Sound Reasoning about Integral Data Types with a Reusable SMT Solver Interface

Régis Blanc Viktor Kuncak

June 14, 2015

Abstract

We extend the Leon verification system for Scala with support for bit-vector reasoning, thus addressing one of its fundamental soundness limitation with respect to the treatment of integers primitives. We leverage significant progresses recently achieved in SMT solving by developing a solver-independent interface to easily configure the back-end of Leon. Our interface is based on the emerging SMT-LIB standard for SMT solvers, and we release a Scala library offering full support for the latest version of the standard.

We use the standard BigInt Scala library to represent mathematical integers, whereas we correctly model Int as 32-bit integers. We ensure safety of arithmetic by checking for division by zero and correctly modeling division and modulo. We conclude with a performance comparison between the sound representation of Ints and the cleaner abstract representation using mathematical integers, and discuss the trade-off involved.

1 Introduction

Leon is a verification and synthesis system for a purely functional subset of Scala. Leon's input language is Turing-complete thanks to the expressivity of recursive functions. Leon verifier works by mapping Scala expressions into logical formulas, which are typically quantifier-free. Leon attempts to prove postconditions of functions expressed using the `ensuring` constructs of Scala, as well as preconditions of functions and completeness of pattern matching. Leon searches for inputs that violate the verification conditions using SMT solvers [2, 10]. In addition to mapping the semantics of a subset of Scala to logic, Leon implements a lazy unrolling algorithm to deal with recursive functions, by alternating between an abstraction using uninterpreted functions and adding assumptions to only consider fully unrolled paths [19, 3].

Previous state. Previously, Leon mapped Scala's `Int` data type into the mathematical integers of the underlying theorem prover. One could argue that the mathematical integers correspond to many typical uses of integer type, and are appropriate for a high-level language. In languages such as Haskell, the convenient to use integer type, `Int`, denotes unbounded integers, so mapping as used in Leon up to recently would be correct. Having access to the type of mathematical integers indeed enables the construction of programs through well-behaved components, making programs easier to understand and reason about. However, to model or to specify code that performs bit manipulation or code that has strong requirements on performance and memory use, a developer may wish to use bounded integers, which are the most common choice in Scala and Java. In any case, a verification tool should provide meaningful guarantees, so it is essential that the verification semantics conforms to the runtime semantics.

As an illustration of the semantics differences between mathematical and 32-bit integers, consider the infamous implementation of the binary search algorithm used in the java JDK[1]. This standard algorithm is present in most algorithm textbooks and was even proven correct mathematically. However, a naive proof of correctness would assume that the following statement:

```
val mid = (low + high)/2
```

returns a number in the interval of `low` and `high`. This implicitly assumes integer semantics, as such a property does not hold with bit-vector arithmetic:

```
scala> (1000000000 + 2000000000)/2
res1: Int = -647483648
```

Because such index overflows occur only for very large data sets, it may take a very long time to detect in the field that this implementation of a binary search algorithm is not correct. A program verifier using mathematical integers to represent native integers would be fooled in the exact same way as mathematicians believing they proved the correctness of that algorithm.

Contributions. We present a modification of Leon that distinguishes between the mathematical integers, modeled as `BigInt`, and the lower level concept of bit-vectors, modeled as `Int`. We show how reasoning about bit-vector formulas interacts with the core Leon algorithm. We discuss the treatment or the integer division, whose definition in Scala and Java differs slightly from the accepted one in mathematics, and the addition of new verification conditions to detect division by zero statically. Together, these techniques give flexibility to the developers, while maintaining sound answers from a verification tool. We also release a Scala library to work with the SMT-LIB standardized solvers. We used the library in our implementation, but it is of general interest for Scala applications that need portable SMT solving functionality.

