Integers post-processing

and precise post-condition diagnosis

Florian Briant

École Polytechnique Fédérale de Lausanne

27 May 2015

Motivations

Integers post-processing: Provide simpler alternative models to Leon's weird integers counter-examples.
Example :
Cons(1234, Cons(1233, Nil))
might be simplified as
Cons(1, Cons(0, Nil)).

Precise post-condition diagnosis: Provide more precise information about post-condition violation ; what part(s) of the condition is violated ?

Simple case

Consider this following funtion :

```
def AddOne(x:Int, y:Int) : Int = {
  x + y
} ensuring(res => res == 5)
```

Leon (as of last version of GitHub) returns the following counter-example :

- x = 1073741826
- y = 1073741827

Idea : Here, the model x' = 0, y' = 1 might be applicable. Might be a good idea to substract the **minimal** value.

Negative values and Requirements

What if x and y had negative values ?
Easy, substract the **maximal** value.

What if the function had **require(x >z)** ($z >= 0$)?
Easy, the minimal value becomes **minimal** - z - 1.

Mixed Sign

What if x and y had **opposed sign** ?
More tricky. Substracting is not the solution here. Might here explore cases such as x = -y.

There is also the fact that when mixed sign counter-examples are found, they might be because of some integer **overflow** error.
Example : This following function gives weird counter-examples which are borderline to overflow. If you compute the result, you indeed violate the post-condition because of overflow.

```
def dontGet3(x:Int, y:Int) : Int = {
  require(x != 0 && y != 0)
  x + y
} ensuring(res => res != 3*x)
```

- x = 1073741824
- y = -2147483648

Integers and Case classes

Now what if integers are mixed with/inside **case classes** ?
Idea - Consider separate lists of integers :

- One list and one minimal value with all global integers (outside of case classes). Same as x and y in previous case.
- One list and one minimal value for each **CaseClass** object and the integers inside them.

Integers and Case classes

Consider the following example :

- t = Cons(1234, Cons(1233, Nil))
- x = 5555
- y = 5554

Here, there will be one list for x and y with minimal value 5554 and one list for t with minimal value 1233. In the end, we get the following alternative model :

- t' = Cons(1, Cons(0, Nil))
- x' = 1
- y' = 0

Validation of alternative model

The alternative model might not be applicable. For example, for the simple case, what if counter-examples were of form x = 2*y ?
We can of course try to explore this case, but we have to know minimal value substraction is not applicable in this case.

We need to have a mean to validate our now model before giving it as output.
Idea : Re-launch the solver with giving our new model values to the function's verification condition.

Validation of alternative model

With this idea, we can :

- Give new model as output if the re-validation still fails
- Explore other solution than minimal value substraction if the re-validation succeeded.
- If no other solution works (i.e. all re-validation succeed), we give the original model as output.

But if the new model re-validation fails, does it fail for **the same reasons** as the original model ?
To be explained later.

Post-condition diagnosis

Goal : Provide information about what parts of the post-condition is violated.

Initial step : Separate the post-condition with conjunctions.

Validation of each conjunction

Idea : Re-Launch the solver **for each** conjunction of the post-condition. Just as Integers post-processing, need to assign counter-example's values to the requirement part of the function verification condition. But here, we need to modify the post-condition such that it contains only one of the conjunction.
Each validation that fails indicates that its corresponding conjunction is violated by the counter-example.

Problem : Speed perfomance is hurt. Need to launch a new solver **for each** conjunction.

Validation of one conjunction

Alternative Idea : Do the same, but stop the first time a validation fails.
We lose precision by providing only one part of the post-condition's conjunctions that are violated but we may gain speed.
=>Tradeoff.

Intergers Post-processing validation

Back to Integers post-processing =>how can we be sure that the new model fails for the same reasons as the original one ?

Solution : The new model will fail for the same reasons iff the conjunctions it violates are the same than with the orignal model
Drawbacks : However, we need to compute twice all the conjunctions that are violated =>Speed perfomance decreases.

Implementation - What of Leon is modified ?

All of the computation part of the project needs to happen when Leon identifies invalid results.
This is found in **AnalysisPhase.scala**, right before *checkVC()* reports a *VCResult* which is invalid.

The project also modifies the structure of invalid *VCResult* such that is holds possible new models and possible list of conjunctions violated and report them correctly.

Further work

- Explore other Integers post-processing solution than minimal value substraction.
- Explore other Leon's purescala structure than case classes.
- Explore BigInt counter-examples.
- Explore the solver to see whether we can pick up the violated conjunctions.

Demo

Thank you for your attention!