Counter-Example Complete Verification for Higher-Order Functions

N. Voirol, E. Kneuss, V. Kuncak
EPFL

Scala Symposium 2015

Program Verification

- Aims to guarantee (or disprove) properties about programs
- Performed statically at compile time
  - Benefits from other analyses (typing, CFG, etc.)
  - Independent of program inputs
- Can be viewed as an extremely precise and powerful type system

Verification Systems

Interesting properties :

- **Soundness** = proofs (or disproofs) are valid
- **Completeness** = if a proof (or disproof) exists, it will be reported
- Performance
- Expressivity

Merge Sort Implementation

```
def split(list: List[Int]): (List[Int], List[Int]) = list match {
}
```

```
def merge(l1: List[Int], l2: List[Int]): List[Int] = (l1, l2) match {
}
```

```
def mergeSort(list: List[Int]): List[Int] = list match {
  case Cons(h1, t1 @ Cons(h2, t2)) =>
    val (l1, l2) = split(list)
    merge(mergeSort(l1), mergeSort(l2))
  case _ => list
}
```

Verifying Sortedness

```
def isSorted(list: List[Int]): Boolean = list match {
  case Cons(h1, t1 @ Cons(h2, xs)) => h1 <= h2 && isSorted(t1)
  case _ => true
}
```

Result of mergeSort for *any* input must be sorted (*i.e.* isSorted must return **true**)

Verification Condition

- Boolean property on program
- Encoded into quantifier-free (QF) formula

$\forall$ list: List[Int]. isSorted(mergeSort(list))

- or equivalently -

!isSorted(mergeSort(list)) $\in$ UNSAT

Program Verification in *Leon*

- Transform boolean expression into formula

  verification condition $p \rightarrow$ formula f

- Use SMT solver to verify $\neg f$
  - $\neg f \in$ UNSAT

    no inputs can break condition
  - $\neg f \in$ SAT

    produces a breaking model : counter-example

Leon Verification System

First-Order Verification in *Leon*

- Encoding to formulas well supported for many language features
- How to encode recursive definitions?

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

Naive Recursive Definitions

Just use universal quantification :

$\forall$ list: List[T]. size(list) = list **match** {
  case Cons(x, xs) => 1 + size(xs)
  case Nil() => 0
}

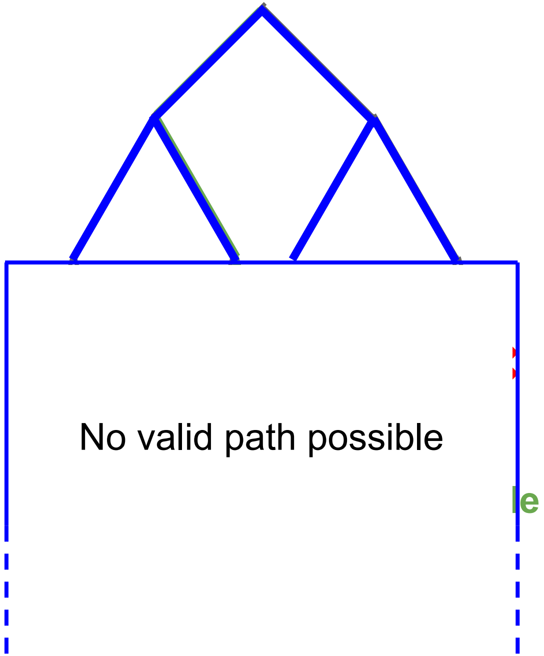
Unfortunately not (yet) well supported by SMT solvers

