Minimizing Stainless counter-examples

CS550 Formal Verification, Project Report

Matthieu Bovel (matthieu.bovel@epfl.ch)
Solal Pirelli (solal.pirelli@epfl.ch)

January 12, 2022

Abstract

Stainless is a tool for verifying Scala programs. It can prove that some invariants inside a function hold for all possible arguments, or show counter-examples when they exist.

These synthesized counter-examples are unfortunately often sub-optimal in terms of human-readability. For example, when asked to provide two Java integers whose sum is smaller than 10, Stainless outputs "2'146'959'355 and -2'146'959'347". While correct, the answer "0 and 0" would probably look more intuitive to most human brains.

We address this problem by extending Inox—the engine behind Stainless—to make it able to find models that are not only valid, but also minimal with respect to some notion of size. To this end, we leverage the optimizing capabilities of the Z3 backend.

1 Introduction

Many complementary techniques exist to make that programs work as expected. Among which are *type systems* that give guarantees about the possible set of states in which a program can be. *Testing* is also used to check that programs are correct for *some inputs*. However, to make sure that a piece of code behaves as expected *for all possible inputs*, one needs to go further.

One way is to write proofs manually or with the help of a proof assistant, but this is time-consuming and requires a significant mathematical background. Another solution is to automatically convert a program to a set of constraints and use an *SMT solver* to prove whereas these constraints are satisfiable or not. Stainless is such a tool.

It allows developers to automatically verify functions written in a subset of Scala by verifying the validity of *contracts* embedded in the code, for example in the form of pre- or post-conditions. In the following example, the function

a `intMinus` has a post-condition `.ensuring(_ == y - x)` aiming to check if Java integers subtraction is commutative:

```scala
def intMinus(x: Int, y: Int) = {x - y}.ensuring(_ == y - x)
```

To verify this program, Stainless applies a succession of semantic-preserving transformations until it fits into the pure higher-order functional language fragment supported by Inox. Inox in turns transforms this fragment into constraints to are fed to an SMT solver. Because the goal is to find a counter-example to a property p, the SMT solver is queried for a model satisfying $\neg p$. The previous example is encoded as:

```
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(assert (and
    (or
        (= (bvand x #b100...0) (bvand y #b100...0))
        (= (bvand x #b100...0) (bvand (bvsub x y) #b100...0))
    )
    (not (= (bvsub x y) (bvsub y x)))))
(check-sat)
```

This defines two vectors of 32 bits `x` and `y` and a constraint $\neg((x-y) = (y-x))$ (right hand side of the logical **and**). The left hand side of the **and** constraints the model further so that $x-y$ does not overflow, because Stainless checks for overflows separately. It is of course possible to find such a model, therefore Stainless outputs a counter-example:

```
Found counter-example:
    y: Int -> -107401216
    x: Int -> -270336
```

While this example indeed violates our post-condition as $-107401216 - -270336 \neq -270336 - -107401216$, it is needlessly verbose and unintuitive to human eyes which would probably prefer something like $x = 0$ and $y = 1$. In other words, we are interested not only in a valid counter-example, but we also aim for a *minimal* counter-example.

This is an *optimization* problem which is stronger than the *satisfiability* problems traditionally handled by SMT solvers. Nevertheless, it has been shown in [1] that such solvers can be extended to efficiently deal with SMT problems where models M are sought such that a given cost function $f(M)$ is minimized. We give a detailed summary of this paper in the annex background report. We also provide a minimal example of the Optimizing DPLL modulo theories written in Scala.

Z3—one of the SMT solvers that Inox can use—implements these optimization capabilities for linear arithmetic objectives. For example, the following query asks Z3 to optimize $x + y$ given the constraints $x \leq 2$ and $x - y \geq 1$.

