Verifying and Synthesizing Software with Recursive Functions
(Invited Contribution)

Viktor Kuncak*

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{firstname.lastname}@epfl.ch

Abstract. Our goal is to help people construct software that does what they wish. We develop tools and algorithms that span static and dynamic verification, constraint solving, and program synthesis. I will outline the current state our verification and synthesis system, *Leon*, which translates software into a functional language and uses SMT solvers to reason about paths in programs and specifications. Certain completeness results partly explain the effectiveness of verification and synthesis procedures implemented within Leon, in particular results on decidability of sufficiently surjective abstraction functions, and the framework of complete functional synthesis.

1 Introduction

Software is more widespread than ever, thanks to trends such as mass adoption of smartphones and tablets, as well as complex software controllers in, e.g. personal vehicles. At the same time, it is still too difficult to construct software that does something meaningful, let alone enforcing that software conforms to rigorous correctness standards. This motivates our current research on making software construction more accessible to large user bases, as well as increasing the confidence in software artifacts being constructed. These tasks require automated reasoning about requirements, specifications, and implementations. A core problem is automatically mapping users' requirements into efficiently executing systems. The problem has traditionally found home in programming languages, formal methods, software engineering, and design automation, but is also related to automated reasoning, human-computer interaction, machine learning, and natural language processing. We can evaluate our progress on addressing this problem by building software development tools that help software developers and users. Their development often required new algorithms for program verification, analysis, program synthesis, and new decision procedures.

In the sequel I use an example to illustrate several usage scenarios of interest and define the corresponding algorithmic questions. The examples use the syntax

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

of the Scala programming language (`http://scala-lang.org/`, [59,60]) and are mostly based on the capabilities of the Leon verification and synthesis system for a subset of Scala (`http://leon.epfl.ch`) on which we have been working for the past few years [10, 17, 35, 40, 45–47, 52, 74–76], but we have also explored related ideas in several other works [15, 16, 24, 26–28, 31, 32, 34, 41–44, 49, 50, 56, 63, 64, 67, 70, 71, 77–81].

2 Problem Definitions through Examples

Consider computations that take inputs $i \in I$ and produce outputs $o \in O$. We view a program as a function $f : I \to O$ from inputs to outputs. We view a specification $P : (I \times O) \to \{\text{false}, \text{true}\}$ as a predicate that takes a potential input and a potential output and returns true iff the output is considered acceptable for the given input. We fix this notation throughout this section.

2.1 Four Types of Problems

We consider the following categories of problems:

(RV) **Runtime Verification**: given P, f and a specific input $i_1 \in I$, compute the value $f(i_1)$, then compute whether $P(i_1, f(i_1))$ holds.
(SV) **Static Verification**: given P, f, either prove $\forall i \in I.\, P(i, f(i))$, or find a counterexample $i_1 \in I$ such that $\neg P(i_1, f(i_1))$.
(RC) **Runtime Constraint solving**: given P and an input $i_1 \in I$, find one corresponding output $o_1 \in O$ such that $P(i_1, o_1)$. This succeeds iff $\exists o.\, P(i_1, o)$.
(SC) **Static Constraint solving**: given P, find a computable function f such that $\forall i \in I.\ P(i, f(i))$. This is a form of *program synthesis* [46, 54].

The classification uses two criteria. The first criterion (R/S) is whether the problem being solved takes place at *runtime* (R), that is, during program execution, or *statically* (S), at compile time, before the program runs. The second criterion (V/C) is whether the task is verification (V), when both the program and the specification are given, or constraint solving (C), when only the specification is given, and we aim to compute values that satisfy it.