Related Work. This paper extends the Leon verification system for Scala [19, 18, 3]. Leon attempts to model the semantics of a subset of Scala precisely and soundly. The original system supported a first-order functional subset [19,

[1]`http://googleresearch.blogspot.ch/2006/06/extra-extra-read-all-about-it-nearly.html`

18]; additional support for imperative programming [3] and higher-order functions [20] were added in later works. There are other extensions of Leon that leverage the core solving algorithm for program synthesis [14], program repair [13], constraint solving [15], as well as extensions for reasoning about floating points and roundoff errors relative to real-valued semantics [7, 6].

Isabelle [17] and ACL2 [12] are interactive theorem provers that enable users to elaborate very expressive and complex proofs about programs. They provide their own language, although ACL2 is very close to Common Lisp, that distinguishes between integers and bit-vectors. Leon is more automated in typical use, but less expressive in general. These systems typically have their own internal rule systems to correctly model their language, including bit-vector arithmetic.

Verifun [21] attempts to automatically prove properties on a small functional programming language. It is a hybrid system that involves the user when the proof fails to complete. Their functionnal language only provides natural numbers as numerical types. Dafny [16], similarly to Leon, is fully automated and relies on SMT solvers. Dafny supports a `int` type that represents mathematical integers, as well as `nat` that supports natural numbers (which, of course, include 0), but there appears to be no support for bit-vectors at the time of writing. Why3 [9] is another mostly automated verification tool based on SMT solvers. Much like our solution, it provides two different types for integers and bit-vectors.

SBV is a Haskell package[2] that relies on SMT solvers to solve properties about Haskell program. SBV supports multiple SMT solvers. Compared to Leon, SBV is a lighter abstraction over SMT solvers, using Haskell as a front-end to SMT solvers. SBV does not implement an independent algorithm to handle recursive functions.

2 Sound Reasoning about Integers

In this section, we build on previous work on Leon and Satisfiability Modulo Theory (SMT) solvers in order to propose a sound and efficient automated reasoning on programs involving primitive types and recursive functions.

We consider a very simple functional subset of Scala, one supporting the two types `Int` and `Boolean` and a list of functions in a top level `object` definition. The functions can be mutually recursive. We build expressions with a combination of conditional expressions and standard integer and boolean operators.

Such a language is Turing-complete, and capable enough to write interesting functions, such as:

```
def factorial(n: Int): Int = {
  require(n >= 0)
  if(n == 0) 1 else n * factorial(n − 1)
} ensuring(res ⇒ res >= 0)
```

[2]`https://hackage.haskell.org/package/sbv`

This function is defined recursively. We use `ensuring` to state properties that should hold for the output of a function given any input that passes a precondition (expressed with `require`).

Leon encodes the above implementation into an equivalent logical formula, as a set of clauses:

- $r = f(n)$
- $n = 0 \implies r = 1$
- $n \neq 0 \implies r = n \cdot f(n-1)$

The last clause corresponds to an unrolling step, and Leon uses an induction hypothesis to add the clause $f(n-1) \geq 0$ to the set. We are trying to prove the following property:

$$n \geq 0 \implies f(n) \geq 0$$

We can carry the proof manually with a simple case analysis. If $n = 0$, then $r = 1 \geq 0$, and if $n > 0$, then $n \neq 0$ and $r = n \cdot f(n-1) \geq 0$ because both n and $f(n-1)$ are positive.

This simple proof is done automatically in Leon by dispatching it to an SMT solver. The SMT solver can check for satisfiability of the conjunction of clauses with the negation of the property:

$$n \geq 0 \wedge f(n) < 0$$

A model would represent a counter-example. In the present example, the conjunction is unsatisfiable as the previous proof shows.

Notice how, in the proof, the exact meaning of the function f does not matter. Leon actually models such function with the theory of unintepreted functions. The formula is still unsatisfiable even without constraining the value of f. The only constraints are from the concrete unrolling steps, introducing a constraint for its value at $n - 1$. This abstraction means that Leon cannot trust a counter-example to the set of clauses as a concrete counter-example to the property. A counter-example can only be trusted if it does not uses uninterpeted parts of the functions. Leon keeps unrolling the program by introducing more and more definitions of functions, until either a concrete counter-example is found or an unsatisfiable set of clauses is derived.

