Stainless Verification System Tutorial

Viktor Kunčak
LARA Research Group
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
viktor.kuncak@epfl.ch

Jad Hamza
LARA Research Group
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
jad.hamza@epfl.ch

***Abstract*—Stainless (https://stainless.epfl.ch) is an open-source tool for verifying and finding errors in programs written in the Scala programming language. This tutorial will not assume any knowledge of Scala. It aims to get first-time users started with verification tasks by introducing the language, providing modelling and verification tips, and giving a glimpse of the tool's inner workings (encoding into functional programs, function unfolding, and using theories of satisfiability modulo theory solvers Z3 and CVC4).**

Stainless (and its predecessor, Leon) has been developed primarily in the EPFL's Laboratory for Automated Reasoning and Analysis in the period from 2011-2021. Its core specification and implementation language are typed recursive higher-order functional programs (imperative programs are also supported by automated translation to their functional semantics). Stainless can verify that functions are correct for all inputs with respect to provided preconditions and postconditions, it can prove that functions terminate (with optionally provided termination measure functions), and it can provide counter-examples to safety properties. Stainless enables users to write code that is both executed and verified using the same source files. Users can compile programs using the Scala compiler and run them on the JVM. For programs that adhere to certain discipline, users can generate source code in a small fragment of C and then use standard C compilers.

***Index Terms*—verification, formal methods, proof, counter-example, model checking, Scala, functional programming, satisfiability modulo theories**

I. Introduction

Stainless [1] is a tool for verifying and finding errors in programs written in a subset of the Scala [2] programming language. Stainless is open source (distributed under Apache license) and hosted on GitHub at:

https://github.com/epfl-lara/stainless/
https://epfl-lara.github.io/stainless/

Stainless (and its predecessor, Leon) have been developed primarily in the EPFL's Laboratory for Automated Reasoning and Analysis in the period from 2011-2021, see, in particular [1], [3] as well as [4]–[14]. The core specification and implementation language of Stainless are typed recursive higher-order functional Scala programs. It also supports certain imperative programs [4], [6]. Stainless can verify that functions are correct for all inputs with respect to provided preconditions and postconditions, it can prove that functions terminate (with optionally provided termination measure functions), and it can also provide counter-examples to safety properties.

Stainless can be used to write programs that are directly executable and proven correct. In particular, because it uses Scala's syntax and type system, users can execute Stainless programs using the standard Scala compiler (version 2.12.13 at the time of writing). In addition, there are passes that eliminate non-executable (ghost) code from source to make sure that it does not result in run-time overhead after compilation. For programs that adhere to certain discipline the "genc" option of Stainless can be used to generate C source code that compiles with common compilers such as gcc.

A. Outline

In this tutorial, we show examples demonstrating how to use Stainless to develop verified models and programs. We will mostly use basic notation for functional programming, which we will introduce along the way. We will use Stainless version 0.9 or later.

In addition to basic introduction, we will suggest strategies for specifying programs and helping Stainless prove them correct. An example is using lemmas and proving them by induction expressed through terminating recursion.

To help users be more effective when using Stainless, we also outline key mechanisms that Stainless uses in proof and counterexample search: encoding into functional programs, function unfolding, and using rich theories of satisfiability modulo theory solvers Z3 and CVC4.

II. Getting Started

Stainless is a command line application that runs on the Java virtual machine, version 1.8. We mostly test it on Ubuntu Linux. We provide releases for Linux and Mac. Others use it on Windows as well, where it may be simplest to use Windows Subsystem for Linux to get started. Download the release file from

https://github.com/epfl-lara/stainless/releases/

then unzip the file and put a link to `stainless` in your path.

The following is a simple program, call it `MaxBug.scala`, containing a function `max`. Max attempts to compute maximum of the two 32-bit integers by returning one of them, depending on the sign `d` of their difference.

