Provably Live Exception Handling

Bart Jacobs

iMinds-DistriNet Research Group, Department of Computer Science, KU Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

Writing concurrent Java programs that provably terminate, i.e. that terminate in all executions allowed by the language specification, is difficult, because of the combination of two language "features": firstly, the virtual machine is allowed to throw a `VirtualMachineError` exception at any point in the execution of the program; secondly, if a thread terminates because of an exception, a stack trace is printed to the console, but other threads continue to execute normally. As a result, no program where threads wait for other threads is provably live.

Furthermore, even if we ignore the `VirtualMachineError` issue, dealing with exceptions in a way that preserves liveness and compositionality is nontrivial. For example, in the .NET Framework, if a thread terminates because of an exception, the program is terminated. This preserves liveness, but it is not compositional.

At ECOOP 2009, we proposed the *failboxes* language extension. We showed how it facilitates writing programs that deal with exceptions in a way that preserves safety, liveness, and compositionality. We proposed proof rules for proving safety, but not for proving liveness. In this paper, we present our ongoing research on writing and verifying provably live programs. In particular, building on Chalice's approach for proving deadlock-freedom in the absence of exceptions of programs that use channels, we propose preliminary proof rules for proving deadlock-freedom in the presence of synchronous and asynchronous exceptions of programs that use semaphores and failboxes.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms Verification

Keywords exception handling, separation logic, concurrency

1. Introduction

Consider the following Scala program:

```
val queue = new LinkedBlockingQueue[String]()
fork { queue.put("Hello, world") }
queue.take()
```

where fork is defined in the obvious way. This program is not provably live. Indeed, it is entirely conceivable that the call of put encounters a resource limitation (such as when trying to allocate a new linked list node) and throws an exception instead of completing the operation and unblocking the main thread. Furthermore, per [1, §11.1.3], the Java virtual machine may throw a so-called *asynchronous exception* "at any point in the execution of a program", either as a result of an invocation of the deprecated Thread.stop method, or as a result of "An internal error or resource limitation in the Java Virtual Machine that prevents it from implementing the semantics of the Java programming language."[1] Furthermore, any such exception in the forked thread would cause the forked thread to terminate, but it would not prevent the main thread from blocking forever on the take call.

Perhaps the most straightforward approach to fix this would be to catch the exception and terminate the program:

```
val queue = new LinkedBlockingQueue[String]()
fork {
  try {
    queue.put("Hello, world")
  } catch {
    case _ ⇒ System.exit(1)
  }
}
queue.take()
```

Note, firstly, that this approach achieves liveness in the presence of exceptions thrown by the put call, but it is not clear that it is fully hardened against asynchronous exceptions: can an asynchronous exception happen before the **try** block is entered, or after the **catch** block is entered? Secondly, this approach is not compositional: if this code block is part of a larger system, failure of this code block should not necessarily terminate the entire application. For example, if the code block is part of the request handling code of a server, a command shell, or some other command handling application, it might be more appropriate to inform the client of the failure and then continue processing other requests.

Here is another attempt:

```
new Failbox().enter {
  val queue = new LinkedBlockingQueue[String]()
  Failbox.fork { queue.put("Hello, world") }
  queue.take()
}
```

where class Failbox and its companion **object** are defined in Fig. 1. The basic idea of this approach is that when an exception occurs in the forked thread, instead of terminating the application, we interrupt the waiting thread, provided it is still waiting for us. Specifically, when an exception occurs inside an *fb*.enter block, for some failbox *fb*, all other threads currently executing inside an *fb*.enter block are interrupted. Method Failbox.fork runs the new thread inside the current failbox.

[1] More correct wording would have been "that prevents it from *otherwise* implementing the semantics of the Java programming language," since by throwing an asynchronous exception, the virtual machine is staying within the boundaries set by the language specification.

