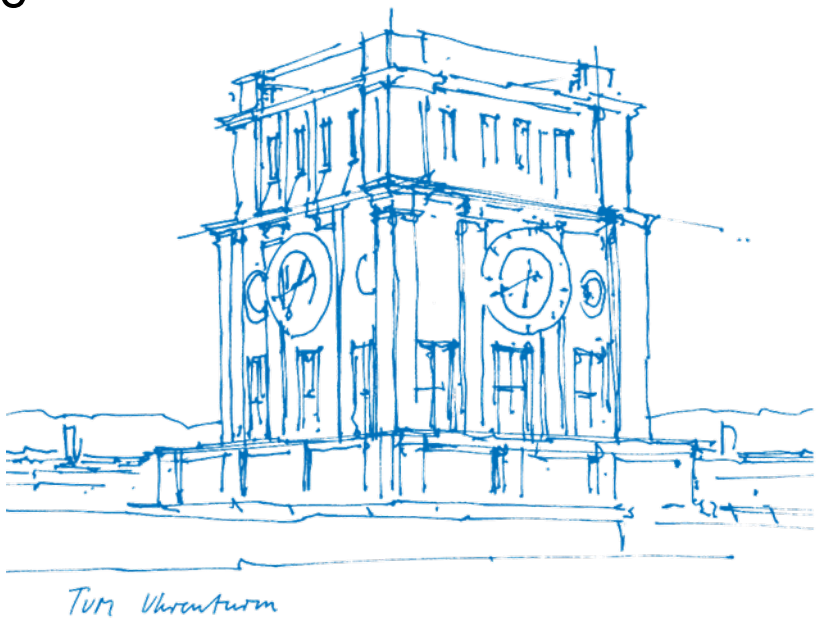
Translating Scala Programs to Isabelle/HOL

Lars Hupel
Technical University of Munich

Viktor Kuncak
École polytechnique fédérale de Lausanne

June 30, 2016



A Taste of Scala & Isabelle

Scala

- functional & object-oriented language
- initially developed at EPFL by Martin Odersky

Scala

- functional & object-oriented language
- initially developed at EPFL by Martin Odersky
- runs on JVM, Javascript and LLVM

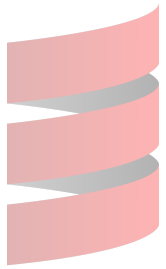
Scala

- functional & object-oriented language
- initially developed at EPFL by Martin Odersky
- runs on JVM, Javascript and LLVM
- strong industry support

Scala Example

```
sealed abstract class List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case class Nil[A]() extends List[A]

def size[A](l: List[A]): BigInt = l match {
  case Nil() => BigInt(0)
  case Cons(_, xs) => 1 + size(xs)
}
```



Isabelle

- interactive proof assistant
- powerful automation

Isabelle

- interactive proof assistant
- powerful automation
  - classical and equational reasoning
  - decision procedures (e.g. linear arithmetic)
  - integration with external automated theorem provers
  - ...

Isabelle/HOL

- *LCF* tradition: small proof kernel

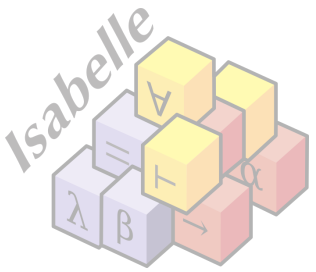
Isabelle/HOL

- *LCF* tradition: small proof kernel
- most prominent logic: *HOL*
  - offers commands for functional programming
  - recursive functions, datatypes, ... are definitional

Isabelle/HOL Example

datatype α list $=$ Nil $|$ Cons α (α list)

primrec size :: α list $\Rightarrow$ int **where**
  size Nil $= 0$
$|$ size (Cons x xs) $= 1 +$ size xs