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_2

```
object TestMax {
  def max(x: Int, y: Int): Int = {
    val d = x - y
    if (d > 0) x
    else y
  } ensuring(res =>
    x <= res && y <= res && (res == x || res == y))
}
```

We use `object` to group functions into modules. We define functions using `def` and provide their parameters (here: `x` and `y`) and their types, as well as the return type. We define local immutable values using `val` keyword. Scala infers the type of `d` as `Int`.

After the usual body, we introduced an `ensuring` statement. The first identifier, `res`, binds the return value of the function. After the arrow `=>` we state the property we would like the result to satisfy. In this case, the result should be greater than each argument and it should be equal to one of them.

Invoke `stainless MaxBug.scala` and you may get output containing some of the following.

```
MaxBug.scala:7:49: warning:  => INVALID
  x <= res && y <= res && (res == x || res == y))
                                               ^

warning: Found counter-example:
warning:  y: Int -> -2147483648
          x: Int -> 1
Verified: 0 / 3

 stainless summary

MaxBug.scala:3:13: max Subtraction overflow invalid
MaxBug.scala:7:37: max postcondition        invalid
MaxBug.scala:7:49: max postcondition        invalid
................................................
total: 3    valid: 0    (0 from cache) invalid: 3
```

Use `--timeout=5` to set time out to 5 seconds. and `--no-colors` to request clean ASCII output with parsable line numbers in reports.

Why did Stainless report a counterexample? Indeed, executing `max` with the two provided values computes using signed 32-bit arithmetic the value `-11` for `d`, so the function returns `y` as the result `res` so `y <= res` is false. We can repair this example in at least two ways:

- Use `if (x <= y)` instead of the value `d`.
- Use `BigInt` instead of `Int`, thus adopting unbounded integers instead signed 32-bit ones.

If you run your program several times, you may notice that Stainless reports that a valid verification condition was persistently cached (inside `.stainless-cache`). You can turn off caching with `--vc-cache=false`.

You may find the `--watch` option useful when modifying a file several times, which makes Stainless run verification whenever the source file is changed.

By default, Stainless uses a version of `z3` (4.7.1) which is packaged inside Stainless (`--solvers=nativez3`). This allows Stainless to interact with `z3` through Java calls. You may also use an externally built version of `z3` (for instance, `z3 4.8.12` is shipped with the release) by specifying `--solvers=smt-z3`. In that case, Stainless will communicate with `z3` using SMT-LIB files, which might be slower than Java calls, but has two benefits. First, you get to use the newest release of `z3`. Second, `smt-z3` is more likely to respect timeouts than `nativez3`.

You can also use CVC4 as the solver if you download and put `cvc4` executable on your path. You can use both with `--solvers=smt-cvc4,smt-z3`. Use `--debug=smt` to preserve the generated SMT-LIB files and look for them in the `smt-sessions` directory.

III. Verified Functional Programming

We will now implement a simple function that computes differences of successive elements of a list. Let us start our file with `import stainless.collection._` so we can use the immutable `List` library of Stainless. You can find the sources of this and other library files at following URL:

https://github.com/epfl-lara/stainless/blob/master/frontends/library/stainless/collection/List.scala

Let's try to write a function `diffs` that takes a list of elements, for example x_1, x_2, x_3, x_4 and keeps the first element and then follows it by the list of their differences. In this case we would like to obtain $x_1, x_2 - x_1, x_3 - x_2, x_4 - x_3$. For empty and one-element list the output equals input. Let us write this as the default implementation. We can also state the example of four-element list as a symbolic test case. To state it, we use another function with a dummy body and a postcondition that invokes `diffs`.

```
import stainless.collection._
object Diffs {
  def diffs(l: List[BigInt]): List[BigInt] = {
    l match {
      case Nil() => l
      case _ :: Nil() => l
      // missing cases
    }
  }
  def test(x1: BigInt, x2: BigInt,
           x3: BigInt, x4: BigInt): Unit = {
  } ensuring(_ =>
    diffs(List(x1,x2,x3,x4)) ==
     List(x1, x2 - x1, x3 - x2, x4 - x3))
}
```