A list definition in Scala. To make the discussion more concrete, consider the simple example of describing operations on sorted lists of integers. We choose a purely functional subset of the Scala to implement these operations (`http://scala-lang.org/`, [59, 60]). We rely only on a small subset of Scala, so our functions correspond to the mathematical notion of mutually recursive functions defined over discrete domains. Listing 1 shows the definition of lists as an algebraic data type with a zero-arity constructor *Nil* and a binary constructor $\mathit{Cons} : (\mathit{Int} \times \mathit{List}) \to \mathit{List}$. We define an algebraic data type in Scala using class inheritance; for pure functions this corresponds to a term algebra [30, 53] with the corresponding constructors (here: *Nil* and *Cons*) and selectors (here: *head* and *tail*). The domain thus represents finite sequences of integers.

Insertion into the list. Listing 2 defines a recursive function *sortedIns* that inserts a given integer into the sorted list, while preserving the property of being sorted. This is a concrete example of a program denoted f above. The ***match*** construct performs the usual case analysis on whether the list is empty or not, and, in the non-empty case, binds the provided variables x, xs to the head and tail of the list. Such code follows a standard approach for defining recursively defined structures and functions. Our methodology uses this same executable language of recursive functions to also describe the desired properties P of programs.

```
abstract class List
case class Cons(head: Int, tail: List) extends List
case object Nil extends List
```

Listing 1. List of Integers Defined Using Custom Case Classes

```
def sortedIns(e: Int, l: List): List = l match {
    case Nil ⇒ Cons(e,Nil)
    case Cons(x,xs) ⇒ if (x ≤ e) Cons(x,sortedIns(e, xs)) else Cons(e, l) }
```

Listing 2. Conventional Implementation of Insertion into a Sorted List

```
def sortedIns(e: Int, l: List): List = {
  require(isSorted(l))
  l match {
    case Nil ⇒ Cons(e,Nil)
    case Cons(x,xs) ⇒ if (x ≤ e) Cons(x,sortedIns(e, xs)) else Cons(e, l) }
} ensuring(result ⇒ isSorted(res) && content(result) == content(l) ++ Set(e))
```

Listing 3. The Insertion into a Sorted List together with its Specification

Writing specifications for functions. Suppose that we wish to specify that *sortedIns* indeed inserts the given element into the set of elements it stores, and that it maintains the ordering of the elements in the list. Listing 3 shows how to write such specification in Scala. We indicate that the input list needs to be sorted using the ***require*** operator, which takes a predicate that should hold for function arguments (function precondition). In this example we use the *isSorted* predicate to define the precondition. Listing 4 shows the definition of *isSorted* as a recursive function. To specify the postcondition, we use the ***ensuring*** clause, which also takes a predicate, but this time involving not only parameters of the function, but also its result, here bound to the *result* variable. The specification in Listing 3 indicates that the result should also be a sorted list. Moreover, it says that the set of elements stored in the resulting list, *content(result)*, should be equal

to the union of the content of the argument list *content(l)* and the singleton set $\{e\}$, denoted in Scala as *Set(e)*. Listing 5 defines the *content* function recursively. We call such function an abstraction function; it abstracts away from the list ordering and computes a set. Algebraically, it is a homomorphism, mapping values of the list algebraic data type with constructor operations into finite sets with union. The property P that we would like to ensure about the result of *sortedIns* when invoked with arguments e and l is thus:

$$\mathit{isSorted(l)} \rightarrow (\mathit{isSorted(result)} \wedge \mathit{content(result)} = \mathit{content(l)} \cup \{e\})$$

In the terminology of our classification, the above predicate is the specification P. the input i is the tuple of arguments (e, l), the output o is *result*, and the function f is *sortedIns*. The overall correctness condition $P((e, l), f(e, l))$ is therefore:

$$\mathit{isSorted(l)} \rightarrow (\mathit{isSorted(sortedIns(e,l))} \wedge \mathit{content(sortedIns(e,l))} = \mathit{content(l)} \cup \{e\}) \quad (1)$$

