Repair in the Leon tool

Manos Koukoutos

LARA, EPFL, Switzerland

May 1, 2017

Problems in SAV

- Verification problem:
  Given a **correct specification** and an **implementation**, prove if the implementation is correct or not (for every input)
- Synthesis problem:
  Given a **correct specification** and **no implementation**, come up with a correct implementation
- Repair problem:
  Given a **correct specification** and an **erroneous implementation**, come up with a correct implementation.

Specification: either a logical formula, or input-output examples.

Example: Max Heap merging as a verification problem

Input:

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  (h1,h2) match {
    case (Leaf(), _) ⇒ h2
    case (_, Leaf()) ⇒ h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
      if(v1 ≤ v2)
        Node(v2, l2, merge(h1, r2))
      else
        Node(v1, l1, merge(r1, h2))
  }
} ensuring { res ⇒
  isLegalHeap(res) &&
  h1.content ++ h2.content == res.content
}
```

Output: Correct for every input!

Example: Max Heap merging as a synthesis problem

With logical specification:

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  choose( (res: Heap) ⇒
    isLegalHeap(res) &&
    h1.content ++ h2.content == res.content
  )
}
```

Example: Max Heap merging as a synthesis problem

With examples:

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  choose( (res: Heap) ⇒
    ((h1, h2), res) passes {
      case (Leaf(), Leaf()) ⇒ Leaf()
      case (Leaf(), Node(0, Leaf(), Leaf())) ⇒
        Node(0, Leaf(), Leaf())
      case (
        Node(1, Leaf(), Leaf()),
        Node(0, Leaf(), Leaf())
      ) ⇒
        Node(
          1,
          Leaf(),
          Node(0, Leaf(), Leaf()))))}}
```

Output: Implementation of previous slide.

Example: Max Heap merging as a repair problem

Input:

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  (h1,h2) match {
    case (Leaf(), _) ⇒ h2
    case (_, Leaf()) ⇒ h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
      if(v1 ≥ v2)
        Node(v2, l2, merge(h1, r2))
      else
        Node(v1, l1, merge(r1, h2))
  }
} ensuring { res ⇒
  isLegalHeap(res) &&
  h1.content ++ h2.content == res.content
}
```

Example: Max Heap merging as a repair problem

Input:

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  (h1,h2) match {
    case (Leaf(), _) ⇒ h2
    case (_, Leaf()) ⇒ h1
    case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
      if(v1 ≥ v2)
        Node(v2, l2, merge(h1, r2))
      else
        Node(v1, l1, merge(r1, h2))
  }
} ensuring { res ⇒
  isLegalHeap(res) &&
  h1.content ++ h2.content == res.content
}
```

Output: the above code where $\geq$ has been replaced with $\leq$.

Overview of approach for Repair

A human programmer could

- Write some tests, run them and classify them as "passing" and "failing".

Overview of approach for Repair

A human programmer could

- Write some tests, run them and classify them as "passing" and "failing".
- Assume the code is generally correct, and some specific part(s) are responsible for the failing tests.
  In a debugger, follow the trace of a failing test until localizing the problem to a specific code snippet

Overview of approach for Repair

A human programmer could

- Write some tests, run them and classify them as "passing" and "failing".
- Assume the code is generally correct, and some specific part(s) are responsible for the failing tests.
  In a debugger, follow the trace of a failing test until localizing the problem to a specific code snippet
- Try to find a small change that would fix the snippet. If that fails, throw it away and write it from scratch.

Overview of approach for Repair

A human programmer could

- Write some tests, run them and classify them as “passing” and “failing”.
- Assume the code is generally correct, and some specific part(s) are responsible for the failing tests.
  In a debugger, follow the trace of a failing test until localizing the problem to a specific code snippet
- Try to find a small change that would fix the snippet. If that fails, throw it away and write it from scratch.
- Rerun the test suite (or verifier!). If there are still issues, repeat.

Stages of the repair algorithm

- Test generation and (trace) minimization
- Fault Localization
- Synthesis of similar expressions
- Verification of the solution

Test generation and verification

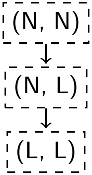
Our algorithm needs at least one *failing test*, which leads to erroneous program execution
We obtain tests from various sources:

- Input-output examples given by the user
- Enumeration of programs
- Counterexamples from SMT solver

Trace minimization

In the presence of recursive functions, a given test may fail within one of its recursive invocations.

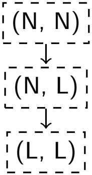
```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  (h1,h2) match {
    case (Leaf(), _) ⇒ h1 // Buggy
  ...
}
```



Trace minimization

In the presence of recursive functions, a given test may fail within one of its recursive invocations.

