Verification of Functional Programs in Scala

Philippe Suter

(joint work w/ Ali Sinan Köksal and Viktor Kuncak)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, SWITZERLAND

Photo: Hisao Suzuki

```
~$ ./demo
```

Leon

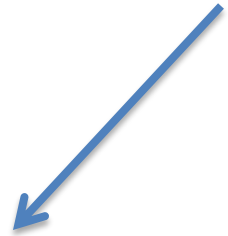
- A verifier for Scala programs.
- The programming and specification languages are the same, purely functional, subset.


```
def content(t: Tree) = t match {
  case Leaf ⇒ Set.empty
  case Node(l,v,r) ⇒
    (content(l) ++ content(r)) + e
}
```

```
def insert(e:Int, t: Tree) = t match {
  case Leaf ⇒ Node(Leaf,e,Leaf)
  case Node(l,v,r) if e < v ⇒
    Node(insert(e,l),v,r)
  case Node(l,v,r) if e > v ⇒
    Node(l,v,insert(r))
  case _ ⇒ t
} ensuring(
    res ⇒ content(res) == content(t) + e)
```

Postconditions

```
def size(lst: List) = (lst match {
  case Nil ⇒ 0
  case Cons(_, xs) ⇒ 1 + size(xs)
}) ensuring(res ⇒ res ≥ 0)
```



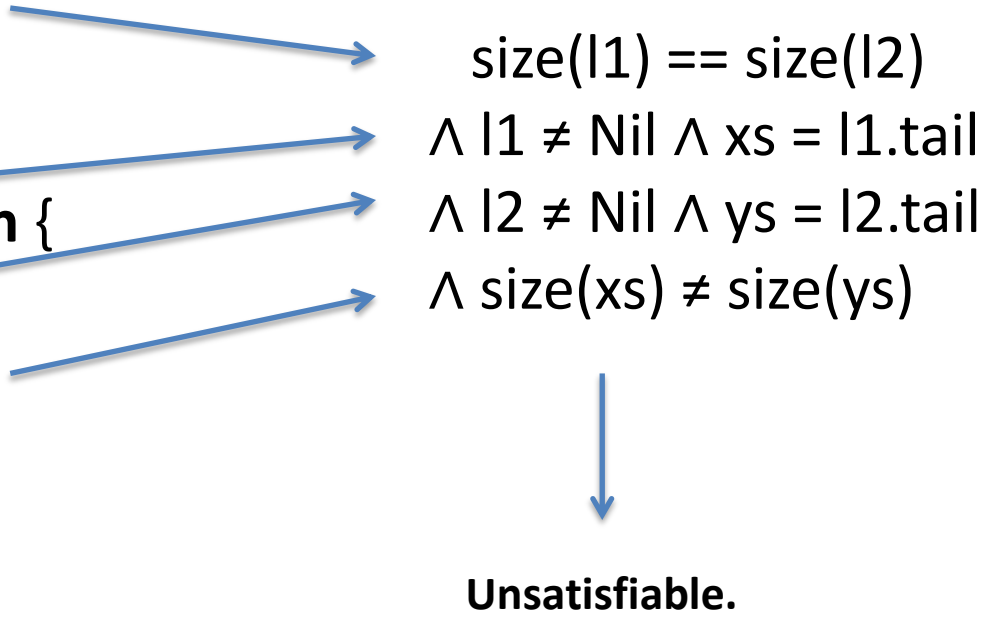
```
(size(xs) ≥ 0) ⇒ (lst match {
  case Nil ⇒ 0
  case Cons(_, xs) ⇒ 1 + size(xs)
}) ≥ 0
```

```
(size(xs) ≥ 0) ∧ (lst match {
  case Nil ⇒ 0
  case Cons(_, xs) ⇒ 1 + size(xs)
}) < 0
```

Preconditions

- We encode the *path condition* and use it to prove that precond. can't be violated.