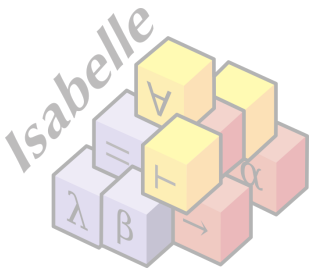


Isabelle/HOL Example

datatype α list $=$ Nil $|$ Cons α (α list)

primrec size :: α list $\Rightarrow$ int **where**
  size Nil $= 0$
$|$ size (Cons x xs) $= 1 +$ size xs

lemma size $xs \geq 0$
by (induct xs) simp+



Leon

A Verification & Synthesis Toolkit for Scala

Leon

- automated verification system for a subset of Scala

Leon

- automated verification system for a subset of Scala
- functions & datatypes can be annotated with assertions (**require**, **ensuring**, **assert**)

Leon

- automated verification system for a subset of Scala
- functions & datatypes can be annotated with assertions (**require**, **ensuring**, **assert**)
- uses SMT solvers to discharge conditions

Leon

- automated verification system for a subset of Scala
- functions & datatypes can be annotated with assertions (**require**, **ensuring**, **assert**)
- uses ~~SMT solvers~~ Isabelle to discharge conditions

Scala Example

```
sealed abstract class List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case class Nil[A]() extends List[A]

def size[A](l: List[A]): BigInt = l match {
  case Nil() => BigInt(0)
  case Cons(_, xs) => 1 + size(xs)
}
```

Leon Example

```
sealed abstract class List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
case class Nil[A]() extends List[A]

def size[A](l: List[A]): BigInt = (l match {
  case Nil() => BigInt(0)
  case Cons(_, xs) => 1 + size(xs)
}) ensuring(_ >= 0)
```

Leon Undo Redo Link Console Params About Documentation WebBuilder Login

```
1  import leon.lang._
2
3  object Example {
4
5    sealed abstract class List[A]
6    case class Cons[A](head: A, tail: List[A])
7      extends List[A]
8    case class Nil[A]() extends List[A]
9
10   def size[A](l: List[A]): BigInt = (l match {
11     case Nil() => BigInt(0)
12     case Cons(_, xs) => 1 + size(xs)
13   }) ensuring(_ >= 0)
14
15 }
16
```

Load an example

-- Select an example

Analysis Compiled ✔

Function	Inv.	Verif.
Style	?	✔
apply	?	✔
ensuring	?	✔
size	?	✔

Bridging the Gap

Bridging the Gap

... but why?

Motivation

Isabelle brings a lot to the table ...

- 20+ years of development
- 1M+ lines of libraries
- trustworthiness through LCF approach
- rich specification & proof language

Motivation

Leon brings a lot to the table ...

- readily-available verification for a wide-spread language
- Goal: Introduce developers to formal methods?

Bridging the Gap

Bridging the Gap

... with as little unverified code as possible

Bridging the Gap

Problem

- Input: Scala AST
- Translation: ???
- Output: Isabelle ???

Possible solution

Textual code generation

Bridging the Gap

Problem

- Input: Scala AST
- Translation: ???
- Output: Isabelle ???

Possible solution

Textual code generation

Bridging the Gap

Problem

- Input: Scala AST
- Translation: ???
- Output: Isabelle API calls

Reasonable solution

Talk directly to Isabelle's API

Bridging the Gap

Problem

- Input: Scala AST
- Translation: ???
- Output: Isabelle API calls

Reasonable solution

Talk directly to Isabelle's API

Talking to Isabelle: libisabelle

- there is a Scala API for Isabelle, but it's very low-level
- libisabelle is a high-level-wrapper

Talking to Isabelle: libisabelle

- there is a Scala API for Isabelle, but it's very low-level
- libisabelle is a high-level-wrapper
  - supports multiple Isabelle versions
  - supports multiple simultaneous processes
  - supports Java & Scala
  - can be used like an asynchronous RPC library
  - setup-free

Talking to Isabelle: libisabelle

- there is a Scala API for Isabelle, but it's very low-level
- libisabelle is a high-level-wrapper
  - supports multiple Isabelle versions
  - supports multiple simultaneous processes
  - supports Java & Scala
  - can be used like an asynchronous RPC library
  - setup-free
- can be used for other projects

Talking to Isabelle: libisabelle

- there is a Scala API for Isabelle, but it's very low-level
- libisabelle is a high-level-wrapper
  - supports multiple Isabelle versions
  - supports multiple simultaneous processes
  - supports Java & Scala
  - can be used like an asynchronous RPC library
  - setup-free
- can be used for other projects
- still: manual work required

Translating Datatypes

Datatypes are almost straightforward.