Unfolding Procedure in *Leon*

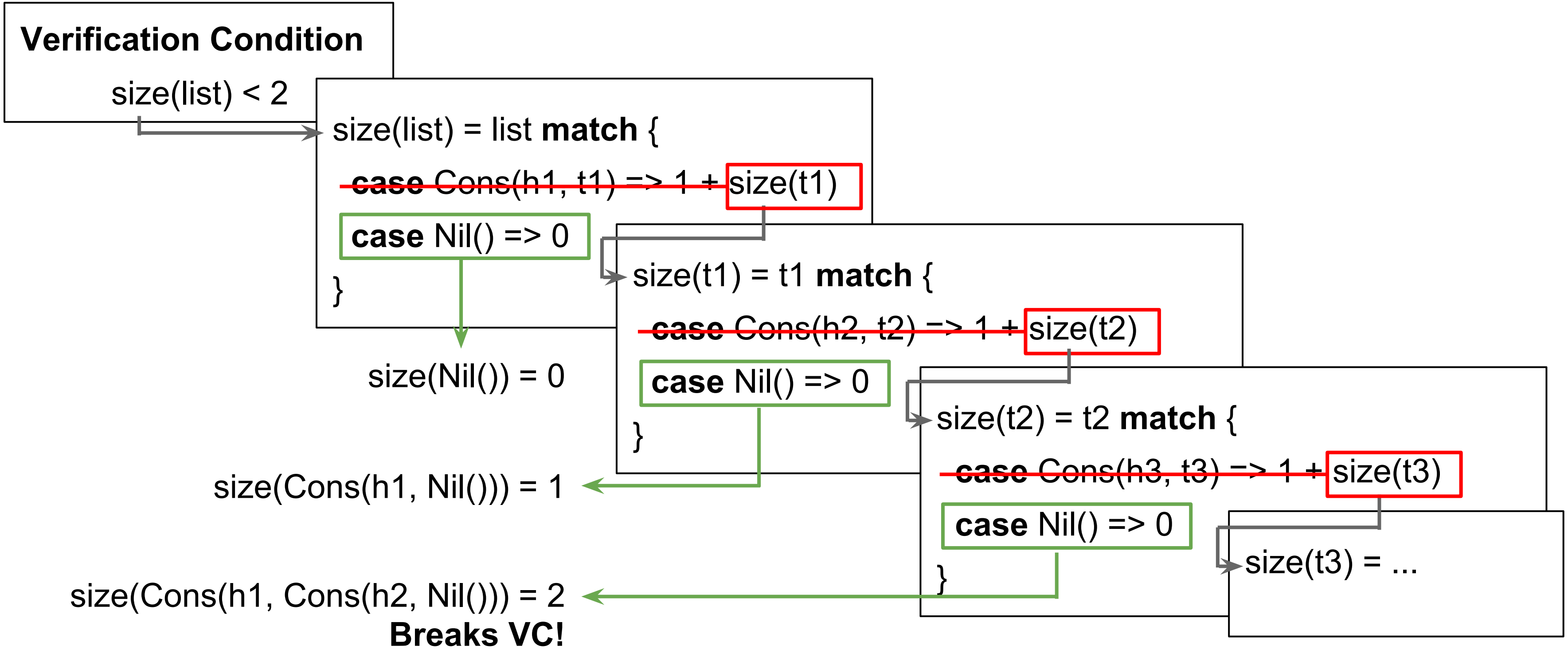
- Progressively inline function calls
- Instrument decision tree so execution tree can be limited to subset that doesn't depend on further inlinings
- At each inlining step :
    - if ¬*f* with blocked branches ∈ SAT

      model is a counter-example
    - if ¬*f* ∈ UNSAT

      VC is valid



Unfolding Procedure - Example I



Unfolding Procedure - Example II

Verification Condition

size(list) < 0

First call is simply inlined to avoid circular logic

size(list) = list **match** {
  case Cons(h1, t1) => 1 + size(t1)
  case Nil() => 0
}

size(t1) = t1 **match** {
  case Cons(h2, t2) => 1 + size(t2)
  case Nil() => 0
}

size(t1) >= 0

No result of size(t1) can break VC!

Why Higher-Order Functions?

- Important feature of functional languages
- Interesting extension to first-order case
  - can’t statically track closure definitions for unfolding
  - decision tree branches that need blocking can’t be statically determined
  - no natural encoding in the formula domain

HOF Examples

First-Class Functions - Approach

Key observation:

we cannot track arbitrary closures through the program …

… but we can track the set of all closures generated or input into the program

Use dynamic dispatch!

First-Class Functions - Dispatching

Set of all closures is Λ = { (x: Int) => x + 1, (x: Int) => x + 2, (x: Int) => 2 }

$$f(x) = \begin{cases} x+1 & \textbf{if } f = \textit{Ident}[\text{(x: Int) => x + 1}] \\ x+2 & \textbf{if } f = \textit{Ident}[\text{(x: Int) => x + 2}] \\ 2 & \textbf{if } f = \textit{Ident}[\text{(x: Int) => 2}] \\ \text{uninterpreted} & \text{otherwise} \end{cases}$$

When new closures are discovered during unfolding,
add them to Λ and expand results of f(x)

First-Class Functions - Blocking

How do we know when the *right* closure has been inlined for a given application?

Block tree branch as long as $f \notin \Lambda$

Note that the procedure doesn't support inputs that are containers for first-class functions (such as List[Int => Int]) as these can't be added to Λ

Theoretical Results

Proved for boolean and function types

- Soundness for proofs

  If the procedure reports valid, there exists no counter-example to the VC

- Soundness for counter-examples

  If the procedure reports a counter-example, evaluating the VC with it as input will result in **false**

- Completeness for counter-examples

  If there exists an input to the VC such that evaluation results in **false**, the procedure will eventually report a counter-example

Demo

Conclusion

- Higher-order functions can be supported in *Leon* without resorting to sacrifices and/or tradeoffs
- Limitations interesting avenues for extension
  - Unfolding data-structures to accept first-class function containers (and more)
  - Limited universal quantification support for specifications