Leon relies on a satisfiability modulo theories (SMT) solver. In the above program, we need a solver supporting the theories of integers, uninterpreted functions and propositional logic.

Unfortunately there is a semantic gap between Scala and pure mathematics. Scala defines primitive integers as machine integers, with only a finite range, so Scala's `Int` is really a `BitVector` of size 32. It does not take much for a function such as factorial to diverge from its mathematical definition. While $13! = 6,227,020,800$, running the above implementation on 13 will give:

```
scala> factorial(13)
res0: Int = 1932053504
```

which significantly differs from the correct mathematical definition. Although the above actually verifies the postcondition of being positive, `factorial(17)` returns a *negative* number, violating the postcondition and throwing a runtime exception if contracts are checked dynamically.

This poses the question whether Leon should follow the natural mathematical meaning of the code, or adhere to the exact Scala semantics. We argue for the latter. Matching Scala semantics would enable the use of Leon in real systems—those concerned with actually delivering working applications. In addition, nothing is lost because there is a Scala type, `BigInt`, whose semantics closely matches the one of mathematical integers. Efficiency concerns put aside, programmers should be using `Int` when they expect bit-vector semantics and `BigInt` when true mathematical integers are expected. This helps the program carry more information on its intent, and gives static analysis tools a better understanding of the properties.

The proof in our example does not extend to bit-vectors. The problematic step is assuming the product of two positive numbers is always positive. This property does not translate from integers arithmetic to bit-vector arithmetic because of overflows. Many important properties of integers are not verified by bit-vectors. This lack of mathematical properties complicates the task of theorem prover for a formula over bit-vectors, when compared to the same formula over integers. However, SMT solvers with the backgrond theory of bit-vectors are still reasonably fast [11, 5, 8, 1]. Additionally, due to the finite nature of the domain, there are some problems that are easier to solve when considered over bit-vector arithmetics. We discuss and compare performance of the two approaches in Section 4.

We follow the same technique as with integers when generating verification conditions. We generate the set of clauses corresponding to the implementation of the function. We then attempt to prove unsatisfiability of the negation of the property as before. This time, however, we interpret constants and operators over the domain of bit-vectors instead of integers. We find a concrete counter-example for $f(17) < 0$ and report a bug to the user.

2.1 Semantic of Division

There are many possible definitions for the integer division [4]. In mathematics, one usually defines the division of integers x and y as the quotient q and remainder r such that $q \cdot y + r = x$ and $0 \leq r < |y|$. This is known as the Euclidean definition [4] and is the definition used by the SMT-LIB standard and thus supported by SMT solvers. The Scala programming languages, following Java, defines integer division as "rounding towards zero", which differs from the Euclidean definition. In particular, the remainder — the value returned by the `%` operator — is sometimes negative. This definition is used both for the primitive `Int` type and the math class `BigInt`.

Leon interprets `BigInt` as mathematical integers. We ensure that Leon supports integer division according to Scala behavior by encoding Scala semantics of division using the Euclidean definition. Mathematically we define the result

q, and denote the Euclidean division of x and y as $\frac{x}{y}$. A direct encoding of the Scala division as a case split is as follows:

- $x \geq 0 \wedge y > 0 \implies q = \frac{x}{y}$
- $x \geq 0 \wedge y < 0 \implies q = \frac{x}{y}$
- $x < 0 \wedge y > 0 \implies q = -\frac{-x}{y}$
- $x < 0 \wedge y < 0 \implies q = \frac{-x}{-y}$

When expressed in SMT-LIB, this encoding uses the `ITE` operator to do the case splitting for the different possible signs of the operands. This results in a relatively complex term with nested conditional expressions to express a simple division operation. The only solution to avoid such a heavy encoding would be for the mathematical meaning of division (of SMT solvers) and the programming language meaning (of Scala) to match. As an optimization, we can actually group the branches with positive x and rewrite the last branch and we obtain the following expression for q:

$$\text{if } x \geq 0 \text{ then } \frac{x}{y} \text{ else } -\frac{-x}{y}$$

We are using the latter one in our implementation, though the presence of a branching condition in the middle of an arithmetic expression is still potentially costly for the solver.

