Automating Induction with an SMT Solver

K. Rustan M. Leino

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

Abstract. Mechanical proof assistants have always had support for inductive proofs. Sometimes an alternative to proof assistants, satisfiability modulo theories (SMT) solvers bring the hope of a higher degree of automation. However, SMT solvers do not natively support induction, so inductive proofs require some encoding into the SMT solver's input.

This paper shows a surprisingly simple tactic—a rewriting strategy and a heuristic for when to apply it—that has shown to be useful in verifying simple inductive theorems, like those that can occur during program verification.

The paper describes the tactic and its implementation in a program verifier, and reports on the positive experience with using the tactic.

0 Introduction

Mathematical induction is an important element of just about any kind of formal proof. This paper concerns the use of induction in an automatic program verifier. More specifically, it is concerned with providing more automation for inductively proving some properties in the kind of program verifier that uses a satisfiability modulo theories (SMT) solver [10,22] as its reasoning engine.

Mechanical proof assistants have always provided support for inductive proofs, most famously starting with the Boyer-Moore prover whose powerful heuristics tried to determine which variables to use in induction schemes [5]. That work has been continued in proof assistants like PVS [23] and ACL2 [15]. Another technique for automatically trying to discover how to construct an inductive proof for a given property is *rippling* [6,14]. As is well known, it is frequently necessary to strengthen an induction hypothesis in order to make a proof go through, and techniques like rippling heuristically try to determine when it might be appropriate to strengthen or generalize a property to be proven. Whereas rippling is goal directed, the technique employed by Zeno [24] is more opportunistic in the way it proceeds.

Unsurprisingly, any system that reasons about infinitely many possible executions of a software program also makes use of induction, either explicitly or implicitly. For example, the KeY system [2] lets a user explicitly specify which induction scheme to apply when reasoning about the executions of a loop. Program verification (and theorem proving) in Coq [3] and VeriFun [25] also tend to make heavy use of induction. Other program verifiers, like Dafny [19], VCC [8], and VeriFast [12], implicitly rely on induction: the loop invariants used to reason about loop executions and the pre/-post specifications used to reason about recursive calls essentially play the role of an induction hypothesis.

V. Kuncak and A. Rybalchenko (Eds.): VMCAI 2012, LNCS 7148, pp. 315–331, 2012.

The implicit support of induction lets a user write programs whose correctness implies the validity of user-provided mathematical properties, essentially giving a manual way to write proofs using a program verifier [15,17,26,13]. In this paper, I go one step further, introducing a tactic that heuristically identifies programmer-supplied properties whose proof may benefit from induction, then automatically sets up the induction hypothesis, and finally passes the proof obligation to an SMT solver. I have implemented the technique in the Dafny program verifier [19][0] and have used it, for example, to automatically prove 45 of the first 47 problems in an evaluation corpus for automatic induction. The tactic is not nearly as powerful as what is used in provers like ACL2 or Zeno; indeed, it never strengthens or generalizes a property to be proved. Instead, the strong appeal of the present tactic lies in its simplicity and surprising effectiveness.

1 Background on Dafny

Before getting to the induction tactic, let me review two properties about program verifiers like Dafny. First, I will explain the verifier architecture, how to think about going from source-language semantics to SMT-solver input. Second, I will show how lemmas to be proved by induction arise in the context of a program verifier.

1.0 Verifier Architecture

A standard program-verifier architecture is to translate the source language of interest into an intermediate verification language (IVL) and then to pass the resulting IVL programs to a verification engine for the IVL [21,1,11]. In other words, the semantics of a given source-language program are encoded into an IVL program such that the correctness of the IVL program implies the correctness of the source program. The verification engine for the IVL attempts to establish the correctness of IVL programs by generating verification conditions that it passes to a reasoning engine, typically an SMT solver.

In this paper, the source language used is Dafny [19], the IVL is Boogie 2 [20], and the SMT solver is Z3 [9], but the tactic described is applicable to other program verifiers and reasoning engines as well. For the purpose of this paper, it is not necessary to understand the details of the Dafny-to-Boogie translation [18], and even less so the Boogie-to-Z3 translation [1,20]. It suffices to realize that the semantics of a given source-language program sometimes provides certain guarantees and sometimes dictates some proof obligations, and that the program verifier encodes these guarantees and proof obligations by translation into the IVL. Next, I will explain the form of this encoding.

A program verifier explicates source-language proof obligations by encoding them as *assert statements* in the IVL. Such proof obligations arise from the semantics of the executable constructs of the source language (for example, expressions used to index into an array must evaluate to a value within the bounds of the array) and from programmer-supplied specifications (for example, method postconditions). The program verifier also explicates source-language guarantees by encoding them as *assume statements* in the IVL.

[0] Dafny is available as open source and can also be used without installation in a web browser, see `research.microsoft.com/dafny`.

