Verified Calculations

K. Rustan M. Leino[0] and Nadia Polikarpova[1]

[0] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[1] ETH Zurich, Zurich, Switzerland
nadia.polikarpova@inf.ethz.ch

Manuscript KRML 231, 8 March 2013.

Abstract. Calculational proofs—proofs by stepwise formula manipulation—are praised for their rigor, readability, and elegance. It seems desirable to reuse this style, often employed on paper, in the context of mechanized reasoning, and in particular, program verification.
This work leverages the power of SMT solvers to machine-check calculational proofs at the level of detail they are usually written by hand. It builds the support for calculations into the programming language and auto-active program verifier Dafny. The paper demonstrates that calculations integrate smoothly with other language constructs, producing concise and readable proofs in a wide range of problem domains: from mathematical theorems to correctness of imperative programs. The examples show that calculational proofs in Dafny compare favorably, in terms of readability and conciseness, with arguments written in other styles and proof languages.

0 Introduction

There is no automatic way to decide, for any given statement, if it holds or not. Both in mathematics and in program verification, people construct proofs to convince themselves and others of the validity of statements, as well as to gain a better understanding of those statements and why they hold. Naturally, people strive to come up with a good notation for writing proofs: one that is easy to read and understand, and, if possible, guards against mistakes.

One such notation is the *calculational method* [3], whereby a theorem is established by a chain of formulas, each transformed in some way into the next. The relationship between successive formulas (for example, equality or implication) is notated, and so is a *hint* that justifies the transformation (see Fig. 3 in Sec. 1.1 for an example). The calculational method encourages making the transformation steps small in order to help the author avoid mistakes and to convince readers by fully explaining what is going on.

Even though calculational proofs are written in a strict format, they still tend to be informal. Manolios and Moore [23] discovered several errors in calculations written by Dijkstra (one of the biggest proponents of the style) and posed a challenge to mechanize calculational proofs, asking if "any attempt to render calculational proofs suitable for mechanical checking will result in ugly, hard to comprehend proofs" and if "the mechanized proof checker [will] necessarily be hard to learn and difficult to use".

Indeed, constructing proofs within *interactive proof assistants* (like Coq [7], Isabelle/HOL [24], or PVS [25]) has a reputation of being a demanding task[0]. The purpose of these tools is to give the user maximum control over the proof process, which is why they favor predictability over automation. Interaction with such an assistant consists of issuing low-level commands, called *tactics*, that guide the prover through the maze of proof states. This mode of interaction is biased towards expert users with a good knowledge of the tool's inner workings.

In contrast, *auto-active* program verifiers [22], like Dafny [20], VCC [11], or VeriFast [16], take a different approach to mechanised reasoning: they provide automation by default, supported by an underlying SMT solver. All the interaction with such a verifier happens at the level of the programming language, which has the advantage of being familiar to the programmer. So far, these tools have been used mostly for verifying functional correctness of programs, but their domain is gradually expanding towards general mathematical proofs.

It would be wonderful if we could just take a pen-and-paper calculational proof and get it machine-checked completely automatically. In this paper, we add support for proof calculations to the programming language and auto-active program verifier Dafny [20, 19]. The extension, which we call *program-oriented calculations* (poC), is able to verify calculational proofs written with the same level of detail and the same degree of elegance as they would be on paper, thereby addressing the challenge of Manolios and Moore.

The main contributions of the present work are as follows. We integrate proof calculations as a statement in a programming language. The supporting syntax uses, for structuring proofs, constructs already familiar to programmers (such as conditionals and method calls). We develop tool support for machine-checking the proofs. Thanks to the underlying SMT solver, the resulting tool provides a high degree of automation. We provide a sound encoding of the new statement that tries to reduce the overhead of specifying that formulas are well defined (in the presence of partial expressions). We give a number of examples that use the proof format and provide a comparison with proof structuring in existing tools. And by adding this streamlined proof support to an auto-active program verifier, we are bringing SMT-based tools closer to what previously has been territory exclusive to interactive proof assistants.

1 Background and Motivation

This section reviews existing features for writing proofs in auto-active program verifiers and then argues in favor of adding support for calculations. While we use Dafny in our examples, other auto-active verifiers have a similar range of features.