Runtime verification. Runtime verification checks the above correctness condition when the values of arguments (in our example, e and l) are known. In addition, before invoking the function *isSorted*, the program system checks its precondition, so the function body only executes when the assumption of the above implication is true. In general, we assume that both f and P are expressed in a language of computable recursive functions. Therefore, the applications of f and P to their arguments are simply computations according to the semantics of the language. As a result, runtime verification RV is a computable problem in our context. In fact, the ***require*** and ***ensuring*** are simply library functions of the Scala programming language [58], and runtime checking in principle requires no further support. Runtime checking is nonetheless not a trivial problem, because the specification P is often written aiming for clarity and provability, and not aiming for efficiency of evaluation. A naturally written specification often leads to naive and repeated computation if executed at runtime. Leon can substantially improve the predictability and performance of runtime checks using static techniques, as we discuss below.

```
def isSorted(l: List): Boolean = l match {
  case Cons(x, Cons(y, ys)) ⇒ x ≤ y && isSorted(Cons(y, ys))
  case _ ⇒ true }
```

Listing 4. Sortedness Property of a List

```
def content(l: List): Set[Int] = l match {
  case Cons(x, xs) ⇒ Set(x) ++ content(xs)
  case Nil ⇒ Set() }
```

Listing 5. Set of Elements Stored in a List

Static verification. Whereas runtime verification checks (1) for the particular e, l, static verification attempts to prove it for all e, l. The Leon verifier takes the same input as for runtime checking, and attempts to either prove correctness of (1) or find a counterexample for its correctness. For this purpose, we build on the field of Satisfiability Modulo Theory (SMT) solvers [2,19,22], which contain decision procedures for many theories that support useful combinations of operations present in programs. For property (1), the unfolding of *isSorted* requires reasoning about algebraic data types, handled by the theory of recursive data types [3] and reasoning about the ordering on integers, handled by integer linear arithmetic on integers within Z3 [36]. Reasoning about operations on sets is handled using array combinators [20] although more expressive theories of sets with cardinalities were evaluated in previous versions of Leon [76]. A careful reader will observe that a specification using sets may be weaker than desired, because it allows a sorting routing to remove or introduce duplicates. A more precise specification can naturally be written using multisets, whose simple fragments can be also encoded using arrays combinators [20], and for which the development of decision procedures of optimal complexity is a result of relatively recent developments [62–64].

In contrast to operators built into Leon's language subset, recursively defined functions are not directly supported in SMT solvers, so Leon implements its own algorithm to handle them [73–75]. Leon's algorithm [75] is related to bounded model checking ideas [1,7] and k-induction [23,37], but applies to recursive functions. In our example, when proving correctness of the condition (1), the system performs satisfiability checks while increasingly unfolding the definitions. For a relatively small unfolding depth, in this example the formula becomes unsatisfiable. In other examples, the system finds a counterexample for certain unfolding depth. In general, it need not terminate because the problem is undecidable.

However, there are interesting classes of recursive functions for which the system is a decision procedure [73,74]. The class that we have considered have the form of homomorphism functions, such as the *content* function in Listing 5. We have identified a number of such functions, which we call "sufficiently surjective abstracts" in [74], thus deriving several families of extensions of term algebras for which satisfiability of quantifier-free formulas is decidable.

The work can also be viewed [34] from the point of view of Ψ-local theory extensions [33], where further decidable extensions of term algebras have been identified [68].

In general, particular recursive functions may require reasoning specialized to this class. What is remarkable is that for a class of sufficiently surjective abstraction functions, the unrolling (a form of bounded model checking for recursive functions) becomes a uniform decision procedure [73]. Therefore, we have a series of decidability results, but the underlying algorithm need not be aware of them, they simply ensure its completeness in some cases. Our experience suggests that the algorithm works in practice for many other cases as well, which suggests that there may be further completeness results to be discovered.