For example, the Dafny verifier translates the allocation statement `a := new int[E];` into the following Boogie code:

$$\textbf{assert } 0 \leqslant E; \quad \ldots \quad \textbf{assume } dtype(a) = array(type\_int());$$

where the assert statement encodes the proof obligation that the requested array size not be negative and the assume statement says that the dynamic type of a, after the assignment, is an array of integers (where $dtype$, $array$, and $type\_int$ are some functions defined elsewhere in the translation).

Semantically, $\textbf{assert } P;$ is equivalent to:

$$\textbf{assert } P; \ \textbf{assume } P; \tag{0}$$

(see [0]). Intuitively, this says that after proving P, one is entitled to assume it. This means that a proof obligation P arising in a source language can be translated into the intermediate verification language as the two statements (0). More generally, it is sound to translate a proof obligation P into:

$$\textbf{assert } Q; \ \textbf{assume } R; \tag{1}$$

where Q implies P and P implies R. This can be useful if, to the SMT solver, Q is easier to prove and R is easier to use.

1.1 Inductive Lemmas in a Program Verifier

On the way to proving the correctness of a program, it happens that one needs to state and prove lemmas about functions or data structures that are used by the program. When such a lemma requires an inductive proof, the induction tactic described in this paper can be useful.

Dafny is a programming language and a program verifier; it has no special constructs for stating and proving lemmas. Instead, a lemma can be stated as an inline assertion (via Dafny's **`assert`** statement) or as a call to a method whose postcondition is the statement of the lemma (*cf.* [13]). Such a method in Dafny is usually declared to be a *ghost* method; the Dafny verifier treats ghost methods and other ghost constructs like their non-ghost counterparts, but the Dafny compiler generates no code for ghost constructs [19].

For example, the Fibonacci function can be defined in Dafny as follows:

```
function Fib(n: nat): nat
{
  if n < 2 then n else Fib(n-2) + Fib(n-1)
}
```

A **`function`** in Dafny is a mathematical function and it denotes an expression (given in the body of the function). A **`method`**, on the other hand, denotes a behavior and is implemented by statements with possible control flow and mutations. To state a lemma about the `Fib` function, we introduce a (ghost) method with the desired lemma as the postcondition; for example:

```
ghost method FibLemma()
  ensures ∀ n: nat • Fib(n+1) + Fib(n+2) ≤ Fib(n+3);
{ }
```

Dafny will attempt to verify that the method terminates and that it terminates in a state where the postcondition holds. A successful verification thus implies that the property stated in the postcondition is a valid lemma. To use this lemma in some code, one simply invokes method `FibLemma`. Because the method is ghost, it is not included in the executable code, and thus the lemma has no effect on the run-time behavior of the program.

2 The Induction Tactic

The induction tactic builds on simple concepts working in concert. I will explain each concept in a separate subsection and then combine them to describe the tactic.

2.0 Induction Principle

The *Induction Principle* says that the formula

$$\forall n \bullet P(n) \tag{2}$$

where $P(n)$ is any expression that may have free occurrences of n, is equivalent to the formula

$$\forall n \bullet (\forall k \bullet k \prec n \implies P(k)) \implies P(n) \tag{3}$$

where $\prec$ is any well-founded order. The antecedent in (3) is known as the *induction hypothesis*. Thus, by the Induction Principle, to prove the validity of the formula (2), we may elect to proceed by proving the validity of the ostensibly weaker formula (3).

I feel compelled to make a remark, which for readers familiar with some mechanical proof assistants may clear up a point about what I refer to as induction. The fact that the induction hypothesis in (3) quantifies over all k smaller than n is known as *strong induction*. Many times, a weaker induction hypothesis suffices, namely the one that quantifies only over those k that are "one smaller" than n. For example, if n and k range over natural numbers, the weaker induction hypothesis can be stated as:

$$\forall k \bullet k = n - 1 \implies P(k)$$

Furthermore, by distinguishing those n that have a value "one smaller" and those n that do not, this condition is often formulated as:

$$n = 0 \quad \vee \quad (0 < n \ \wedge \ P(n-1))$$

Using this condition in place of the antecedent in (3) and simplifying, we get:

$$P(0) \quad \wedge \quad \forall n \bullet 0 < n \ \wedge \ P(n-1) \implies P(n)$$

or equivalently:

$$P(0) \quad \wedge \quad \forall n \bullet P(n) \implies P(n+1) \tag{4}$$

The two conjuncts in formula (4) are referred to as the *base case* and the *induction step*, which is how induction is used in some proof assistants and is probably how most of us learned about induction in our education. However, note that the formulation of the Induction Principle above (that is, (2) = (3)) does not necessitate bringing in the concept of a case distinction when defining induction. (This is somewhat analogous to the use of recursion in programming, where a conditional statement and a call statement are independent constructs that usefully come together in the body of a recursive procedure.)

2.1 Induction Translation

Here is why observation (1) in Sec. 1.0 is interesting for induction. If a proof obligation in a given source program takes the form (2), then the program verifier has the option to translate it into the IVL as:

$$\textbf{assert}\ (3);\ \textbf{assume}\ (2); \tag{5}$$

This *Induction Translation* has the effect that the SMT solver will be asked to establish the validity of (3), after which it is entitled to assume (2). Note that SMT solver does not need to know anything about induction. Instead, the (source-to-IVL translation of the) program verifier takes responsibility for the soundness of the translation into (5), and that soundness is justified by the Induction Principle.