1.0 Proofs in an Auto-Active Program Verifier

Automatic verification attempts may fail for several reasons. One reason is that the program correctness may depend on mathematical properties that the program verifier

[0] A popular quote from an old version of the PVS website states that it takes six months to become a moderately skilled user [28].

```
datatype List⟨T⟩ = Nil | Cons(T, List⟨T⟩);

function length(xs: List): nat {
  match xs
  case Nil ⇒ 0
  case Cons(x, xrest) ⇒ 1 + length(xrest)
}
```

Fig. 0. Definition of a generic datatype `List⟨T⟩` and a function that returns the length of a list. We omit the type parameter of `List` where Dafny can infer it automatically, as for function `length` here.

is unable to figure out by itself. In such cases, the user of an auto-active verifier may augment the program text with *lemmas*, which will give the verifier clues about how to proceed.

The simplest form of a lemma is an **assert** statement. It directs the verifier's focus by adding a proof obligation that subsequently can be used as an assumption. As an example, consider an algebraic `List` datatype with a `length` function in Fig. 0. Suppose we want to prove that there exists a list of length one:

$$\exists\ \texttt{xs}\ \bullet\ \texttt{length(xs)} = 1$$

The proof of such an existential property requires supplying the witness manually, and we can do so by inserting the following assert statement:

```
assert length(Cons(496, Nil)) = 1;
```

In more complex cases, an **assert** statement is not enough, perhaps because we want to reuse a multiple-step proof, or, in case of an inductive argument, because we want to apply the same proof recursively. For example, suppose we want to generalize our previous statement and assert that there exists a list of any non-negative length:

$$\forall\ \texttt{n}\ \bullet\ 0 \leq \texttt{n} \Longrightarrow \exists\ \texttt{xs}\ \bullet\ \texttt{length(xs)} = \texttt{n}$$

Since this property may be useful in several places, we would like to state and prove it once as a theorem and later be able to refer to it by name.

To state a theorem in the programming language, we declare a method (see Fig. 1) whose postcondition is the conclusion of the theorem, whose precondition is the assumption made by the theorem, and whose in-parameters show how the theorem can be parameterized. Since we intend this method only for the program verifier, we declare it to be *ghost*, which means that the compiler will not generate any code for it.

To use a lemma stated in this way, the program then simply calls the method, upon return of which the caller gets to assume the postcondition (as usual in program verification). Given a method declaration, the program verifier will, as usual, set out to prove that the method body is terminating and that it terminates with the given postcondition. Thus, the method body essentially gives the proof of the theorem.

In Fig. 1, the method body introduces two cases, $\texttt{n} = 0$ and $\texttt{n} \neq 0$. In the first case, the verifier can automatically come up with the witness using the definition of

```
ghost method LemmaLength(n: int)
  requires n ≥ 0;
  ensures ∃ xs • length(xs) = n;
{
  if (n = 0) {
    // trivial
  } else {
    LemmaLength(n - 1); // invoke induction hypothesis
    var xs :| length(xs) = n - 1;
    assert length(Cons(496, xs)) = n;
} }
```

Fig. 1. The ghost method states a theorem and its body gives the proof. More precisely, the body of the method provides some code that convinces the theorem prover that all control-flow paths terminate in a state where the postcondition holds.

`length`, and therefore no further code needs to be given in the **`then`** branch. The **`else`** branch performs a method call, so the verifier checks that the callee's precondition holds. Furthermore, since the call is recursive, the verifier needs to check termination, which it does (in the standard program-verification way) by checking that some variant function decreases in some well-founded ordering. In this example, the default variant function—the argument `n`—is good enough, but in general it may need to be supplied by the user. Upon return from the call, the verifier assumes the postcondition of the call: `∃ xs • length(xs) = n - 1`. With this fact at hand, we use the Dafny "`:|`" ("assign such that") operator to save an arbitrary list of length `n - 1` in a local variable `xs`. From that list, we construct one of length `n`, whereby convincing the verifier that the enclosing postcondition holds. The body of method `LemmaLength` is essentially a proof by induction, where the recursive call denotes the appeal to the induction hypothesis.