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (<= x 2))
(assert (>= (- x y) 1))
(maximize (+ x y))
```

For this query, Z3 outputs the model $x = 2$ and $y = 1$, which is indeed optimal with respect to the given constraints.

Our project consisted in making this optimization capabilities accessible from Inox, and then using them to create a solver that automatically minimize the *size* of all free variables. In the following section, we detail how we implemented this in Inox, how we defined sizing constraints for different variable types. We then compare the new minimizing solver performance to the existing solvers in the Benchmarks section, before summarizing and discussing possible enhancements and steps in the Conclusion.

2 Implementation

Inox already has the notion of a *solver* interface that takes in constraints and outputs a satisfying assignment or UNSAT. The existing solvers include Z3, CVC4, and Princess; furthermore, Z3 can be used either in "native" mode, talking to Z3 via its own API, or in "SMT-LIB" mode, talking to Z3 via standard SMT-LIB text queries. The advantage of the SMT-LIB mode is that the same queries can be sent to any solver that supports SMT-LIB, modulo solver specificities that should not exist in theory but happen in practice.

Inox already had a `Z3Optimizer` solver which supported a form of weighted Max-SMT: instead of merely asserting a constraint, one could give the constraint a weight and the optimizer would try to maximize the sum of weights of satisfied constraints. However, this does not work for our purposes, as we wish to maximize *variables*, not *constraint weights*.

We added support for maximization and minimization to Inox's internal *optimizer* interface in pull request #171, which was merged.

Using this support, we then added a `Z3Minimizer` solver in pull request #176, which is currently open, that implements the minimization described in this report. This solver can be used as `smt-z3-min`, for instance using Stainless's `--solvers=` argument. It will automatically ask Z3 to minimize the variables in all queries, using the techniques we describe below.

2.1 Numbers

Before discussing the minimization of numbers, we note that we use *integers* to refer to the arithmetic integers, which are unbounded and represented in Scala as `BigInts`, not to Scala `Ints` and friends, which are *bit vectors* of fixed size. This is important in the context of SMT, as the two cannot mix without explicit conversions, and bit vectors are (on their own) decidable while integers are not always decidable (e.g., non-linear integer arithmetic is undecidable).

Minimizing a number has an intuitive connotation of "making it smaller", but we must be more formal. Asking Z3 to make an integer as small as possible is pointless, since integers are not bounded, and the result will thus be some very large negative integer. Instead, we minimize *the absolute value*, i.e., to minimize x we minimize `if (x >= 0) x else -x`.

The naïve approach of generating one big addition for all variables does not work because variables are of different sorts. One cannot add bit vectors of different sizes together without an explicit conversion, which would require a first pass over all variables to first check what the largest bit vector is. Even if we extended the smaller bit vectors to make them all the same size, we would need to convert all bit vectors to integers if any variable is an integer, an operation that Z3 supports but Inox currently does not expose (issue #108). Even if we added that support, the resulting query would be suboptimal since we would generate a large query with lots of explicit conversions which would hinder performance (see Z3 issue #1481).

Instead, we generate one minimizing query per number. We could in theory also generate minimization queries for other "primitive" types such as Booleans, but we did not see a need to do so.

In practice, for the given Scala code:

```scala
def add(x: Int, y: Int): Int = { x + y } ensuring(res => res >= 10)
```

An SMT-LIB query similar to this will be generated:

```
...
(declare-fun x!110 () (_ BitVec 32))
(minimize (ite (bvsge x!110 0) x!110 (bvneg x!110)))
(declare-fun y!21 () (_ BitVec 32))
(minimize (ite (bvsge y!21 0) y!21 (bvneg y!21)))
...
```

The result will then be `0` and `0`, which are not just satisfactory assignments but also minimal.

2.2 ADT

ADTs present two challenges for minimization: there is no formal notion of "minimum" for any ADT, and ADTs can be recursive.

One can design strategies for simple cases, such as preferring constructors with fewer parameters or constructors whose parameters are themselves smaller, but the intersection of these goals has no obvious solution. For instance, given an ADT with one constructor `A` taking one 64-bit vector and one constructor `B` taking two 16-bit vectors, which constructor should be preferred for a counter-example? We decided to prefer fewer parameters always.

Handling recursive ADTs must be done explicitly, to ensure the code generating the query terminates instead of generating the equivalent of "make the first item in the list minimal, and the second item in the list minimal, and . . . " *ad infinitum.* However, this naturally conflicts with the goal of minimizing all items. We chose to not generate any minimization queries once we reach the first recursive definition in a path, such as an ADT containing a field of its own type, or an ADT containing a field of another ADT type itself containing a field of the first ADT type.

[Example]

2.3 Recursion

[Recursive functions and unrolling]

[Loops]

3 Benchmarks

[Quick benchmarks and a nice boxplot with our examples comparing run times between (–solvers=smt-z3-min and –solvers=smt-z3-opt)].

[TODO add a sentence to the conclusion with a summary of the perf]

4 Conclusion

In summary, we have successfully implemented a prototype of minimization in Inox, and are on the way to getting it merged to mainline so that any user of Inox or Stainless can use it by changing command-line arguments.

We encountered two key problems, one at the design level and one at the implementation level. At the design level, the notion of "minimization" is not as well-defined as one would think in the context of real code: one has to make choices as to what to favor, such as fewer fields vs. smaller types

in bits. At the implementation level, one cannot create a giant "sum of all variables" expressions as the variables may be of different kinds, and the types may be recursively defined, requiring the solver to stop at some point that must be explicitly chosen.

The next step is for our pull request to be merged, and then perhaps improved to support more scenarios or more complex minimization policies; this is no longer a pure formal methods problem but also enters human-computer interactions territory as to what users of verification tools prefer.

References

[1] Robert Nieuwenhuis and Albert Oliveras. On SAT modulo Theories and Optimization Problems. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, page 156–169, Berlin, Heidelberg, 2006. Springer-Verlag.