2.2 Induction Heuristic

A program verifier has the option of translating proof obligation like (2) into the IVL statements (5), but when would that be a good idea? Always doing so can lead to bad performance, and always requiring the source-language programmer explicitly to say when to use the Induction Translation could be a nuisance. Better would be to use a good heuristic, possibly with a way to manually override the outcome of the heuristic.

A heuristic that is both simple and seems to work well, and which I will refer to as the *Induction Heuristic*, is to apply the Induction Translation if the bound variable n in (2) is used in $P(n)$ as part of an argument to a recursive function.

I tried and rejected a less discriminating heuristic, namely to apply the Induction Translation also if n is used in $P(n)$ as part of an index expression into an array or sequence. The Dafny test suite contains hundreds of methods and mentions more than 700 quantifiers, but most of the quantifiers can be proved without induction. I tried the less discriminating heuristic on the test suite. This resulted in out-of-memory failures for 8 of the method verifications and ten-minute timeouts for 2 others. Evidently, the additional induction hypotheses caused the SMT solver too much distraction.

The default heuristic used by Dafny is actually more discriminating than the Induction Heuristic, in two ways. First, n cannot be just any subexpression of an argument to a recursive function; the argument must *prominently* feature n. An expression prominently features n if the expression is n, or if the expression has the form $E + F$ or $E - F$ where E or F prominently features n. For example, in the postcondition of

method `FibLemma` in Sec. 1.1, all three calls to `Fib` prominently feature `n` as an argument. Second, not all argument positions to recursive functions are considered; the formal parameter corresponding to the argument must contribute to the variant (which is used for proving termination) of the function. However, these two additional discriminating factors have not made any appreciable difference in the experiments I have run.

2.3 Well-Founded Orders in Dafny

The Induction Principle holds for any well-founded order $\prec$. Dafny fixes a well-founded order for each of its types and for lexicographic tuples. These are used in Dafny when reasoning about termination of loops and recursive calls [19]. The same ordering is used for $\prec$ in the Induction Translation.

The types most frequently used with induction are integers and inductive datatypes. For integers x and y, Dafny defines $x \prec y$ as:

$$x < y \quad \wedge \quad 0 \leqslant y$$

The lower bound 0 is somewhat arbitrary, and note that the order is not total when both x and y are negative; however, this simple order is easy for a programmer to remember and works well in practice.[1]

The ordering on inductive datatypes associates a *rank* with each datatype value and defines the rank of a constructed value to be strictly above the rank of each of its arguments. For example, given the definition:

```
datatype List = Nil | Cons(int, List);
```

Dafny defines $xs \prec \mathit{Cons}(x, xs)$ for any integer x and list xs.

2.4 Datatypes and Case Distinctions

The current version of Dafny does not use native SMT support for inductive datatypes. Instead, it encodes constructors and destructors of datatype values using suitably axiomatized functions. Every datatype value is generated by some constructor, but unrestricted use of this property can lead to expensive and unfruitful case distinctions in the SMT solver, so Dafny encodes this property only in certain places [19]. The property does tend to be useful when proving properties inductively, so Dafny makes the case distinction available to the SMT solver when applying the Induction Translation.

For example, suppose n in (2) is of the type `List` defined in Sec. 2.3. Then Dafny includes in the Induction Translation an additional antecedent:

$$n = \mathit{Nil} \quad \vee \quad \exists x, xs \bullet n = \mathit{Cons}(x, xs)$$

[1] For reasoning about termination of loops and recursive calls, the lower bound can be adjusted, because Dafny supports programmer-supplied variant expressions [19].

Actually, Dafny distributes these cases and produces one assert statement for each constructor, because this allows Dafny to give more precise error messages. So, the Induction Translation will produce the following 3 statements:

$$\begin{array}{l}\textbf{assert}\ \forall n \bullet\ (\forall k \bullet\ k \prec n \implies P(k)) \wedge n = Nil \implies P(n);\\ \textbf{assert}\ \forall n \bullet\ (\forall k \bullet\ k \prec n \implies P(k)) \wedge (\exists x, xs \bullet\ n = Cons(x, xs))\\ \qquad\qquad \implies P(n);\\ \textbf{assume}\ \forall n \bullet\ P(n);\end{array}$$

If the first assertion cannot be proved, Dafny reports that (2) might not hold for values constructed by `Nil`; if the second assertion cannot be proved, Dafny reports that (2) might not hold for values constructed by `Cons`.

2.5 Multiple Bound Variables

If a proof obligation has multiple bound variables, Dafny evaluates the Induction Heuristic for each one. If the Induction Heuristic applies to any of the bound variables, then Dafny applies the Induction Translation for all the bound variables to which the Induction Heuristic applies, combining these into a lexicographic tuple.