After developing a function that meets this partial specification, we can see whether it meets a stronger specification. For example, we can define the inverse function `undiff` that takes $y_0, y_1, \ldots, y_n$ and computes $y_0, y_0 + y_1, \ldots, \sum_{i=0}^{n} y_i$. Being masters of functional programming, we recognize that this is just a prefix sum of a list, so we define it by

```
def undiff(l: List[BigInt]): List[BigInt] =
  l.scanLeft(BigInt(0))(_ + _).tail
```

where `scanLeft` is defined in our `List` library. Now we can add as the `ensuring` condition of `diffs` the condition `ensuring (res => (undiff(res)== l))`. It so happens that Stainless proves this condition automatically using its algorithm. As an off-line exercise, try to prove this result with pen and paper. This might give you a sense on how Stainless is able to prove this property.

The algorithm of Stainless initially treats called functions as unknown (uninterpreted) mathematical functions. It then

iteratively expands each call by defining the function to be equal to one unfolding of its body and also inserts the `ensuring` clause as an assumption.

IV. Amortized Queue

We have found Stainless to work very well for verification of purely functional data structures. Let us examine the case of an amortized queue such as the one from [15, Section 5.2, Page 42]. We will start by writing down an *abstract class*. In this class we define methods with dummy bodies denoted by `???` but with `ensuring` clauses that specify the desired behavior of operations. To specify the behavior we use `toList` function, which is also left unspecified in the abstract class.

```
import stainless.collection._
import stainless.lang._
abstract class Queue[A] {
  def enqueue(a: A) = (??? : Queue[A])
    .ensuring(res =>
    res.toList == this.toList ++ List(a))

  def dequeue: Option[(A, Queue[A])] =
    (??? : Option[(A, Queue[A])])
  .ensuring(res => res match {
    case None() =>
      this.toList == Nil[A]()
    case Some((a, q)) =>
      this.toList == a :: q.toList
    })

  def toList: List[A]
}
```

When we extend the abstract class, Scala requires us to define `toList`, whereas Stainless ensures that our implementation meets the specifications in the abstract class. We can implement an inefficient queue using a single list.

```
case class SimpleQueue[A](l: List[A])
    extends Queue[A] {
  def enqueue(a: A) = SimpleQueue(l ++ List(a))

  def dequeue = l match {
    case Nil() => None()
    case Cons(x, xs) => Some((x, SimpleQueue(xs)))
  }

  def toList = l
}
```

Stainless successfully verifies that the properties required by a queue are satisfied by this implementation. Even if correct, this implementation is inefficient because `enqueue` takes linear time in the current number of queue elements. We will thus try to develop and prove correct the implementation like one from [15, Section 5.2, Page 42] that uses two lists and that has constant time amortized complexity.

```
case class AmortizedQueue[A](front: List[A],
                             rear: List[A])
    extends Queue[A] {

  def toList = front ++ rear.reverse
```

The `toList`, which we use only for specification, gives us a hint on how to implement `enqueue` efficiently. For `dequeue` we will need a `reverse` operation on lists, which we can implement in linear time. Despite its complexity, our version of `dequeue` will be verified automatically. As for `enqueue`, its implementation is simple, yet its proof turns out to require some well known property of lists that we need to tell Stainless to invoke explicitly!

```
def enqueue(a: A): Queue[A] = {
  val res: Queue[A] = // to fill

  // You can state using assertions things you know are true,
  // to see if Stainless is able to prove them:
  assert(res.toList == front ++ (a :: rear).reverse)

  // Alternatively, you can use an equation style reasoning.
  // Here Stainless should timeout from the second to the third
  // step, because some steps are missing.
  (
    res.toList ==:| trivial |:
    front ++ (a :: rear).reverse ==:| trivial |:
    // Add missing steps here to arrive to the result.
    // For complicated steps, you need to invoke lemmas
    // instead of writing 'trivial'.
    this.toList ++ List(a)
  ).qed

  res
}
```

