Automating Theorem Proving with SMT

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Manuscript KRML 234, 19 May 2013.

Abstract. The power and automation offered by modern satisfiability-modulo-theories (SMT) solvers is changing the landscape for mechanized formal theorem proving. For instance, the SMT-based program verifier Dafny supports a number of proof features traditionally found only in interactive proof assistants, like inductive, co-inductive, and declarative proofs. To show that proof tools rooted in SMT are growing up, this paper presents, using Dafny, a series of examples that illustrate how theorems are expressed and proved. Since the SMT solver takes care of many formal trivialities automatically, users can focus more of their time on the creative ingredients of proofs.

0 Introduction

A growing number of theorems about mathematics, logic, programming-language semantics, and computer programs are formalized and proved using mechanized proof assistants. Examples of such proof assistants are ACL2 [23], Agda [8, 34], Coq [5], Guru [39], HOL Light [18], Isabelle/HOL [33], PVS [35], and Twelf [36]. The assistants vary in their level of expressivity and automation as well as in the size of their trusted computing base. Satisfiability-modulo-theories (SMT) solvers (for example, Alt-Ergo [6], CVC3 [2], OpenSMT [9], Simplify [13], and Z3 [12]) are collections of (semi-)decision procedures for certain theories. SMT solvers provide a high degree of automation and have, in the last couple of decades, undergone impressive improvements in power. Therefore, it has become increasingly common for proof assistants to use SMT solvers as subroutines, as is done for example in PVS [35] and in Isabelle/HOL's Sledgehammer tactic [7].

Although general proof assistants can be used to verify the correctness of computer programs, there are also some verification tools dedicated to verifying programs. These include Chalice [29], Dafny [25], F* [40], Frama-C [11], Hi-Lite Ada [17], KeY [3], KIV [38], Pangolin [37], Spec# [1], VCC [10], VeriFast [20], and Why3 [16]. Many of these use as their underlying reasoning engine an SMT solver, typically accessed via an intermediate verification language like Boogie [0] or Why [15]. This tool architecture facilitates automation, and it also tends to move the user's interaction with the tool from the formula level (like in general proof assistants) to the program level. This lets users express necessary proof ingredients in program-centric declarations like preconditions or loop invariants. We might therefore refer to this kind of program verifier as *auto-active*—a mix of automatic decision procedures and user interaction at the program level [27].

Just as some proof assistants have incorporated special tactics to better handle program verification (*e.g.*, Ynot [32]), auto-active program verifiers are incorporating features to better support user-guided mathematical proofs (*e.g.*, VeriFast [21] and Dafny [25]). While such program verifiers do not yet achieve the full expressivity of some proof assistants and generally have a much larger trusted computing base, their automation can be remarkable. This automation matters, since it affects the amount of human time required to use the tool. Curiously, these SMT-based tools are primarily program verifiers, but that seems to have happened more by serendipity; one can easily imagine similar SMT-based tools that focus on mathematics rather than on programs.

In this paper, and in the invited talk that the paper accompanies, I argue and showcase that the future may hold room for proof assistants that are entirely based on SMT solving. I focus on the programming language and auto-active program verifier Dafny [25], which supports proof features like induction [26], co-induction [28], and declarative calculations [30]. The paper is a collection of examples that give an idea of what these proof features can do and how they are represented in the Dafny input.

Sec. 1 defines a type and a function that will be used throughout the examples. Sec. 2 states a lemma and proves it by induction. It also shows a proof calculation. Sec. 3 then turns to the infinite by considering co-inductive declarations and proofs. Sec. 4 combines the inductive and co-inductive features into a famous "filter" function, and Sec. 5 proves a theorem about filters by simultaneously applying induction and co-induction. Sec. 6 summarizes the examples and concludes.