For example, consider a proof obligation:

$$\forall x, y, z \bullet\ Q(x, y, z)$$

and suppose the Induction Heuristic applies to x and z (that is, both x and z are prominently featured in $Q(x, y, z)$ as arguments to recursive functions). Applying the Induction Translation, Dafny thus produces:

$$\forall x, y, z \bullet\ (\forall k, m \bullet\ (k, m) \prec (x, z) \implies Q(k, y, m)) \implies Q(x, y, z)$$

where the definition of $(k, m) \prec (x, z)$ is the $\prec$ ordering on lexicographic pairs:

$$k \prec x \quad \vee \quad (k = x \wedge m \prec z)$$

Dafny only applies the Induction Translation to quantifiers that appear as positive top-level conjuncts of proof obligations. In particular, the Induction Translation is not applied to nested quantifiers. For example, if the proof obligation above had been formulated as:

$$\forall x, y \bullet\ \forall z \bullet\ Q(x, y, z)$$

then the Induction Translation would be:

$$\forall x, y \bullet\ (\forall k \bullet\ k \prec x \implies \forall z \bullet\ Q(k, y, z)) \implies \forall z \bullet\ Q(x, y, z)$$

2.6 Overriding the Induction Heuristic

As I report in Sec. 4, the Induction Heuristic seems to work well. However, there are times when one may wish to force or suppress the Induction Translation. Dafny has a general mechanism for hanging custom attributes in various places in the source code (akin to custom attributes in .NET and annotations in Java). One of those places is in

quantifiers, just after the declaration of the bound variables. While use of such custom attributes is rare, it is sometimes a convenient feature to have.

Dafny supports an `:induction` attribute. Used with no arguments, this attribute says to apply the Induction Translation to all of the quantifier's bound variables. The `:induction` attribute can also be used by listing those bound variables to which the Induction Translation should apply. Finally, `:induction false` says not to apply the Induction Translation to the quantifier.

For example, consider the following method declaration:

```
ghost method AdjacentImpliesTransitive(s: seq<int>)
  requires ∀ i ● 1 ≤ i < |s| ⟹ s[i-1] ≤ s[i];
  ensures ∀ i,j {:induction j} ● 0 ≤ i < j < |s| ⟹ s[i] ≤ s[j];
{ }
```

The postcondition (keyword **ensures**) follows from the precondition (keyword **requires**), but proving that necessitates induction. Since this example does not involve any recursive functions, the Induction Heuristic does not apply. However, the custom attribute used in this example tells Dafny to apply the Induction Translation for bound variable `j`, which leads to a successful verification of the postcondition. Alternatively, using the attribute `{:induction i,j}` or simply `{:induction}` also leads to a successful proof. Note, however, that `{:induction i}` does not, since even proving the property for `i` being `0` requires induction on `j`.

3 Examples

In this section, I show four applications of the induction tactic in the program verifier: two are used to verify programs, one to verify a simple mathematical property, and one to verify properties of functions over inductive datatypes. I also show an application that exemplifies the operation of the SMT solver given the formula produced by the program verifier.

3.0 A Simple Program

The program in Fig. 0 shows a Dafny method. It takes an array `a` as an in-parameter and returns an integer `r` as an out-parameter. As specified by the postcondition, the method returns an index where the array is `0`, or returns `-1` if the array does not contain a `0`. As specified by the precondition, the array has non-negative elements and has the special property that an element `a[i]` is not much smaller than its preceding neighbor, `a[i-1]`. In particular, if `a[i]` is smaller than `a[i-1]`, then it is smaller only by `1`. This property allows the method implementation to do better than linear search, for if the array element at the current position `n` is non-zero, then the next possible zero occurs `a[n]` array elements later.

The correctness of the method implementation hinges on the fact that the update `n := n + a[n];` maintains the loop invariant, which in turn follows from the special property of the array. However, the special property needs to be applied repeatedly for the proof, which does not happen automatically. Instead, the programmer supplies

```
method FindZero(a: array<int>) returns (r: int)
  requires a ≠ null ∧ ∀ i • 0 ≤ i < a.Length ⟹ 0 ≤ a[i];
  requires ∀ i • 0 ≤ i-1 ∧ i < a.Length ⟹ a[i-1]-1 ≤ a[i];
  ensures 0 ≤ r ⟹ r < a.Length ∧ a[r] = 0;
  ensures r < 0 ⟹ ∀ i • 0 ≤ i < a.Length ⟹ a[i] ≠ 0;
{
  var n := 0;
  while (n < a.Length)
    invariant ∀ i • 0 ≤ i < n ∧ i < a.Length ⟹ a[i] ≠ 0;
  {
    if (a[n] = 0) { return n; }
    assert ∀ m {:induction} •
                n ≤ m < n + a[n] ∧ m < a.Length ⟹ n+a[n]-m ≤ a[m];
    n := n + a[n];
  }
  return -1;
}
```

Fig. 0. A Dafny method that finds the index of a `0` in a given array `a`. Because the array has the special property that successive array elements do not decrease quickly (as stated by the second **`requires`** clause), the search is sub-linear (see the increase of `n`). The **`assert`** statement is proved by induction and then used to show the correctness of the program.

a lemma, here phrased as a Dafny **`assert`** statement. Since no recursive function is involved, the Induction Heuristic does not apply. Instead, the programmer uses the *{:induction}* attribute to indicate that the quantifier is to be proved using induction.