V. Properties and Proofs

How do we state properties in Stainless? We write a property $\forall x : T.F(x)$ as a function `lemmaF` defined by:

```
def lemmaF(x: T): Unit = {
  ()
} ensuring (_ => F(x))
```

When we wish to instantiate the property taking x to be some specific value v, we insert a function invocation `lemmaF(v)` into the part of the code where we need this property. Suppose that proving property $\forall x : T.F(x)$ is not automatic. Then verification of `lemmaF` itself will fail, as stated. If $F(x)$, for example, follows from $G(x, x + 1)$ that is established in `lemmaG(x,y)`, then we can state and prove `lemmaF` as:

```
def lemmaF(x: T): Unit = {
  lemmaG(x,x+1)
} ensuring (_ => F(x))
```

Thus, we can adopt the following strategies for libraries of lemmas:

- introduce a function for a lemma
- use a function parameter for each universally quantified variable
- write lemma statement in the `ensuring` clause
- use the body of the function to encode a high-level proof, with function invocations corresponding to applying previously proven lemmas.

Purely universal statements can return `Unit` type. For existential statements, we can often state their constructive Skolemized form and return a witness for the existential quantifier from the lemma.

It can be helpful to examine some proofs of properties in the `List` library. Remarkably, we can even make recursive invocations of functions in their bodies. Which mathematical reasoning principle do such proofs correspond to?

VI. Digits

For built-in types such as `Int` and `Long`, the SMT solvers will successfully reason about their bitwidth representation. What if we wish to reason about the bits of arbitrarily large numbers? As a simple example, let us define simple addition as a recursive function on lists of bits.

```
import stainless.annotation._
import stainless.lang._
import stainless.collection._
object AddBitwise {
  type Digits = List[Boolean]
  val zero = Nil[Boolean]()

  def add(x: Digits, y: Digits, carry: Boolean):
      Digits = {
    require(x.length == y.length)
    (x,y) match {
      case (Nil(), Nil()) =>
        if (carry) true::zero else zero
      case (Cons(x1,xs), Cons(y1,ys)) => {
        val z = x1 ^ y1 ^ carry
        val carry1 = (x1 && y1) ||
                     (x1 && carry) ||
                     (y1 && carry)
        z :: add(xs, ys, carry1)
      }
    }
  }
}
```

How can we state that such addition is commutative? How can we prove it in Stainless? As an off-line exercise, think about how we can prove that this corresponds to actual addition on integers (`BigInt`).

VII. Termination

The following recursive function searches for an element in a sorted array, but it has a bug. You may run Stainless on this file to spot it. Fix the issue, and add a **`decreases`** clause at the beginning of the function to ensure that Stainless can prove the function terminating.

```
import stainless.lang._

object BinarySearch1 {

  def search(arr: Array[Int], x: Int, lo: Int, hi:
      Int): Boolean = {
    if (lo <= hi) {
      val i = (lo + hi) / 2
      val y = arr(i)
      if (x == y) true
      else if (x < y) search(arr, x, lo, i-1)
      else search(arr, x, i+1, hi)
    } else {
      false
    }
  }
}
```

In Stainless, all functions are required to have a measure (either inferred automatically, or written in a **`decreases`** clause by the user). The system in its current design would be unsound (we would be able to prove false postconditions or assertions) if we allowed non-terminating functions.

VIII. Imperative Features

Stainless supports some imperative features, such as local mutable variables, while loops, return statements, and more (see https://epfl-lara.github.io/stainless/imperative.html). Stainless transforms these constructs into functional programs.

