Formal verification in Scala with Leon

Romain Ruetschi, EPFL

August 2015

Outline

1. Formal verification
2. A few words about Scala
3. Leon, a verification system for Scala
4. Verification conditions
5. Demo
6. Under the hood
7. SMT solver

Formal verification

Traditionally, errors in hardware and software have been discovered empirically, by testing them in many possible situations.

The number of situations to account for is usually so large that it becomes impractical.

Formal verification is an alternative that involves trying to prove mathematically that a computer system will function as intended.

A lot of hardware companies rely extensively on formal verification, eg. Intel.

But it can also be applied to cryptographic protocols, digital circuits, **software**, etc.

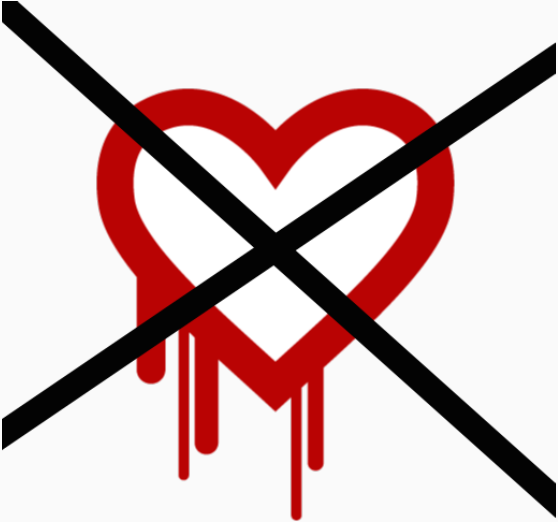
Formal verification of software

Process of proving that a program satisfies a formal specification of its behavior, thus making the program safer and more reliable.

Catches bugs such as integer overflows, divide-by-zero, out-of-bounds array accesses, buffer overflows, etc.

But also helps making sure that an algorithm is properly implemented.

Formal verification of software



A few words about Scala

Statically typed programming language.

Runs on the Java Virtual Machine.

Invented at EPFL by Prof. Martin Odersky.

Version 1.0 released in 2004.

In use at companies such as: Twitter, UBS, LinkedIn, MUFG, Geisha Tokyo Entertainment, M3, etc.

Leon, a verification system for Scala

Leon takes as input a Scala source file, and generates individual *verification conditions* corresponding to different properties of the program.

It then tries to prove or disprove that the verification conditions hold.

Verification conditions

Pre- and post-conditions

```
def neg(x: Int): Int = {
  require(x >= 0)
  -x
} ensuring(_ <= 0)
```

Leon will try to prove that the post-condition always holds, assuming that the pre-condition does hold.

Array access safety

For each array variable, Leon carries along a symbolic information on its length.

This information is used to prove that each expression used as an index in the array is both positive and strictly smaller than its length.

Pattern matching exhaustiveness

Leon takes pre-conditions into account to verify that pattern matches are exhaustive.

```
def getHead(l: List): Int = {
  require(l != Nil)
  l match {
    case x :: _ => x
  }
}
```

Repair and synthesis

Leon can automatically repair your program if it doesn't satisify its specification.

More importantly, it can also synthesize code from a specification!

It does so by attempting to find a counter-example to the claim that no program satisfying the given specification exists.

Demo: leon.epfl.ch

Under the hood

Leon is itself written in Scala.

It delegates parsing and typechecking to the Scala compiler.

The AST it gets from `scalac` is then converted to a *PureScala* AST.

This AST then goes through a number of transformations, before either of the verification, repair or synthesis phases kick in.

Under the hood

Most of the hard work required to prove or disprove various properties of the program is delegated to an SMT solver.

SMT stands for Satisfiability Modulo Theories.

An SMT instance is a first-order logic formulas over various *theories* such as real numbers, integers, lists, arrays, ADTs, and others.

Verification conditions are translated to an SMT instance, then fed to the SMT solver, which attempts to either prove it, or yield a counter-example.

Learn more about Leon

http://leon.epfl.ch
http://leon.epfl.ch/doc
http://lara.epfl.ch/w/leon
https://github.com/epfl-lara/leon

Thank you!

If you have any questions or just want to get in touch:

Twitter: @_romac
GitHub: @romac

Bachelor semester project

An encoding of Any for Leon

Adds support for uni-typed programs such as

```
def reverse(lst: Any): Any = {
  if (lst == Nil()) Nil()
  else reverse(lst.tail) ++ Cons(lst.head, Nil())
} ensuring (_.contents == lst.contents)

def reverseReverseIsIdentity(lst: Any) = {
  reverse(reverse(lst)) == lst
}.holds
```

Mostly an experiment, as using Any is generally frowned upon in the Scala community.

Has nonetheless interesting applications, such as eg. automatically porting theorems from Lisp-based theorem provers like ACL2.

Nothing too fancy. It's just a pre-processing phase, that encodes Any as a sum type and lifts expressions into it.

Allowed us to add support for Any without touching the rest of the system.

Before

```
case class Box(value: Int)

def double(x: Any): Any = x match {
    case n: Int => n * 2
    case Box(n) => Box(n * 2)
    case _      => x
}

double(42)
```

After

```scala
sealed abstract class Any1
case class Any1Int(value: Int) extends Any1
case class Any1Box(value: Box) extends Any1

def double(x: Any1): Any1 = x match {
    case Any1Int(n)      => Any1Int(n * 2)
    case Any1Box(Box(n)) => Any1Box(Box(n * 2))
    case _               => x
}

double(Any1Int(42))
```