1 A Types and a Function

As a basis for all the examples in this paper, let us define a type Stream whose values are infinite lists. Such a type is called a *co-inductive datatype*, but this fancy name need not cause any alarms.

```
codatatype Stream<T> = Cons(head: T, tail: Stream)
```

The stream type is parameterized by the type of its elements, T. The stream type has one constructor, Cons. The names of the parameters to the constructor (head and tail) declare destructors. For example, for any stream s, we have

```
s = Cons(s.head, s.tail)
```

The type of tail is Stream<T>, but here and in other signatures, the type argument T can be supplied automatically by Dafny, so I omit it.

Next, let us declare a function that returns a suffix of a given stream. In particular, Tail(s, n) returns s.tail^{n}. Here is its inductive definition:

```
function Tail(s: Stream, n: nat): Stream
{
  if n = 0 then s else Tail(s.tail, n-1)
}
```

Each function is checked for well-definedness. For a recursive function, this includes a check of well-foundedness among the recursive calls. Well-foundedness is checked using a *variant function*, which conceptually is evaluated on entry to a call. If the value of the variant function is smaller for the callee than for the caller, well-foundedness follows. If no variant function is explicitly supplied, Dafny guesses one. The guess is the lexicographic ordering on the tuple of the function's arguments, omitting arguments whose types have no ordering, like co-inductive datatypes. For Tail, Dafny (correctly) guesses the variant function n and, using it, checks that Tail's recursion is indeed well-founded. No further input is required from the user—the tool automatically initiates the check—and since the check succeeds, the user is not bothered by any messages from the tool (except for a small indication in the margin of the integrated development environment that for a split second shows that the verifier is active).

2 An Inductive Proof

In order to show how lemmas and proofs are set up, let us consider an alternative definition of Tail:

```
function Tail_Alt(s: Stream, n: nat): Stream
{
  if n = 0 then s else Tail_Alt(s, n-1).tail
}
```

and let us prove that Tail and Tail_Alt give the same result.

A lemma is expressed as a *method* in the programming language, that is, as a code procedure with a pre- and postcondition. As usual (*e.g.*, [19]), such a method says that for any values of the method parameters that satisfy the method's precondition, the method will terminate in a state satisfying the postcondition.[0] This corresponds to what is done in mathematics for a lemma: a lemma says that for any values of the lemma parameters that satisfy the lemma's antecedent, the lemma's conclusion holds. So, we express our lemma about Tail and Tail_Alt by declaring the following method:

```
ghost method Tail_Lemma(s: Stream, n: int)
  requires 0 ≤ n;
  ensures Tail(s, n) = Tail_Alt(s, n);
```

The precondition of this method (keyword **requires**) says that n is a natural number (which, alternatively, we could have indicated by declaring the type of n to be **nat**). The postcondition (keyword **ensures**) gives the property we want to prove. The designation of the method as a *ghost* says that we do not want the Dafny compiler to emit any executable code. In other words, a ghost method is for the verifier only; the compiler ignores it.

To get the program verifier to prove the lemma, we supply a method body and let the verifier convince itself that all code control paths terminate and establish the postcondition. The body we supply typically consists of **if** statements and (possibly recursive) method calls, but other statements (*e.g.*, **while** loops) can also be used. A recursive call

[0] In general, methods can have effects on the memory of the program. Such effects are declared with a **modifies** clause in the method specification. However, since there is no need for such effects here, I ignore further discussion of them.

corresponds to invoking an inductive hypothesis, because the effect on the proof is to obtain the proof goal (stated in the postcondition) for the callee's arguments, which in a well-foundedness check are verified to be "smaller" than the caller's arguments. It is common to supply assertions that act as in-place lemmas: a statement **assert** `Q` tells the verifier to check that the boolean condition `Q` holds, after which the verifier can make use of that condition. A more systematic way to direct the verifier is provided by Dafny's **calc** statement, whose verified calculations take the form of human-readable equational proofs [30].