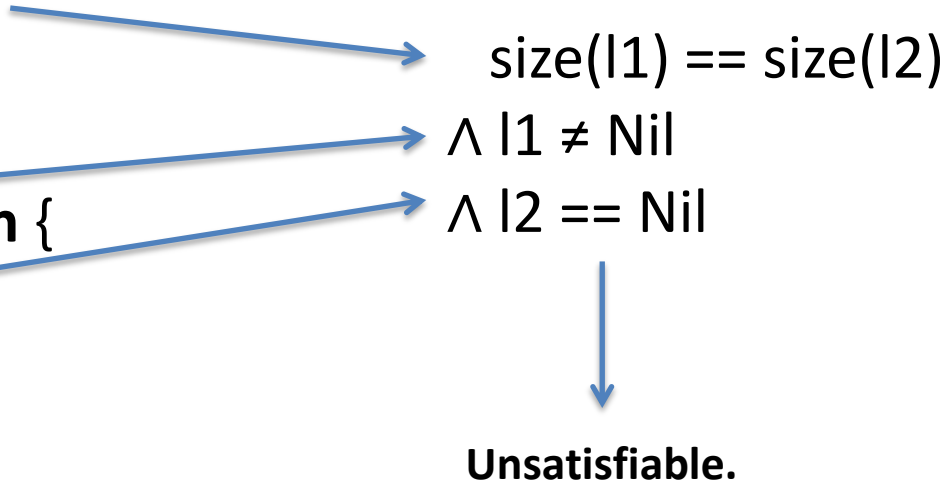
```
def zip(l1: List, l2: List) : PairList = {
  require(size(l1) == size(l2))
  l1 match {
    case Nil() ⇒ PNil()
    case Cons(x, xs) ⇒ l2 match {
      case Cons(y, ys) ⇒
        PCons(P(x, y), zip(xs, ys))
    }
  }
}
```



Pattern-Matching Exhaustiveness

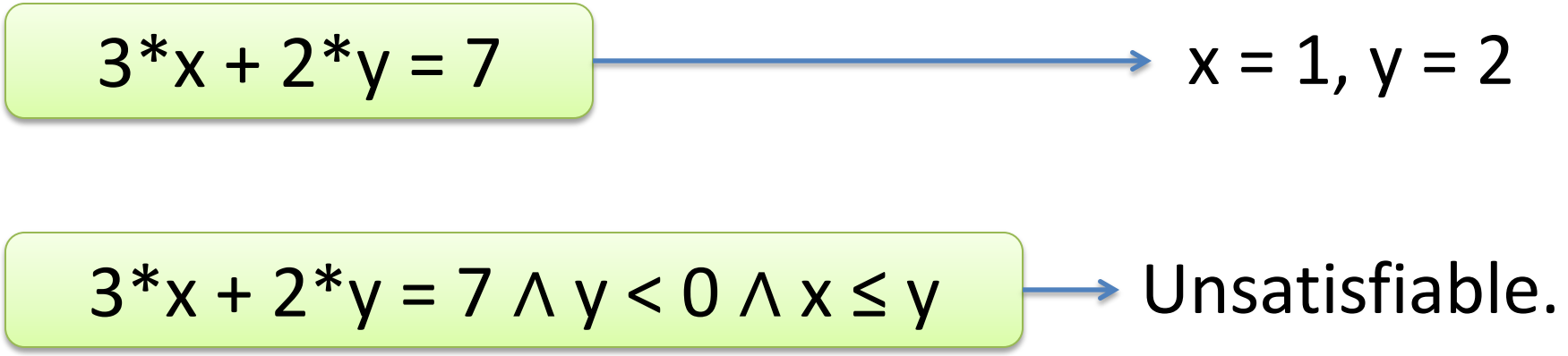
- We generate a formula that expresses that no **case** matches, and prove it unsatisfiable.

```
def zip(l1: List, l2: List) : PairList = {
  require(size(l1) == size(l2))
  l1 match {
    case Nil() ⇒ PNil()
    case Cons(x, xs) ⇒ l2 match {
      case Cons(y, ys) ⇒
        PCons(P(x, y), zip(xs, ys))
    }
  }
}
```



Decision Procedures

- Algorithms that answer a satisfiability/validity question for a class of formulas.



φ is valid $\Leftrightarrow$ *¬φ is unsatisfiable.*

Satisfiability Modulo Theories Solvers

- Essentially, efficient implementations of decision procedures.
- Decide the satisfiability of a formula modulo a *combination* of theories.
- Usually for quantifier-free formulas.