FTfJP'15, July 7, 2015, Prague, Czech Republic.
 ISBN 978-1-4503-3656-7/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2786536.2786543

```
class Failbox {
  var failed = false
  val threads = new ArrayList[Thread]()
  def enter(body: ⇒ Unit) {
    synchronized {
      if (failed) throw new FailboxException()
      threads.add(Thread.currentThread())
    }
    try {
      try {
        Failbox.current.withValue(this) { body }
      } finally {
        synchronized {
          threads.remove(Thread.currentThread())
        }
      }
    } catch {
      case e ⇒
        synchronized {
          failed = true
          for (t ← threads) t.interrupt()
        }
        throw e
    }
  }
}
object Failbox {
  val current = new DynamicVariable[Failbox](null)
  def fork(body: ⇒ Unit) {
    ThreadUtils.fork { current.value.enter { body } }
  }
}
```

Figure 1. A minimal failboxes implementation

Assuming no asynchronous exceptions occur inside the code of Fig. 1, this approach achieves liveness and compositionality. The assumption can be eliminated by implementing the methods of Fig. 1 in native code or directly in the virtual machine.

The code of Fig. 1 is a partial implementation of the *failboxes* language extension we proposed in earlier work [2] to facilitate writing programs that are provably safe and live in the presence of unchecked exceptions. In [2], we proposed proof rules for proving safety properties of programs using failboxes, but we did not propose proof rules for proving liveness properties.

In the remainder of this paper, we present some preliminary results in our ongoing research into writing and verifying provably live programs. In particular, in Sec. 2, we present preliminary proof rules for verifying termination of programs using failboxes such as the one above. We offer a conclusion and discuss future work in Sec. 3.

2. Proof rules

In this section, we propose an extension of separation logic [5] for proving deadlock-freedom[2] of concurrent programs that use semaphores for wait-notify synchronization and failboxes for dealing with exceptions. We work up to our final program logic by first proposing in Sec. 2.1 a program logic for proving deadlock-freedom of programs with semaphores but without exceptions, based on the approach used in Chalice [4] for channels, locks, and thread joining. To focus on the deadlock-freedom issue, we elide all features related to safety and data consistency; for those, see [4]. We then extend this program logic in Sec. 2.2 to one that is sound in the presence of synchronous and asynchronous exceptions.

[2] For a complementary approach for modular verification of absence of infinite recursion, see [3].

2.1 Deadlock-freedom of programs with semaphores

We consider the following programming language, where c ranges over commands and x ranges over variables:

$$c ::= \mathbf{new\ sema} \mid \mathbf{fork}\ c \mid x.\mathbf{V} \mid x.\mathbf{P} \mid \mathbf{let}\ x := c\ \mathbf{in}\ c$$

Command **new sema** creates a new semaphore with initial value 0, $x.\mathbf{V}$ releases (i.e. increments) semaphore x and $x.\mathbf{P}$ acquires (i.e. decrements) it, blocking while the value is 0.

The example program translates into this language as follows:

$$\mathbf{let}\ \mathsf{s} := \mathbf{new\ sema\ in\ fork}\ \mathsf{s}.\mathbf{V};\ \mathsf{s}.\mathbf{P}$$

(Here, we use $c; c'$ as a shorthand for $\mathbf{let}\ x := c\ \mathbf{in}\ c'$ with x fresh.)

The assertions P of our assertion language are as follows:

$$P ::= s.\mathsf{sema}(w) \mid s.\mathsf{credit} \mid s.\mathsf{debit} \mid t.\mathsf{obs}(W) \mid P * P \mid \mathsf{true}$$

where s ranges over semaphores, t ranges over thread identifiers, w ranges over *wait levels*, and W ranges over multisets of wait levels. We assume a partially ordered set of wait levels; we will use it in the usual way to prevent wait cycles.

Assertion $s.\mathsf{sema}(w)$ asserts that semaphore s has been associated with wait level w. $s.\mathsf{credit}$ asserts ownership of one credit (acquire permission) for semaphore s. $s.\mathsf{debit}$ asserts ownership of one debit (release obligation) for semaphore s. $t.\mathsf{obs}(W)$ asserts that W is the current multiset of obligations (each associated with a wait level) of thread t.

