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

Fig 1. AddOne – a dumb and simple function

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

[TODO (will be done in the next 10 minutes)]