Developing Verified Software Using Leon

Viktor Kuncak*[1]

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract. We present Leon, a system for developing functional Scala programs annotated with contracts. Contracts in Leon can themselves refer to recursively defined functions. Leon aims to find counterexamples when functions do not meet the specifications, and proofs when they do. Moreover, it can optimize run-time checks by eliminating statically checked parts of contracts and doing memoization. For verification Leon uses an incremental function unfolding algorithm (which could be viewed as k-induction) and SMT solvers. For counterexample finding it uses these techniques and additionally specification-based test generation. Leon can also execute specifications (e.g. functions given only by postconditions), by invoking a constraint solver at run time. To make this process more efficient and predictable, Leon supports deductive synthesis of functions from specifications, both interactively and in an automated mode. Synthesis in Leon is currently based on a custom deductive synthesis framework incorporating, for example, syntax-driven rules, rules supporting synthesis procedures, and a form of counterexample-guided synthesis. We have also developed resource bound invariant inference for Leon and used it to check abstract worst-case execution time. We have also explored within Leon a compilation technique that transforms real-valued program specifications into finite-precision code while enforcing the desired end-to-end error bounds. Recent work enables Leon to perform program repair when the program does not meet the specification, using error localization, synthesis guided by the original expression, and counterexample-guided synthesis of expressions similar to a given one. Leon is open source and can also be tried from its web environment at leon.epfl.ch .

1 Overview

We present Leon, a system supporting the development of functional Scala [21] programs. We illustrate the flavor of program development in Leon, and present techniques deployed in it. Leon supports a functional subset of Scala. It has been observed time and again that one of the most effective ways of writing software that needs to be proved correct is to write it in a purely functional language. ACL2 [8] and its predecessors have demonstrated the success of this approach, resulting in verification of a number of hardware and software systems.

* This work is supported in part by the European Research Council (ERC) Grant *Implicit Programming* and Swiss National Science Foundation Grant "Constraint Solving Infrastructure for Program Analysis".

Unlike ACL2, the input language supported by Leon has Hindley-Milner style type system [6, 19]. Leon currently delegates parsing and type analysis to the existing Scala compiler front end; a Leon program is a valid Scala program. For convenience, Leon also supports local functions, local mutable variables and while loops, which are expanded into recursive functions [1]. Among related tools to Leon as far as verification functionality is concerned are liquid types [27], though Leon has a real model checking flavor in that it returns only valid counterexamples.

Leon functions are annotated with preconditions and postconditions using Scala syntax for contracts [20]. They manipulate unbounded integer and bitvector numerical quantities, algebraic data types expressed as case classes, lists, functional arrays, and maps. An ambitious research direction introduces a Real data type that compiles into a desired finite-precision data type that meets given precision guarantees [2,3]. The main challenge in this work is automatically computing the accumulation of worst-case error bounds though non-linear computations, which requires also precisely computing ranges of variables in programs using constraint solving.

Contracts in Leon can themselves refer to recursively defined functions, which makes them very expressive. Leon aims to find counterexamples when functions do not meet the specifications, and proofs when they do. For verification Leon uses an incremental function unfolding algorithm (which could be viewed as k-induction) and SMT solvers. The foundations of this work have been presented in [25], with first presentation of experimental results appearing in [26]. This algorithm simultaneously searches for proofs and counterexamples and has many desirable properties [24]. To speed up search for counterexamples, Leon also makes use of specification-based test generation, though this direction could be pushed further using, for example, techniques deployed in the domain-specific Scala language for test generation [17].

Leon has so far primarily relied on the Z3 SMT solver [4]; its performance and support for numerous theories including algebraic data types has proven to be very useful for automating a functional program verifier such a Leon. Particularly convenient have been extended array operations in Z3 [5] which have allowed us to encode Leon's sets, arrays, and maps efficiently. More recently we have built a more generic SMT-LIB interface and are exploring the possibility of using other solvers, as well as many of the unique features of CVC4, such as its increasingly sophisticated support for quantifiers [23] and automated mathematical induction [22].

