

Propositions as Programs, Proofs as Programs

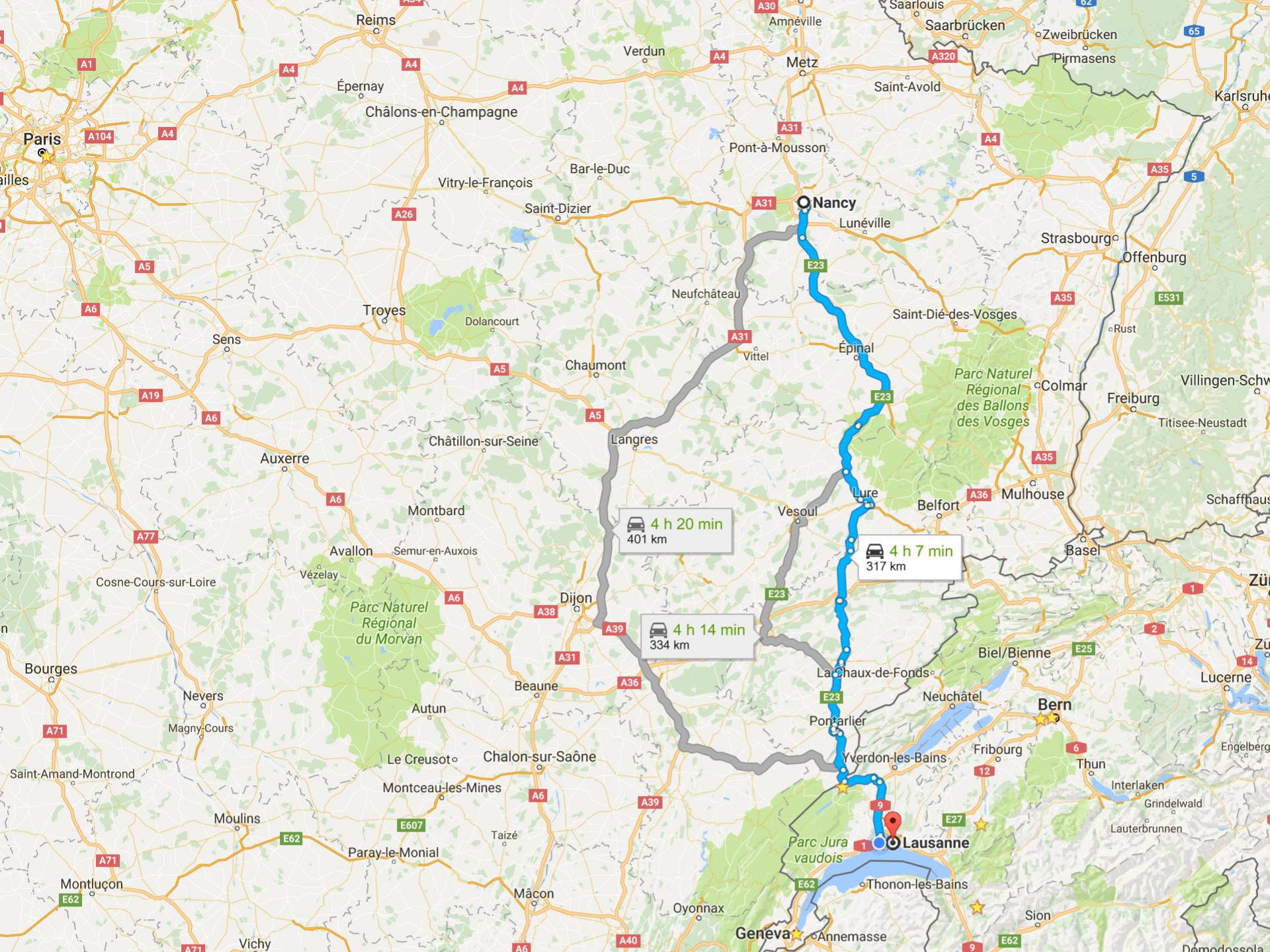
Viktor Kuncak

EPFL

Laboratory for Automated Reasoning and Analysis

<http://lara.epfl.ch>

<http://leon.epfl.ch>



EPFL



France

Lac Léman

10 minute walk from EPFL



Mont Blanc

Lac Léman



Steam Ship (1926 look and engine)