Here is a proof of the lemma:

```
ghost method Tail_Lemma(s: Stream, n: int)
  requires 0 ≤ n;
  ensures Tail(s, n) = Tail_Alt(s, n);
{
  if n < 2 {
    // def. of Tail and Tail_Alt
  } else {
    calc {
      Tail(s, n);
    =  // def. Tail, since n ≠ 0
      Tail(s.tail, n-1);
    =  { Tail_Lemma(s.tail, n-1); }  // induction hypothesis
      Tail_Alt(s.tail, n-1);
    =  // def. Tail_Alt, since n-1 ≠ 0
      Tail_Alt(s.tail, n-2).tail;
    =  { Tail_Lemma(s.tail, n-2); }  // induction hypothesis
      Tail(s.tail, n-2).tail;
    =  // def. Tail, since n-1 ≠ 0
      Tail(s, n-1).tail;
    =  { Tail_Lemma(s, n-1); }  // induction hypothesis
      Tail_Alt(s, n-1).tail;
    =  // def. Tail_Alt, since n ≠ 0
      Tail_Alt(s, n);
    }
  }
}
```

The method body provides two code paths. For the `n < 2` branch, the verifier can prove the postcondition by unwinding the definitions of each of `Tail` and `Tail_Alt` once or twice (which the verifier is willing to do automatically). The else branch uses a **calc** statement with a number of equality-preserving steps, each of which is verified. Some steps are simple and need no further justification; the code comments give explanations for human consumption. Other steps are justified by *hints*, which are given as code blocks (in curly braces). Here, each hint makes a recursive call to `Tail_Lemma`, which in effect invokes the induction hypothesis.

In more detail, for each step in a calculation, the verifier checks that the equality entailed by the step is provable after the code in the associated hint (if any). In the calculation above, each of the provided hints consists of a single recursive call. As usual in program verification, the verifier thus checks that the precondition of the callee is met, checks that the variant function is decreased for the call (to ensure that the recursion will terminate), and can then assume the postcondition of the callee. For example, the postcondition that holds after the first recursive call is:

```
Tail(s.tail, n-1) = Tail_Alt(s.tail, n-1)
```

which is essentially the induction hypothesis for `s,n := s.tail,n-1`. Since no variant function is supplied explicitly, Dafny guesses `n`, which it verifies to decrease. Thus, the recursion—and indeed, the induction—is well-founded.

For brevity, the equality signs between the lines in the calculation can be omitted. Or, if desired, they can be replaced by different operators, like $\Longrightarrow$, $\Longleftarrow$, or <.

The calculation in the example gives more detail than the Dafny verifier needs, but, as given, yields a presentation of the proof that is better suited for a human. In fact, the proof calculation is quite readable; it looks almost identical to how one would write an equational-style proof by hand. For a comparison with other styles of declarative proofs, like Isar [41], and with tactic-based proofs, see [30].

3 Co-recursion and a Co-inductive Proof

In this section, we consider how values of a co-datatype are constructed and how one states and proves properties of such values.

Values of co-inductive datatypes may be of an infinite nature. For example, a stream represents an infinite list of elements. Here is a function that defines such a value, namely the stream whose elements are the integers from `n` upward in increasing order:

```
function Up(n: int): Stream<int>
{
  Cons(n, Up(n+1))
}
```

It may look as if invocations of `Up` will never terminate, but the self-call of `Up` is identified by Dafny as being *co-recursive*, because it is positioned as an argument to a co-datatype constructor. Co-recursive calls are compiled into lazily evaluated code, so that the arguments to the constructor are not evaluated until their values are used by the executing program (if ever). Consequently, for a co-recursive call, there is no need for the verifier to enforce a decrease of a variant function.

Here is another function on streams:

```
function Prune(s: Stream): Stream
{
  Cons(s.head, Prune(s.tail.tail))
}
```

It defines a stream consisting of half of the elements of the given stream: every other element, starting with the first. Note that the self-call to `Prune` is co-recursive, so the verifier does not need to check termination.