The encoding of the modulo operator is based on the result of the division operator, ensuring the correct relation between the quotient and remainder:

$$r = x - y \cdot q$$

So far we discussed the semantic of the pure mathematical integers. The theory of bit-vectors comes with its own `bvsdiv` and `bvsrem` operators with distinct definitions from the corresonding operators on integers. It always performs the unsigned division on corresponding absolute values. The remainder is then defined to be consistent with the quotient of the division. This definition actually matches the definition of Scala for primitive `Int` and allows us to use a straightforward encoding of Scala division expressions into bit-vectors.

We also added support to Leon for preventing division by zero. For each division expression over integers, or bit-vector, Leon verifies that the divisor is never zero in any possible execution of the program. Leon processes such checks in the same way it handles postcondition to a function, finding a counterexample if the condition does not hold.

3 Implementation

We integrated the above techniques into the Leon system. We show an overview of the architecture of Leon in Figure 1. First, Leon runs the input program

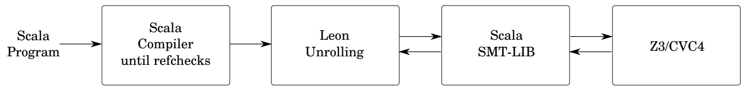

Figure 1: Architecture of Leon.

Table 1: Comparing performence of verification using bit-vectors (BV) and integers.

Solver	**Z3**		**CVC4**	
Bench.	BV	Integer	BV	Integer
List Ops.	1.167	1.088	2.025	2.053
Insert. Sort	0.851	0.702	1.215	0.978
Merge Sort	0.821	0.269	N.A.	N.A.
Sorted List	1.088	1.152	1.751	1.717
Red-Black Tree	6.254	3.743	6.755	6.512
Amort. Queue	4.477	3.225	7.011	6.384
AVL Tree	3.494	2.836	8.146	7.103

through the first part of the standard Scala 2.11.* compiler pipeline, until phase `refchecks`. We extract the resulting tree and translate it into an internal representation used by Leon. We derive verification conditions and attempt to prove them.

The unrolling works as described in the previous section. It generates clauses and then sends them to the SMT-LIB module. This module handles the communication with a native SMT solver: Z3 or CVC4 in the current version of Leon. For better performances, that communication must be incremental because Leon is refining the formulas more and more based on the feedback from the solver.

3.1 Reusable SMT Solver Interface

This section presents how we use the SMT-LIB interface to make Leon solver-agnostic. Based on that interface, we added support for CVC4 [1] and are now officially supporting it as well as Z3 [8]. Being independent of the solver is particularly important as designing efficient decision procedures for theories used in programming languages are still a research topic and is evolving at a very fast pace.

Leon embraces the SMT-LIB standard for communicating with SMT solvers [2]. Many state-of-the-art solvers, including Z3 and CVC4, implement a robust support for that standard. SMT-LIB version 2 provides a scripting language to communicate with SMT solvers. This scripting language supports, in particular, a notion of stack of assertions that enable incremental solving if the underlying solver supports it properly.

The solving backend of Leon is an abstraction over SMT-LIB, which essentially defines a transformation from the Leon representation of Scala programs

Table 2: Evaluation of programs using bit-vectors, showing the numbers of valid (V), invalid (I), and unknown (U) verification conditions and the total time for the benchmark in seconds.

Solver	**Z3**				**CVC4**			
Benchmark	V	I	U	T. (s)	V	I	U	T. (s)
Bin. Search	0	1	0	0.32	0	1	0	0.11
Bit Tricks	24	0	3	0.03	25	0	2	0.08
Identities	4	1	0	4.89	4	1	0	3.68

to a first-order logic representation of programs. It performs unrolling of recursive functions in a lazy manner, asserting more and more clauses to the solver.