Feeding the program in Fig. 0 to the Dafny verifier proves the program (with no further user interaction) instantly (in 0.04 seconds on a single thread on a modern laptop with an Intel Core i7-M620 clocked at 2.67 GHz and running 64-bit Windows 7).

3.1 A Difficult Program

Floyd's "tortoise and hare" algorithm is a simple method for detecting whether a given linked-list node reaches a cycle or reaches **`null`** [16]. Its formal proof is not equally simple. One Dafny program for the algorithm, its specification (here shown slightly simplified)

```
method TortoiseAndHare() returns (reachesCycle: bool)
  ensures reachesCycle ⟺
    ∃ n • n ≠ null ∧ Reaches(n) ∧ n.next ≠ null ∧ n.next.Reaches(n);
```

and supporting definitions, and the lemmas needed for its verification is 233 (non white-space) lines long, of which 16 lines get compiled.[2] More than half of dozen of the quantified formulas given as lemmas require the Induction Translation to be verified. For example, one of them (slightly simplified) is:

[2] See the `FloydCycleDetect.dfy` program in the `Test/dafny2` folder of the `boogie.codeplex.com` source repository.

```
function Sum(n: nat): nat { if n = 0 then 0 else Sum(n-1) + n }
function CubeSum(n: nat): nat { if n = 0 then 0 else CubeSum(n-1) + n*n*n }
ghost method ArithmeticTheorem()
  ensures ∀ n: nat • CubeSum(n) = Sum(n) * Sum(n)  ∧  2*Sum(n) = n*(n+1);
{ }
```

Fig. 1. A Dafny encoding of the arithmetic theorem $\sum_{i=0}^{n} i^3 = (\sum_{i=0}^{n} i)^2$. Because of the induction tactic, the theorem is proved automatically.

```
assert ∀ j • 0 ≤ j ⟹ Nexxxt(x).Nexxxt(j) = Nexxxt(x + j);
```

where `Nexxxt(k)` returns the node obtained after `k` applications of the `.next` field. The program relies entirely on the Induction Heuristic and does not use any occurrence of the *{:induction}* attribute. The verification of the entire program takes just under 60 seconds.

3.2 Integers

Figure 1 states a familiar theorem about arithmetic. Functions `Sum` and `CubeSum` are defined recursively, and the theorem `CubeSum(n) = Sum(n) * Sum(n)` is stated as the postcondition of a method. Proving this arithmetic equality requires an additional fact about function `Sum`, which is also stated in the postcondition. Thus, the proof does what in mathematics is known as simultaneous induction.

The program in Fig. 1 is verified as shown (and with no further user interaction) in 0.09 seconds.

Remark: Alternatively, the closed-form property of `Sum` could have been given as a postcondition of function `Sum`:

```
function Sum(n: nat): nat
  ensures 2*Sum(n) = n*(n+1);
{ if n = 0 then 0 else Sum(n-1) + n }
```

The proof of this postcondition does not require the Induction Translation; the standard rules for postconditions and reasoning about (recursive) calls suffice.

3.3 Inductive Datatypes

Inductive datatypes are common in functional languages and in interactive proof assistants like Isabelle/HOL, Coq, PVS, and ACL2, which are based around functional languages. When proving properties of functions over such datatypes, it is natural to use induction. Dafny also supports inductive datatypes as well as user-defined recursive functions. Figure 2 defines two simple datatypes and three functions, the structure of whose definitions is representative of functions on inductive datatypes. Method `P19` states a theorem about these functions. The theorem is proved automatically (in 0.025 seconds), thanks to the induction tactic.

```
datatype Nat = Zero | Suc(Nat);
datatype List = Nil | Cons(Nat, List);
function minus(x: Nat, y: Nat): Nat {
  match x
  case Zero ⇒ Zero
  case Suc(a) ⇒ match y
    case Zero ⇒ x
    case Suc(b) ⇒ minus(a, b)
}
function len(xs: List): Nat {
  match xs  case Nil ⇒ Zero  case Cons(y, ys) ⇒ Suc(len(ys))
}
function drop(n: Nat, xs: List): List {
  match n
  case Zero ⇒ xs
  case Suc(m) ⇒ match xs
    case Nil ⇒ Nil
    case Cons(x, tail) ⇒ drop(m, tail)
}
ghost method P19()
  ensures ∀ n, xs • len(drop(n, xs)) = minus(len(xs), n);
{ }
```

Fig. 2. Two user-defined inductive datatypes in Dafny along with three functions defined on those datatypes. The postcondition of the method gives a theorem, which is proved automatically by Dafny's induction tactic.