To define a property of a co-inductive datatype, one uses a *co-predicate*. For example, the following co-predicate holds for streams that consist of even integers:

```
copredicate AllEven(s: Stream<int>)
{
  s.head % 2 = 0 ∧ AllEven(s.tail)
}
```

Co-predicates are defined by greatest fix-points, that is, as the greatest solutions of the recursive equations to which their definitions give rise. Applied to the example, this means that AllEven(s) evaluates to **true** as long as there is no suffix t of s such that t.head % 2 $\neq$ 0. Eager evaluation of a co-predicate may fail to terminate and lazy evaluation would not be meaningful, so co-predicates are always ghost. In other words, they are never part of executing code, but they can be used to describe and reason about executing code.

We have now seen three features from the quartet of co-inductive features in Dafny: co-datatypes define possibly infinite data structures, co-recursive function calls make it possible to define values of co-datatypes, and co-predicates define properties of co-datatypes. The fourth feature is *co-methods*, whose purpose is to enable co-inductive proofs. Let us consider an example.

We will state a theorem that for any even n, Prune(Up(n)) consists only of even integers. Because we intend to prove the theorem by co-induction, we use a co-method:

```
comethod Theorem(n: int)
  requires n % 2 = 0;
  ensures AllEven(Prune(Up(n)));
{
  Theorem(n+2);
}
```

Ignoring the issue of termination, this proof can be understood in the same manner as inductive proofs: AllEven says something about the head of the stream Prune(Up(n)), which is proved automatically. It also says something about the tail of the stream, which follows from the postcondition of the call Theorem(n+2) and the definitions of the functions involved. To make this argument more explicit, the call could have been preceded by the following calculation (where, for brevity and variety, I have chosen to omit the optional equality signs between lines in the left margin):

```
calc {
  Prune(Up(n)).tail;
  Prune(Up(n).tail.tail);
  { assert Up(n).tail.tail = Up(n+2); }
  Prune(Up(n+2));
}
```

In contrast to methods, whose recursive calls are checked to terminate (by checking that they decrease the variant function), calls to co-methods are always allowed. In other words, the co-induction hypothesis can always be obtained; however, the *use* of it is restricted. Intuitively, the co-induction hypothesis can be used to discharge only those conjuncts that show up after one unwinding of the co-predicate in the co-method's postcondition. I will give some details about this in Sec. 5.

For example, suppose the body of co-method Theorem were replaced by the call Theorem(n). With unrestricted use of the postcondition of this call, the co-induction hypothesis obtained would trivially prove the theorem itself. However, because the co-induction hypothesis can be used only on conjuncts from an unwinding of the postcondition, the call Theorem(n) provides no benefit here.

4 A Filter Function

Let us now consider a more difficult function definition, namely that of a *filter* function on streams. For any stream s, we want the filter function to return the stream consisting of those elements of s that satisfy some predicate P. A filter function like this is used, for example, in the prime number sieve of Eratosthenes (*cf.* [4, 24, 14]).

Conceptually, the filter function and all related lemmas are parameterized by the predicate P. Lacking the higher-order features necessary to take P as a parameter, we represent an arbitrary predicate by declaring a (here, global) generic predicate without a defining body:[1]

```
predicate P<T>(x: T)
```

The definition of Filter has the following form:

```
function Filter(s: Stream): Stream
  //...specification to be written...
{
  if P(s.head) then
    Cons(s.head, Filter(s.tail))
  else
    Filter(s.tail)
}
```

The first branch of this definition is fine, because its call to Filter is co-recursive. However, the other call to Filter is not co-recursive, so it is subject to a termination check. This makes sense, because if the given stream has no elements that satisfy P, then Filter would never terminate in its computation to produce the next element of the resulting stream. Note, thus, how Dafny allows one function to be involved in both recursive and co-recursive calls. Next, we will consider how to deal with the termination of the recursive call.