Function unfolding is justified for terminating functions, and corresponds to inductive reasoning according to the well-founded relation that implies termination of functions. Leon currently does not check termination by default; for some approaches to check termination, see [13, 65]. Sufficiently surjective abstraction functions are, however, terminating by their syntactic structure. Note also that the properties that we are checking are safety properties, which means that we could generate logical encoding that uses relations instead functions and is not sensitive to termination. It remains to be investigated how much we would lose in practice by using relations instead of functions.

Inductive generalization. In our experience, more properties turned out to be k-inductive than what we initially expected. The general-purpose algorithm of Leon, based on function unfolding, is therefore surprisingly effective in practice. For some cases, though, it fails to perform the required generalization and find an invariant that implies the desired property. To address these cases, we have started incorporating some of the ideas from model checking and constraint-based static analysis.

One approach is to search for invariants of a particular template form [6]. We have recently also implemented a refinement of such an approach in Leon and showed that it is able to compute worst-case execution bounds for sequential and parallel execution of functional programs [52], which often involve difficult-to-find numerical constants.

An alternative approach is to generalize predicate abstraction to recursive functions. State of the art methods use counterexample-guided refinement of the set of predicates, often based on interpolation. We have applied this approach to linear integer arithmetic models of programs [67]. The technique requires tree interpolants [29, 57], which generalizes interpolation problem to more complex cases of proving consistency of Horn clauses [8, 25, 66].

In both of these approaches to verification with inductive invariant inference, we were greatly influenced by the works of Andrey Rybalchenko.

Why an executable specification language. The choice of executable predicates for specifications (as opposed to, for example, logic with quantifiers or dynamic logic) is somewhat restrictive but very practical. First, the class of properties is rather large, because we are allowed to use a Turing-complete language for predicates. Second, it is not an obscure language that happens to be Turing complete, but a functional language that is already used for implementations, and which many schools teach to undergraduates. Whereas software developers may be hesitant to use logical notation, here they just use assertions. Third, it is an executable language, which immediately enables runtime checking in program runs and test runs, as discussed above. It also leads to automated generate-and test approaches to bug finding, which are as complete as theoretically possible, given that the problem of finding counterexamples is recursively enumerable. The computable specifications approach is also related to reasons why bounded model checking is effective for such specifications. Finally, as we have seen, the absence of counterexamples can also be automated through inductive reasoning. For many of these reasons this has been a popular choice in other verification

tools as well, most notably the ACL2 prover and its predecessors [12, 38, 39]. That said, given numerous other heavier-weight specification and verification approaches, we feel that the elegance and the advantages of the approach of using executable functions can never be emphasized enough.

Runtime constraints solving (constraint programming). Using the abstraction function and the invariant, we can concisely specify an *insert* operation for sorted lists using a constraint as in Listing 6.

```
def insert(l: List, v: Int) = {
  require(isSorted(l))
  choose{ (x: List) ⇒ isSorted(x) && (content(x) == content(l) ++ Set(v)) }
}
```

Listing 6. Insertion Specified using Constraint

Runtime constraint solving allows the developer to describe computations using predicates alone, avoiding the need to write an explicit function from inputs to inputs. In other words, they allow programming with "implicit" functions.

Observe that, given f we can define P that characterizes it by defining $P(x, y) \iff (f(x) = y)$. Therefore, input/output specifications (which are relations) subsume implementations (which we consider to be functions). They are more expressive because they can describe the desired properties of the output, without specifying it uniquely. This allows us to specify orthogonal properties separately and then combine them using conjunction to obtain a function as an intersection of several relations.

The advantages of specifications become even more apparent for more complex examples. The following method describes the insertion into a red-black tree.[1]

```
def insert(t: Tree, v: Int) = {
  require(isRedBlack(t))
  choose{ (x: Tree) ⇒ isRedBlack(x) && (content(x) == content(t) ++ Set(v)) }
}
```

In such scenario, the run-time waits until the argument t and the value v are known, and finds a new tree value x such that the constraint holds. Thanks to our constraint solver, which has a support recursive functions and also leverages the Z3 SMT solver, this approach works well for small red-black trees. It is therefore extremely useful for prototyping and testing and we have previously explored it as a stand-alone technique for constraint programming in Scala [43].