We have also developed resource bound invariant inference for Leon by encoding the inference problem into non-linear arithmetic, and used this approach to check abstract worst-case execution time [18]. In this approach we have also shown that function postconditions can be inferred or strengthened automatically.

Constructs for preconditions (require) and postconditions (ensuring) have run-time checking semantics in standard Scala; they are simply particular assertions. Executing precise specifications at run time may change not only constant factors but also asymptotic complexity of the original program, changing, for example, insertion into a balanced tree from logarithmtic into quadratic operation. In Leon, even when contracts cannot be checked fully statically, they can be optimized by eliminating statically checked parts of contracts and doing memoization [12]. Using these techniques, it is often possible to speed up runtime checks and recover the asymptotic behavior of the original program.

Leon can also execute specifications alone (e.g. functions without body, given only by postconditions), by invoking a constraint solver at run time [13]. This mechanism reuses counterexample-finding ability as a computation mechanism [11]. Leon thus supports an expressive form of constraint programming with computable functions as constraints. While convenient for prototyping, constraint programming can be slow and unpredictable, often involving exponential search for solutions.

As a step towards more efficient and predictable approach, Leon supports deductive synthesis of functions form specifications. This functionality was originally aimed at being fully automated [10]. Synthesis in Leon is based on a custom deductive synthesis framework incorporating, for example, syntax-driven rules, rules supporting synthesis procedures [7, 14–16], and a form of counterexample-guided synthesis [10]. Subsequently we have worked on interfaces to perform this synthesis interactively, which allows the developer both to explore different alternatives if the solution is not unique, and to guide synthesis using manual steps.

Recent work enables Leon to perform program repair when the program does not meet the specification, using error localization, synthesis guided by the original expression, and counterexample-guided synthesis of expressions similar to a given one [9].

Leon is under active development and has been used in teaching courses at EPFL. It is open source and can also be tried from its web environment at the URL `http://leon.epfl.ch`.

References

1. 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.
2. E. Darulova. *Programming with Numerical Uncertainties.* PhD thesis, EPFL, 2014.
3. E. Darulova and V. Kuncak. Sound compilation of reals. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2014.
4. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
5. L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design*, Nov. 2009.
6. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.
7. S. Jacobs, V. Kuncak, and P. Suter. Reductions for synthesis procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.

8. M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.
9. E. Kneuss, M. Koukoutos, and V. Kuncak. On deductive program repair in Leon. Technical Report EPFL-REPORT-205054, EPFL, February 2014.
10. E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.
11. A. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.
12. E. Koukoutos and V. Kuncak. Checking data structure properties orders of magnitude faster. In *Runtime Verification (RV)*, 2014.
13. V. Kuncak, E. Kneuss, and P. Suter. Executing specifications using synthesis and constraint solving (invited talk). In *Runtime Verification (RV)*, 2013.
14. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2010.
15. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Communications of the ACM*, 2012.
16. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional synthesis for linear arithmetic and sets. *Software Tools for Technology Transfer (STTT)*, 15(5-6):455–474, 2013.
17. I. Kuraj and V. Kuncak. SciFe: Scala framework for effcient enumeration of data structures with invariants. In *Scala Workshop*, 2014.
18. R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification (CAV)*, 2014.
19. R. Milner. A theory of type polymorphism in programming. *JCSS*, 17(3):348–375, 1978.
20. M. Odersky. Contracts for Scala. In *Int. Conf. Runtime Verification*, 2010.
21. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide.* Artima Press, 2008.
22. A. Reynolds and V. Kuncak. Induction for SMT solvers. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2015.
23. A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *FMCAD*, pages 195–202. IEEE, 2014.
24. P. Suter. *Programming with Specifications.* PhD thesis, EPFL, December 2012.
25. 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.
26. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis Symposium (SAS)*, 2011.
27. N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 209–228. Springer, 2013.