To avoid non-termination, we must restrict Filter's input to streams that contain infinitely many elements that satisfy P. We give the following definitions:

```
predicate HasAnother(s: Stream)
{
```

[1] Dafny allows such a body-less predicate to be placed in a *module*. Other modules can then be declared as *refinements* of this module, and each refinement module can give its own specific definition of the predicate.

```
    ∃ n • 0 ≤ n ∧ P(Tail(s, n).head)
  }
  copredicate AlwaysAnother(s: Stream)
  {
    HasAnother(s) ∧ AlwaysAnother(s.tail)
  }
```

Predicate `HasAnother(s)` says that, after some finite prefix of `s`, there is an element that satisfies `P`, and `AlwaysAnother(s)` says that `HasAnother` holds at every point in the stream. We can now restrict the input to `Filter` by adding a precondition:

```
    requires AlwaysAnother(s);
```

Even with this precondition, the verifier complains that it cannot prove termination. To remedy the situation, we supply a variant function explicitly. As the variant function, we will use the length of the non-`P` prefix of `s`, that is, the number of steps to the next element satisfying `P`. Using `StepsToNext(s)` to denote that number of steps, we add to the specification of `Filter` the following clause:

```
    decreases StepsToNext(s);
```

Given a stream that satisfies `AlwaysAnother`, function `StepsToNext` returns a natural number. It is tempting to define it with a body like

```
    if P(s.head) then 0 else 1 + StepsToNext(s.tail)
```

but to prove that this recursive call to `StepsToNext` terminates, we would need a variant function like `StepsToNext` itself. Instead, we find a number of steps that will yield some `P` element, and then we use this number as an upper bound in a linear search to the first `P` element:

```
  function StepsToNext(s: Stream): nat
    requires AlwaysAnother(s);
  {
    var n :| 0 ≤ n ∧ P(Tail(s, n).head);
    Steps(s, n)
  }
  function Steps(s: Stream, n: nat): nat
    requires P(Tail(s, n).head);
    ensures P(Tail(s, Steps(s, n)).head);
    ensures ∀ i • 0 ≤ i < Steps(s, n) ⟹ ¬P(Tail(s, i).head);
  {
    if P(s.head) then 0 else 1 + Steps(s.tail, n-1)
  }
```

These definitions require some explanation.

The "let such that" expression **var** `x :| Q; E` evaluates to `E` in which all free occurrences of `x` are bound to a value that satisfies `Q`. The expression is well-defined only if there exists a value for `x` that satisfies `Q`. In `StepsToNext`, this proviso follows from the precondition `AlwaysAnother(s)`. Note that `n` may be set to any number of steps that will reach a `P` element in `s`, not necessarily the smallest.

The specification of the auxiliary function Steps requires s to reach a P element in n steps. It ensures that the result value, which in the **ensures** clause is denoted by Steps(s, n), is not only a number of steps that reaches a P element (first **ensures** clause) but also the smallest such number (second **ensures** clause).

The body of Steps encodes a straightforward linear search.

To prove the recursive call in the body of Filter(s) to be well-founded, the verifier checks that the given variant function decreases, that is,

```
StepsToNext(s.tail) < StepsToNext(s)
```

This condition rests on the fact that StepsToNext returns the smallest number of steps to reach a P element, which is spelled out by the postcondition of Steps. Note that StepsToNext does not need to declare such a postcondition, because Dafny unwinds the definition of StepsToNext and obtains an expression in terms of Steps. Since Steps is recursive, the needed property is not evident from any bounded number of unwindings, so the presence of the postcondition essentially facilitates an inductive argument.

Finally, rather than introducing the auxiliary function Steps, one could consider replacing the body of StepsToNext with one whose let-such-that condition is stronger:

```
var n :| 0 ≤ n ∧ P(Tail(s, n).head) ∧
         ∀ i • 0 ≤ i < n ⟹ ¬P(Tail(s, i).head);
n
```

However, Dafny is unable to prove the existence of such an n directly from the precondition AlwaysAnother(s). The use of Steps is one way to set up the necessary inductive argument.