3.4 Operation of the SMT Solver

As I have shown, the induction tactic is encoded in the translation from Dafny into Boogie, that is, the translation from the source language to the intermediate verification language. After that, Boogie and Z3 operate as usual. In other words, the induction tactic does not require any change to Boogie or Z3. Let us take a look at an example end to end, that is, from Dafny to Boogie to Z3 and let us also consider the operation of Z3 on its given verification condition.

Suppose we start with the following recursive function in Dafny:

```
function Fac(n: int): int { if n ≤ 1 then 1 else Fac(n-1) * n }
```

Dafny's translation of this function sets up proof obligations that will check that the function is well defined. Mimicking the Dafny function definition, the translation also includes the following Boogie declarations:

$$\begin{array}{l}\textbf{function } \mathit{Fac}(n\colon \textbf{int}) : \textbf{int};\\ \textbf{axiom } (\forall n\colon \textbf{int} \bullet \ \{\mathit{Fac}(n)\}\\ \qquad \mathit{Fac}(n) = (\textbf{if } n \leqslant 1 \textbf{ then } 1 \textbf{ else } \mathit{Fac}(n-1) * n));\end{array}$$

where the expression in curly braces specifies the *matching trigger* for the universal quantifier. The matching trigger tells the SMT solver how to select instantiations for the

quantifier [10]. This definition of Fac in Boogie is a bit of a simplification, because Dafny also takes some measures that will reduce the chances of running into matching loops, where the SMT solver would keep instantiating universal quantifiers forever.

Suppose further that the proof obligation is to show that `Fac` only returns positive integers, as expressed in Dafny by the following lemma:

```
ghost method FacPos() ensures ∀ n • 1 ≤ Fac(n);  { }
```

The Dafny verifier detects in this postcondition proof obligation that the bound variable `n` in passed as an argument to a recursive function. So the Induction Heuristic applies and Dafny applies the Induction Translation, obtaining the following Boogie statements (in the Boogie procedure corresponding to the Dafny method `FacPos`):

$$\begin{array}{l}\textbf{assert } (\forall n\colon \textbf{int} \bullet\ (\forall k\colon \textbf{int} \bullet\ 0 \leqslant k \wedge k < n \implies 1 \leqslant Fac(k)) \\ \qquad\qquad \implies\ 1 \leqslant Fac(n)); \\ \textbf{assume } (\forall n\colon \textbf{int} \bullet\ 1 \leqslant Fac(n));\end{array}$$

Boogie then translates this into Z3 input, which essentially amounts to:

$$\begin{array}{l}(\forall n\colon \textbf{int} \bullet\ \{Fac(n)\}\ Fac(n) = (\textbf{if } n \leqslant 1 \textbf{ then } 1 \textbf{ else } Fac(n-1) * n)) \\ \implies \\ (\forall n\colon \textbf{int} \bullet\ (\forall k\colon \textbf{int} \bullet\ 0 \leqslant k \wedge k < n \implies 1 \leqslant Fac(k)) \implies 1 \leqslant Fac(n))\end{array}$$

When Z3 tries to prove that this formula is valid, it negates it and starts looking for a satisfying assignment to the negation. The negation produces the following two conjuncts:

$$\begin{array}{l}(\forall n\colon \textbf{int} \bullet\ \{Fac(n)\}\ Fac(n) = (\textbf{if } n \leqslant 1 \textbf{ then } 1 \textbf{ else } Fac(n-1) * n)) \\ (\exists n\colon \textbf{int} \bullet\ (\forall k\colon \textbf{int} \bullet\ 0 \leqslant k \wedge k < n \implies 1 \leqslant Fac(k)) \wedge \neg(1 \leqslant Fac(n)))\end{array}$$

Z3 then Skolemizes the existential, calling it, say, Sk. Since no matching trigger was indicated for the quantifier over k in the Z3 input, Z3 will at this time select a matching trigger for it; here, I show that selected trigger explicitly:

$$\begin{array}{l}(\forall n\colon \textbf{int} \bullet\ \{Fac(n)\}\ Fac(n) = (\textbf{if } n \leqslant 1 \textbf{ then } 1 \textbf{ else } Fac(n-1) * n)) \\ (\forall k\colon \textbf{int} \bullet\ \{Fac(k)\}\ 0 \leqslant k \wedge k < Sk \implies 1 \leqslant Fac(k)) \\ Fac(Sk) < 1\end{array} \tag{6}$$

At this time, the presence of the term $Fac(Sk)$ will cause both quantifiers to be instantiated, with $n := k$ and $k := Sk$, yielding:

$$\begin{array}{l}\text{conjuncts (6)} \\ Fac(Sk) = (\textbf{if } Sk \leqslant 1 \textbf{ then } 1 \textbf{ else } Fac(Sk-1) * Sk) \\ 0 \leqslant Sk \wedge Sk < Sk \implies 1 \leqslant Fac(Sk)\end{array}$$

The last of these formulas will evaporate, since Z3 knows that $Sk < Sk$ is unsatisfiable. Z3 will now do a case distinction on the **if** expression.

For the **then** case, it gets:

$$\begin{array}{l}\text{conjuncts (6)} \\ Sk \leqslant 1 \\ Fac(Sk) = 1\end{array}$$

which (from $Fac(Sk) < 1$ and $Fac(Sk) = 1$) it will realize is unsatisfiable.

