Project Report

Synthesis, Analysis and Verification

Leon's integer post-processing and post-condition diagnosis

Florian Briant

SAV Project Report – Florian Briant

Project's goals and motivations :

Integer post-processing :
Leon often gives weird counter-examples to the verification phase results whenever it deals with integers (*Int* or *BigInt*). As an example, it may find counter-example such as :

Cons(1234, Cons(1233, Nil))

In this case, we might prefer an alternative simpler model such as this one :

Cons(1, Cons(0, Nil))

The first part of this project is to find interesting post-processing optimizations that could help Leon find simpler counter-examples and thus helping the programmer to better understand why Leon proved wrong.

Post-condition diagnosis :
The other part of the project, still with the aim to help the programmer to better understand why Leon proved wrong, is to provide more information about what part of a post-condition was violated.

Integer post-processing – Ideas :

Minimal value subtraction :
Here is a very simple and dumb function that yields weird counter-example results :

```
def Add(x: Int, y: Int) : Int = {
      x + y
} ensuring(res => res == 6)
```

The results are the following :

- **x** = 1073741826
- **y** = 1073741827

Here we see that this following simpler model is applicable :

- **x'** = 0
- **y'** = 1

With this we get a first hint of optimization : minimal value subtraction.
The idea is to subtract the minimal value to all integers (we consider however two set of integers, one for *Int*, the other one for *BigInt*).

With this idea, we need to get a different behavior when all integers are negatives (simply need to subtract the maximal value) or are of mixed signs (we separate the integers in two lists, one for positives, and one for negatives). We need as well to take care of requirements (if there is requirement x >= 5, then we should not give x = 0 but x = 5)

However, after trying to optimize the older counter-example model, we need to get sure that the newer model still holds (is still a counter-example). For this, we only need to re-evaluate the function's verification condition while adding requirements where we give the variables

our newer counter-example results. If the re-evaluation succeeds (is still false) then we may give this model as an alternative simpler model.

Binary search :
The intuitive idea here is that the “minimal value” in the eye of the programmer is 0 ; the more the result is close to 0, the easier it will be to understand why it is a counter-example. Then we can try to (binary) search between the invalid result and 0 for each integer and re-valuate the function's verification condition until we get a minimal value for each integer.

The drawback of this method is that if integer results are dependent to each-other, this method may not give an improved result.

Minimal value VS Binary search :
Both of them are good for certain cases, and bad for others. For example, minimal value subtraction should be good with any dependency such as x = y + n, whereas binary search should be better with non-dependent integers. The intuitive rule of thumb to decide which alternative is better is to compute the sum of all optimized integer results and see which is lower.

Other complex structures :
It is often the case than more complex type of arguments (such as case classes) have integers inside, and they do not escape the weird counter-example results. The idea, which is the same as the separation of *BigInt* and *Int* results, is to consider for each complex structures a list of integers inside them.
For example, consider these following results :

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

The optimization will yield this new model :

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

Because it has considered a list for **x** and **y**, and a list for **t**.

Validation of new model :
As stated before, the idea to validate a new model is to re-evaluate the function's verification-condition and see whether Leon still proves wrong. However, how can we be sure that the newer model fails for “the same reasons” than the older model ? The second part of the project will help to answer this question.

SAV Project Report – Florian Briant

Post-condition diagnosis – Ideas :

Post-condition's conjunctions separation :
The intuitive idea for this part is quite simple, we need to first separate the post-condition into “parts”. The simpler idea is to separate the post-condition's conjunctions ; we know the post-condition is false, it means at least one of the conjunctions is false and all of the disjunctions are false. There is in this case no need for information about disjunctions.

Validation of each conjunction :
Then, the idea is to modify the function's verification condition for each conjunction such that the post-condition only holds the conjunction and such that there is requirements that arguments are set to the counter-example results. We then re-evaluate the function's verification condition (for each conjunction). If a re-evaluation is still false, it means the corresponding conjunction is violated in the original verification condition with the counter-example results. This allows us to precisely know which conjunctions are violated.

The slight drawback of this method is that, whenever there is two conjunctions or more, Leon automatically creates a *val res = […]* in the post-condition. However, if the post-condition is only one conjunction, it does not, and simply replaces *res* with the whole function body, which is not very convenient.

Back to Integer post-processing :

New model validation :
The results of the post-condition diagnosis helps to solve the previous question ; intuitively, the newer model fails “for the same reasons” than the older model *iff* both model violate the exact same conjunctions of the post-condition.

Implementation – What was modified ? :

Integer post-processing :
All that is related to integer post-processing is available in *IntOpti.scala* (new file) in the verification package.

Post-condition diagnosis :
All related works to post-condition diagnosis is available in *PostCondDiag.scala* (new file) in the verification package.

Verification phase modification :
All of this project “had to happen” when Leon finds an invalid result. This is found in *VerificationPhase.scala* in the verification package. When Leon knows that there was an invalid result, then the whole optimization procedure of this project is called.

As well, the structure *VerificationResult.scala* in the verification package had to be modified a bit such that it holds and prints a possible alternative model (the “best” one of the optimizations) and possible post-condition diagnosis (a list of conjunction). There is also a small line in *Repairman.scala* in the repair package (for compatibility reason).

Further works :

Integer post-processing :

Binary search : Still needs to be implemented.

Other complex structures : Yet only work for CaseClasses.

Post-condition processing :

Find solution for “one-conjunction” post-condition.

Interesting results :

TestFlo.scala :

TestFlo.scala which is found in *SAVProject/testcases* is a test case that tries to high light features of this project. Here is some of these cases :

```
def AddWithRequire1(x:Int, y:Int) : Int = {
      require(x > 2 && y > 2)
      x + y
} ensuring(res => res == 5)
```

```
[  Info  ]  - Now considering 'postcondition' VC for AddWithRequire1 @16:14...
[ Error  ]  => INVALID
[ Error  ] Found counter-example:
[ Error  ]   x -> 1073741826
[ Error  ]   y -> 1073741827
[ Error  ] Following counter-example is also applicable :
[ Error  ]   x -> 3
[ Error  ]   y -> 4
[ Error  ] Following post conditions are violated :
[ Error  ]   x + y == 5
```

Fig 1. Example of **minimal value subtraction**

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

```
[  Info  ]  - Now considering 'postcondition' VC for AddOneNeg @39:14...
[ Error  ]  => INVALID
[ Error  ] Found counter-example:
[ Error  ]   x -> -7
[ Error  ]   y -> 6
[ Error  ] Following counter-example is also applicable :
[ Error  ]   x -> 0
[ Error  ]   y -> 0
[ Error  ] Following post conditions are violated :
[ Error  ]   -x + y == 5
```

Fig 1. Example of **minimal value subtraction** with mixed sign

```
def AddOneWithZAndConjs(x:Int, y:Int, z:Int) : Int = {
      x + y
} ensuring(res => (res == 5 && res == 6) && (res != 4 && res !=3))
```

```
[  Info  ]  - Now considering 'postcondition' VC for AddOneWithZAndConjs @35:14...
[ Error  ]  => INVALID
[ Error  ] Found counter-example:
[ Error  ]   x -> 2048
[ Error  ]   y -> 0
[ Error  ] Following post conditions are violated :
[ Error  ]   res == 5
[ Error  ]   res == 6
```

Fig 3. Example of **post-condition diagnosis**