We developed and released an open-source Scala library, `scala-smtlib`, that provides a nearly complete support for the upcoming 2.5 version of the standard. The library is open-source and available on GitHub[3] as a separate package on which Leon depends.

`scala-smtlib` is a lightweight interface on top of the SMT-LIB standard that exposes a tree representation mirroring the abstract grammar of the SMT-LIB language. At its core, the API offers a `Parser` that transforms an input stream into the tree representation, and a `Printer` that transforms a tree into a SMT-LIB complient textual output. Building on that abstraction, `scala-smtlib` wraps solver processes as an interpreter for SMT-LIB scripts. This gives Scala programmers access to a type-safe interface to an SMT solver. The wrapper handles low level communication with an external process, communicating over the textual standard input and output. The library comes with two implementations of that wrapper for Z3 and CVC4, but very little solver-specific code is required to add additional wrappers.

We refer to the online repository for more extensive documentation on the library.

4 Experiments

With the change introduced in the present work, previous benchmarks using `Int` as a data type are rewritten as benchmarks that use `BigInt`, capturing our original intent behind those benchmarks. We now additionally consider the `Int` benchmarks, but now correctly interpreted using 32-bit integers. Certain specification-only functions, such as `size`, still use `BigInt`, which suits their purpose in specification and allows us to prove basic properties such as that size is non-negative. We ran a set of experiments to evaluate the difference in verification performance between these two versions of benchmarks. The extensions presented in this paper are available on the official version of Leon[4]. A snapshot of Leon containing all the benchmarks reported here, is available on the `submission/scala-2015-bv` branch of the official Leon repository.

[3]`https://github.com/regb/scala-smtlib`
[4]`https://github.com/epfl-lara/leon`

Table 1 compares the performence of bit-vectors and mathematical integers on a few different benchmarks. The experiments have been run on an Intel core i7-4820K @ 3.70GHz with 64 GB RAM. We report the average of several run of Leon on the benchmark for each of the configurations reported. The running time is shown in seconds. Not available (N.A.) are due to CVC4 not supporting non linear arithmetic.

The use of integers in these benchmarks is not subject to problems of overflow, hence the use of bit-vector instead of integers does not influence the correctness of these particular properties. We can see that there is some overhead to the use of bit-vectors, in particular when implementing more complex data structures. However, in sorting benchmarks, the impact of using bit-vector is less noticeable.

We tried to use benchmarks representative of the use of integers. List operations verifies standard operations over lists of integers. They are mostly transparent to the properties of their element and the results show, as expected, close to no difference between using bit-vectors or integers. The sorting and sorted list benchmarks rely on the comparison operators in order to insert elements. Data structure benchmarks are similar in its use of comparison, however the more complex shapes of formulas makes reasoning more complicated for the bit-vector solver.

Table 2 summarizes experiments involving bit-vectors only. The results list the different kind of verification conditions generated for each benchmark. A valid (V.) verification condition corresponds to proving a property, an invalid (I.) corresponds to finding a bug and an unknown (U.) is due to a timeout. The timeout was set to 30 seconds. The time is in seconds and is the average for solving all verification condition that did not time out.

The binary search benchmark illustrates a typical bug that implementations of binary search can suffer from. One step of the search algoirthm consist in looking up the value at the mean of the two indices. The natural implementation of the mean is $(x + y)/2$, which unfortunately can overflow when x and y are large. However, this is only an artifact due to the computation, as the average is always in the interval between x and y. Leon, with support for bit-vectors, finds a counter-example on the former implementation. A correct implementation of the mean with bit-vector arithmetic is $x + (y - x)/2$. Notice that using mathematical integer, Leon does not report any counter-example, as in such case the two versions are equivalent.