For the **else** case:

conjuncts (6)
$1 < Sk$
$Fac(Sk) = Fac(Sk - 1) * Sk$

There is now a new term, $Fac(Sk - 1)$, which matches the given triggers, so Z3 will produce two more instantiations, namely with $n := Sk - 1$ and $k := Sk - 1$:

conjuncts (6)
$1 < Sk$
$Fac(Sk) = Fac(Sk - 1) * Sk$
$Fac(Sk - 1) = (\mathbf{if}\ Sk - 1 \leqslant 1\ \mathbf{then}\ 1\ \mathbf{else}\ Fac(Sk - 1 - 1) * (Sk - 1))$
$0 \leqslant Sk - 1 \wedge Sk - 1 < Sk \Longrightarrow 1 \leqslant Fac(Sk - 1)$

By $1 < Sk$, the antecedent $0 \leqslant Sk - 1 \wedge Sk - 1 < Sk$ simplifies to $true$, and thus the consequent $1 \leqslant Fac(Sk - 1)$ emerges as a fact. Z3 now realizes that the conjuncts:

$Fac(Sk) < 1$
$1 < Sk$
$Fac(Sk) = Fac(Sk - 1) * Sk$
$1 \leqslant Fac(Sk - 1)$

are unsatisfiable, since the product of two positive numbers is not non-positive.

And that completes the proof.

4 Evaluation on a Test Suite for Induction

Theorem `P19` in Fig. 2 is part of a test suite for automatic induction, collected and used by the authors of the IsaPlanner system to evaluate their technique [14]. The suite contains 87 problems, of which IsaPlanner (which uses rippling [6] and an analysis of case statements) automatically solves the first 47. Beyond problem 47, the problems require various forms of abstraction, strengthenings, and new-lemma discovery. According to a paper on Zeno [24], ACL2s [7] automatically proves 74 of the 87 problems (with manually supplied type information) and the Zeno tool automatically proves 82 of them. The report on Zeno provides a detailed comparison of these tools on the test suite and some other problems [24].

Of the 47 problems that IsaPlanner can prove, Dafny can prove 45. For all the problems, the Induction Heuristic applies, so Dafny automatically uses the Induction Translation. For an induction tactic that is as simple as the one in Dafny, proving 45 of the 47 problems that IsaPlanner can solve seems surprisingly good.

Like the other tools, most of the proofs are instantaneous. Dafny spends 0.21 seconds on problem 39, 0.17 seconds on problem 45, and less than 0.10 seconds for each of the others.

Dafny cannot automatically prove problem 47 (and neither can ACL2s). However, problem 47 is verified using problem 23 as a lemma:

```
ghost method P47() ensures ∀ a • height(mirror(a)) = height(a); {
  P23();  // invoke the statement of problem 23 as a lemma
}
```

(I am unsure whether each problem in the test suite is allowed to use the preceding problems as lemmas. If so, Dafny also proves this one. Dafny proves each of the other 45 problems independently of each other.)

The other problem of the 47 that Dafny cannot automatically prove is problem 20. It requires the use of the preceding problem 15 as a lemma, and also requires an additional case distinction, which needs to be supplied manually:

```
ghost method P20()
  ensures ∀ xs • len(sort(xs)) = len(xs);
{
  P15();  // invoke the statement of problem 15 as a lemma
  // and manually introduce a case distinction:
  assert ∀ ys • sort(ys) = Nil ∨ ∃ z,zs • sort(ys) = Cons(z, zs);
}
```

The verification of this method requires 0.39 seconds.

5 Induction for Ghost Methods

The lemma expressed by the postcondition of method `P19` in Fig. 2 says something about all `n` and `xs`. Alternatively, the following method:

```
ghost method P19'(n: Nat, xs: List)
  ensures len(drop(n, xs)) = minus(len(xs), n);
{ }
```

states the same property, but just for the particular (but arbitrary) parameters `n` and `xs`. By universal generalization, the two methods express the same thing. Therefore, if the program verifier can prove `P19` automatically, we would expect it also to be able to prove `P19'` automatically.

Dafny's translation of the method makes this possible: at the beginning of the body of `P19'`, it effectively inserts recursive calls to `P19'` for all values of the parameters that satisfy the method's precondition (here, just **`true`**) and are smaller than the given `n,xs`. This *induction translation for methods* is analogous to the Induction Translation for quantifiers, and the heuristic for when to apply this method translation is also analogous to the Induction Heuristic for quantifiers. Two differences are noteworthy. One is that the "smaller" ordering on parameter values is determined by the method's (implicit or explicit) variant expression (see [19]), as required for the recursive calls to terminate. The other difference is that instead of inserting an induction-hypothesis antecedent, the induction translation for methods inserts code. Inserting that many calls could severely degrade the performance of a program, but Dafny performs this translation only for result-less effect-free ghost methods, so there is no impact on the program's run-time performance.

```
ghost method Lemma_RevConcat(xs: List, ys: List)
  ensures reverse(concat(xs, ys)) = concat(reverse(ys), reverse(xs));
{
  match (xs) {
    case Nil ⇒  assert ∀ ws  •  concat(ws, Nil) = ws;
    case Cons(t, rest) ⇒
      assert ∀ a,b,c  •  concat(a, concat(b, c)) = concat(concat(a, b), c);
} }
```

Fig. 3. A lemma about the list reversal and concatenation operations (whose standard recursive definitions are elided from the figure). Dafny automatically verifies the two assert statements and the postcondition, which altogether require 3 appeals to induction.