We have the following laws:

$$\begin{array}{rcl} s.\mathsf{sema}(w) & \Rightarrow & s.\mathsf{sema}(w) * s.\mathsf{sema}(w) \\ t.\mathsf{obs}(W) * s.\mathsf{sema}(w) & \Leftrightarrow & t.\mathsf{obs}(W \uplus \{\!|w|\!\}) * s.\mathsf{sema}(w) \\ & & * s.\mathsf{debit} * s.\mathsf{credit} \end{array}$$

In words: $s.\mathsf{sema}(w)$ assertions are duplicable, and a thread t can take upon itself an obligation to release a semaphore s, which produces both a debit and a credit for the semaphore. Conversely, a thread can cancel an obligation by handing in a corresponding debit-credit pair. $\{\!|w|\!\}$ denotes the singleton multiset containing element w once.

We define the provable correctness judgment $t \vdash \{P\}\ c\ \{Q\}$, where t is the current thread, P is the precondition (an assertion with no free variables), c is the command, and Q is the postcondition (an assertion with a single free variable res denoting the result value of c), inductively as follows:

$$t \vdash \{\mathsf{true}\}\ \mathbf{new\ sema}\ \{\mathsf{res}.\mathsf{sema}(w)\}$$

$$\frac{\forall t'.\ t' \vdash \{t'.\mathsf{obs}(W') * P\}\ c\ \{t'.\mathsf{obs}(\emptyset) * \mathsf{true}\}}{t \vdash \{t.\mathsf{obs}(W \uplus W') * P\}\ \mathbf{fork}\ c\ \{t.\mathsf{obs}(W)\}}$$

$$t \vdash \{\mathsf{true}\}\ s.\mathbf{V}\ \{s.\mathsf{credit}\}$$

$$t \vdash \{t.\mathsf{obs}(W) * s.\mathsf{sema}(w) * s.\mathsf{credit} \wedge w < W\}\ s.\mathbf{P}\ \{t.\mathsf{obs}(W)\}$$

$$\frac{t \vdash \{P\}\ c\ \{Q\} \qquad \forall v.\ t \vdash \{Q[v/\mathsf{res}]\}\ c'[v/x]\ \{R\}}{t \vdash \{P\}\ \mathbf{let}\ x := c\ \mathbf{in}\ c'\ \{R\}}$$

$$\frac{t \vdash \{P\}\ c\ \{Q\}}{t \vdash \{P * R\}\ c\ \{Q * R\}}$$

$$\frac{P \Rightarrow P' \qquad t \vdash \{P'\}\ c\ \{Q\} \qquad Q \Rightarrow Q'}{t \vdash \{P\}\ c\ \{Q'\}}$$

In words: when creating a new semaphore, the proof author can associate an arbitrary wait level with it. When forking a new thread, the parent thread can delegate some of its obligations to the child thread. The child thread must discharge (or delegate) all of its obligations before it terminates. Releasing a semaphore produces a credit (i.e. an acquire permission). Acquiring a semaphore consumes a credit; furthermore, it requires that the semaphore's wait level is below the wait levels of the current thread's obligations. The frame, let and consequence rules are as usual.

Theorem 1 (Soundness)**.** *If* $t \vdash \{t.\mathsf{obs}(\emptyset)\}\ c\ \{t.\mathsf{obs}(\emptyset) * \mathsf{true}\}$*, then* c *is deadlock-free, i.e. it does not reach a non-finished state where all threads are blocked.*

Proof. Observe that we have the following invariants:

- For any semaphore s, the total number of credits for s owned by all threads equals the value of s plus the total number of debits for s.
- The multiset union of all threads' obligation multisets equals the multiset union of the wait levels of the semaphores associated with the debits present in the system.