5 A Property of Filter

The interesting property to prove about Filter(s) is that it returns the subsequence of s that consists of exactly those elements that satisfy P. The notion of such a subsequence can be divided up into the property that Filter returns the right set of elements:

```
∀ x • x ∈ Filter(s) ⟺ x ∈ s ∧ P(x)
```

(where I have taken the liberty of using operator ∈ as if stream were sets) and the property that Filter(s) preserves the order of elements in s. Let us look at one possible way to state and prove the latter.

To simplify matters, let us suppose that there is a function Ord from the elements of streams to the integers.

```
function Ord<T>(x: T): int
```

Using a co-predicate, we define what it means for a stream's elements to be strictly increasing:

```
copredicate Increasing(s: Stream)
{
  Ord(s.head) < Ord(s.tail.head) ∧ Increasing(s.tail)
}
```

Now we can state the order-preservation theorem that we want to prove:

```
ghost method Theorem_FilterPreservesOrdering(s: Stream)
  requires AlwaysAnother(s) ∧ Increasing(s);
  ensures Increasing(Filter(s));
```

This theorem is not the most general order-preservation theorem we can state, but it suffices for our purpose of showing an interesting proof.

To prove the theorem, we introduce an alternative definition of `Increasing`:

```
copredicate IncrFrom(s: Stream, low: int)
{
  low ≤ Ord(s.head) ∧ IncrFrom(s.tail, Ord(s.head) + 1)
}
```

The two definitions are interchangeable, as the following two lemmas show.

```
comethod Lemma_Incr0(s: Stream, low: int)
  requires IncrFrom(s, low);
  ensures Increasing(s);
{
}
comethod Lemma_Incr1(s: Stream)
  requires Increasing(s);
  ensures IncrFrom(s, Ord(s.head));
{
  Lemma_Incr1(s.tail);
}
```

The co-inductive proof of `Lemma_Incr0` is done automatically, whereas the other requires an explicit appeal to the co-induction hypothesis.

We can now write the theorem in terms of `IncrFrom`:

```
comethod Lemma_FilterPreservesIncrFrom(s: Stream, low: int)
  requires AlwaysAnother(s) ∧ IncrFrom(s, low) ∧ low ≤ Ord(s.head);
  ensures IncrFrom(Filter(s), low);
  decreases StepsToNext(s);
{
  if P(s.head) {
    Lemma_FilterPreservesIncrFrom(s.tail, Ord(s.head) + 1);
  } else {
    Lemma_FilterPreservesIncrFrom#[_k](s.tail, low);
  }
}
```

The proof of this lemma is interesting because it uses co-induction and induction together. The first branch of the **`if`** statement makes an appeal to the co-induction hypothesis. Dafny will actually fill it in automatically, so the proof also goes through with that call omitted. In the else branch, we cannot use the co-induction hypothesis, because, as discussed above, the co-induction hypothesis can be used only after one unwinding of the proof goal. To make use of the lemma's postcondition for `s.tail` directly, we instead make a recursive call to the lemma. Syntactically, this is achieved by the characters "`#[_k]`". The recursive call gives rise to a proof obligation of termination, which is addressed by the explicit **`decreases`** clause.

