Semester project: An encoding for Any within Leon

Romain Ruetschi, LARA - EPFL

Spring 2015

Abstract

We describe how to add support for Scala's top type `Any` to Leon, by encoding it as a sum type, and by lifting expressions into it.

Contents

1 Introduction

Leon is an automated verification, synthesis and repair system for a purely functional subset of Scala, called *Pure Scala*. Leon already support a good chunk of Scala's features, such Algebraic Data Types (ADT) definitions in the form of an abstract class and case classes/objects, as well as pattern-matching, a restricted form of generics, and a few built-in types such as Sets, Maps and Arrays (Lists are defined in the library as a regular ADT). To verify a given program, Leon expresses verification conditions as first-order logic formulas where some functions and symbols have additional meaning with respect to various theories (eg. ADT, linear arithmetic, real numbers) (Wikipedia 2015). Those formulas are then fed to an SMT solver (Blanc et al. 2013). Leon is thus constrained by what the current state-of-the-art SMT solvers are capable of, and must thus sometime encode *Pure Scala* features in a certain way in order to be able to express it in a way that the SMT solver will understand.

Top types such as `Any`, ie. types which all types are subtypes of, are not readily meaningful to the SMT solver. To overcome this limitation, we can view `Any` as a sum type with one variant per type defined in the program, and encode it as such. We thus need to declare a constructor for each type in use in the program, or at least for each type that is upcasted to `Any` at some point, and to wrap plain values into the appropriate constructor.

As an example, the following code

```
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)
```

would become

```
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))
```

Here, `Any1` is the sum type, `Any1Int` is the constructor for values of type `Int`, and `Any1Box` the one for values of type `Box`.

Note how we need to:

- Rewrite functions types to replace occurrences of `Any` with `Any1`
- Rewrite pattern matches to peel the wrappers off
- Lift expressions upcasted to `Any` into the `Any1` subtype by wrapping them in the corresponding constructor.

As we will show in the following section, this process is completely mechanical and can thus be fully automated. Another nice thing about it is that this is only a tree transformation that leaves the most complex parts of Leon untouched.

2 Implementation

2.1 Outline

Leon works very much like a compiler and is composed of two mandatory phases (Extraction and Preprocessing) which are followed by one of five phases (Synthesis, Repair, Termination, Evaluation and Analysis).

We are now going to describe the modifications we made to Leon, which are entirely self-contained into a new sub-phase of the Preprocessing phase, with the exception of a few new extractors.

2.2 Extraction

The Extraction phase is the first phase run by Leon. It invokes the official `scalac` compiler on the provided input file, and then transforms the Abstract Syntax Tree returned by `scalac` after its typing phase into a *Pure Scala* AST. The only modifications

2.3 Pre-processing

The Preprocessing phase is actually composed of several other phases listed below, and is in charge of turning the *Pure Scala* AST generated by the extraction phase into an AST that will be easier to work with for the subsequent phases. It does so by eg. translating methods into top-level functions.

1. ScopingPhase
2. MethodLifting
3. *EncodeAny*
4. TypingPhase
5. ConvertWithOracle
6. ConvertHoles
7. CompleteAbstractDefinitions
8. InjectAsserts

This phase is where most of the work required to support `Any` takes place, as evidenced by the *EncodeAny* phase emphasized above, which is described in detail in section 2.5.

2.4 Library

The `Any1` type itself is defined as an `abstract class` in the library, rather than synthetically within Leon. This allows us to define implicit conversions and add extension methods straight in the library. See section 2.6.1 for more details.

```
abstract class Any1
```

The class isn't `sealed` but this is not a problem in practice as Leon treats all classes hierarchy as if they were sealed. We could also mark it as `sealed`, as Leon doesn't actually enforce it and thus allows us to add descendants programatically, but that could perhaps confuse a user wading through the standard library, as she wouldn't find any descendants for that class.

2.5 Encoding

The `EncodeAny` phase performs the tree transformations described in section 1 and is a composed of three sub-phases, of which two are of interest:

- *ReplacesTypes* which replaces references to the `Any` type in classes fields, functions and lambdas signatures with `Any1`. It is described in subsection 2.5.1
- *LiftExprs* which lifts values upcasted to `Any1` into their associated constructors, generating the corresponding `case class` on the fly. It is described in subsections 2.5.2, 2.5.3, and 2.5.4.

The following subsections break down the transformations performed by those phases.