We prove the theorem by contradiction. Consider a non-finished state where all threads are blocked. We say thread t *waits for* thread t' *at level* w if t is blocked on a command $s.\mathbf{P}$ and thread t''s obligation multiset contains the element w, where w is the wait level associated with semaphore s. Notice that each thread waits for some thread, and furthermore, if t waits for t' at some level w, then t' waits at some level $w' < w$. Since the number of threads is finite, there is a cycle in the waits-for graph and $w < w$ for some w, which contradicts the fact that relation $<$ on wait levels is a strict partial order. □

We can prove deadlock-freedom of the example program as follows:

$\{t.\mathsf{obs}(\emptyset)\}$
let s := **new sema in**
$\{t.\mathsf{obs}(\emptyset) * \mathsf{s.sema}(0)\}$
$\{t.\mathsf{obs}(\{0\}) * \mathsf{s.sema}(0) * \mathsf{s.debit} * \mathsf{s.credit}\}$
fork
  $\{t'.\mathsf{obs}(\{0\}) * \mathsf{s.sema}(0) * \mathsf{s.debit}\}$
  s.$\mathbf{V}$;
  $\{t'.\mathsf{obs}(\{0\}) * \mathsf{s.sema}(0) * \mathsf{s.debit} * \mathsf{s.credit}\}$
  $\{t'.\mathsf{obs}(\emptyset) * \mathsf{s.sema}(0)\}$
$\{t.\mathsf{obs}(\emptyset) * \mathsf{s.sema}(0) * \mathsf{s.credit}\}$
s.$\mathbf{P}$
$\{t.\mathsf{obs}(\emptyset)\}$

The main thread, after creating the semaphore at wait level 0, creates a debit and passes the obligation to send on to the forked thread, while using the resulting credit to acquire the semaphore. The forked thread releases the semaphore, and uses the resulting credit to cancel its obligation before finishing.

2.2 Deadlock-freedom of programs with semaphores and exceptions

Notice that the program logic presented in the preceding section is not sound in the presence of exceptions: in the example program, if an exception occurs in the forked thread before the release operation completes, the main thread remains blocked on the acquire operation forever.

The core of the problem is that exceptions may cause obligations to remain unfulfilled, causing threads that wait on those obligations to block forever.

The solution we consider in this paper is to use failboxes to make sure that if an obligation is lost due to an exception, any thread blocking or attempting to block on that obligation will itself receive an exception. In particular, we associate a failbox with each obligation, and furthermore, 1) the thread that holds this obligation must be inside this failbox, and 2) any thread that blocks on this obligation must also be inside this failbox.

More specifically, we associate a failbox with each semaphore, and we associate the semaphore's failbox with each obligation to release this semaphore.

We extend the programming language as follows:

$$c ::= \cdots \mid \mathbf{new\ failbox} \mid x.\mathbf{enter}\ c$$

We modify the semantics of the **fork** c command so that it executes the forked thread inside the current failbox, if any.

A translation of the hardened example program into the formal language is as follows:

let fb := **new failbox in**
fb.**enter** (
  let s := **new sema in**
  fork s.$\mathbf{V}$; s.$\mathbf{P}$
)

The semantics of the language is extended with a step rule THROW that arbitrarily picks a thread t and throws an exception in it. In our simple language, this means that:

- All failboxes that t is directly or indirectly running inside, i.e., all failboxes f such that an f.enter block is on t's stack, are marked as failed.
- All threads directly or indirectly running inside of these failboxes are marked as interrupted. In our simple language, a thread's interrupted flag is never reset. An execution state where a thread is blocked and marked as interrupted is not considered a deadlocked state, since an InterruptedException will eventually be thrown in this thread.
- Thread t enters the finished state.

Our assertion language is now as follows:

$$\begin{aligned} P \quad ::= \quad & s.\mathsf{sema}(w, f) \mid s.\mathsf{credit} \mid s.\mathsf{debit} \\ & \mid t.\mathsf{obs}(W, \tilde{f}, W) \mid P * P \mid \mathsf{true} \end{aligned}$$

Here, f ranges over failboxes, and $\tilde{f}$ ranges over the failboxes plus the special value $\perp$ denoting that the thread is currently not inside a failbox. Assertion $s.\mathsf{sema}(w, f)$ asserts that semaphore s is associated with wait level w and failbox f. $t.\mathsf{obs}(W, f, W')$ asserts that thread t is currently directly inside failbox f and holds obligations $W \uplus W'$ of which W' were taken on while inside the current innermost enclosing enter block, and therefore are associated with failbox f. (Some of the obligations in W may also be associated with f, but the simple logic we propose here does not track that information.) $t.\mathsf{obs}(W, \perp, W')$ asserts that thread t is currently not inside a failbox; it follows that $W = W' = \emptyset$.

