CHC solvers in Stainless: Work in Progress

Sankalp Gambhir
EPFL
sankalp.gambhir@epfl.ch

Viktor Kunčak
EPFL
viktor.kuncak@epfl.ch

Stainless is a program verifier for Scala programs. Its solver component, Inox, uses fair unfolding of functions and their contracts to find counterexamples and perform k-inductive proofs. When invariants are not k-inductive, solving verification conditions typically leads to infinite unfolding. In practice, this results in a timeout or, when the underlying SMT solver is fast, stack overflow in Inox due to many unfoldings. To overcome this limitation, this paper outlines how we started integrating into Inox (and, hence, into Stainless) Horn clause solvers, which are designed to infer inductive invariants. Using Eldarica and z3's Spacer, we present encouraging preliminary experiments with successful proofs of integer-manipulating Scala programs whose verification previously diverged in Stainless.

1 Introduction

Consider the program in Fig. 1 showing a recursive implementation of the function `posTwice(x)`, computing $\max(0, 2x)$. We wish to ensure that the function cannot evaluate to -1 with a postcondition (ensuring). Attempting to verify this program with the Stainless verifier [9] results in a stack overflow caused by unbounded unfolding. Similar divergence of the underlying algorithm appears for other non-inductive invariants, such as statements that the result is not equal to 1001.

```
def posTwice(x: BigInt): BigInt = {
  if x <= 0 then BigInt(0)
  else 2 + posTwice(x - 1)
} ensuring(_ != -1)
```

Figure 1: Scala implementation of `posTwice`(x), computing $\max(0, 2x)$.

In this paper, we describe the preliminary support for the Horn solvers such as Eldarica [10] and Spacer [8] in the Inox solver framework [4], which is used as the back end solver for the Stainless verifier [9, 6] (historically, Stainless and Inox were derived by refactoring of the Leon verifier, first presented in [12]). We briefly describe the encoding of programs and the interpretation of solver responses and present examples showing successful invariant inference. We conclude by describing ongoing and future work on further integrating CHC solvers and on checking and further automating high-level proofs about programs.

2 Stainless and Inox

At its core, Inox relies on unrolling recursive function definitions, followed by a quantifier-free encoding of the resulting program. This encoding is sent to an SMT solver for verification. If the SMT solver finds

Submitted to:
HCVS 2024

a counterexample, or succeeds in proving the validity of the assertion, the result is returned to the user. If neither occurs, the query is incrementally expanded by unrolling any remaining function calls and their invariants in the query, ad infinitum. This unrolling procedure as implemented in Stainless and Inox is complete for counterexample search [13], but as in this example, does not necessarily terminate for valid assertions. The reason that SMT solvers use axiomatizations that, even if complete for quantifier-free formulas, are not complete in the presence of recursive functions with respect to intended canonical models, so inductive reasoning is required [11].

Looking again at the program in Fig. 1: in a manual proof attempt, one would attempt to show the required property by instead noting the stronger and inductive invariant that the result of the function is non-negative. The current verification procedure lacks the ability to infer and reason with such invariants.

This paper presents an approach to translate the correctness of the program to an SMT-LIB query [1], interpreting the single-argument function `posTwice(x)` as a binary predicate $posTwice(x,y)$, such that $posTwice(x,y) \iff y = \texttt{posTwice}(x)$. We provide the query as input to a Horn clause solver and interpret it so that models imply the existence of inductive invariants. Given the convention of Inox, this means that a model is reported as unsatisfiability of the query that checks the existence of a counterexample. Using this approach, we are able to prove examples such as above and many others.

The model returned by Eldarica exhibits the invariant we are looking for, with the result of `posTwice` (here, `B`) being non-negative:

```
(define-fun posTwice ((A Int) (B Int)) Bool (>= B 0))
```

An equivalent model is also output by z3's Spacer.

Adding this invariant to the original problem allows non-inductive modes of solvers such as z3 and cvc5 to prove the result; storing such inferred invariants would be one way to speed-up future verification attempts.

3 Problem Encoding

We encode the problem as a set of Horn clauses interpreting functions as predicates, such that a function $f(x_1,\ldots,x_n,y)$ is interpreted as a predicate $F(x_1,\ldots,x_n,y') \iff y' = f(x_1,\ldots,x_n)$. Finally, the safety assertion is added to the query.

Note that in the Horn clause encoding, the satisfiability results are inverted compared to the usual semantics Inox assumes for SMT solvers. For a predicate abstraction problem, a satisfiable result implies the discovery of an invariant, while an unsatisfiable result implies the discovery of a counterexample, presented as a *proof* of unsatisfiability [2].

The solver interface must intercept these results and reconstruct the violating trace from the unsatisfiability proof. In the case of a satisfiable result, the invariant can additionally be extracted and added to the call-site in the original problem, which aids the solving of future queries. (Automatic mapping of invariants to Scala source is not yet supported in the current implementation.)