If we now considered writing a removal operation for trees, an approach based on conventional imperative or functional code would require writing a separate

[1] We omit here the definition of the tree invariant for brevity, which is non-trivial [14, 61], but still rather natural to describe using recursive functions.

removal algorithm, which is non-trivial for red-black trees. Using specifications, the desired behavior is given simply by replacing ++ sign with -- sign:

```
def remove(t: Tree, v: Int) = {
  require(isRedBlack(t))
  choose{ (x: Tree) ⇒ isRedBlack(x) && (content(x) == content(t) -- Set(v)) }
}
```

Static constraint solving (synthesis). Analogously to verification, we would like to obtain efficiency and predictability advantages of static computation also in the case of implicitly defined computations. For this purpose, we aim to statically solve specifications and convert them into directly executable functions. This process is typically referred to as program synthesis [46, 54]. The synthesis techniques in Leon [40] heavily rely on the underlying verification techniques, but also on complete functional synthesis [35, 46–48]. Our current implementation of synthesis in Leon [40] is able to translate the specification into the complete implementation shown in Listing 7.

```
def insert(l: List, v: Int) = {
  require(isSorted(l))
  l match {
    case Cons(head, tail) ⇒
      if (v == head) {
        l
      } elseif (v < head) {
        Cons(v, l)
      } else {
        insert(t, v)
      }
    case Nil ⇒
      Cons(v, Nil)
  }
}
```

Listing 7. Result of Synthesis of Code Shown in Listing 6

Theoretical questions of completeness for synthesis have interesting connection to logic and automata. For example, synthesis for Presburger arithmetic turns out to be related to constructive quantifier elimination [46–48]. On the other hand, we can obtain better theoretical bounds for synthesized code using automata techniques [28, 70]. Parameterized complexity of problems is likely to be important when theoretically characterizing when synthesis is useful, because we wish to consider the specification P and the input i as two distinct input parameters.

Our implemented technique [40] also builds on counterexample-guided approaches [69]. Techniques for learning representations in logic and automata [51] are a likely to have further fruitful applications in software synthesis.

In addition to synthesis over discrete domains, an important problem is synthesis of numerical computations that conform to the desired precision guarantees [17, 18].

2.2 Relationship Between Different Problems

This classification gives an overview of typical different tasks, though some of the most interesting questions arise by considering combinations and relationships between these four problems.

Optimizing runtime checks using static verification. The Leon verifier can remove the runtime checks that are provable statically, and transform the programs to avoid duplicate checks. This allows automatic static verification for as many properties as possible, but still allows the resulting programs to run and to detect if any leftover checks fail for the values arising during program use and testing. This is part of an ongoing work with Emmanouil Koukoutos at EPFL.

Invoking static verification at runtime for complex programs. When program state is complex, static verification techniques face their limitations. It is therefore interesting to invoke static analyzers at runtime, in parallel with actual program executions. We have done this in the context of distributed system implementations [80] and Java programs (EPFL MSc thesis of Sebastian Gfeller), and it could also be done to perform eager checks of higher-order function contracts in functional programs.

Using counterexamples of static verification for constraint solving. A very important connection shows that runtime constraint solving is a dual to static verification. We perform static verification in Leon, showing validity of $\forall i.P(i, f(i))$, by proving unsatisfiability of $C(x)$ defined as $\neg P(x, f(x))$. A satisfying assignment for $C(x)$ is a counterexample to the validity. Counterexample search is thus search for values that satisfy a constraint $C(x)$, expressed in logics that contain operations from theories, as well as recursive function invocations. On the other hand, runtime constraint solving tries to find, for a given i_1 a value o such that $P(i_1, o)$ holds. If we define $C'(x)$ as $P(i_1, x)$, then runtime constraint solving also corresponds to solving the constraints $C'(x)$ expressed in the same language as before. Therefore, in our work on constraint programming for Scala, we were able to use the same constraint solving implementation to solve constraints [43]. This mechanism is also avaible in the current Leon system [45], and allows us to execute very expressive specifications between inputs and outputs.