SMT Solving

$$
\begin{aligned}
& l_1 = \mathrm{Cons}(e_1, l_2) \\
\wedge\ & (l_2 = \mathrm{Nil} \vee e_1 = 0) \\
\wedge\ & (f(e_1) \neq f(e_2) \vee l_1 = \mathrm{Nil}) \\
\wedge\ & (e_2 = 0 \vee f(e_2) = 0)
\end{aligned}
$$

$$
\begin{aligned}
& l_1 = \mathrm{Cons}(e_1, l_2) \\
& f(e_1) \neq f(e_2) \\
& \neg(f(e_2) = 0) \\
& \sout{\neg(l_2 = \mathrm{Nil})} \\
& \sout{e_1 = 0} \\
& \sout{e_2 = 0} \\
& l_2 = \mathrm{Nil} \\
& e_2 = 0
\end{aligned}
$$

$l_1 \rightarrow \mathrm{Cons}(e_1, l_2), l_2 \rightarrow \mathrm{Nil},$
$e_1 \rightarrow 1, e_2 \rightarrow 0,$
$f : \{\ 1 \rightarrow 1,\ \_ \rightarrow 0\ \}$

Assignment to the free variables, and a model for the functions symbols that satisfy the axiom: $a = b \Rightarrow f(a) = f(b)$.

SMT + Computable Functions

Tree ::= Leaf | Node(Tree, Int, Tree)

content(Leaf) = ∅

content(Node(t_1, e, t_2)) = content(t_1) ∪ { e } ∪ content(t_2)

$$
\begin{aligned}
& t_1 = \text{Node}(t_2, e_1, t_3) \\
\wedge\ & e_1 > e_2 \\
\wedge\ & \text{content}(t_4) = \text{content}(t_2) \cup \{ e_2 \} \\
\wedge\ & \text{content}(\text{Node}(t_4, e_1, t_3)) \neq \text{content}(t_1) \cup \{ e_2 \}
\end{aligned}
$$

…of quantifier-free formulas in a decidable base theory…

Satisfiability Modulo Computable Functions

…pure, total, deterministic, first-order and terminating on all inputs…

- Semi-decidable problem worthy of attention.
- What are general techniques for proving and disproving constraints?
- What are interesting decidable fragments?

Proving with Inlining

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

```
def sizeTR(lst: List, acc: Int) = lst match {
  case Nil ⇒ acc
  case Cons(_, xs) ⇒ sizeTR(xs, 1 + acc)
} ensuring(res ⇒ res = size(lst) + acc)
```

size(lst) = sizeTR(lst, 0)

```
def size(lst: List) = if(lst = Nil) {
  0
} else {
  1 + size(lst.tail)
}
```

```
def sizeTR(lst: List, acc: Int) = if (lst = Nil) {
  acc
} else {
  sizeTR(lst.tail, 1 + acc)
} ensuring(res ⇒ res = size(lst) + acc)
```

Proving with Inlining

```
def size(lst: List) = if(lst = Nil) {
  0
} else {
  1 + size(lst.tail)
}
```

```
def sizeTR(lst: List, acc: Int) = if (lst = Nil) {
  acc
} else {
  sizeTR(lst.tail, 1 + acc)
} ensuring(res ⇒ res = size(lst) + acc)
```

∀ lst, ∀ acc : (**if**(lst = Nil) acc **else** sizeTR(lst.tail, 1 + acc)) = size(lst) + acc

∃ lst, ∃ acc : (**if**(lst = Nil) acc **else** sizeTR(lst.tail, 1 + acc)) ≠ size(lst) + acc

lst → Nil, acc → 0, size : { Nil → 1, _ → 0 }, sizeTR : { _ → 0 }

Proving with Inlining

∃ lst, ∃ acc :

(**if**(lst = Nil) acc **else** sizeTR(lst.tail, 1 + acc)) ≠ size(lst) + acc

∧ size(lst) = **if**(lst = Nil) 0 **else** 1 + size(lst.tail)

∧ sizeTR(lst.tail, 1 + acc) = size(lst.tail) + 1 + acc

~~lst → Nil, acc → 0, size : { Nil → 1, _ → 0 }, sizeTR : { _ → 0 }~~

~~lst → Cons(0, Nil), acc → 1, size : { _ → 0 }, sizeTR : { _ → 0 }~~

⇒ Unsatisfiable.

