An encoding for Any within Leon

Romain Ruetschi, LARA - EPFL

Spring 2015

Motivation

Proving properties of unityped programs

```
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
```

Implementation

Example

```
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)
```

Example

```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))
```

Outline

Declare constructors

Rewrite functions types to replace Any with Any1

Rewrite pattern matches to peel the constructors off

Lift expressions into Any1

Extract implicit classes as case classes

Add extension methods to Any

Declare constructors

```
sealed abstract class Expr
case class Lit(n: Int)           extends Expr
case class Add(l: Expr, r: Expr) extends Expr

abstract class Any1
case class Any1Expr(value: Expr) extends Any1
case class Any1Int(value: Int)   extends Any1
```

Rewrite functions types

```scala
def id(x: Any): Any = x

case class Box(value: Any) {
    def map(f: Any => Any): Box = Box(f(value))
    def contains(other: Any): Boolean = value == other
}
```

Rewrite functions types

```
def id(x: Any1): Any1 = x

case class Box(value: Any1) {
    def map(f: Any1 => Any1): Box = Box(f(value))
    def contains(other: Any1): Boolean = value == other
}
```

Lift expressions

```
def id(x: Any1): Any1   = x
def toAny(x: Int): Any1 = x

def double(x: Any1): Any1 = x match {
  case i: Int     => x * 2
  case b: Boolean => !b
  case _          => x
}
```

Lift expressions

```scala
def id(x: Any1): Any1   = x
def toAny(x: Int): Any1 = Any1Int(x)

def double(x: Any1): Any1 = x match {
  case i: Int     => Any1Int(i * 2)
  case b: Boolean => Any1Boolean(!b)
  case _          => x
}
```

Rewrite pattern matches

```
def double(x: Any1): Any1 = x match {
  case Any1Int(i: Int)         => Any1Int(i * 2)
  case Any1Boolean(b: Boolean) => Any1Boolean(!b)
  case _                       => x
}
```

Extract implicit classes

```
@library
implicit class BooleanOps(val underlying: Boolean) {
  def holds : Boolean = {
    assert(underlying)
    underlying
  }
  def ==> (that: Boolean): Boolean = {
    !underlying || that
  }
}
```

Extract implicit classes

```
@library
case class BooleanOps(underlying: Boolean) {
  def holds: Boolean = {
    assert(underlying)
    underlying
  }
  def ==> (that: Boolean): Boolean = {
    !underlying || that
  }
}

@library
def BooleanOps(x: Boolean): BooleanOps = BooleanOps(x)
```

Add extension methods to Any

```
@library
implicit class Any1Ops(val lhs: Any) {
  def *(rhs: Any): Any = (lhs, rhs) match {
    case (l: Int, r: Int)       => l * r
    case (l: BigInt, r: BigInt) => l * r
    case (l: String, r: BigInt) => l repeat r
    case _                      =>
      error[Any]("operation not supported")
  }
}
```

Limitations

Generic types

Because child types must form a simple bijection with parent class type, it is currently impossible to synthesize constructors for polymorphic type such as `Option[T]`:

```
abstract class Any1
case class Any1Option[T](x: Option[T]) extends Any1
```

Structural types parametrized by Any

Because ADTs cannot contain recursive references through non-structural types such as Set, it is currently impossible to define types such as:

```
abstract class Any1
case class Any1SetAny(value: Set[Any1]) extends Any1
```

Hack

Mutable types

```
id.setType(Any1Type)
```

Current implementation mutates `Identifiers` type.

Only meant as a temporary hack, will be fixed soon.

What's left?

Preconditions

Currently not possible to write:

```
def reverse(lst: Any): Any = {
  require(lst.isInstanceOf[List])
  if (lst == Nil()) Nil()
  else reverse(lst.tail) ++ Cons(lst.head, Nil())
} ensuring (_.size == lst.size)
```

Can't handle this in translation to Z3:
require(lst.isInstanceOf[Any1List])

Without those preconditions, counter-examples are yielded for postconditions.

What's left?

Maybe more, cannot tell at the moment.

Other applications

Other applications

Type inference for uni-typed programs

Other applications

Type inference for uni-typed programs

Port theorems from ACL2 to Leon

Thank you