Combining constraint solving and synthesis. Our deductive synthesis framework performs a search over different steps that transform specification into a set of potentially simpler ones. This architecture allows us to combine synthesis and run-time constraint solving. We illustrate this using an example

of a *red-black tree with a cache*. Such a tree contains a red-black tree, but also redundantly stores one of its elements.

```
case class CTree(cache: Int, data: Tree)
```

The specification of the invariant *inv* formalizes the desired property: the cache value must be contained in the tree unless the tree is empty.

```
def inv(ct: CTree) = {
  isRedBlack(ct.data) &&
  (ct.cache ∈ content(ct.data)) || (ct.data == Empty)
}
```

The *contains* operation tests membership in the tree.

```
def contains(ct: CTree, v: Int): Boolean = {
  require(inv(ct))
  choose{ (x: Boolean) ⇒ x == (v ∈ content(ct)) }
}
```

While not being able to fully synthesize it, the deductive synthesis procedure decomposes the problem and partially synthesizes the constraint. One of its possible results is the following *partial* implementation that combines actual code and a sub-constraint:

```
def contains(ct: CTree, v: Int): Boolean = ct.data match {
  case n: Node ⇒
    if (ct.cache == v) {
      true
    } else {
      choose { (x: Boolean) ⇒ x == (v ∈ content(n)) }
    }
  case Empty ⇒
    false
}
```

We notice that this partial implementation makes use of the cache in accordance with the invariant. The code accurately reflects the fact that the cache may not be trusted if the tree is empty. The remaining constraint is in fact a simpler problem that only relates to standard red-black trees. Our system can then compile the resulting code, where the fast path is compiled as the usual Scala code, and the *choose* construct is compiled using the run-time solving approach.

Using synthesis to verify existential statements. Note that constraint solving by itself does not have static guarantees that the synthesized value can be produced. By successfully solving a synthesis problem, the system establishes the truth of an existentially quantified statement. Therefore, such synthesis capability can be used to prove quantified statements. Here we do not mean to imply that constructive interpretation of quantifiers is the only one possible, nor that it should be built into the semantics. We merely observe that synthesis is an interesting method for proving existential statements containing, for example

recursive functions. It is interesting to note that the authors of classical deductive synthesis [54,55] concluded that better inductive theorem proving is needed to enable synthesis of recursive programs. Given recent advances in software synthesis including ours and others [11, 72], it is interesting to re-examine the use of synthesis algorithms for theorem proving.

3 Towards Case Studies as Mathematical Statements

One challenge in this research is that software manipulates inputs from very large or infinite domains, with rich algebraic structure. Examples include numerical domains such as integers or approximations of real numbers, as well symbolic domains, such as algebraic data types, sequences, sets, multisets, and maps. To handle this complexity, it is essential to further advance the field of Satisfiability Modulo Theories, including the development of new decision procedures for structures such as multisets and algebraic data types. Moreover, the applications in embedded and cyber-physical systems call for systematic support for reasoning about approximations of real numbers, something that we have recently started exploring as well [17], and that connects the area of verification to numerical analysis and to decision procedures for theories of real numbers [21].

Another challenge is that programs have complex control and language structure, including conditionals, recursion, higher-order functions, dynamic dispatch, and concurrency. Much of this complexity can be handled by translation into first-order side-effect-free functional or logic programming language. This supports the use of standardized formats for software analysis and program synthesis problems using either Horn clauses in the language of SMT-LIB theories [8], or pure subsets of popular programming languages such as Scala.