Plage du pélican

EPFL is Hiring

- MSc students (stipends, exchanges, thesis)
- PhD students: phd.epfl.ch/edic
- Postdocs (in my group)
- Faculty hiring in 2016 (check later ic.epfl.ch)

Scala programming language

Invented at EPFL by Prof. Martin Odersky <http://scala-lang.org/>

- hundreds of thousands of Scala programmers. Used by:

[Twitter](#), [Foursquare](#), [Coursera](#), [The Guardian](#), [New York Times](#), [Huffington Post](#), [UBS](#), [LinkedIn](#), [Meetup](#), [Verizon](#), Intel, Markus Wenzel, Lars Hupel
Libghtbend Inc. supports Scala commercially

EPFL Scala Center: industrial advisory board, courses, open source development

Apache Spark: “an open-source cluster computing framework with in-memory processing to speed analytic applications up to 100 times faster compared to technologies on the market today. Developed in the AMPLab at UC Berkeley, Apache Spark can help reduce data interaction complexity, increase processing speed and enhance mission-critical applications with deep intelligence.”

“...IBM is making a major commitment to the future of [Apache Spark](#), with a series of initiatives announced today. IBM will offer Apache Spark as a service on [Bluemix](#); commit **3,500 researchers** to work on Spark-related projects; donate IBM SystemML to the Spark ecosystem; and offer courses to train 1 million data scientists and engineers to use Spark.”

Leon:
a system to verify Scala programs

Try at [http://leon.epfl.ch](http://leon.epfl.ch/doc/) /doc/

Check <http://github.com/epfl-lara/>

Thanks to Leon Contributors

M. Antognini

R. Blanc

S. Gruetter

L. Hupel

E. Kneuss

M. Koukoutos

R. K. Madhavan

M. Mayer

R. Ruetschi

S. Stucki

P. Suter

N. Voirol

R. Edelmann

E. Darulova

A.S. Koeksal

S. Kulal

I. Kuraj

R. Piskac

G. Schmid

Simple Binary Search Trees in Leon

```
import leon.lang._; import leon.collection._
sealed abstract class Tree {
  def content: Set[BigInt] = this match {
    case Leaf() => Set.empty[BigInt]
    case Node(l, v, r) => l.content ++ Set(v) ++ r.content
  }

  def insert(value: BigInt): Node = (this match {
    case Leaf() => Node(Leaf(), value, Leaf())
    case Node(l, v, r) =>
      if (v < value) Node(l, v, r.insert(value))
      else if (v > value) Node(l.insert(value), v, r)
      else Node(l, v, r)
  }) ensuring(_ .content == content ++ Set(value))

  def contains(value: BigInt): Boolean = (this match {
    case Leaf() => false
    case Node(l, v, r) =>
      if (v == value) true
      else if (v < value) r.contains(value)
      else l.contains(value)
  }) ensuring(_ == (content contains value))
}

case class Leaf() extends Tree
case class Node(left: Tree, value: BigInt, right: Tree) extends Tree {
  require(forall((x: BigInt) => (left.content contains x) ==> x < value) &&
    forall((x: BigInt) => (right.content contains x) ==> value < x)) }
```

Writing Properties and Lemmas

We write universally quantified properties as functions taking arguments, returning Boolean:

```
def containsInserted(value: BigInt): Boolean = {  
  insert(value).contains(value)  
} holds
```

`holds` is a shorthand for:

```
ensuring(res => res==true)
```

`res` binds to the result of the function

`this` – an implicit receiver parameter, desugared into explicit one

$\forall \text{ this:Tree. } \forall \text{ value:Z. contains(insert(this, value), value)$

Proves immediately: decision procedure for (finite) sets after unfolding ensuring of **insert** and **contains**

How Leon Proves Expression Valid (by default)

Repeat:

- **Pretend user functions are arbitrary** functions and check if valid using an SMT solver (Z3, CVC4)
 - if there is a counterexample not relying on any user function, report it as a real counterexample, else:
- **Unfold** *body* and the corresponding **ensuring** *spec*, once for each user function application

“Bounded model checking for recursive functions”

“k-induction on termination relation for defined functions”

On top of this, we add quantifier instantiation, and modeling higher-order functions using dynamic dispatch

Example: List Size

sealed abstract class List

case class Cons(head:BigInt,tail:List) **extends** List

case object Nil **extends** List

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

```
def notFifteen(l:List): Boolean = {  
  size(l) != 15  
} holds
```

