\}\ \mathbb{s}$. The contract of a method call is built by instantiating the formal object names, contained in the method contract, with the actual ones. For example, the contract of the method `m` of Fig. 2 is derived using the rule

$$\frac{\Gamma + \texttt{this} : (\texttt{C}, a[\,]) \vdash_a \texttt{new C()} \ : \ (\texttt{C}, b[\,]),\ \mathbb{0}}{\Gamma \vdash \ \texttt{C m ()\{return new C();\}} : a[\,]\,()\{\mathbb{0}\}\, b[\,] \ \ \text{IN}\ \texttt{C}}$$

where in $a[\,]\,()\{\mathbb{0}\}\, b[\,]$ we have $a[\,]$ as the receiver object (a is the object name and `[]` means an empty sequence of fields), no parameters, and $b[\,]$ as the returned object. The contract is empty ($= \mathbb{0}$) in this case, meaning that `m`'s body contains no method invocations. Let's consider method `r`'s contract: $a[\,]\,(c[\,])\{\,\texttt{C.m}\ c[\,]\,() \rightarrow b[\,]\,\centerdot\,(a, c)\,\}\, b[\,]$, where the contract body gives an abstract account of `r`'s behavior: it invokes method `m` of the same class on the object $c[\,]$, passed to it as a parameter, and finally returns a third object $b[\,]$. The `get` operation specifies that the task inside a has to wait for another task inside c to return: this information is thus added to contract $\mathbb{c}$ with the pair (a, c). The general rules for `get` and `await` expressions are:

$$\frac{\Gamma \vdash_a \texttt{e} \ : \ (\texttt{Fut(T)}, a' \rightsquigarrow \mathbb{s}),\ \mathbb{c}}{\Gamma \vdash_a \texttt{e.get} \ : \ (\texttt{T}, \mathbb{s}),\ \mathbb{c} \between (a, a')} \qquad \frac{\Gamma \vdash_a \texttt{e} \ : \ (\texttt{Fut(T)}, a' \rightsquigarrow \mathbb{s}),\ \mathbb{c}}{\Gamma \vdash_a \texttt{e.await} \ : \ (\texttt{Fut(T)}, a' \rightsquigarrow \mathbb{s}),\ \mathbb{c} \between (a, a')^{\mathrm{a}}}$$

where the $\between$ operator can be read as "add pair (a, a') to $\mathbb{c}$". As expected, the `get` operation, retrieving the result, on an expression of type `Fut(T)` returns an object of type `T` and the returned future record $\mathbb{s}$. The `await` operation instead, dealing only with the availability of the result, leaves the type and record part unchanged. They both affect the contract $\mathbb{c}$ by documenting that in the former (resp. the last) case, the continuation (resp. correct termination) of the execution of object a's activity is bound to the termination of a task inside an object a'.

The analysis. Once contracts have been inferred we transform them into automata and by composition we obtain a finite model of an `FJf` program. The states of this automaton contain dependencies and the transitions model how dependencies are activated or discarded during program's execution. A potential misbehavior is signaled by the presence of a circularity in some state of it. Our analysis is based on over-approximations and as such: if the analysis certifies a program to be lock-free it certainly is, otherwise it means that a locked configuration *might* be reached at run-time (as is the case of deadlock depending on the scheduler's choices). The framework we have just described can be found in [**?**]. In the following section we discuss some extensions and their impact on the analysis.

2 Extensions

Field updates. `FJf`, just as `FJ`, is a functional language as fields are initialized by the constructor and are immutable. In this contribution we introduce a notion of state by enabling the private field updates, namely by adding `this.f = e` to the syntax of expressions. To see how this enhancement is reflected upon our framework let us consider a sequence of two method invocations `x.m(); x.n()`, both called on the same object and both modifying the same field. Say `m` does `this.f = e1` and `n` does `this.f = e2`. Due to the asynchronous nature of method invocation, there is no way to know the order in which the updates will take place at run-time. Working statically, we have to keep track of all the different possibilities, thus the type system must assume for an expression a set of possible objects it can reduce to. The updated syntax of future records is the following: $\mathbb{r} ::= X \ \mid\ \mathcal{A}[\bar{\texttt{f}} : \bar{\mathbb{r}}] \ \mid\ \mathcal{A} \rightsquigarrow \mathbb{r}$, where $\mathcal{A}$ is a set of object names.

