Project Report:
Improving Leon's Termination Checker

Samuel Grütter

5 June 2015

Introduction

When writing recursive functions, it's a common mistake to forget about base cases, and to accidentally write non-terminating functions. And if one adds a postcondition, Leon will accept it, because the verification assumes that all functions terminate. For example, the following function passes verification:

```
def fib(n: BigInt): BigInt = {
  fib(n-1) + fib(n-2)
} ensuring {res => res == (5*n + 1)*(5*n - 1)}
```

There's a termination checker in Leon [1], but unfortunately, it doesn't give any answer for the above example. It also has some other shortcomings, and was never tested on a bigger set of tests, so that's why it's disabled by default. The goal of this project is to improve the termination checker and its test coverage, so that it can be enabled by default.

The rest of this report describes the improvements made in this project, and it contains one section per feature.

1 Detecting non-termination using a call graph under-approximation

Leon uses call graphs in many places, but it always uses over-approximations of the "true" call graph, i.e., whenever the body of a function f_1 contains a call to a function f_2, a "call edge" from f_1 to f_2 is added to the graph. In order to prove non-termination, such a graph is not (directly) useful. But if we have an under-approximation of the call-graph, i.e., an edge from function f_1 to function f_2 is added only if we are sure that every execution of f_1 will call f_2, we get a simple way of detecting (some) non-terminating functions by simply checking if this graph contains a cycle. If we find such a cycle, we know that all functions on the cycle are non-terminating for all inputs.

This is implemented in the newly added `SelfCallsProcessor`, which performs a depth first search in the call graph under-approximation. It's not necessary to construct the graph explictly,

because in order to know the outgoing edges of a function, we can simply look at its implementation on the fly while doing the depth first search.

2 Abstracting over the well-founded order used by the Relation Processor

The Relation Processor (described in section 3.3 of [1]) proves termination of functions by checking that all recursive sub-calls are *decreasing* [1], i.e. if a function's argument tuple is $\overline{a}_1$, it checks for all recursive sub-calls with argument tuple $\overline{a}_2$ that $\overline{a}_2 \prec \overline{a}_1$.

The $\prec$ relation is defined as $\overline{a}_2 \prec \overline{a}_1 \iff size(\overline{a}_2) < size(\overline{a}_1)$, and $size$ is a function mapping Leon expressions to $\mathbb{N}$ (defined in section 2.1 of this report).

However, the strategy used by the Relation Processor also works for different definitions of the $\prec$ relation. All we need to ensure is that $\prec$ is a well-founded order, i.e. that there are no infinite chains $a_1 \succ a_2 \succ a_3 \succ \dots$.

With very little refactoring, it was possible to make the Relation Processor abstract over the definition of $\prec$, so that it can be used with different implementations of $\prec$. Each of the following subsections describes one such implementation.

2.1 Structural size of the argument tuples

The termination checker developed in [1] uses the notion of "structural size" of an expression. It's defined as follows:

$$size(x) = \begin{cases} \sum_{i=1}^{N} size(x_i) & \text{if } x : Tuple_N \\ 1 + \sum_{i=1}^{N} size(x.field_i) & \text{if } x : ADT_N \\ 0 & \text{otherwise} \end{cases}$$

It can be used to define a $\prec$ relation as follows:

$$\overline{a}_2 \prec \overline{a}_1 \iff size(\overline{a}_2) < size(\overline{a}_1)$$

Originally, the termination checker only used this definition of $\prec$.

The testing (section 4) showed that in many cases with BigInt arguments, the other processors could not prove termination, and that it would be useful to give the Relation Processor a means of reasoning about integers. This can be achieved by modifying the size function as follows:

$$size(x) = \begin{cases} \sum_{i=1}^{N} size(x_i) & \text{if } x : Tuple_N \\ 1 + \sum_{i=1}^{N} size(x.field_i) & \text{if } x : ADT_N \\ abs(x) & \text{if } x : BigInt \\ 0 & \text{otherwise} \end{cases}$$

[1] To be more precise, this requirement can even be slightly relaxed by requiring only *transitive decreasing*, i.e. that the size is preserved in the sub-call, but strictly decreases in the sub-sub-calls made by the sub-call.

It would have been even better to include also a case for bitvectors, but the size function has to return a BigInt, and the solvers don't support conversion from bitvectors to BigInt. Of course, one could implement the conversion by hand, but the real challenge would be to give the solver the hint that the conversion preserves all inequalities. Therefore, the approach described in section 2.3 was preferred.

2.2 Comparing function call arguments lexicographically by structural size

Another possible definition of $\prec$ is to compare function argument lists lexicographically, using the *size* function defined in 2.1 to compare individual expressions in the argument lists. This was implemented in `LexicographicRelationComparator`.

Note that in general, lexicographic orders over sequences of different length are not well-founded: For instance, in the alphabetic ordering of lower-case strings, we have an infinite decreasing chain `b` $\succ$ `ab` $\succ$ `aab` $\succ$ `aaab` $\succ \ldots$.