4 Implementation

Stainless formally incorporates its unfolding solver, Inox [4], as a dependency. To support Horn clause solvers, we extend the scala-smtlib [5] backend of Inox, which provides a common Scala interface to solvers implementing the SMT-LIB standard. The implementation can be used either through the Stainless verifier, by pointing to the updated Inox implementation as a dependency, or through the Inox CLI, with problems encoded in either the SMT-LIB or TIP [3] formats. The resulting implementation generates SMT-LIB files and can, in principle, be used with various Horn clause solvers. Our experience comes from using it with the Spacer solver [8] inside z3, as well as with Eldarica [10].

5 Results

We implement preliminary support for Horn solvers extending the Inox solver framework, targeting the scala-smtlib [5] backend, allowing portable interfacing with different solvers. We have experimented with Eldarica and z3/Spacer.

The extension to Inox is available at:

`https://github.com/sankalpgambhir/inox/tree/hcvs24`

A version of Stainless with only the dependency pointing to the updated version of Inox for testing may also be found at `https://github.com/sankalpgambhir/stainless/tree/hcvs24`. Both repositories contain a folder `inductiveExamples/` containing the examples described in the paper and appendices with the respective solver outputs.

The solvers, provided the recursive definitions of several integer programs annotated with *non-inductive* properties to be verified, infer inductive invariants that lead to successive verification where the standard SMT-backed verification of Inox diverges.

We provide several tested examples in Appendix B. Further examples and their outputs may be found in the folder `inductiveExamples/` in the above repository.

6 Conclusion and Future Work

In this work, we describe preliminary support for Horn solvers in the Inox component of the Stainless program verifier. Horn solvers provide powerful reasoning capabilities for a subset of problems arising from program verification, and it is not always feasible to adapt verification conditions to fit the input language of the Horn solver.

We consider this to be the most promising avenue for invariant inference in Stainless. Currently, Stainless performs only a very limited form of inference, as part of its termination checker. We expect that Stainless users will continue to provide most of the subtle inductive invariants. That said, we hope that Horn clause solvers will prove to be a robust technique that eliminates the need to state some very simple invariants, as well as the need to repeat properties that are stated already in the source code, but not in the place where Stainless uses them as inductive invariants.

We intend to improve the current state of the solver framework in two main ways. First, we will use Horn solvers on smaller subproblems to infer invariants, and augment the original problem with the results. This would allow further calls to the solver, as well as to other SMT solvers, to utilize the discovered information to effectively reason about the problem.

Second, we will perform Horn solver validation by exporting the results of the Horn solver to our interactive theorem prover, Lisa [7], also implemented in Scala, to allow the user to augment the reasoning by providing high-level lemmas or hints about the Horn clauses, allowing for reuse of reasoning across problems, and increasing the trust in results by providing proof certificates from the prover.

The combination of these two directions would allow us to improve the current verification pipeline to establish the correctness of programs with proofs grounded in well-studied mathematical foundations, while maintaining a high level of automation.

References

[1] Clark Barrett, Aaron Stump, Cesare Tinelli et al. (2010): *The SMT-LIB standard: Version 2.0*. In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, 13, p. 14.

[2] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan & Andrey Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner & Wolfram Schulte, editors: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, *Lecture Notes in Computer Science* 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2. Available at `https://doi.org/10.1007/978-3-319-23534-9_2`.

[3] Koen Claessen, Moa Johansson, Dan Rosén & Nicholas Smallbone (2015): *TIP: Tons of Inductive Problems*. In: *International Conference on Intelligent Computer Mathematics*, Springer, pp. 333–337.

[4] EPFL LARA: *inox*. `https://github.com/epfl-lara/inox`.

[5] EPFL LARA: *scala-smtlib*. `https://github.com/epfl-lara/scala-smtlib`.

[6] EPFL LARA: *Stainless*. `https://github.com/epfl-lara/stainless`.

[7] Simon Guilloud, Sankalp Gambhir & Viktor Kunčak (2023): *LISA - A Modern Proof System*. In Adam Naumowicz & René Thiemann, editors: *14th International Conference on Interactive Theorem Proving (ITP 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 268, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 17:1–17:19, doi:10.4230/LIPIcs.ITP.2023.17. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.17`.

[8] Arie Gurfinkel (2022): *Program Verification with Constrained Horn Clauses (Invited Paper)*. In Sharon Shoham & Yakir Vizel, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 19–29.

[9] Jad Hamza, Nicolas Voirol & Viktor Kunčak (2019): *System FR: Formalized foundations for the Stainless verifier*. *Proceedings of the ACM on Programming Languages* 3(OOPSLA), pp. 1–30.