```
def merge(h1: Heap, h2: Heap) : Heap = {
  require(isLegalHeap(h1) && isLegalHeap(h2))
  (h1,h2) match {
    case (Leaf(), _) ⇒ h1 // Buggy
  ...
}
```



A failing test should also be blamed for the failure of all other tests that invoke it transitively.
In this case, only (Leaf(), Leaf()) is maintained as a failing example.

Error Localization

Follow the trace of failing tests to find in which branch of the program they lead us.

Suppose we have identified as failing tests:

Node(1, Leaf(), Leaf()), Node(0, Leaf(), Leaf())

Node(2, Leaf(), Leaf()), Node(0, Leaf(), Leaf())

```
(h1,h2) match {
  case (Leaf(), _) ⇒ h2
  case (_, Leaf()) ⇒ h1
  case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
    if(v1 ≥ v2)
      Node(v2, l2, merge(h1, r2))
    else
      Node(v1, l1, merge(r1, h2))
}
```

Error Localization

A realistic set of failing tests is

Node(1, Leaf(), Leaf()), Node(0, Leaf(), Leaf())

Node(0, Leaf(), Leaf()), Node(1, Leaf(), Leaf())

```
(h1,h2) match {
  case (Leaf(), _) ⇒ h2
  case (_, Leaf()) ⇒ h1
  case (Node(v1, l1, r1), Node(v2, l2, r2)) ⇒
    if(v1 ≥ v2)
      Node(v2, l2, merge(h1, r2))
    else
      Node(v1, l1, merge(r1, h2))
}
```

Focusing on if-condition

Do we need to focus on the condition? In testing terms,
is there an alternative condition which makes all tests succeed?

Focusing on if-condition

Do we need to focus on the condition? In testing terms,
is there an alternative condition which makes all tests succeed?

To find out, replace the condition with havoc and run the tests, i.e. nondeterministically consider both branches of the **if** for each test.

Focusing on if-condition

Do we need to focus on the condition? In testing terms,
is there an alternative condition which makes all tests succeed?

To find out, replace the condition with havoc and run the tests, i.e. nondeterministically consider both branches of the **if** for each test.

If testing succeeds now,
i.e. there exists a valid execution exists for each failing test,
it means that the answer to the question is true.

We have localized the error on the if-condition. Now, we have to synthesize an alternative solution.

We describe interesting programs with a term grammar. For repair, the grammar should describe small variations to the original program and simple arbitrary programs.
E.g.

$$\mathit{Boolean} ::= \mathit{Int} \geq \mathbf{v2} \mid \mathbf{v1} \geq \mathit{Int} \mid \mathbf{v2} \geq \mathbf{v1} \mid \mathbf{true} \mid \mathbf{false} \mid \ldots$$

Synthesis - The CEGIS algorithm

Once we have a grammar representing interesting programs, we can synthesize a solution with the CEGIS algorithm.
Basic idea of CEGIS:

1. Use concrete tests to filter out candidate programs.
2. From those remaining, pick one and send it to the verifier.
3. If verification successful, we are done
4. Otherwise, the verifier generates a counterexample.
   Add it to the set of tests and jump back to (1).
   (note: step (1) will now filter out more programs)

CEGIS will generate $v2 \geq v1$ and verify it as the correct solution

Demos!