Using a while loop and a return statement, rewrite the `findIndexOpt` function:

```
def findIndexOpt(ar: Array[Int], v: Int):
                                      Option[Int] = {

}
```

that finds an index of element `v` in a sorted array `ar`. Prove that, when your function returns `Some(i)`, then `ar(i)== v`. To prove that array indices are within bounds, you will need a loop invariant, for which the syntax is:

```
(while(...) {
  decreases(...)
  ...
}).invariant(...)
```

Does Stainless help you if you make an overflow mistake when computing the middle of an interval using bounded arithmetic?

Note that while loops require **`decreases`** clauses as well (when the measure cannot be inferred automatically), because they are translated into recursive functions by Stainless. To see how the while loop and the return statement are transformed, you may run the command below on your file. Stainless has a pipeline containing several phases, and `ReturnElimination` is the one that removes while loops and return statements. The `--debug-objects` option tells Stainless to only display the `findIndexOpt` function in the debug output.

```
stainless --debug=trees
    --debug-objects=findIndexOpt
    --debug-phases=ReturnElimination FindIndex.scala
```

As a harder exercise, identify and prove a stronger postcondition of `findIndexOpt`: what can we state in the postcondition for the case when the function returns `None`? What assumptions and loop invariants do we need to be be able to prove this postcondition?

IX. Design Principles

A number of verification systems have been developed in the past decades. Stainless tries to borrow many of the features that others and us have found useful in other systems. At the same time, it is driven by a somewhat unique combination of principles, whose understanding may help set the expectations from the tool.

A. Searching for Both Proofs and Counterexamples

From the beginning [13], the system was designed to search for both counterexamples and proofs in a unified iterative loop. Thanks to this design, on many programs Stainless behaves like a combination of a bounded model checker and a k-inductive prover such as [16]: we can often expect a definite answer, whether the program verifies or has a counterexample.

B. Recursive programs as foundation, not transition systems.

Operational semantics tells us that we can translate functional (and many other) programs into transition systems. This has even been used in verification tools with success []. Nonetheless, we believe that it carries significant overhead, especially for proofs. Thus, like in ACL2 [17], [18] our intermediate representation is based on recursive functions [13] and we hope to leverage high-level structure to make verification more feasible, much like Liquid Haskell [19] which needs to be complemented with symbolic execution to also generate counterexamples [20]. Consequently, iterative unfolding of our recursive functions in Stainless gives a different sequence of approximations than the one we would obtain by representing programs using control-flow graphs and explicit stacks [21].

C. Top-down verification for each function.

Stainless verifies each desired function one by one. When verifying a function f, it does not check which other parts of code invoke f. In particular, it will, in its current design, not infer preconditions for a function automatically. Preconditions need to be explicitly specified using a `require` clause at function entry. On the other hand, when Stainless examines the body of f and finds a function g, then it will examine not only the specification of g, but also its body. If g is recursive, this process will continue, with a check for counterexample and check for unsatisfiability performed at each step. This process treats functions more transparently than some modular verifiers. The process is also breadth-first, instead of having the form of directed rewriting as in some other systems. The effectiveness of this process is explained in part by the fact that it results in a decision procedure for certain classes of functions [14], [22], [23]. Furthermore, we continue to be surprised by how well this simple strategy works in practice, even if we have no theoretical reason to know that it will succeed.

D. Scala subset as the input language.

Stainless uses Scala as a language that has substantial user base, regularly ranked higher than Haskell and LISP in Stack Overflow developer surveys [24], which is relevant for maintaining the correspondence between what executes and that is verified. As a functional language, Scala contains an expressive purely functional fragment which can be used for specification and modelling. The users of Stainless thus largely avoid the need to learn a separate specification language, because functional programs are a great specification vehicle. At the same time, the system supports polymorphism and subtyping with a type system that eliminates many nonsensical programs before they waste user's time inside the program verifier's loop. That said, Stainless purposely avoids by design certain Scala 2 features, such as null references and complex initalization. Other features, such as machine integers, are modelled precisely: it is certainly necessary in practice to have machine integers of various width available (for example, 32-bit Int and 64-bit Long), but it is also helpful to use unbounded BigInt data types, especially for specifications, and these different types should not be confused. Stainless provides the user a choice and maps these data types and operations on them to the appropriate types and theories inside SMT solvers [8]. Subtyping is currently implemented via a translation into a language with disjoint types [3]; its use requires additional encoding and may slow down verification. Imperative features are supported as a choice of either unshared mutable state [6] or using a model [4] that, at user level, is similar to dynamic frames [25] of Dafny [26].