2.5.1 Rewriting types

We first need to replace any reference to `Any` in functions and lambdas' signatures with to `Any1`. We do not need to care about the methods themselves as this phase comes right after the *MethodLifting* phase, which lifts methods into plain functions that takes the receiving object as first argument.

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

becomes

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

Note: We show here code as if the *MethodLifting* pass was not performed, the actual tree that comes out of that phase is actually different, but only in that regard.

2.5.2 Declaring constructors

To actually be able to lift an expression of some type `T` into the `Any1` sum type, we need a constructor that takes a value of type `T`. Those constructors are declared as `case classes` extending `Any1`, which is the idiomatic way of

defining a sum type in Scala. Concretely, when we encounter an expression of type `T` that needs to be lifted, we first check whether this type can be lifted according to the following rules:

- If `T` is a type parameter, then it cannot be lifted .
- If `T` s a polymorphic class type, then it cannot be lifted either.
- If `T` is a built-in type (ie. one of `Set`, `Multiset`, `Map`, `Array`, or `Function`) parametrized by `Any` or that can itself not be lifted, then it cannot be lifted.
- Otherwise, if can be lifted.

See section 3 for more details about those limitations.

We must now generate a synthetic case class with a single field of type `T`, in the following way.

- If there is already an constructor for type `T`, there is nothing to do.
- If `T` is a "primitive" type (ie. `Int`, `BigInt`, etc.), we synthesize the appropriate case class straight away, if it hasn't been done before.
- If `T` is a class type, we first walk up the class hierarchy to find the hierarchy root type which we will call `R`, and we synthesize a case class with a single field of type `R`. Note that if `T` is already the top of its hierarchy, then `R = T`.

The case class just synthesized is now added to the know descendants of the `Any1` abstract class in order to make it a variant of the sum type.

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

lift[Int]  => case class Any1Int(value: Int)  extends Any1
lift[Box]  => case class Any1Box(value: Box)  extends Any1
lift[Lit]  => case class Any1Expr(value: Expr) extends Any1
lift[Add]  => case class Any1Expr(value: Expr) extends Any1
lift[Expr] => case class Any1Expr(value: Expr) extends Any1
```

2.5.3 Lifting expressions

We can lift an expression itself in `Any1` by wrapping it in the corresponding constructor (declared as described in the previous section). But first we need to figure out which expressions actually needs to be lifted. To do so, a new abstract `Transformer`, `TransformerWithType`, recursively walks down the tree of the program, starting at every function definition, keeping track of what is

the expected type of the expression under focus. Another concrete transformer, extending the abstract one, wraps the expression under focus with the appropriate constructor when the expected type is `Any` but the expression's type is not.

As a result, the following program

```
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
}
```

becomes

```
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)
  e _          => x
}
```

2.5.4 Rewriting pattern matches

An astute reader will notice that the program above would fail to type-check, the compiler complaining that both pattern types in the pattern match of the `double` function are incompatible with expected type. We thus need to rewrite the patterns as well, in order to account for the fact that the scrutinee is now embedded in a case class constructor. To do so, we recursively walk down every pattern tree and add the required `CaseClassPattern` around the original pattern in order to effectively peel off the constructor.

As result, the program above now becomes

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

This is overall quite straighforward, except for tuple patterns, where we need to guess what is the most general type of values that would successfully match against the pattern in order to synthesize the appropriate constructor, and wrapping the pattern itself into the corresponding case class pattern.

2.5.5 Ensuring match exhaustivity

The Scala compiler doesn't require pattern matches on a value of type `Any` to be exhaustive, so the following code will not trigger any warnings:

```
def negate(x: Any): Any = x match {
  case i: Int     => - + 2
  case b: Boolean => !b
}
```

On the other hand, this is something we would like to enforce within Leon. That very same code would currently result in an valid *match exhaustivity* verification condition, because the only two `Any1` constructors that have been declared at that point are those for `Int` and `Boolean`. To avoid that, we could either declare constructors for every types available to the user, or we can generate an additional, synthetic nullary constructor for `Any1`, inaccessible to the user, called `Any1Unexpected`. By doing so, the program above will now result in an invalid *match exhaustivity* verification condition, forcing the user to effectivly handle all possible types by adding an additional case with a wildcard pattern:

```
def negate(x: Any): Any = x match {
  case i: Int     => -x
  case b: Boolean => !b
  case _          => error[Any]("can only negate Ints and Booleans")
}
```

It is currently unclear whether this feature really provides additional guarantees and whether it can potentially prevent further analysis, such as *occurrence typing* (see section 4).

2.6 Extending `Any` with implicit classes

In order to enable a kind of dynamically-typed (or untyped) programming within Leon, it would be nice to be able to call various methods on values of type `Any`, such as `+`, `-`, etc. and let Leon prove properties about code making use of such operations. Unfortunately, it is not possible in Scala to add methods to `Any` itself, via implicit classes or conversions. Luckily for us, since we encode `Any` as a standard hierarchy of classes, we are able to do so for `Any1`.

2.6.1 Implicit classes

The Leon library already makes use of implicit conversions to extend types such as `Boolean` with new methods, but these implicit conversions are treated in an ad-hoc way during extraction. Each implicit conversion is handled as a special case and is the resulting tree doesn't match what the Scala compiler initially produced. As such, they are quite painful to use since it is required to modify the extraction phase, and thus Leon itself, to add a new one.

As we might want to add a potentially large number of methods to `Any1` as we evolve the testcases and explore new possibilities, it was suggested to us to actually implement proper support for implicit classes and conversions in Leon, as an orthogonal (but nonetheless useful to our ends) side-project. The implementation was quite straightforward and we won't talk about it in the present report. The end result is that an implicit classes such as

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

which are desugared by the Scala compiler as

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

are now represented in the resulting *Pure Scala* AST as

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

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

The main difference with the output of the Scala compiler is that the implicit class becomes a case class, and that the implicit conversion loses a few modifiers which is of no consequence as Leon, at the time of writing this, doesn't make use of/respect those in any way for the moment.

Another benefit of the new implementation is that these implicit classes do not need to be `@ignored` by Leon anymore, and it thus now possible to both verify their properties but also use them when synthesizing programs.

2.6.2 Extending `Any`

In the end, we can now write:

```
@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")
  }
}
```

which will be treated by Leon as any other top-level case class we could have defined. This means that `Any` will be encoded as described in the rest of the report, and that the newly defined `*` operation will be available on values of type `Any1` and thus give the impression to the end-user that they are available on `Any` itself.

3 Limitations

3.1 Generic types

Because child types should form a simple bijection with parent class type (e.g. C[T1, T2] extends P[T1, T2]), it is currently impossible to synthesize constructors for polymorphic type such as `Option[T]`. The following program will thus be rejected:

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

While we could add in that case a single type parameters to `Any1`, what if we also need to lift a type with two or more parameters? This could be overcome by making `Any1` itself polymorphic over a fixed number of type parameters, and by having the constructors carry those type parameters as well, at the expense of syntactic noise (which can maybe be avoided by inserting missing type parameters wherever needed).

3.2 Structural types parametrized by `Any`

Due to a limitation of both the Z3 and CVC4 SMT solvers, it is currently not possible to define a type such as

```
case class A(x: Set[A])
```

because *ADTs cannot contain recursive references through non-structural types* such as `Set`. This makes it impossible to define an `Any1` constructor for values of type `Set[Any1]`.

4 Applications

We briefly describe below a few applications that the presented work might enable.

4.1 Verify, synthesize and repair uni-typed programs

We can consider a program where all values, methods and arguments are ascribed the `Any` type as a uni-typed (or dynamically-typed) program, and could thus eventually prove properties of such programs.

After extending `Any` with the appropriate methods, and fixing a few bugs still present in the current implementation, it might eg. be possible to prove properties of a program such as the following:

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

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

4.2 Type inference for uni-typed programs

Given an untyped program, it would be interesting to be able to perform some kind of occurence typing (Tobin-Hochstadt and Felleisen 2010), before eventually adding the inferred types back into the program.

5 Acknowledgments

I am very grateful to Prof. Viktor Kuncak for allowing me to do this semester project at LARA, and for providing both the project, the main insight for its implementation (described in section 1). Many thanks also to Etienne Kneuss for helping and guiding me during the whole duration of the project, and for his valuable insights as well.

References

Blanc, Régis, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. 2013. "An Overview of the Leon Verification System." http://lara.epfl.ch/~kuncak/papers/BlancETAL13VerificationTranslationRecursiveFunctions.pdf.

Tobin-Hochstadt, Régis, and Matthias Felleisen. 2010. "Logical Types for Untyped Languages." http://www.ccs.neu.edu/racket/pubs/icfp10-thf.pdf.

Wikipedia. 2015. "Satisfiability Modulo Theories." https://en.wikipedia.org/wiki/Satisfiability_modulo_theories.