The example above illustrates how auto-active program verifiers exploit the analogy between programs and proofs to reuse programming constructs for proof structuring: **`if`** (and **`match`**/**`case`**) statements for case splits, procedures for stating and reusing lemmas, recursion for inductive arguments. This spares programmers the effort of learning a new language and, more importantly, a new paradigm.

To convince the verifier of some non-trivial fact, one may need to appeal to multiple lemmas (either **`assert`** statements or method calls), and the discovery of which lemmas are useful is non-trivial. One way to discover what is needed is to start writing a detailed proof, proceeding in small steps and trying to figure out in which step the verifier gets stuck. It is for writing such detailed proofs that poC provides helpful features. Once a detailed proof has been constructed, one often finds that the tool would have been satisfied with some smaller set of lemmas; the user then has the choice of either deleting the steps that are not needed or keeping the proof steps, which may facilitate better readability and maintainability.

```
function reverse(xs: List): List {
  match xs
  case Nil ⇒ Nil
  case Cons(x, xrest) ⇒ append(reverse(xrest), Cons(x, Nil))
}
function append(xs: List, ys: List): List {
  match xs
  case Nil ⇒ ys
  case Cons(x, xrest) ⇒ Cons(x, append(xrest, ys))
}
```

Fig. 2. Definition of a function `reverse` that reverses the order of elements in a list and its helper function `append` that concatenates two lists.

1.1 Calculational Proofs

Having reviewed how manual proofs are supplied in an auto-active program verifier, let us now turn our attention to some properties that require a somewhat larger manual effort.

As a motivating example, we consider proving that reversing a list is an involution:

$$\forall\ \mathtt{xs}\ \bullet\ \mathtt{reverse(reverse(xs))} = \mathtt{xs}$$

There are systems for automatic induction that can prove this property automatically (*e.g.* [18, 10]), but it requires some manual effort in Dafny. The relevant definitions are found in Fig. 2, along with the definition of `List` from Fig. 0.

Here is how one would prove this fact by hand. Because the definition of `reverse` is inductive, it is natural that the proof should proceed by induction on `xs`. The base case $reverse(reserve(Nil)) = Nil$ holds trivially by definition of `reverse`. For the step case, we can write a little calculation shown in Fig. 3. This calculation consists of five *steps*, each stating an equality between two consecutive *lines* and accompanied by a *hint*, which explains why the equality holds. Typically, hints reference definitions or well-known properties of used operations, invoke auxiliary lemmas, or appeal to the induction hypothesis. The *conclusion* of this calculation is $reverse(reverse(x : xs)) = x : xs$ (the first line is equal to the last line), which holds due to transitivity of equality.

In Fig. 4, we show the same proof written in Dafny, in a “classical” style with each step represented by an **`assert`** statement. This proof is harder to read than the one in Fig. 3: it is less structured and more verbose (each inner line appears twice). In the next section, we show how poC improves the situation.

2 Calculations in Dafny

Using poC, we can write the proof of `LemmaReverseTwice` as shown in Fig. 5. The calculation is introduced using the **`calc`** statement, which adheres to the following gram-

$$\begin{array}{ll}
 & \mathit{reverse}(\mathit{reverse}(x :: \mathit{xs})) \\
= & \quad \{ \text{ def. } \mathit{reverse} \ \} \\
 & \mathit{reverse}(\mathit{reverse}(\mathit{xs}) +\!\!+ [x]) \\
= & \quad \{ \text{ lemma } \forall \mathit{xs}, \mathit{ys} \bullet \ \mathit{reverse}(\mathit{xs} +\!\!+ \mathit{ys}) = \mathit{reverse}(\mathit{ys}) +\!\!+ \mathit{reverse}(\mathit{xs}) \ \} \\
 & \mathit{reverse}([x]) +\!\!+ \mathit{reverse}(\mathit{reverse}(\mathit{xs})) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & \mathit{reverse}([x]) +\!\!+ \mathit{xs} \\
= & \quad \{ \text{ def. } \mathit{reverse}, \mathit{append} \ \} \\
 & [x] +\!\!+ \mathit{xs} \\
= & \quad \{ \text{ def. } \mathit{append} \text{ twice } \} \\
 & x :: \mathit{xs}
\end{array}$$