Co-methods are used to establish the validity of co-predicates (including equality on co-datatype values, which is a built-in co-predicate). These co-inductive proof obligations are actually carried out by induction, in conjunction with a meta-theorem. For any co-predicate `Q(x)`, let the *prefix predicate* `Q#[_k](x)` denote the first `_k` unrollings of `Q`, defined inductively. For example, the prefix predicate for co-predicate `AllEven` is:[2]

```
predicate AllEven#[_k: nat](s: Stream<int>)
{
  if _k = 0 then
    true
  else
    s.head % 2 = 0 ∧ AllEven#[_k-1](s.tail)
}
```

Similarly, for each co-method `M(x)`, Dafny generates a *prefix method* `M#[_k](x)`, where each call `M(E)` in the co-method's body is turned into a call `M#[_k-1](E)` in the corresponding prefix method. In more detail, the following co-method:

```
comethod M(x: T)
  ensures Q(x);
  decreases D(x);
{
 ... M(E); ...
}
```

is turned into:

```
ghost method M#[_k: nat](x: T)
  ensures Q#[_k](x);
  decreases _k, D(x);
{
  if _k ≠ 0 {
   ... M#[_k-1](E); ...
  }
}
```

Any explicit prefix-method call in the body of `M` (like the one in the else branch of the filter lemma co-method above) is left unchanged in the corresponding prefix method. A recursive call to `M#[K]` where `K < _k` corresponds to obtaining the co-induction hypothesis (for use after `_k - K` unwindings of the co-predicate in the proof goal), whereas

[2] Prefix predicates are declared automatically. The made-up declaration syntax shown here is suggestive of how prefix predicates are actually invoked, with the unrolling-depth argument in square brackets, set apart from the other arguments.

a call to M#[_k] is just an ordinary recursive call corresponding to the induction hypothesis. By verifying the inductive prefix method for any _k, the postcondition of the co-method follows on account of the following meta-theorem [31, 28]:

```
∀ x: T • Q(x) ⟺ ∀ _k: nat • Q#[_k](x)
```

For more details, see [28].

Finally, the proof of Theorem_FilterPreservesOrdering is given as follows:

```
{
  Lemma_Incr1(s);
  Lemma_FilterPreservesIncrFrom(s, Ord(s.head));
  Lemma_Incr0(Filter(s), Ord(s.head));
}
```

6 Conclusion

In this paper, I have conveyed a flavor of Dafny's proof features by showing examples of inductive and co-inductive definitions, proofs by induction and by co-induction, as well as human-readable proofs. These are features that until recently were confined to interactive proof assistants, but they can now be supported by auto-active verifiers.

To try the examples in the Dafny tool,[3] the only input given to the tool are the lines shown in this paper—no additional proof tactics need to be supplied.

The given examples showcase the high degree of automation that is possible in a tool powered by an SMT solver and designed to keep the interaction at the problem level (and not, for example, at the level of defining and using necessary prover tactics). Users are not bothered with trivial details (like the associativity of logical and arithmetic operators) and the human involvement to prove that user-defined functions are mathematically consistent is small. Even the tricky recursive call of Filter is solved by defining and using StepsToNext as a variant function, which does not require an excessive amount of human effort. When more information is needed, human-readable calculations can be used, putting proofs in a format akin to what may be done by hand.

For each function and method in the examples shown, the verifier needs to spend only a small fraction of a second. This makes performance good enough to be running the verifier continuously in the background of the integrated development environment, which is what Dafny does. Most changes of the program text yield a near-instant response, which is important when developing proofs. Note that performance is at least as important for failed proofs as for successful proofs, because failed proofs happen on the user's time (see [27] for some research directions for auto-active verification environments).

In the future, I expect a higher degree of automation to become available in proof assistants. For tools like Dafny that already provide a high degree of automation, I expect to see a richer set of features (for example, higher-order functions, drawing inspiration

[3] Dafny can be installed from http://dafny.codeplex.com. It can also be run directly in a web browser at http://rise4fun.com/dafny.

from Pangolin [37], Who [22], and F* [40], and user-defined theories, drawing inspiration from Coq [5] and Why3 [16]) as well as work that will seek to reduce the currently large trusted computing base for SMT-based verifiers.

Acknowledgments I am grateful to Maria Christakis, Sophia Drossopoulou, Peter Müller, and David Pichardie for comments on an earlier draft of this paper.

