Verifying Higher-Order Functions

SAV 2017

Verification Problem

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 *Stainless*

- 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 : counterexample

First-Order Verification

First-Order Verification in *Stainless*

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

```scala
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 :

```
∀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

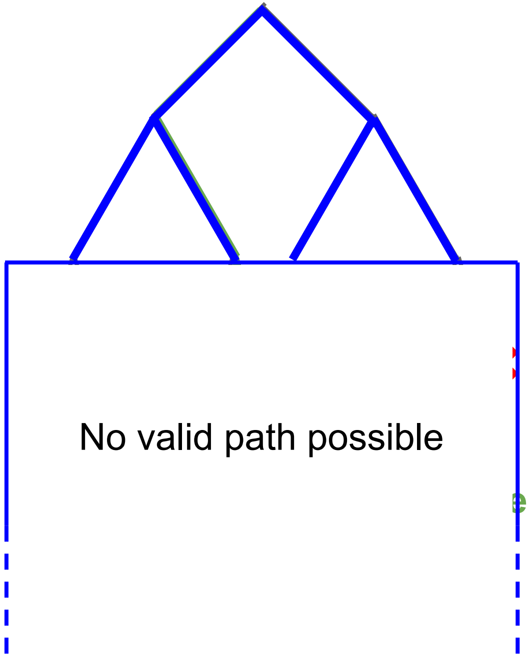
Unfolding Procedure in *Leon*

- 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 $\neg f$ with blocked branches $\in$ SAT

    model is a counterexample

  - if $\neg f$ $\in$ UNSAT

    VC is valid



Unfolding Procedure - Example I

Verification Condition

size(list) >= 2

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(t2) = t2 **match** {

~~**case** Cons(h3, t3) => 1 +~~ size(t3)

case Nil() => 0

}

size(t3) = ...

size(Nil()) = 0

size(Cons(h1, Nil())) = 1

size(Cons(h1, Cons(h2, Nil()))) = 2

Breaks VC!

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(list) can break VC!

Higher-Order Functions

Challenges

- 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

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 }

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

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 we need **all** lambdas to appear in Λ at some point to make progress!

Input Functions

- Finding closures during unfolding is easy
- What about input functions?
  - Input f: Int => Int

    □ add f to Λ
  - Input tuple: (Int => Int, Int => Int)

    □ add {tuple._1, tuple._2} to Λ
  - Input list: List[Int => Int]

    ?? how do we find **all** functions in list?

Input Functions in Recursive ADTs

- Idea: unfold the datatype

```
def allFunctions(list: List[Int => Int]): Unit = list match {
  case Cons(f, fs) => /* register function f */ allFunctions(fs)
  case Nil() => /* do nothing */
}
```

- Specialized unfolding to register functions

Function Equality

Remember, branches are blocked by $f \notin \Lambda$

- Consider $g \in \Lambda$, $f \notin \Lambda$

	$\forall x.\ f(x) = g(x)$	$\exists x.\ f(x) \neq g(x)$
encode(f) = encode(g)	model ✓, proof ✓	model x, proof x
encode(f) ≠ encode(g)	model ✓, proof x	model ✓, proof ✓

Function Equality - Tradeoffs

- We want semantics that can be evaluated
  - $\text{encode}(f) = \text{encode}(g) \Leftrightarrow \forall x.\ f(x) = g(x)$ x
- We want sound counterexamples
  - $\text{encode}(f) = \text{encode}(g)$ x
- We want to preserve proofs when possible
  - $\text{encode}(f) \neq \text{encode}(g)$ x
- Idea: use function structure

Function Equality - Structural

encode(f) = encode(g) iff

structure(f) = structure(g) - static

closures(f) = closures(g)   - dynamic

Note that these are **not** Scala semantics.

Theoretical Results

- Soundness for proofs

  If the procedure reports valid, there exists no counterexample to the VC
- Soundness for counterexamples

  If the procedure reports a counterexample, evaluating the VC with it as input will result in **false**
- Completeness for counterexamples

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