For instance, let us consider a field `f` containing an object with a field `g`. If two different updates of `f` occur in the program, with two expressions of future record $\mathbb{r} = b[\texttt{g} : \mathbb{r}']$ and $\mathbb{s} = c[\texttt{g} : \mathbb{s}']$, respectively. The future record of `f` must then take into account both updates and therefore it will be $\mathbb{r} \bigvee \mathbb{s} = \{b, c\}[\texttt{g} : \mathbb{r}' \bigvee \mathbb{s}']$. A typing rule for the new construct is introduced:

$$\frac{\Gamma \vdash_a \texttt{this} : (\texttt{C}, a[\bar{\texttt{f}} : \bar{\mathbb{r}}', \texttt{f} : \mathbb{r} \bigvee \mathbb{s}), \mathbb{0} \qquad \Gamma \vdash_a \texttt{e} : (\texttt{C}', \mathbb{r}), \mathbb{c} \qquad \texttt{C}' <: \texttt{C}}{\Gamma \vdash_a \texttt{this.f} = \texttt{e} : (\texttt{C}', \mathbb{r}), \mathbb{c}}$$

As for the analysis, the transformation of contracts into automata must be adapted to treat sets of names instead of single object names. While the analysis is less precise, since it predicts a set of dependencies for each actual one, it is still correct: if a program is recognized to be lock-free, its execution will proceed without encountering a locked configuration.

Task Dependencies. By extending `FJf` with field update we introduce the possibility of having *pure livelocks*: configurations made up of only `await`-generated dependencies in which there is a circularity of task dependencies. Fig. 5 depicts such a configuration in which `await`-pairs are shown with a dashed arc. Without field updates, it is not possible to write a program that leads to such a configuration. Therefore, the analysis could safely ignore an `await`-circularity between objects (as in [**?**]). Consider for example the situation of Fig. 4.

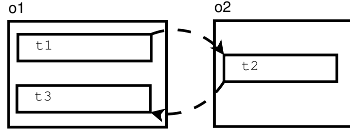

Fig. 4. Non-problematic `await`-circularity

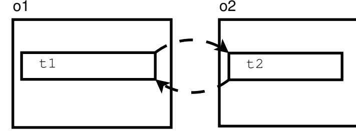

Fig. 5. Pure livelock

The (object) circularity due to $(o_1, o_2)^{\mathsf{a}}$ and $(o_2, o_1)^{\mathsf{a}}$ does not lead to a livelock since task $\texttt{t}_3$ can acquire o_1's lock and get into execution. $\texttt{t}_2$ and $\texttt{t}_1$ can therefore terminate. As soon as pure livelocks become expressible, we have to refine the analysis to work at a finer granularity by also taking into consideration task pairs. In facts, when the analysis finds a circularity of `await`-pairs, it needs additional information on task pairs in order to discriminate pure livelocks from non-dangerous configurations. (This additional discriminating power only concerns `await`-circularities, as a circularity with one or more `get` is always a dangerous configuration.) The typing judgments must then include information about the task in which a given expression is to be typed. They thus take the form: $\Gamma \vdash^{\texttt{t}}_a \texttt{e} : (\texttt{T}, \mathbb{r}), \mathbb{c}$.
We adopt a technique similar to the one for object names described in Sec. 1. Namely, we introduce a fresh task name (picked from a countable set of task names $\texttt{t}_1, \texttt{t}_2, \ldots$) for every method invocation much in the same way as we introduced a fresh object name for every `new` construct in the program. That is, we employ task names for tagging a method invocation with the task responsible for its computation: $\mathcal{A} \rightsquigarrow_{\texttt{t}} \mathbb{s}$. The rule for `await` becomes:

$$\frac{\Gamma \vdash^{\texttt{t}}_a \texttt{e} : (\texttt{Fut(T)}, \mathcal{A} \rightsquigarrow_{\texttt{t}'} \mathbb{s}), \mathbb{c}}{\Gamma \vdash^{\texttt{t}}_a \texttt{e.await} : (\texttt{Fut(T)}, \mathcal{A} \rightsquigarrow_{\texttt{t}'} \mathbb{s}), \mathbb{c} \between (\texttt{t}, \texttt{t}') \between (a, a_i)^{\mathsf{a}} \ \forall a_i \in \mathcal{A} \rightsquigarrow_{\texttt{t}'} \mathbb{s}}$$

adding both a task pair $(\texttt{t}, \texttt{t}')$ and a set of object pairs $(a, a_i)^{\mathsf{a}}$, one for each a_i in the set $\mathcal{A}$.