E. Embracing SMT solver theories, avoiding quantifiers.

Instead of using axioms to encode program semantics and data types, Stainless leverages algebraic data types, sets, and arrays. Stainless thus currently emits quantifier-free queries to solvers (either Z3 or CVC4). The hope with this choice is that SMT solvers will remain predictable for both proofs and counterexamples. In contrast, the use of quantifiers may lead to more automation and sometimes excellent performance for proofs, but quickly leads outside of the space where the solvers can reliable report counterexamples.

F. Executability of programs and specifications.

In Stainless we aim to write programs that can be compiled using the standard Scala compiler. Specification constructs in Stainless are defined in a Scala library and they have dummy execution semantics. In some cases, even such dummy semantics may result in overhead, so we have developed passes that eliminate some of the specification code altogether. In addition, Stainless has a subset that can be used to generate C code suitable for embedded systems, an enhanced version of such functionality developed for Leon [27].

Acknowledgements. Research on Stainless has been funded in part by (i) the Swiss Science Foundation grants 200021_132176, 200020_138204, 200020_146649, 200021_144503, 200020_159949, and 200021_175676. (ii) European Research Council (ERC) Starting Grant PE6-306484-IMPRO, (iii) The Swiss State Secretariat for Education, Research and Innovation, Swiss Space Office grant "Embedded Flight Software Verification–ESOVER" and (iv) the envelope budget for the LARA group from the EPFL School of Computer and Communication Sciences.

Stainless and Inox were created from parts of Leon code by Nicolas Voirol. In addition to Nicolas and the two authors of this tutorial, contributors to Stainless and Inox include: Roman Ruetschi, Georg Stefan Schmid, Marco Antognini, Ravichandhran Madhavan, Etienne Kneuss, Lars Hupel, Emmanouil Koukoutos, Philippe Suter, Roman Edelmann, Utkarsh Upadhyay, Ivan Kuraj, Sandro Stucki, Ruzica Piskac, Tihomir Gvero, Czipó Bence, Sumith Kulal, Lucien Iseli, Regis Blanc, Iulian Dragos, Dragana Milovančević, Antoine Brunner, Mirco Dotta, Yann Bolliger, Rodrigo Raya, Samuel Gruetter, Mikaël Mayer, Guillaume Massé. Romain Jufer worked with Jad Hamza on a fork for smart contract verification and Solidity code generation, Romain Edelmann and Rodrigo Raya developed an interactive proof assistant concept

based on Inox. Regis Blanc developed a Scala library for input and output of SMT-LIB files. ScalaZ3 interface to the Z3 dynamically linked library additionally received contributions from Ali Sinan Köksal and Thorsten Tarrach. Contributors to Stainless Bolts case studies include additionally Samuel Chassot and Clément Burgelin. We thank users of Stainless from Ateleris GmbH including Simon Felix, Filip Schramka, and Ivo Nussbaumer. We also thank MSc students at EPFL taking the Formal Verification course, completing interesting case studies and identifying bugs in the system.

REFERENCES

[1] J. Hamza, N. Voirol, and V. Kunčak, "System FR: Formalized foundations for the Stainless verifier," *Proc. ACM Program. Lang*, no. OOPSLA, November 2019.

[2] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 4th ed. Artima Inc, 2008.