However, in Leon, function argument lists are of finite length, and for each program, we can statically find the maximum argument list length $l_{\max}$, and be sure that during the termination checking, no argument lists longer than $l_{\max}$ will occur, and since the order defined by the *size* function which is used to compare the entries of the argument lists is well-founded as well, the lexicographic order we're using here is well-founded.

2.3 Comparing bitvector arguments lexicographically by absolute value

A third definition of $\prec$ focusses on bitvectors: Given two tuples of expressions, it first filters them by the type of the expression, and only keeps the expressions of type `Int32Type`. Then, these filtered tuples are compared lexicographically, using the absolute value of the individual entries.

Another possibility would be to compare the sum of all absolute values of bitvector arguments, but this turned out to be useless, because in most cases, we cannot guard against overflows when summing up the absolute values, so the solver cannot give us any proofs.

3 Verification and termination results in one run

By giving the options `--verify` and `--termination` together, we can now get the results of verification and of the termination checker in just one run of Leon. The beginning of the pipeline, which is the same for verification and termination (i.e. the parsing with `scalac`, the `ExtractionPhase` and the `PreprocessingPhase`) are run only once, and the obtained program is then fed to both the `AnalysisPhase` and the `TerminationPhase`.

This feature will hopefully make the use of the termination checker in Leon more popular, because it removes the need for an extra call to get termination results.

4 More testing

In order to empirically test the robustness of the implementation, the `TerminationRegression` test was extended so that it tests the termination checker not only on the regression suite written for the termination checker, but also on the regression suite written for testing the verification phase.

Without using any of the improvements made in this project, 6 out of the 68 files of the verification regression suite contained functions for which termination could not be proven:

- `regression/verification/purescala/valid/Monads3.scala`
- `regression/verification/purescala/valid/FlatMap.scala`
- `regression/verification/purescala/valid/ParBalance.scala`
- `regression/verification/purescala/valid/MergeSort.scala`
- `regression/verification/purescala/valid/Nat.scala`
- `regression/verification/purescala/valid/BitsTricks.scala`

And among the tests added during this project, 3 files contained functions for which termination could not be proven:

- `regression/termination/valid/CountTowardsZero.scala`
- `regression/termination/looping/OddEven.scala`
- `regression/termination/looping/WrongFibonacci.scala`

Moreover, the leon library contained some methods for which termination could not be proven:

- `library/collection/List.scala`

The goal of this project was to make them all succeed. This was achieved on one hand by implementing the features described in the previous sections, and on the other hand by slightly rewriting some problematic functions in the test cases, so that they calculate the same result, but that it's more obvious why they terminate.

The following subsections describe for each of the above listed files why termination can now be proven for them.

Since some files had to be modified, it's best to look at them in the modified version at `https://github.com/samuelgruetter/leon/tree/a5f519a395/src/test/resources/`.

And in order to see the edits made to the files, it's best to look at the commits of the pull request at `https://github.com/epfl-lara/leon/pull/115`.

4.1 `Monads3.scala` and `FlatMap.scala`

Both of these files contain a lemma called `associative_lemma_induct` which does lexicographic induction on its first three arguments (which all are lists). Thanks to the lexicographic argument lists comparison described in section 2.2, they now succeed, without any modification of the test files.

4.2 `ParBalance.scala` and `MergeSort.scala`

In `ParBalance.scala`, the problematic function is `reverse_reverse_equivalence`: It proves that reversing a list twice yields the original list. The lemma is recursively called on a reversed list, and the termination checker does not recognize that the *size* of the argument (according to the structural *size* function of section 2.1) is preserved by the `reverse` operation. So we have to edit the `reverse_reverse_equivalence` function to include the `size` (number of elements, according to the function definition in the Scala file) as an additional argument, which just serves as termination witness and would be removed by any decent optimizer.

Now the lexicographic measure applies, because it suffices to show that the first argument always decreases, and it doesn't matter if we don't know anything about the structural size of the second argument.

In `MergeSort.scala`, the same problem happened with the `weirdSort` function, and it was solved the same way.

4.3 `Nat.scala`

In `Nat.scala`, the `int2Nat` function converted integers to an ADT type `Nat`, but there was no precondition to exclude negative arguments, so it would have looped for negative arguments. After adding the precondition, termination could be proven, thanks to the *abs* change made to the *size* function (section 2.1).

4.4 `BitsTricks.scala`

In `BitTricks.scala`, the `turnOffRightmostOneRec` and `isRotationLeft` functions, which used recursion to loop through all 32 bits of a bitvector, could not be proven terminating, because they counted from 0 up to 31. By modifying the functions so that they count from 31 down to 0, the bitvector comparison described in section 2.3 could now kick in, and termination was proven.

4.5 `CountTowardsZero.scala`

This testcase contains a function accepting both positive and negative BigInts, and counting down towards zero, so the absolute value always decreases. Thanks to the modifications to the size function described in section 2.1, it now succeeds.