Disproving with Inlining

```
def size(lst: List) = lst match {
  case Nil ⇒ 1
  case Cons(_, Nil) ⇒ 1
  case Cons(_, xs) ⇒ 1 + size(xs)
}
```

```
def sizeTR(lst: List, acc: Int) = lst match {
  case Nil ⇒ acc
  case Cons(_, xs) ⇒ sizeTR(xs, 1 + acc)
}
```

size(lst) = sizeTR(lst, 0)

```
def size(lst: List) = if(lst = Nil) {
  1
} else if(lst.tail = Nil) {
  1
} else {
  1 + size(lst.tail)
}
```

```
def sizeTR(lst: List, acc: Int) = if (lst = Nil) {
  acc
} else {
  sizeTR(lst.tail, 1 + acc)
}
```

Disproving with Inlining

```
def size(lst: List) = if(lst = Nil) {
  1
} else if(lst.tail = Nil) {
  1
} else {
  1 + size(lst.tail)
}
```

```
def sizeTR(lst: List, acc: Int) = if (lst = Nil) {
  acc
} else {
  sizeTR(lst.tail, 1 + acc)
}
```

$\forall$ lst, $\forall$ acc : (**if**(lst = Nil) acc **else** sizeTR(lst.tail, 1 + acc)) = size(lst) + acc

$\exists$ lst, $\exists$ acc : (**if**(lst = Nil) acc **else** sizeTR(lst.tail, 1 + acc)) $\neq$ size(lst) + acc

lst → Cons(0, Nil), acc → 0, size : { _ → 1 }, sizeTR : { _ → 0 }

Disproving with Inlining

∃ lst, ∃ acc :

(**if**(lst = Nil) 0 **else** sizeTR(lst.tail, 1 + acc)) ≠ size(lst) + acc

∧ size(lst) = **if**(lst = Nil ∨ lst.tail = Nil) 1 **else** 1 + size(lst.tail)

∧ sizeTR(lst.tail, 1 + acc) = **if** (lst.tail = Nil) 1 + acc **else**
sizeTR(lst.tail.tail, 2 + acc)

∧ size(lst.tail) = **if**(lst.tail = Nil ∨ lst.tail.tail = Nil) 1 **else** 1 +
size(lst.tail.tail)

∧ sizeTR(lst.tail.tail, 2 + acc) = **if** (lst.tail.tail = Nil) 2 + acc **else**
sizeTR(lst.tail.tail.tail, 3 + acc)

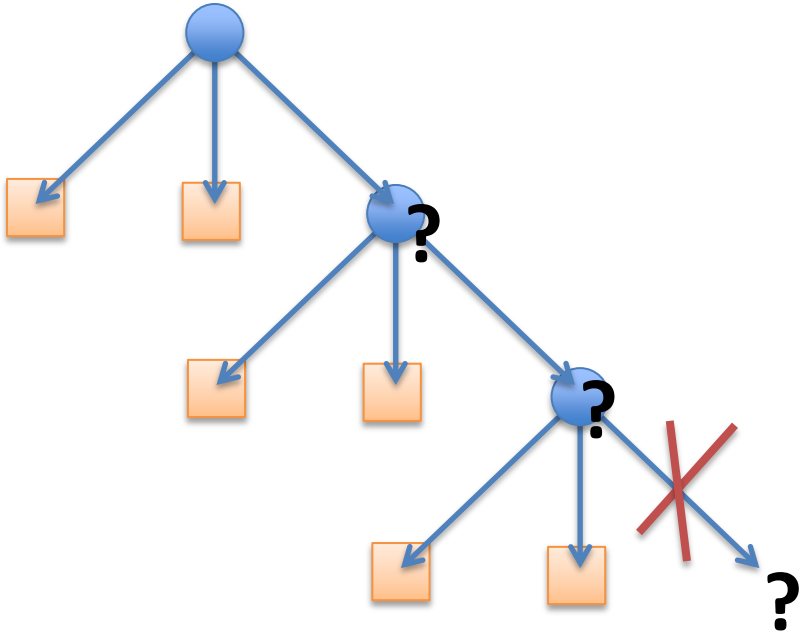
lst → [0, 1], acc → 0, sizeTR : { _ → 0 }
size : { [0] → 1, [0, 1] → 2, _ → 0 }

lst → [0, 1, 2], acc → 0, sizeTR : { _ → 0 },
size : { [0] → 1, [0, 1] → 2, [0, 1, 2] → 3, _ → 0 }