We also evaluated several low level bit manipulation code fragments, many of them taken from the Hacker's Delight book [22]. The operations exploit a small constant number of bit manipulations to obtain a result that one would naively solve using a loop over all the bits. We assert the correctness by comparing the output to what the naive, loop-based, algorithm would have computed. The timeout cases could, in fact, be solved given sufficient time, in this case about a hundred seconds.

Finally we looked at a few arithmetic identities involving non linear arithmetic. Non linear arithmetic is undecidable over unbounded integers, whereas it is decidable but difficult over bit-vectors (indeed, it can encode the breaking of certain problems in cryptography). We use the following types of definitions to prove the validity of an arithmetic simplification:

```
def f(x: Int): Int = {
  require(x > 0 && x < 10000)
  (2*x + x*x) / x
} ensuring(res ⇒ res == x + 2)
```

Both Z3 and CVC4 are currently unable to prove this property over unbounded integers. Due to the finite domain, they do manage to prove it for bit-vectors. Notice the upper bound constraint on the input: without some such upper bound, the identity would actually not hold due to an overflow. The invalid verification conditions is due to one such case.

5 Conclusion

We presented an extension to Leon that addresses a previous semantics mismatch between integral data types of Leon and Scala. With this new support for bit-vector reasoning, Leon is now sound for proving properties about integers, and gives the developer a choice between using mathematical integers or 32-bit bit-vectors, with the semantics used by the verifier matching the actual runtime semantics. Data types such as `Long` and `Short` can be supported entirely analogously as 32-bit integers. We also build on the new SMT-LIB standard to develop a solver-agnostic backend that let Leon profits from advancements in SMT solving algorithms.

Our results show that precise semantics modeling of integers can be more costly than the abstraction with mathematical integers. However, the overhead is often acceptable and sometimes even unnoticeable. Moreover, we demonstrated cases where bit-vector semantics was necessary in order to catch real bugs. In addition to checking division by zero, it is also straightforward to check for expression that could lead to overflows and issue a warning in such cases.

Because Scala and Java do not consider overflows of `Int` as an error but as well-behaved modular arithmetic data types, we are exploring the addition of bounded integers libraries that would automatically check for overflows. These data types would simultaneously encode developer's expectations that the integers remain small and efficient yet have mathematical properties of `BigInt`s. Preliminary results showed that simple Scala programs written with `BigInt` instead of `Int` could lead to a difference in performance of two orders of magnitude. This naturally pushes developers to write code using `Int` even when the intent is simply to use a mathematical integer. We believe that with the infrastructure present in Leon, we might be able to combine the correctness of using `BigInt` with the efficiency of using `Int` via an automated optimization step.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[3] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.

[4] R. T. Boute. The euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, Apr. 1992.

[5] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[6] E. Darulova. *Programming with Numerical Uncertainties.* PhD thesis, EPFL, 2014.

[7] E. Darulova and V. Kuncak. Sound compilation of reals. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2014.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[9] J.-C. Filliâtre and A. Paskevich. Why3 – Where Programs Meet Provers. In *ESOP'13 22nd European Symposium on Programming*, volume 7792, Rome, Italy, Mar. 2013. Springer.

[10] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In R. Alur and D. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.

[11] S. Jha, R. Limaye, and S. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. 2009.

[12] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-aided reasoning: an approach.* Kluwer Academic Publishers, 2000.

[13] E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer-Aided Verification (CAV)*, 2015.

[14] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM.

[15] V. Kuncak, E. Kneuss, and P. Suter. Executing specifications using synthesis and constraint solving (invited talk). In *Runtime Verification (RV)*, 2013.

[16] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] L. C. Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

[18] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 199–210, New York, NY, USA, 2010. ACM.

[19] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability Modulo Recursive Programs. In *SAS*, 2011.

[20] N. Voirol, E. Kneuss, and V. Kuncak. Counterexample-complete verification for higher-order functions. In *SCALA*, 2015.

[21] C. Walther and S. Schweitzer. About verifun. In *Automated Deduction–CADE-19*, pages 322–327. Springer, 2003.

[22] H. S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.