Thanks to the induction translation for methods, `P19'` verifies as given above (in 0.01 seconds). A more interesting example, which illustrates the induction translation for both quantifiers and methods, is shown in Fig. 3. First, the proof of the postcondition requires the properties that `Nil` is a right unit of `concat` (first **`assert`**) and that `concat` is associative (second **`assert`**), both of which are handled by the Induction Translation. Second, the proof of the postcondition requires induction on `xs` (in particular, it requires knowing the postcondition for `rest,ys` in the `Cons` case), which is handled by the induction translation for methods. Method `Lemma_RevConcat` is verified as given in the figure in 0.10 seconds.

6 Conclusion

In conclusion, the simple and straightforward (some may say brute force) approach of inserting induction-hypothesis assumptions at heuristically chosen points in the verification conditions passed to the SMT solver seems to be a win. Without unduly cluttering up the source program, it automatically sets up the induction for the SMT solver, and the SMT solver seems to do well at discharging the resulting proof obligations. The soundness of the approach is justified by a simple appeal to the Induction Principle.

The tactic performs on par with one serious participant in the quest for automatic induction, and yet is far simpler. The tactic introduces SMT solvers as a powerful workhorse in the arena of induction solvers.

I have described the induction tactic as used in a program verifier. However, the same tactic could be implemented in other settings, for example in a full-fledged proof assistant, perhaps as part of Isabelle's "sledgehammer" tactic [4].

Approaches to automatic induction are often accompanied by techniques for lemma discovery. It would be interesting to investigate how they could be incorporated in the context of an SMT solver or a program verifier like Dafny.

Acknowledgments. I am grateful to Will Sonnex, Sophia Drossopoulou, and Susan Eisenbach for their inspiring work on Zeno and for their encouraging comments. Thanks also to Bart Jacobs and Jan Smans who, during a recent visit of mine to Leuven, joined

in a stint to learn more about automatic induction by rolling up their sleeves and coding up a little prototype solver. That effort led to me to wanting better support for induction in Dafny, which in turn led to the work presented in this paper. I appreciate the valuable comments by Jean-Christophe Filliâtre and the referees on an earlier version of this paper. Finally, I am indebted to Michał Moskal for serving as a sounding board for the ideas herein.

References

0. Back, R.-J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science. Springer, Heidelberg (1998)
1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004)
4. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
5. Boyer, R.S., Moore, J.S.: A Computational Logic. ACM Monograph Series. Academic Press (1979)
6. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-level Guidance for Mathematical Reasoning. Cambridge Tracts in Theoretical Computer Science, vol. 56. Cambridge University Press (2005)
7. Chamarthi, H.R., Dillinger, P.C., Manolios, P., Vroon, D.: The ACL2 Sedan Theorem Proving System. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 291–295. Springer, Heidelberg (2011)
8. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
9. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM 52(3), 365–473 (2005)
11. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
13. Jacobs, B., Smans, J., Piessens, F.: VeriFast: Imperative programs as proofs. In: VSTTE Workshop on Tools & Experiments (2010)
14. Johansson, M., Dixon, L., Bundy, A.: Case-Analysis for Rippling and Inductive Proof. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 291–306. Springer, Heidelberg (2010)

15. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
16. Knuth, D.E.: The Art of Computer Programming. Seminumerical Algorithms, vol. II. Addison-Wesley (1969)
17. Leino, K.R.M.: This is Boogie 2. Technical report, Microsoft Research (2008)
18. Leino, K.R.M.: Specification and verification of object-oriented software. In: Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press (2009); Summer School Marktoberdorf 2008 lecture notes
19. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
20. Leino, K.R.M., Rümmer, P.: A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
21. Leino, K.R.M., Saxe, J.B., Stata, R.: Checking Java programs via guarded commands. In: Formal Techniques for Java Programs, Technical Report 251. Fernuniversität Hagen (1999); Also available as Technical Note 1999-002, Compaq Systems Research Center
22. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
23. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE-11 1992. LNCS (LNAI), vol. 607, pp. 748–752. Springer, Heidelberg (1992)
24. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, Imperial College London (2011), http://pubs.doc.ic.ac.uk/zeno/
25. Walther, C., Schweitzer, S.: About VeriFun. In: Baader, F. (ed.) CADE-19 2003. LNCS (LNAI), vol. 2741, pp. 322–327. Springer, Heidelberg (2003)
26. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI 2008, pp. 349–361. ACM (2008)