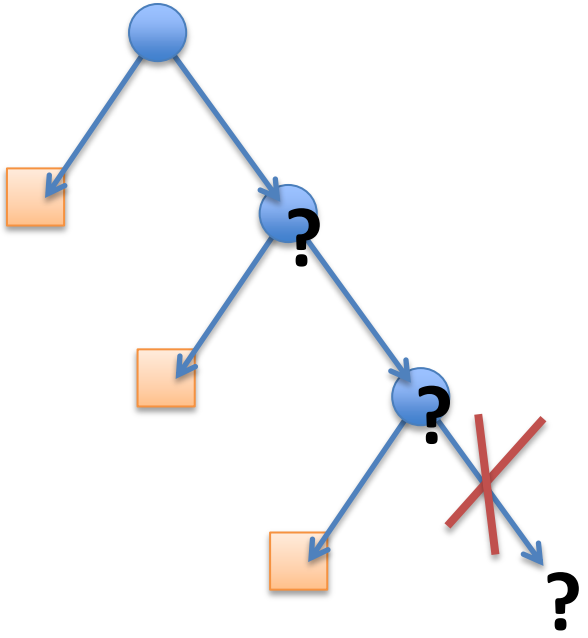
...

Disproving with Inlining

size(lst)



sizeTR(lst, aux)

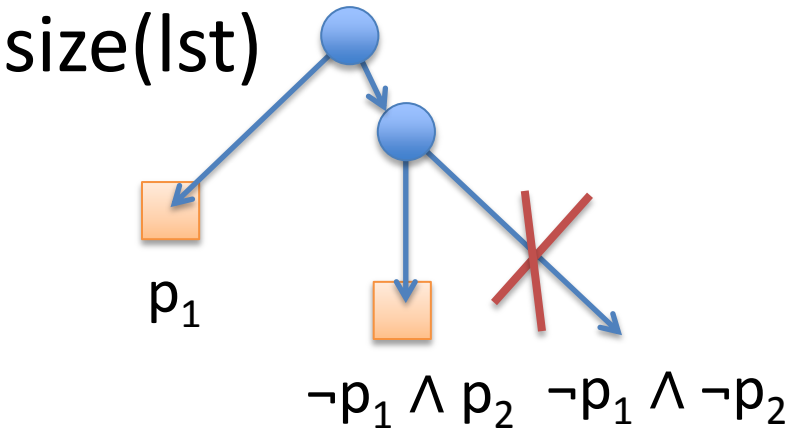


There are always unknown branches in the evaluation tree.
We can never be sure that there exists no smaller solution.

Branch Rewriting

```
size(lst) = if(lst = Nil) {
  1
} else if(lst.tail = Nil) {
  1
} else {
  1 + size(lst.tail)
}
```

$$
\begin{aligned}
&\text{size(lst)} = \text{ite}_1 \\
&\wedge\ p_1 \Leftrightarrow\ \text{lst} = \text{Nil} \\
&\wedge\ p_1 \Rightarrow \text{ite}_1 = 1 \\
&\wedge\ \neg p_1 \Rightarrow \text{ite}_1 = \text{ite}_2 \\
&\wedge\ p_2 \Leftrightarrow\ \text{lst.tail} = \text{Nil} \\
&\wedge\ p_2 \Rightarrow \text{ite}_2 = 1 \\
&\wedge\ \neg p_2 \Rightarrow \text{ite}_2 = 1 + \text{size(lst.tail)}
\end{aligned}
$$



…

$\wedge\ p_2$

Algorithm

```
(φ, B) = unroll(φ, _)
while(true) {
  solve(φ ∧ B) match {
    case "SAT" ⇒ return "SAT"
    case "UNSAT" ⇒ solve(φ) match {
      case "UNSAT" ⇒ return "UNSAT"
      case "SAT" ⇒ (φ, B) = unroll(φ, B)
    }
  }
}
```

Some literals in B may be implied by φ : no need to unroll what they guard.

"I'm feeling lucky"

Inlining must be *fair*

Inlines some postconditions and bodies of function applications that were guarded by a literal in B, returns the new formula and new set of guards B.

Assumptions & Guarantees

1) All functions terminate on all inputs.
2) All functions satisfy their postconditions.
3) All function invocations satisfy the precondition.
4) All **match** expressions are exhaustive.
5) The SMT solver is sound and complete.

(We currently prove 2) – 4).)

Properties of the Algorithm

Three Facts

1) The algorithm terminates when there exists an assume/guarantee style inductive proofs.

2) The algorithm terminates when the formula admits a counter-example.

3) (The algorithm is a decision procedure for *sufficiently surjective* abstraction functions.)