The format of Horn clauses has a particularly promising future as a way of taming the complexity of programming languages as well as the methodological complexity of precise and automated verification algorithms, as witnessed by a number of successful approaches in this direction. Apart from those mentioned already, exciting new work includes addressing more complex quantification patterns that go beyond verification of safety properties [4, 5, 9].

When Horn clauses are expressed in SMT-LIB2 format (`http://www.smt-lib.org`), rich theories present in the format, combined with the ability to encode complex control flow, result in a versatile format for precisely stating mathematical problems about software. Building tools that handle such benchmarks requires new theoretical insights as well as important software development and experimental work.

Acknowledgements. The authors would like to thank his group members, his collaborators. The author also thanks researchers gathered around the EU COST Action "Rich Model Toolkit" (`http://richmodels.epfl.ch`), 2009-2013.

References

1. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *SPIN*, pages 146–162, 2006.

2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
3. C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. *Electronic Notes in Theoretical Computer Science*, 174(8):23–37, 2007.
4. T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234, 2014.
5. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, pages 869–882, 2013.
6. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, pages 378–394, 2007.
7. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *TACAS*, pages 193–207, 1999.
8. N. Bjørner, K. L. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT@IJCAR*, pages 3–11, 2012.
9. N. Bjørner, K. L. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS*, pages 105–125, 2013.
10. 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.
11. R. Bodík. Algorithmic program synthesis with partial programs and decision procedures. In *SAS*, page 1, 2009.
12. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1975.
13. M. Codish, J. Giesl, P. Schneider-Kamp, and R. Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *J. Autom. Reasoning*, 49(1):53–93, 2012.
14. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.
15. E. Darulová and V. Kuncak. Trustworthy numerical computation in scala. In *OOPSLA*, 2011.
16. E. Darulova and V. Kuncak. Certifying solutions for numerical constraints. In *Runtime Verification (RV)*, 2012.
17. E. Darulova and V. Kuncak. Sound compilation for reals. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2014.
18. E. Darulova, V. Kuncak, R. Majumdar, and I. Saha. Synthesis of fixed-point programs. In *Embedded Software (EMSOFT)*, 2013.
19. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
20. L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design*, Nov. 2009.
21. L. M. de Moura and G. O. Passmore. Computation in real closed infinitesimal and transcendental extensions of the rationals. In *CADE*, pages 178–192, 2013.
22. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
23. A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *SAS*, pages 351–368, 2011.
24. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering (ICSE)*, 2010.

25. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.
26. T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, 2013.
27. T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *Computer Aided Verification (CAV) Tool Demo*, 2011.
28. J. Hamza, B. Jobstmann, and V. Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, 2010.
29. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.
30. W. Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.
31. H. Hojjat, R. Iosif, F. Konečný, V. Kuncak, and P. Rümmer. Accelerating interpolants. In *Automated Technology for Verification and Analysis (ATVA)*, 2012.
32. H. Hojjat, F. Konecny, F. Garnier, R. Iosif, V. Kuncak, and P. Ruemmer. A verification toolkit for numerical transition systems (tool paper). In *16th International Symposium on Formal Methods (FM)*. Springer, 2012.
33. C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *TACAS*, pages 265–281, 2008.
34. S. Jacobs and V. Kuncak. Towards complete reasoning about axiomatic specifications. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.
35. S. Jacobs, V. Kuncak, and P. Suter. Reductions for synthesis procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.
36. D. Jovanovic and L. M. de Moura. Cutting to the chase - solving linear integer arithmetic. *J. Autom. Reasoning*, 51(1):79–108, 2013.
37. T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. In *PDMC*, pages 55–62, 2011.
38. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.
39. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
40. E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.
41. E. Kneuss, V. Kuncak, and P. Suter. Effect analysis for programs with callbacks. In *Fifth Working Conference on Verified Software: Theories, Tools and Experiments*, 2013.
42. E. Kneuss, P. Suter, and V. Kuncak. Runtime instrumentation for precise flow-sensitive type analysis. In *International Conference on Runtime Verification*, 2010.
43. A. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.
44. V. Kuncak and R. Blanc. Interpolation for synthesis on unbounded domains. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013.
45. V. Kuncak, E. Kneuss, and P. Suter. Executing specifications using synthesis and constraint solving (invited talk). In *Runtime Verification (RV)*, 2013.
46. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2010.
47. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Communications of the ACM*, 2012.