Fig. 3. Hand proof of $\mathit{reverse}(\mathit{reverse}(x :: \mathit{xs})) = x :: \mathit{xs}$ written in calculational style. The proof uses Coq-style notation with $x :: \mathit{xs}$ for `Cons(x, xs)`, $[x]$ for `Cons(x, Nil)`, and $\mathit{xs} +\!\!+ \mathit{ys}$ for `append(xs, xy)`.

mar:

$$\begin{array}{rcl}
\mathit{CalcStatement} & ::= & \text{“}\mathbf{calc}\text{”}\ \mathit{Op}^{?}\ \text{“\{”}\ \mathit{CalcBody}^{?}\ \text{“\}”} \\
\mathit{CalcBody} & ::= & \mathit{Line}\ (\mathit{Op}^{?}\ \mathit{Hint}\ \mathit{Line})^{*} \\
\mathit{Line} & ::= & \mathit{Expression}\ \text{“;”} \\
\mathit{Hint} & ::= & (\mathit{BlockStatement} \mid \mathit{CalcStatement})^{*} \\
\mathit{Op} & ::= & \text{“}{=}\text{”} \mid \text{“}{\leq}\text{”} \mid \text{“}{<}\text{”} \mid \text{“}{\geq}\text{”} \mid \text{“}{>}\text{”} \mid \text{“}{\Longrightarrow}\text{”} \mid \text{“}{\Longleftarrow}\text{”} \mid \text{“}{\Longleftrightarrow}\text{”} \mid \text{“}{\neq}\text{”}
\end{array}$$

Non-syntactic rules further restrict hints to only ghost and side-effect free statements, as well as impose a constraint that only *chain-compatible* operators can be used together in a calculation.

The notion of chain-compatibility is quite intuitive for the operators supported by poC; for example, it is clear that $<$ and $>$ cannot be used within the same calculation, as there would be no relation to conclude between the first and the last line. We treat this issue more formally in Sec. 3. Note that we allow a single occurrence of the intransitive operator $\neq$ to appear in a chain of equalities (that is, $\neq$ is chain-compatible with equality but not with any other operator, including itself).

Calculations with fewer than two lines are allowed, but have no effect. If a step operator is omitted, it defaults to the calculation-wide operator, defined after the **`calc`** keyword. If that operator if omitted, it defaults to equality.

In the following, we review more features and usage patterns of poC. Most of them we get for free, simply reusing language constructs already present in Dafny. This indicates that calculations integrate well with the rest of the language.

2.0 Contextual Information

It is often desirable to embed a calculation in some context (for example, when proving an implication, the context of the antecedent). Dijkstra gives an example [13] of how choosing an appropriate context can significantly simplify a calculational proof.

```
ghost method LemmaReverseTwice(xs: List)
  ensures reverse(reverse(xs)) = xs;
{
  match xs {
    case Nil ⇒
    case Cons(x, xrest) ⇒
      // by def. reverse, we have:
      assert reverse(reverse(xs))
        = reverse(append(reverse(xrest), Cons(x, Nil)));
      LemmaReverseAppendDistrib(reverse(xrest), Cons(x, Nil));
      assert reverse(append(reverse(xrest), Cons(x, Nil)))
        = append(reverse(Cons(x, Nil)), reverse(reverse(xrest)));
      LemmaReverseTwice(xrest); // induction hypothesis
      assert append(reverse(Cons(x, Nil)), reverse(reverse(xrest)))
        = append(reverse(Cons(x, Nil)), xrest);
      // by def. reverse and append:
      assert append(reverse(Cons(x, Nil)), xrest)
        = append(Cons(x, Nil), xrest);
      // by def. append applied twice:
      assert append(Cons(x, Nil), xrest)
        = xs;
} }

ghost method LemmaReverseAppendDistrib(xs: List)
  ensures reverse(append(xs, ys)) = append(reverse(ys), reverse(xs));
```

Fig. 4. Theorem that `reverse` is involutive stated and proven in Dafny without poC. The proof shown here makes use of an auxiliary lemma `LemmaReverseAppendDistrib`, which shows how `reverse` distributes over `append` (and whose proof we have omitted for brevity). The proof can be improved with poC.

Inside a Dafny method, a proof is always embedded in the context of the method's precondition; however, whenever additional context is required, this is easily achieved with a conventional **`if`** statement.