References

0. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
1. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.
2. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer, July 2007.
3. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*. Springer, 2007.
4. Yves Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005*, volume 3461 of *LNCS*, pages 102–115. Springer, April 2005.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
6. François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in SMT solvers. In *SMT '08/BPR '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5. ACM, July 2008.
7. Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173 of *LNCS*, pages 107–121. Springer, July 2010.
8. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, August 2009.
9. Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *LNCS*, pages 150–153. Springer, March 2010.
10. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, August 2009.

11. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C — a software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods — 10th International Conference, SEFM 2012*, volume 7504 of *LNCS*, pages 233–247. Springer, October 2012.
12. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, March–April 2008.
13. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
14. Marco Devillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In Elsa L. Gunter and Amy P. Felty, editors, *Theorem Proving in Higher Order Logics, 10th International Conference, TPHOLs'97*, volume 1275 of *LNCS*, pages 89–104. Springer, August 1997.
15. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007*, volume 4590 of *LNCS*, pages 173–177. Springer, July 2007.
16. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems — 22nd European Symposium on Programming, ESOP 2013*, volume 7792 of *LNCS*, pages 125–128. Springer, March 2013.
17. Jérôme Guitton, Johannes Kanig, and Yannick Moy. Why Hi-Lite Ada? In K. Rustan M. Leino and Michał Moskal, editors, *BOOGIE 2011: First International Workshop on Intermediate Verification Languages*, August 2011.
18. John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96*, volume 1166 of *LNCS*, pages 265–269. Springer, November 1996.
19. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.
20. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.
21. Bart Jacobs, Jan Smans, and Frank Piessens. VeriFast: Imperative programs as proofs. In *VSTTE workshop on Tools & Experiments*, August 2010.
22. Johannes Kanig and Jean-Christophe Filliâtre. Who: A verifier for effectful higher-order programs. In *ACM SIGPLAN Workshop on ML*. ACM, August 2009.
23. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
24. François Leclerc and Christine Paulin-Mohring. Programming with streams in Coq — a case study: the sieve of Eratosthenes. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93*, volume 806 of *LNCS*, pages 191–212. Springer, May 1993.
25. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, April 2010.
26. K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, January 2012.

27. K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In Tom Ball, Lenore Zuck, and N. Shankar, editors, *UV10 (Usable Verification) workshop*. `http://fm.csl.sri.com/UV10/`, November 2010.
28. K. Rustan M. Leino and Michał Moskal. Co-induction simply: Automatic co-inductive proofs in a program verifier. Technical Report MSR-TR-2013-49, Microsoft Research, May 2013.
29. K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *LNCS*, pages 378–393. Springer, March 2009.
30. K. Rustan M. Leino and Nadia Polikarpova. Verified calculations. In *Fifth Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2013)*, May 2013.
31. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1982.
32. Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008*, pages 229–240. ACM, September 2008.
33. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
34. Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.
35. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction — CADE-11, 11th International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 748–752. Springer, June 1992.
36. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *Automated Deduction — CADE-16, 16th International Conference on Automated Deduction*, volume 1632 of *LNAI*, pages 202–206. Springer, July 1999.
37. Yann Régis-Gianas and François Pottier. A Hoare logic for call-by-value functional programs. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008*, volume 5133 of *LNCS*, pages 305–335. Springer, July 2008.
38. Wolfgang Reif. The KIV system: Systematic construction of verified software. In Deepak Kapur, editor, *Automated Deduction — CADE-11, 11th International Conference on Automated Deduction*, volume 607 of *LNCS*, pages 753–757. Springer, June 1992.
39. Aaron Stump, Morgan Deters, Adam Petcher, Todd Schiller, and Timothy Simpson. Verified programming in Guru. In Thorsten Altenkirch and Todd Millstein, editors, *PLPV '09 — Proceedings of the 3rd workshop on Programming Languages meets Program Verification*, pages 49–58. ACM, January 2009.
40. Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011*, pages 266–278. ACM, September 2011.
41. Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.