48. 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.
49. V. Kuncak, R. Piskac, and P. Suter. Ordered sets in the calculus of data structures (invited paper). In *CSL*, 2010.
50. V. Kuncak, R. Piskac, P. Suter, and T. Wies. Building a calculus of data structures (invited paper). In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2010.
51. A. Lemay, S. Maneth, and J. Niehren. A learning algorithm for top-down xml transformations. In *PODS*, pages 285–296, 2010.
52. R. Madhavan and V. Kuncak. Symbolic resource bound inference for functional programs. In *Computer Aided Verification (CAV)*, 2014.
53. A. I. Mal'cev. Axiomatizable classes of locally free algebras of various types. In *The Metamathematics of Algebraic Systems*. North-Holland, 1971. (Translation, original in Doklady, 1961).
54. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
55. Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
56. M. Mayer and V. Kuncak. Game programming by demonstration. In *SPLASH Onward!*, 2013.
57. K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, Jan. 2013.
58. M. Odersky. Contracts for Scala. In *Int. Conf. Runtime Verification*, 2010.
59. M. Odersky and T. Rompf. Unifying functional and object-oriented programming with Scala. *Commun. ACM*, 57(4):76–86, 2014.
60. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.
61. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
62. R. Piskac and V. Kuncak. Fractional collections with cardinality bounds, and mixed integer linear arithmetic with stars. In *Computer Science Logic (CSL)*, 2008.
63. R. Piskac and V. Kuncak. Linear arithmetic with stars. In *Computed-Aided Verification (CAV)*, volume 5123 of *LNCS*, 2008.
64. R. Piskac and V. Kuncak. Munch - automated reasoner for sets and multisets (system description). In *IJCAR*, 2010.
65. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.
66. P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and solving horn clauses for verification. In *Fifth Working Conference on Verified Software: Theories, Tools and Experiments*, 2013.
67. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *Computer Aided Verification (CAV)*, 2013.
68. V. Sofronie-Stokkermans. Locality results for certain extensions of theories with bridging functions. In *CADE*, pages 67–83, 2009.
69. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
70. A. Spielmann and V. Kuncak. Synthesis for unbounded bitvector arithmetic. In *International Joint Conference on Automated Reasoning (IJCAR)*, LNAI. Springer, 2012.

71. A. Spielmann, A. Nötzli, C. Koch, V. Kuncak, and Y. Klonatos. Automatic synthesis of out-of-core algorithms. In *SIGMOD*, 2013.
72. S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.
73. P. Suter. *Programming with Specifications*. PhD thesis, EPFL, December 2012.
74. 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.
75. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis Symposium (SAS)*, 2011.
76. P. Suter, R. Steiger, and V. Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.
77. T. Wies, M. Muñiz, and V. Kuncak. An efficient decision procedure for imperative tree data structures. In *Computer-Aideded Deduction (CADE)*, 2011.
78. T. Wies, M. M. niz, and V. Kuncak. Deciding functional lists with sublist sets. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, LNCS, 2012.
79. T. Wies, R. Piskac, and V. Kuncak. Combining theories with shared set operations. In *FroCoS: Frontiers in Combining Systems*, 2009.
80. M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM Transactions on Computer Systems*, 28(1), 2010.
81. K. Yessenov, R. Piskac, and V. Kuncak. Collections, cardinalities, and relations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2010.