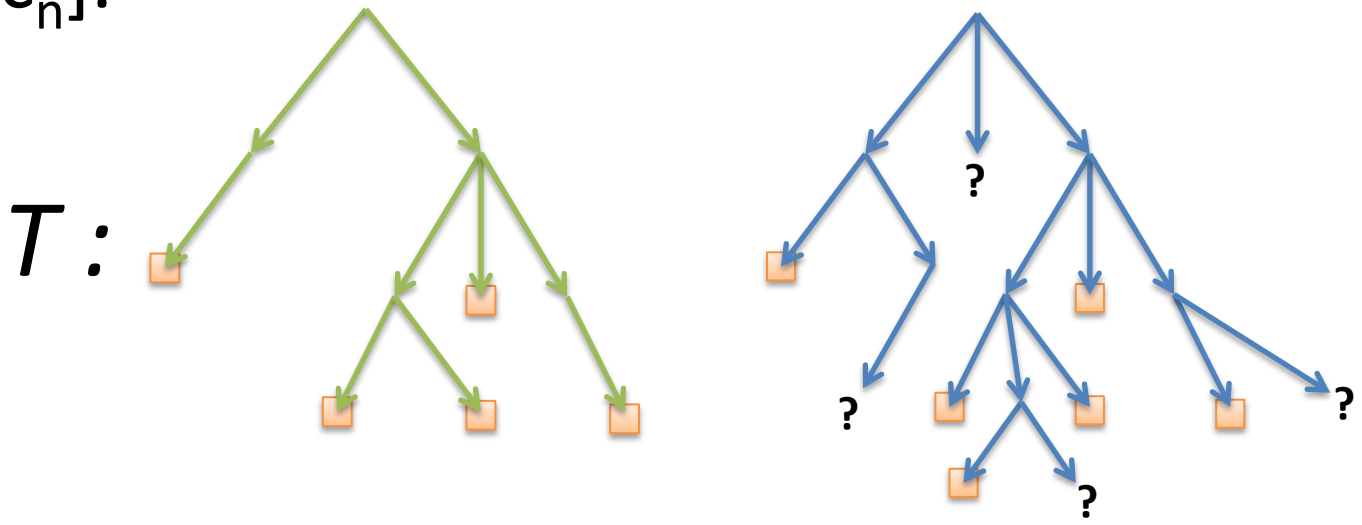
Inductive Proofs

- If there exists a proof in assume/guarantee style, it will be found by sufficient inlining of the postconditions.

- Also succeeds when the property becomes inductive only after a certain number of iterations (à la *k*-induction).

Counter-examples

- Let $a_1,\ldots a_n$ be the free variables of φ, and $c_1,\ldots,c_n$ be a counter-example to φ.
- Let *T* be the evaluation tree of $\varphi[a_1 \rightarrow c_1, \ldots, a_n \rightarrow c_n]$.