Translating Functions

Scala input

```
def size(xs: List[A]): BigInt = xs match {
  case Nil() => 0
  case Cons(_, xs) => 1 + size(xs)
}
```

Translating Functions

Scala input

```
def size(xs: List[A]): BigInt = xs match {
  case Nil() => 0
  case Cons(_, xs) => 1 + size(xs)
}
```

Isabelle output

```
fun size xs = (case xs of Nil ⇒ 0 | Cons x xs ⇒ 1 + size xs)
```

Translating Functions

Scala input

```
def size(xs: List[A]): BigInt = xs match {
  case Nil() => 0
  case Cons(_, xs) => 1 + size(xs)
}
```

Ambitious Isabelle output

```
fun size where
  size Nil = 0
| size (Cons x xs) = 1 + size xs
```

Translating Functions

Scala input

```
def size(xs: List[A]): BigInt = xs match {
  case Nil() => 0
  case Cons(_, xs) => 1 + size(xs)
}
```

Ambitious Isabelle output

```
fun size where
  size Nil = 0
| size (Cons x xs) = 1 + size xs
```

Evaluation

Coverage of Leon's Library

- almost complete

Coverage of Leon's Library

- almost complete
- ... including gnarly pattern matches

Coverage of Leon's Library

- almost complete
- ... including gnarly pattern matches
- Isabelle can prove 71% of the verification conditions automatically

Coverage of Leon's Library

- almost complete
- ... including gnarly pattern matches
- Isabelle can prove 71% of the verification conditions automatically
- missing: array, string operations

Coverage of Leon's Library

- almost complete
- ... including gnarly pattern matches
- Isabelle can prove 71% of the verification conditions automatically
- missing: array, string operations
- takes $\approx$ 40 seconds on a fast machine

Reuse

```
sealed abstract class List[T] {

  def size: BigInt = ... // implementation
}

case class Cons[T](h: T, t: List[T]) extends List[T]

case class Nil[T]() extends List[T]
```

Reuse

```
@isabelle.typ(name = "List.list")
sealed abstract class List[T] {
  @isabelle.function(term = "Int.int o List.length")
  def size: BigInt = ... // implementation
}

@isabelle.constructor(name = "List.list.Cons")
case class Cons[T](h: T, t: List[T]) extends List[T]

@isabelle.constructor(name = "List.list.Nil")
case class Nil[T]() extends List[T]
```

Reuse

```
@isabelle.typ(name = "List.list")
sealed abstract class List[T] {
  @isabelle.function(term = "Int.int o List.length")
  def size: BigInt = ... // implementation
}

@isabelle.constructor(name = "List.list.Cons")
case class Cons[T](h: T, t: List[T]) extends List[T]

@isabelle.constructor(name = "List.list.Nil")
case class Nil[T]() extends List[T]
```

Integration

```
@proof(method = "(clarsimp, induct rule: list_induct2, auto)")
def mapFstZip[A, B](xs: List[A], ys: List[B]) = {
  require(length(xs) == length(ys))
  xs.zip(ys).map(_._1)
} ensuring { _ == xs }
```

Trustworthiness

- Scala and Isabelle/HOL are quite different
  - translation is more intricate than we'd like it to be

Trustworthiness

- Scala and Isabelle/HOL are quite different
  - translation is more intricate than we'd like it to be
  - but: just 3500 lines of code (including verified code)

Trustworthiness

- Scala and Isabelle/HOL are quite different
  - translation is more intricate than we'd like it to be
  - but: just 3500 lines of code (including verified code)
- user doesn't see resulting terms and definitions

Trustworthiness

- Scala and Isabelle/HOL are quite different
  - translation is more intricate than we'd like it to be
  - but: just 3500 lines of code (including verified code)
- user doesn't see resulting terms and definitions
- countermeasure: print *report*
  - looks almost like an Isabelle theory
  - can be copy-pasted into IDE

Conclusion

- prove Scala code correct with high assurance
- co-develop specifications in Isabelle and Scala
- zero setup required, integrates seamlessly
- groundwork for using Isabelle as an API has been laid

Q & A