← counterexample found fast

```
def aboveZero(l:List): Boolean = {  
  size(l) >= 0  
} holds
```

← Without postcondition on size, fails to prove

Example: List Size

sealed abstract class List

case class Cons(head:BigInt,tail:List) **extends** List

case object Nil **extends** List

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

← essential postcondition for size

```
def notFifteen(l:List): Boolean = {  
  size(l) != 15  
} holds
```

← counterexample found fast

```
def aboveZero(l:List): Boolean = {  
  size(l) >= 0  
} holds
```

← Without postcondition on size, fails to prove

Append Associativity

```
sealed abstract class List {  
  def ++(that: List): List = (this match {  
    case Nil => that  
    case Cons(x, xs) => Cons(x, xs ++ that)  
  }) ensuring { res =>  
    (res.size == this.size + that.size) &&  
    ((that == Nil) ==> (res == this))  
  }  
  def size : BigInt = (this match {  
    case Nil => BigInt(0)  
    case Cons(_, t) => 1 + t.size  
  }) ensuring(res => res >= 0)  
}  
  
case class Cons(head: BigInt, tail: List)  
  extends List  
  
case object Nil extends List
```

```
@induct  
def appendAssoc(l1: List,  
                 l2: List,  
                 l3: List)= {  
  (l1 ++ l2) ++ l3 == l1 ++ (l2 ++ l3)  
} holds
```


Append Associativity Explicitly

```
def appendAssocExp(l1: List,  
                    l2: List,  
                    l3: List): Boolean = {  
  (l1 match {  
    case Cons(h,t) => appendAssocExp(t, l2, l3)  
    case _ => true  
  }) &&  
  ((l1 ++ l2) ++ l3 == l1 ++ (l2 ++ l3))  
} holds
```

Append Associativity with Sugar

```
def appendAssocExp(l1: List,  
                    l2: List,  
                    l3: List): Boolean = {  
  ((l1 ++ l2) ++ l3 == l1 ++ (l2 ++ l3)) because  
  (l1 match {  
    case Cons(h,t) => appendAssocExp(t, l2, l3)  
    case _ => true  
  })  
} holds
```

With Refined Sugar

```
def appendAssocExp(l1: List, l2: List, l3: List)= {  
  (l1 ++ l2) ++ l3 == l1 ++ (l2 ++ l3)  
} holds because  
  (l1 match {  
    case Cons(h,t) => appendAssocExp(t, l2, l3)  
    case _ => true  
  })
```

Expression to Stack Machine Compiler

By structural induction on e prove

$$\forall S. \text{run}(\text{compile}(e), S) == \text{interpret}(e) :: S$$

```
def compileInterpretEquivalenceLemma[A](e: ExprTree[A], S: List[A]) = {  
  run(compile(e), S) == interpret(e) :: S  
} holds because  
(e match {  
  case Const(c) => true  
  case Op(e1, e2) =>  
    val c1 = compile(e1)  
    val c2 = compile(e2)  
    runAppendLemma((c1 ++ c2), Cons[ByteCode[A]](OpInst[A](), Nil()), S)&&  
    runAppendLemma(c1, c2, S) &&  
    compileInterpretEquivalenceLemma(e1, S) &&  
    compileInterpretEquivalenceLemma(e2, Cons[A](interpret(e1), S)) })
```


Proof Domain-Specific Language

Course project: S. Stucki and M. Antognini

```
def reverseReverse[T] (l: List[T]): Boolean = {  
  l.reverse.reverse == l  
}.holds because {  
  l match {  
    case Nil()          => trivial  
    case Cons(x, xs) => {  
      (xs.reverse :+ x).reverse ==| snocReverse[T] (xs.reverse, x) |  
      x :: xs.reverse.reverse   ==| reverseReverse[T] (xs)         |  
      (x :: xs)  
    }.qed  
  }  
}}
```

Implemented entirely as a library (no change to Leon!):
implicit conversion to new type that has additional method
such as **==|**

<https://github.com/epfl-lara/leon/tree/master/library/leon/proof>

<http://leondevel.epfl.ch/doc/neon.html>

Resource Verification Problem

```
def sortedIns(e: BigInt, l: List)= {  
  require(isSorted(l))  
  l match {  
    case Nil() => Cons(e, Nil())  
    case Cons(x, xs) =>  
      if (x <= e)  
        Cons(x, sortedIns(e, xs))  
      else  
        Cons(e, l)  
  }  
} ensuring(res =>  
  // completes in O (size(l)) time/space  
)
```