We now have the following laws:

$$s.\mathsf{sema}(w, f) \Rightarrow s.\mathsf{sema}(w, f) * s.\mathsf{sema}(w, f)$$
$$t.\mathsf{obs}(W, f, W') * s.\mathsf{sema}(w, f) \Leftrightarrow$$
$$\quad t.\mathsf{obs}(W, f, W' \uplus \{w\}) * s.\mathsf{sema}(w, f) * s.\mathsf{debit} * s.\mathsf{credit}$$

Notice that a thread may take on an obligation to release a semaphore s only if it is currently directly inside the failbox associated with s.

The proof rules are now:

$$t \vdash \{\mathsf{true}\}\ \mathbf{new\ sema}\ \{\mathsf{res.sema}(w, f)\}$$

$$\frac{\forall t'.\ t' \vdash \{t'.\mathsf{obs}(\emptyset, \tilde{f}, W') * P\}\ c\ \{t'.\mathsf{obs}(\emptyset, \tilde{f}, \emptyset) * \mathsf{true}\}}{t \vdash \{t.\mathsf{obs}(W_0, \tilde{f}, W \uplus W') * P\}\ \mathbf{fork}\ c\ \{t.\mathsf{obs}(W_0, \tilde{f}, W)\}}$$

$$t \vdash \{\mathsf{true}\}\ s.\mathbf{V}\ \{s.\mathsf{credit}\}$$

$$t \vdash \begin{array}{l} \left\{\begin{array}{l} t.\mathsf{obs}(W, f, W') * s.\mathsf{sema}(w, f) * s.\mathsf{credit} \\ \wedge\ w < W \cup W' \end{array}\right\} \\ s.\mathbf{P} \\ \{t.\mathsf{obs}(W, f, W')\} \end{array}$$

$$t \vdash \{\mathsf{true}\}\ \mathbf{new\ failbox}\ \{\mathsf{true}\}$$

$$\frac{t \vdash \{t.\mathsf{obs}(W \uplus W', f', \emptyset) * P\}\ c\ \{t.\mathsf{obs}(W \uplus W', f', \emptyset) * Q\}}{t \vdash \{t.\mathsf{obs}(W, \tilde{f}, W') * P\}\ f'.\mathsf{enter}\ c\ \{t.\mathsf{obs}(W, \tilde{f}, W') * Q\}}$$

(The frame, let and consequence proof rules are unchanged and are not shown again.)

In words: when creating a semaphore, the proof author may associate an arbitrary wait level and an arbitrary failbox with it. When forking a new thread, the child thread executes initially inside the failbox that the parent thread is executing within, if any. The parent thread may delegate some of the obligations that it took on during the current innermost enclosing **enter** block to the child thread. Acquiring a semaphore requires that the thread is executing directly inside the associated failbox. Creating a new failbox yields a failbox value, but no further information about this value is tracked by our simple logic. Before a thread leaves an **enter** block, it must discharge all obligations taken on during the block.

Theorem 2 (Soundness)**.** *If* $t \vdash \{t.\mathsf{obs}(\emptyset, \bot, \emptyset)\}\ c\ \{t.\mathsf{obs}(\emptyset, \bot, \emptyset) * \mathsf{true}\}$*, then* c *is deadlock-free, i.e. it does not reach a non-finished state where all threads are blocked, even in the presence of synchronous or asynchronous exceptions.*

Proof. We instrument the execution semantics as follows: in each state, we associate with each thread an *augmented obligation bag* $\hat{W}$, which is a bag of pairs (w, f). When a debit is created for a semaphore s associated with wait level w and failbox f, the pair (w, f) is added to the bag. Analogously, when a debit is destroyed, the pair is removed.