4.6 `OddEven.scala` and `WrongFibonacci.scala`

Both of these files miss a base-case in the recursive definition, so they count down to minus infinity. The existing termination checker (the `LoopProcessor`, more precisely) could not prove non-termination for these, because it only works for functions which call themselves with the same arguments, but here, the argument always changes.

In this case, the `SelfCallsProcessor` kicks in and proves that these functions are non-terminating for all inputs.

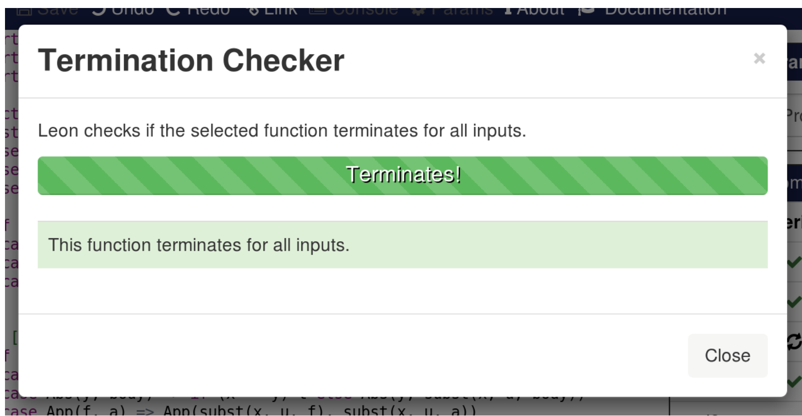
5 Web interface improvements

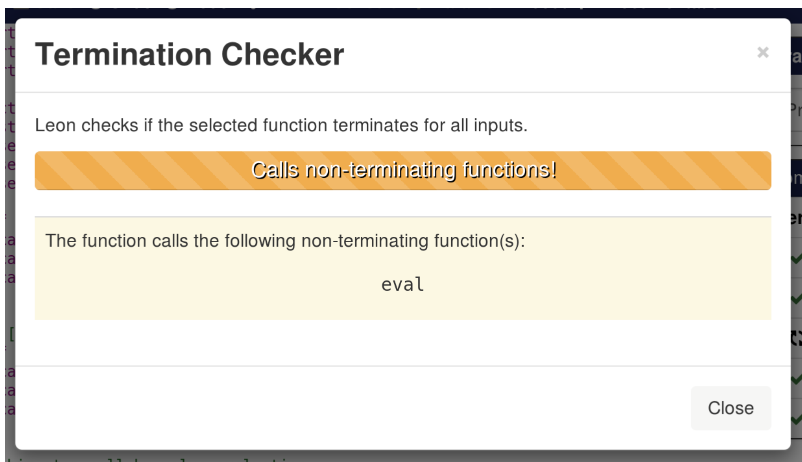
The web interface already displayed output of the termination checker, but not in a very user-friendly way. It did not display counter-examples to termination, and it did not distinguish between the "counter example found" case and the "no guarantee" case (both were just displayed as "no guarantee").

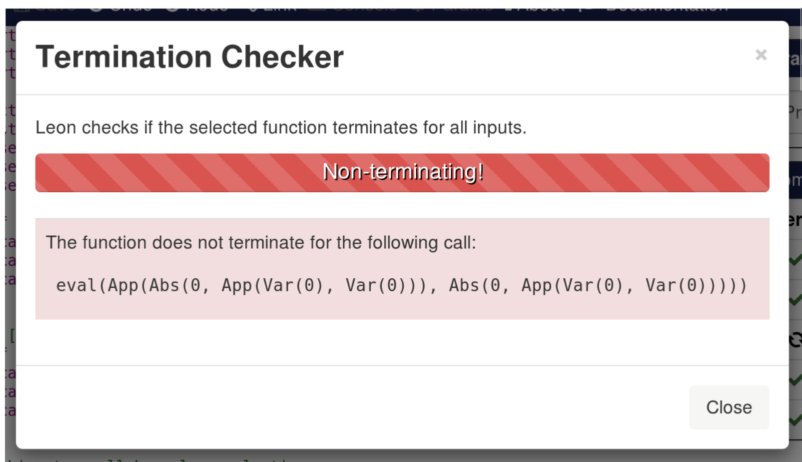
In the overview box, we now distinguish between four cases: "terminates", "conditionally terminates" (i.e. calls non-terminating functions), "loops", and "no guarantee".

Analysis	Compiled ✔	
Function	**Verif.**	**Term.**
fv	✔	✔
subst	✔	✔
eval	↻	!
isValue	✔	✔
eval2	✔	(✔)

And clicking on the icon in the overview box opens a dialog with details, displaying the counterexample or the called non-terminating functions (if any).







References

(1) Nicolas Voirol: Modular and Extended Termination Prover (Semester project report). EPFL, School of Computer and Communication Sciences, Spring 2013.