[3] N. C. Y. Voirol, "Verified functional programming," Ph.D. dissertation, EPFL, thesis number 9479, 2019. [Online]. Available: http://doi.org/10.5075/epfl-thesis-9479

[4] G. Schmid and V. Kunčak, "Proving and disproving programs with shared mutable data," 2021.

[5] R. Madhavan, S. Kulal, and V. Kuncak, "Contract-based resource verification for higher-order functions with memoization," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[6] R. W. Blanc, "Verification by reduction to functional programs," Ph.D. dissertation, EPFL, thesis number 7636, 2017. [Online]. Available: http://doi.org/10.5075/epfl-thesis-9479

[7] N. Voirol, E. Kneuss, and V. Kuncak, "Counter-example complete verification for higher-order functions," in *Scala Symposium*, 2015.

[8] R. Blanc and V. Kuncak, "Sound reasoning about integral data types with a reusable SMT solver interface," in *Scala Symposium*, 2015.

[9] V. Kuncak, "Developing verified software using Leon (invited contribution)," in *NASA Formal Methods (NFM)*, 2015.

[10] E. Koukoutos and V. Kuncak, "Checking data structure properties orders of magnitude faster," in *Runtime Verification (RV)*, 2014.

[11] R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter, "An overview of the Leon verification system: Verification by translation to recursive functions," in *Scala Workshop*, 2013.

[12] A. Köksal, V. Kuncak, and P. Suter, "Constraints as control," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.

[13] P. Suter, A. S. Köksal, and V. Kuncak, "Satisfiability modulo recursive programs," in *Static Analysis Symposium (SAS)*, 2011.

[14] P. Suter, M. Dotta, and V. Kuncak, "Decision procedures for algebraic data types with abstractions," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2010.

[15] C. Okasaki, *Purely Functional Data Structures*. Cambridge University Press, 1998.

[16] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The kind 2 model checker," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9780. Springer, 2016, pp. 510–517.

[17] J. S. Moore, "Milestones from the pure lisp theorem prover to ACL2," *Formal Aspects Comput.*, vol. 31, no. 6, pp. 699–732, 2019.

[18] R. S. Boyer and J. S. Moore, "Proving theorems about LISP functions," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, N. J. Nilsson, Ed. William Kaufmann, 1973, pp. 486–493. [Online]. Available: http://ijcai.org/Proceedings/73/Papers/053.pdf

[19] N. Vazou, "Liquid haskell: Haskell as a theorem prover," Ph.D. dissertation, UNIVERSITY OF CALIFORNIA, SAN DIEGO, 2016.

[20] W. T. Hallahan, A. Xue, M. T. Bland, R. Jhala, and R. Piskac, "Lazy counterfactual symbolic execution," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 411–424. [Online]. Available: https://doi.org/10.1145/3314221.3314618

[21] L. Lamport, "The pluscal algorithm language," in *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, ser. Lecture Notes in Computer Science, M. Leucker and C. Morgan, Eds., vol. 5684. Springer, 2009, pp. 36–60.

[22] V. Sofronie-Stokkermans, "Locality results for certain extensions of theories with bridging functions," in *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, ser. Lecture Notes in Computer Science, R. A. Schmidt, Ed., vol. 5663. Springer, 2009, pp. 67–83.

[23] T. Pham, A. Gacek, and M. W. Whalen, "Reasoning about algebraic data types with abstractions," *J. Autom. Reason.*, vol. 57, no. 4, pp. 281–318, 2016.

[24] S. Overflow, "Annual developer survey," 2021. [Online]. Available: https://insights.stackoverflow.com/survey/

[25] I. T. Kassios, "Dynamic frames: Support for framing, dependencies and sharing without restrictions," in *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 268–283.

[26] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370.

[27] M. Antognini, "Extending safe C support in Leon," Master's thesis, EPFL, 2017. [Online]. Available: https://infoscience.epfl.ch/record/227942/