Specifying Resource Bounds

```
def sortedIns(e: BigInt, l: List)= {  
  require(isSorted(l))  
  l match {  
    case Nil() => Cons(e, Nil())  
    case Cons(x, xs) =>  
      if (x <= e)  
        Cons(x, sortedIns(e, xs))  
      else  
        Cons(e, l)  
  }  
} ensuring(res => time <= ?*size(l)+? && stack <= ?*size(l)+? )  
               time <= a*size(l)+b && stack <= c*size(l)+d
```

Ensuring clauses with holes to be inferred

Example Programs & Templates

AVL tree	$t_{\text{delete}} \leq a * \text{height}(t) + b$
Red-black tree	$t_{\text{insert}} \leq a * \text{blackHeight}(t) + b$
Leftist Heap	$t_{\text{rem}} \leq a * \text{rightHeight}(t.\text{left}) + b$
Binomial Heap	$t_{\text{remove}} \leq a * \text{numTrees}(h) +$ $b * \text{minChildren}(h) + c$
Insertion Sort	$t_{\text{sort}} \leq a * \text{size}(l) * \text{size}(l) + b$

Bounds Inferred by the Tool

AVL tree	$t_{\text{delete}} \leq 145 * \text{height}(t) + 19$
Red-black tree	$t_{\text{insert}} \leq 178 * \text{blackHeight}(t) + 96$
Leftist Heap	$t_{\text{rem}} \leq 44 * \text{rightHeight}(t.\text{left}) + 5$
Binomial Heap	$t_{\text{remove}} \leq 70 * \text{numTrees}(h) +$ $31 * \text{minChildren}(h) + 22$
Insertion Sort	$t_{\text{sort}} \leq 9 * \text{size}(l) * \text{size}(l) + 2$

Also Inferred

Red-black tree	$2^{\text{blackHeight}(t)} \leq \text{size}(t) + 1$
Leftist Heap	$2^{\text{rightHeight}(t)} \leq \text{size}(t) + 1$
Binary inc. (Amortized)	$t_{\text{inc}} \leq 15 * \text{nop} + 3$

Implies logarithmic time for access

Extended to Lazy Evaluation and Memoization

Verified about 10k lines - correctness and resource bounds

- Heaps: Leftist heap, binomial heap
- Sorting algorithms
- Balanced search trees
- Tries
- Stream library
- Cyclic streams: fibonacci stream, hamming stream
- Lazy persistent data structures: real-time queues, dequeues
- Dynamic programming algorithms: packrat parsing instance, Viterbi algorithm

We used proof hints in the more difficult examples

Example Case Study: ConcTrees

- Work of a PhD student of Martin Odersky, basis of a parallel collection data structure

Aleksandar Prokopec, [Martin Odersky](#):
Conc-Trees for Functional and Parallel Programming. [LCPC 2015](#): 254-268

- Part of it explained in the last week of the Coursera course on parallel programming in Scala:
<https://www.coursera.org/learn/parprog1>
- Formalized a lazy data structure and its time bounds
- 800 lines of Leon for data structure operations, specifications, proof hints

Expressive Framework

- If a lemma application appears in formula, it will be eventually unfolded (fair unfolding)
- To request Leon to consider a set of lemmas:
 - Define MyRules datatype: a case for each lemma
 - Define function that traverses List[MyRules] and instantiates corresponding lemma
 - Add List[MyRules] as an extra parameter to the formula that needs to be proven
 - Parameters must encode all possible expressions to which lemmas is to be applied (!)
- Even if it gives provability, having many unfoldings makes solving very slow

What we cannot do?

- Cannot control what *is* not given to SMT solver
 - can: add lemma instances (must give arguments)
 - harder: tell it to ignore some fact as irrelevant (e.g. hide proof part of “because”, prevent unfolding)
part of the problem: for induction to work, want &&
 - smaller modifications to system can address this
- Cannot write general-purpose tactic that proves a formula from a given decidable class
 - No way to programmatically inspect formulas and decide which rules/lemmas to instantiate, which not
 - Solution?

