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:

```
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[1]:

```
(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

[1] The query is slightly simplified. Most notably, duplicated 0s are replaced with "...". Full examples can be found on the GitHub repository of the project.

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 if it exists or UNSAT otherwise. The existing solvers include Z3, CVC4, and Princess. Furthermore, Inox can interface with Z3 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 free variables in all constraints, 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 `BigInt`s, not to Scala `Int`s 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). And 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 chose to generate one minimizing query per number field. 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:

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

Our minimizing solver adds the following optimization goals to the SMT-LIB query generated by Inox:

```
(minimize (ite (bvsge x 0) x (bvneg x)))
(minimize (ite (bvsge y 0) y (bvneg y)))
```

Thanks to these goals, the resulting model is $x = 0$ and $y = 0$, which is not not just a satisfactory assignment 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, and to give a "size" to each parameter types such that BVs are smaller than integers which are themselves smaller than ADTs.

Here is a simple example where we define two non-recursive ADTs, one for boxed big integers and one representing complex numbers:

```
final case class BoxedInt(value: BigInt)
sealed trait Comp
final case class BoxedComp(a: BoxedInt, b: BoxedInt) extends Comp
final case class UnboxedComp(a: BigInt, b: BigInt) extends Comp
```

Given `re` and `im` two accessors functions respectively returning the `BigInt` value of the `a` and `b` field and the above definitions, we define a buggy function `mult` supposed to multiply two complex numbers. It is possible to spot the bug by trying to check that our operation is commutative:

```
def mult(a: Comp, b: Comp): Comp =
    UnboxedComp(re(a) * re(b) - im(a) * im(b),
                re(a) * im(b) + im(a) * re(a))
def multCommutative(a: Comp, b: Comp) =
    { mult(a, b) == mult(b, a) }.holds
```

With the default Z3 solver, Inox returns:

```
    z1: Comp -> BoxedComp(BoxedInt(BigInt("1")), BoxedInt(BigInt("0")))
    z2: Comp -> BoxedComp(BoxedInt(BigInt("0")), BoxedInt(BigInt("1")))
```

For the parameter `z1`, our solver adds the following constraints:

```
(minimize (ite ((_ is UnboxedComp) z1) #b...001010 #b...010100)) ; favors UnboxedComp
(minimize (ite ((_ is UnboxedComp) z1) (ite (>= (a z1) 0) (a z1) (- (a z1))) 0))
(minimize (ite ((_ is UnboxedComp) z1) (ite (>= (b z1) 0) (b z1) (- (b z1))) 0))
```

Which allows to return a smaller counter example:

```
    z1: Comp -> UnboxedComp(BigInt("0"), BigInt("0"))
    z2: Comp -> UnboxedComp(BigInt("1"), BigInt("1"))
```

2.3 Recursion

Inox supports recursive functions and ADTs by *unrolling* them and querying the SMT solver incrementally for increasing depths.

For example, one can define a recursive `MyList` ADT and recursive a recursive function `myMin` as follows in the SMT2 or TIP[2] format:

```
(declare-datatypes () (
    (MyList
        (MyNil)
        (MyCons (x Int) (xs MyList)))))
(define-fun-rec myMin ((t MyList)) Int
    (match t
        (case MyNil 1000)
        (case (MyCons x xs)
            (let ((xsMin (myMin xs)))
            (ite (<= x xsMin) x xsMin)))))
```

Note that we use `1000` as a simplified maximum value guard.

Given a `myMax` function defined in the same way as `myMin` (see linked repository for full example snippets), we can query for a list where the minimum and the maximum are different using:

```
(declare-fun aList () MyList)
(assert (>= (myMin aList) 0))
(assert (<= (myMin aList) 999))
(assert (not (= (myMin aList) (myMax aList))))
```

In such a case, Inox will first try to find a model where `aList` is `MyNil`, then instantiate the recursive function once and try models where `aList` has length 1 and so on. In this case, in will stop at length 2.

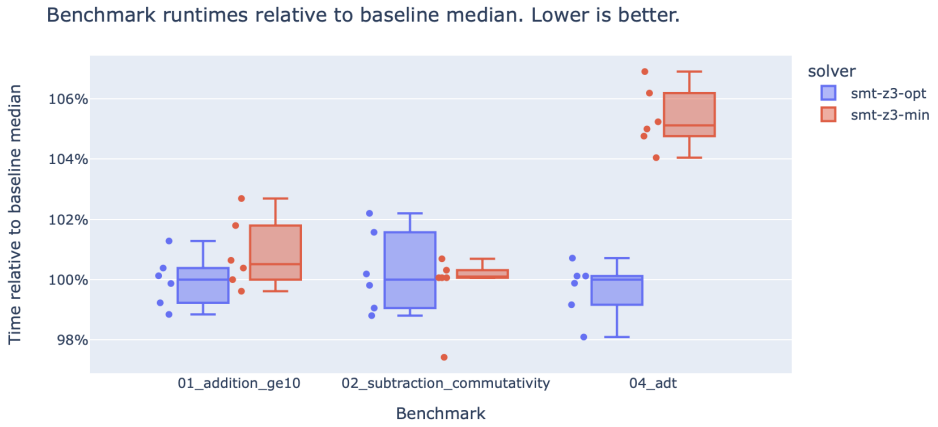
The original solver outputs `MyCons(39, MyCons(38, MyNil()))`, while the minimizing solver outputs `MyCons(0, MyCons(1, MyNil()))`.

Note that in general, this will minimize the depth of the unrolling and not the size of the argument. This is expected as a general procedure for optimizing the result of a recursive function would not be decidable because the number of possibilities is unbounded. For the same reason, Z3 will generally also not be able to optimize goals involving quantifiers (see Z3 issue #1382).

[2]TIP is a superset of SMT2. This example does not use any extra feature from TIP and can therefore be run both by Inox (`inox --solvers=smt-z3-min example.tip`) and by Z3 (`z3 example.tip`). Note that Inox automatically outputs a model, while Z3 needs an explicit `(get-model)` commands to do so.

3 Benchmarks

While minimal models are easier to read, it is a legitimate question to ask if it implies a performance hit, and if so if it is worth it. Therefore, we evaluated the performance of the new `smt-z3-min` solver on the 3 Stainless input examples presented in this report. The graph below summarize the performance `smt-z3-min` relatively to the `smt-z3-opt`:



Each run example code has been run 6 times and each individual run is represented one point on the graph. The first example suffers from a 1% performance hit, the second suffers from no observable performance hit, and the third one is 5% slower on average with the minimizing solver.

Therefore, there *is* a minimal performance overhead induced by model minimization, but we believe that is stays within a very reasonable range. Also, note that this overhead should be induced only if a counter-example is found, but not otherwise.

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. For instance, at what point does an integer become "bigger" than an ADT? Is it better to have two 15-bit values or one 32-bit value?

Our code is in the pull requests #171 (merged) and #176 (open) for Inox. We also have a toy implementation of DPLL in Scala at https://github.com/mbovel/formal-verification-project/tree/main/paper-test-implementation.

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.