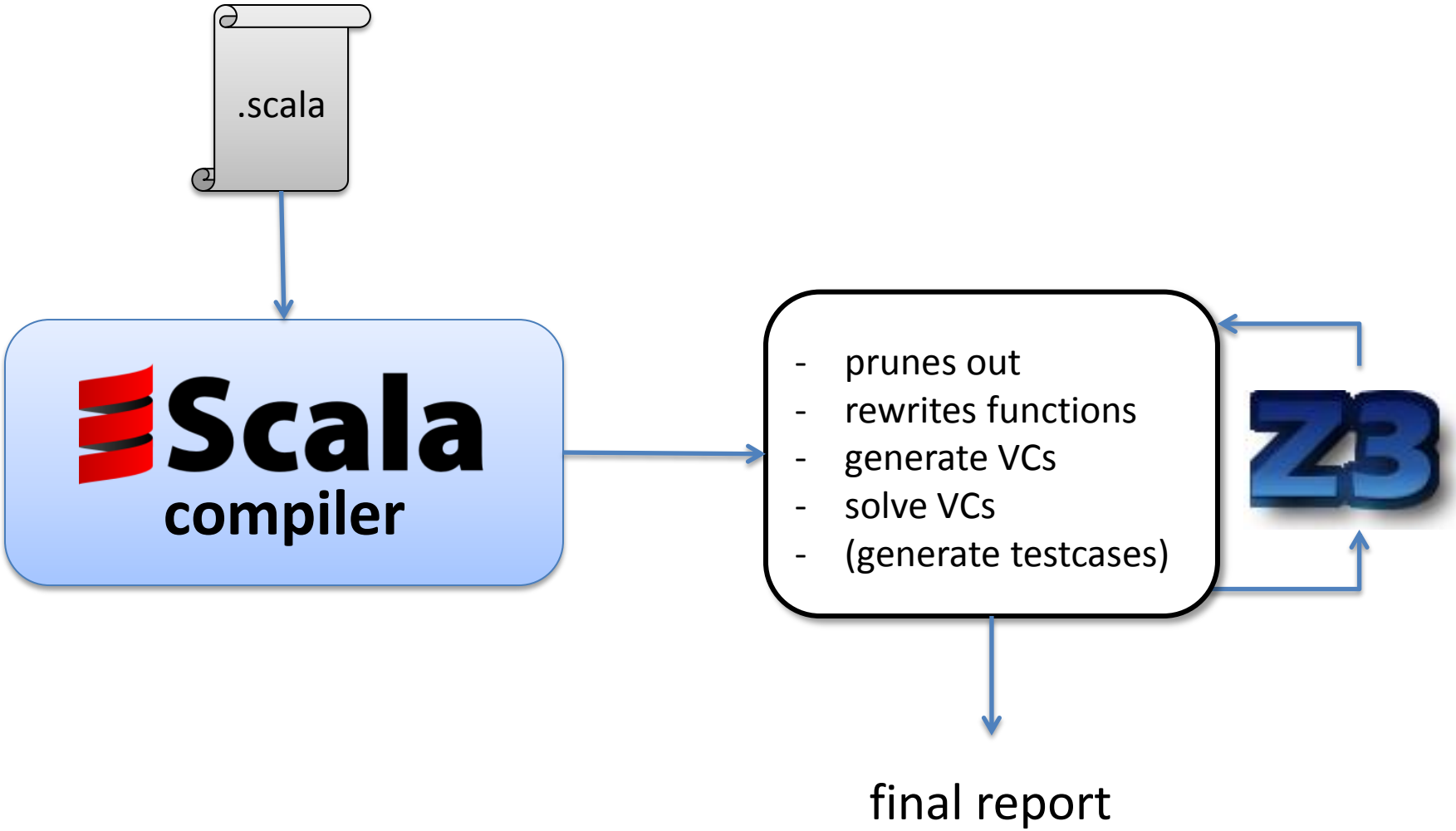


- *Eventually, the algorithm will reach a point where* T *is covered.*

Verification System.

Photo: Hisao Suzuki

Leon



Try out Leon*Online* : http://lara.epfl.ch/leon/

Leon

- Proves that all match expressions are exhaustive.
- Proves that the preconditions imply the postconditions.
- Proves that all function invocations satisfy the preconditions.
- Can generate testcases that satisfy some precondition.

Some Experimental Results

Benchmark	LoC	#Funs.	#VCs.	Time (s)
ListOperations	122	15	22	1.29
AssociativeList	60	5	11	0.91
InsertionSort	86	6	9	0.87
RedBlackTrees	112	10	24	2.98
PropositionalLogic	86	9	23	4.17

Functional correctness properties of data structures: red black trees implement a set and maintain height invariants, associative list has read-over-write property, insertion sort returns a sorted list of identical content, etc. Properties of propositional logic transformations: nnf and removing implications are stable, applying a (wrong) simplification to an nnf formula does not keep it in nnf, etc.

Limitations & Future work

- System features
  - Termination proofs.
  - Generic types.
  - Support for more Scala constructs (tuples, etc.).
- Proof system
  - Currently, supports only a limited form of induction on arguments (**@induct**).
  - Folding is limited to functions present in the formula.

Related Work

- SMT Solvers + axioms
  - can be very efficient for well-formed axioms
  - in general, no guarantee than instantiations are fair
  - cannot in general conclude satisfiability (changing…)
- Interactive verification systems (ACL2, Isabelle, Coq)
  - very good at building inductive proofs
  - could benefit greatly from counter-example generation (as feedback to the user but also to prune out branches in the search)
- Sinha's *Inertial Refinement* technique
  - similar use of "blocked paths"
  - no guarantee to find counter-examples

Related Work cont'd

- DSolve (Liquid Types)
  - because the proofs are based on type inference rules, cannot immediately be used to prove assertions relating different functions
- Bounded Model Checking
  - offer similar guarantees on the discovery of counter-examples
  - applied to the verification of temporal properties
  - reduces to SAT rather than SMT
- Finite Model Finding (Alloy, Paradox)
  - lack of theory reasoning, no proofs