Future: Reflection and LCF Approach

Ongoing work by EPFL PhD student Romain Edelmann

- Reflect Leon's terms and formulas in Leon user space as algebraic data types
- Introduce private Theorem type; stores a term
- If $t:\text{Theorem}$, then $t.\text{term}$ gives formula (cannot go arbitrarily in the other direction!)
- Inspect theorems: **$t.\text{term match \{...\}}$**
 - can write tactics
- Allow terminating computations to compute theorems using a trusted kernel of formula constructing functions

Envisioned And Introduction

```
def andIntro(a: Theorem, b: Theorem): Theorem = {  
  ???[Theorem]  
} ensuring(_.term == And(a.term, b.term))
```

- Can find out the resulting formula by looking at **ensuring** alone!
- Once we check the body and prove it terminates, we can skip executing it
- Avoid re-running such Theorem-producing computations after we verify them, just look up the value according to **ensuring**

Possible Implication Introduction?

```
def implIntro(hyp: Term, concl: Term,  
              p: Theorem => Theorem): Theorem={  
  require(forall((t:Theorem) =>  
                (t.term==hyp) ==> p(t).term==concl))  
  ???[Theorem]  
} ensuring(_.term == Implies(hyp, concl))
```

- Possible because language used to compute theorems is terminating and free of side effects
 - Enforced using Leon's termination and verification (allows many forms of recursion)
- To avoid re-evaluation, need predictable support for contracts used for kernel rules like above

Two Proof Languages

- Proofs as Leon terms of Boolean type
 - for (readable?) proofs of specific theorems
- Proofs as Leon programs computing Theorems
 - tactics, decision procedures
 - larger developments

In both cases, proofs are programs

Program synthesis techniques apply to proofs

Activities Supported in Leon

- a) **Check assertion** while program **p** runs: **$C(i, p(i))$**
- b) **Verify** whether program always meets the spec:
 $\forall i. c(i, p(i))$
- c) **Constraint programming**: once **i** is known, find **o** to satisfy a given constraint: **find o such that $C(i, o)$**
- d) **Synthesis**: solve **C** symbolically to obtain program **p** that is correct by construction, for all inputs: **find p such that $\forall i. C(i, p(i))$** i.e. **$p \subseteq C$**
- repair**

Recent work: verify **time and space, quantifiers, state, higher-order functions, termination**, Isabelle linkup

Insertion Sort Synthesis

```
def content(l: List): Set[BigInt] = l match {  
  case Nil => Set()  
  case Cons(i, t) => Set(i) ++ content(t) }
```

```
def isSorted(l: List): Boolean = l match {  
  case Nil => true  
  case Cons(_, Nil) => true  
  case Cons(x1, Cons(x2, rest)) =>  
    x1 < x2 && isSorted(Cons(x2, rest)) }
```

```
def sort(l: List) = {  
  ???[List]  
} ensuring((res: List) =>  
  isSorted(res) &&  
  content(res) == content(l))
```

```
def sInsert(x: BigInt, l: List): List = {  
  require(isSorted(l))  
  l match {  
    case Nil => Cons(x, Nil)  
    case Cons(e, rest) if (x == e) => l  
    case Cons(e, rest) if (x < e) =>  
      Cons(x, Cons(e, rest))  
    case Cons(e, rest) if (x > e) =>  
      Cons(e, sInsert(x, rest))  
  }  
} ensuring {(res: List) =>  
  isSorted(res) &&  
  content(res) == content(l) ++ Set(x)}
```

Isabelle Linkup

- Developed by Lars Hupel:
[Translating Scala Programs to Isabelle/HOL \(System Description\)](#), by *Lars Hupel* and *V.K.*
International Joint Conference on Automated Reasoning (IJCAR), 2016.
- Largely orthogonal to SMT solver approach discussed in this talk; we can use them together
 - Maps Leon type and function definitions into Isabelle's datatypes, recursive functions, and lemmas
 - Invokes Isabelle to prove lemmas (accepts proof hints)
 - Maps e.g. Leon's list operations to those of Isabelle

Conclusion

Leon started as automated verification system

- Also used for research in synthesis and repair

Functions are proved (terminating and)
valid for all arguments

Lemmas are functions returning Boolean

To do more difficult proofs, we help Leon:

- conjoin goals with applications of useful lemmas
- proof by induction: invoke property recursively
- Scala DSL for proofs: nice equational reasoning syntax

Experience in proofs of data structures, some dist. protocols

To do much more, we are adding reflection

Open positions at EPFL (MSc, PhD, postdoc, faculty)