[10] Hossein Hojjat & Philipp Rümmer (2018): *The ELDARICA Horn solver*. In: *2018 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, pp. 1–7.

[11] Adithya Murali, Lucas Peña, Ranjit Jhala & P Madhusudan (2023): *Complete First-Order Reasoning for Properties of Functional Programs*. *Proceedings of the ACM on Programming Languages* 7(OOPSLA2), pp. 1063–1092.

[12] Philippe Suter, Ali Sinan Köksal & Viktor Kuncak (2011): *Satisfiability Modulo Recursive Programs*. In Eran Yahav, editor: *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, *Lecture Notes in Computer Science* 6887, Springer, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[13] Nicolas Voirol, Etienne Kneuss & Viktor Kuncak (2015): *Counter-example Complete Verification for Higher-order Functions*. In: *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, pp. 18–29.

A posTwice SMT-LIB encoding

```
(set-logic UFLIA)

; interpret function as predicate
(declare-fun posTwice (Int Int) Bool)

; encoding of function definition

; first branch: x <= 0
(assert
  (forall ((x Int))
    (=>
      (<= x 0)
      (posTwice x 0)
    )
  )
)

; second branch: x > 0
(assert
  (forall ((x Int) (y Int))
    (=>
      (and (posTwice (- x 1) y) (> x 0))
      (posTwice x (+ y 2))
    )
  )
)

; encoding of assertion

; assertion => false
(assert
  (forall ((x Int) (y Int))
    (=>
      (and (posTwice x y) (= y (- 1)))
      false
    )
  )
)

(check-sat)
```

Figure 2: SMT-LIB encoding of the `posTwice` function.

Solver arguments and outputs for the query in Fig. 2, we run z3 with and without `(set-logic HORN)`:

```
$ z3 posTwice.smt2 --model # without HORN
unknown
$ z3 posTwiceHorn.smt2 --model # with HORN
```

```
sat
(
  (define-fun posTwice ((x!0 Int) (x!1 Int)) Bool
    (not (<= x!1 (- 1))))
)
$ cvc5 posTwice.smt2 --produce-models
unknown
$ eldarica posTwice.smt2 -hsmt -ssol
sat
(define-fun posTwice ((A Int) (B Int)) Bool (>= B 0))
```

B Examples of Invariants Inferred with Horn Clause Solvers

Here, we provide some small interesting examples and the outputs on attempting to verify their post-conditions through both Eldarica and z3/Spacer. In each case, the invariants are slightly different, with Eldarica generating smaller/visibly simpler invariants in our tests. More examples can be found on the linked repository `https://github.com/sankalpgambhir/inox/tree/hcvs24`.

B.1 Fibonacci

Fibonacci with `fib(x) != -1`:

```
def fib(n: BigInt): BigInt = {
    require(n >= 0)
    decreases(n)
    if n <= 1 then
        BigInt(1)
    else
        fib(n - 1) + fib(n - 2)
} ensuring(_ != -1)
```

Eldarica:

```
sat
(define-fun fib ((A Int) (B Int)) Bool (>= B 1))
```

z3/Spacer:

```
sat
(
  (define-fun fib ((x!0 Int) (x!1 Int)) Bool
    (not (<= x!1 (- 1))))
)
```

Notably, the invariants here are different but both correct!

B.2 McCarthy's 91

McCarthy's 91 function with `m(x) != 50`:

```
def m(n: BigInt): BigInt = {
  if (n > 100) n - 10
  else m(m(n + 11))
} ensuring(_ != 50)
```

Eldarica:

```
sat
(define-fun mccarthy ((A Int) (B Int)) Bool (>= B 91))
```

z3/Spacer:

```
sat
(
  (define-fun mccarthy ((x!0 Int) (x!1 Int)) Bool
    (not (<= x!1 50)))
)
```

Eldarica finds quite the interesting and precise invariant here!

B.3 Recursive Addition

(Natural) Addition function asserting `add(x, y) > -10`:

```
def add(x: BigInt, y: BigInt): BigInt = {
    require(x >= 0 && y >= 0)
    if x <= 0 then y else add(x - 1, y + 1)
} ensuring(_ > -10)
```

Eldarica:

```
sat
(define-fun add ((A Int) (B Int) (C Int)) Bool (and (and (>= A 0) (>= B
   0)) (>= C 0)))
```

z3/Spacer:

```
sat
(
  (define-fun add ((x!0 Int) (x!1 Int) (x!2 Int)) Bool
    (or (and (not (<= x!2 (- 2)))
             (>= x!0 0)
             (>= x!1 0)
             (>= x!0 1)
             (>= x!1 (- 1))
             (not (<= x!0 0)))
        (not (<= x!2 (- 2)))
        (and (>= x!0 0) (>= x!1 0) (<= x!0 0) (= x!2 x!1))))
```