The invariants identified in Thm. 1 still hold. We have the following additional invariants:

- Whenever we have $t.\mathbf{obs}(W, f, W')$, there is an augmented bag $\hat{W}$ such that t's augmented bag equals $\hat{W} \uplus (W' \times \{f\})$ and $W = \mathsf{fst}(\hat{W})$.
- If a thread is holding an obligation (w, f), it is running inside f, or else f is marked as failed.
- The bag of all obligations associated with a failbox f equals the bag of the wait levels of all debits of all semaphores associated with f.
- If a thread is directly or indirectly running inside a failed failbox, then it is marked as interrupted.
- All obligations held by finished threads are associated with failed failboxes.

Consider a deadlocked state. Consider any thread t blocked on an $s.\mathbf{P}$ operation. t is necessarily running inside a non-failed failbox f, since otherwise it would be marked as interrupted and the state would not be deadlocked. Also, s must be associated with f. Therefore, some thread t' must be holding an obligation (w, f), where w is s's wait level. t' must be non-finished. The remainder of the proof is as for Thm 1. □

We can prove deadlock-freedom of the example program as follows:[3]

$\{t.\mathsf{obs}(\emptyset, \bot, \emptyset)\}$
let fb := **new failbox in**
$\{t.\mathsf{obs}(\emptyset, \bot, \emptyset)\}$
fb.**enter** (
  $\{t.\mathsf{obs}(\emptyset, \mathsf{fb}, \emptyset)\}$
  let s := **new sema in**
  $\{t.\mathsf{obs}(\emptyset, \mathsf{fb}, \emptyset) * \mathsf{s.sema}(0, \mathsf{fb})\}$
  $\{t.\mathsf{obs}(\emptyset, \mathsf{fb}, \{0\}) * \mathsf{s.sema}(0, \mathsf{fb}) * \mathsf{s.debit} * \mathsf{s.credit}\}$
  fork
    $\{t'.\mathsf{obs}(\emptyset, \mathsf{fb}, \{0\}) * \mathsf{s.sema}(0, \mathsf{fb}) * \mathsf{s.debit}\}$
    s.**V**;
    $\{t'.\mathsf{obs}(\emptyset, \mathsf{fb}, \{0\}) * \mathsf{s.sema}(0, \mathsf{fb}) * \mathsf{s.debit} * \mathsf{s.credit}\}$
    $\{t'.\mathsf{obs}(\emptyset, \mathsf{fb}, \emptyset) * \mathsf{s.sema}(0, \mathsf{fb})\}$
  $\{t.\mathsf{obs}(\emptyset, \mathsf{fb}, \emptyset) * \mathsf{s.sema}(0, \mathsf{fb}) * \mathsf{s.credit}\}$
  s.**P**
  $\{t.\mathsf{obs}(\emptyset, \mathsf{fb}, \emptyset)\}$
)
$\{t.\mathsf{obs}(\emptyset, \bot, \emptyset)\}$

3. Conclusion

We presented an intermediate result in our ongoing work on verification of liveness properties of concurrent programs in the presence of exceptions. While the proposed program logic leaves much room for further work, it enables us, for the first time, to verify, soundly, liveness properties of concurrent Java programs involving wait-signal-style synchronization.[4]

We are in the process of implementing these ideas in the context of our VeriFast modular verification tool for Java programs. We have translated the simple failboxes API shown in Fig. 1 to Java and encoded the proposed proof rules into VeriFast annotations for this API. We were able to verify the example program against this API.

Future work on the program logic includes integrating support for locking and thread joining; adding support for catching exceptions (and for explicitly throwing exceptions); adding support for discharging obligations held by the current thread but not taken on in the innermost enter block; developing a formal soundness proof; and further experimentation and validation.

Acknowledgements We thank the reviewer who suggested the observation of Footnote 3. Funded by EU project ADVENT.

References

[1] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 Edition, 2015.

[2] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In *ECOOP*, 2009.

[3] Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification: extended version. Technical Report CW 680, Department of Computer Science, KU Leuven, Belgium, January 2015.

[4] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, 2010.

[5] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.

[3] Notice that we could eliminate the need for $s.\mathsf{debit}$ assertions in our logic by having $t.\mathsf{obs}$ take bags of semaphores instead of bags of wait levels as arguments. The choice is largely a matter of taste.

[4] Note that supporting condition variables (Java's built-in wait-signal construct) is a challenging open problem.