Fig. 6 gives an example of the same result being proven in two different ways: with and without additional context. Theorem `Monotonicity` states that for a function `f` of a natural argument, the "single step" characterization of monotonicity, *i.e.* $\forall$ `n • f(n)` $\leq$ `f(n + 1)` is as good as the more common characterization with two quantified variables: $\forall$ `a, b • a` $\leq$ `b` $\Longrightarrow$ `f(a)` $\leq$ `f(b)`. Because it has been embedded in a richer context, the calculation in the second proof in Fig. 6 has fewer steps and manipulates simpler terms than the one in the first proof.

2.1 Structuring Calculations

Since hints are statements, they can themselves contain calculations. This gives rise to nested calculations, akin to *structured calculational proofs* [1] and *window inference* [29]. In fact, since this is a common case, a hint that consists solely of a calculation

```
ghost method LemmaReverseTwice(xs: List)
  ensures reverse(reverse(xs)) = xs;
{
  match xs {
    case Nil ⇒
    case Cons(x, xrest) ⇒
      calc {
        reverse(reverse(xs));
      = // def. reverse
        reverse(append(reverse(xrest), Cons(x, Nil)));
      = { LemmaReverseAppendDistrib(reverse(xrest), Cons(x, Nil)); }
        append(reverse(Cons(x, Nil)), reverse(reverse(xrest)));
      = { LemmaReverseTwice(xrest); } // induction hypothesis
        append(reverse(Cons(x, Nil)), xrest);
      = // def. reverse, append
        append(Cons(x, Nil), xrest);
      = // def. append (x2)
        xs;
} }   }
```

Fig. 5. Proof that `reverse` is involutive using Dafny's **`calc`** statement.

need not be enclosed in curly braces. The example in Fig. 7 uses a nested calculation to manipulate a sub-formula of the original line, in order to avoid dragging along the unchanged part of the formula and focus the reader's attention on the essentials.

As demonstrated by Back *et al.* [1], nesting is a powerful mechanism for structuring calculational proofs, especially if a subderivation can be carried out in its own context. For example, the following structured calculation (where `s0` and `s1` are sets and ∩̸ says that two sets are disjoint) succinctly proves an implication by manipulating subterms of the consequent under the assumption of the antecedent:

```
calc {
  s0 ∩̸ s1 ⟹ f((s0 ∪ s1) ∩ s0) = f(s0);
  { if (s0 ∩̸ s1) {
      calc {
        (s0 ∪ s1) ∩ s0;
        (s0 ∩ s0) ∪ (s1 ∩ s0);
        // s0 ∩̸ s1
        s0;
  } } }
  s0 ∩̸ s1 ⟹ f(s0) = f(s0);
  true;
}
```

Another way of adding context is parametrization: Dafny's **`forall`** statement (which in programming is used to perform aggregate operations) makes it possible in proofs to introduce arbitrary values to be manipulated within a subderivation, whose conclusion

```
function f(n: nat): nat

ghost method Monotonicity(a: nat, b : nat)
  requires ∀ n: nat • f(n) ≤ f(n + 1); // (0)
  ensures a ≤ b ⟹ f(a) ≤ f(b);
  decreases b - a; // variant function
{
  // The first proof:                     // The second proof:
  calc ⟹ {                               if (a < b) {
    a < b;                                  calc ≤ {
    a + 1 ≤ b;                                f(a);
    { Monotonicity(a + 1, b); }               // (0)
    f(a + 1) ≤ f(b);                          f(a + 1);
    // (0)                                    { Monotonicity(a + 1, b); }
    f(a) ≤ f(a + 1) ≤ f(b);                   f(b);
    f(a) ≤ f(b);                            }
  }                                       }
}
```

Fig. 6. Theorem that connects different ways of expressing monotonicity and two proofs thereof. Both proofs establish $\mathtt{a} < \mathtt{b} \implies \mathtt{f(a)} \leq \mathtt{f(b)}$, which is sufficient since the $\mathtt{a} = \mathtt{b}$ case is trivial. The difference is that the second proof uses $\mathtt{a} < \mathtt{b}$ as a context for the calculation, which simplifies it significantly.

is then universally generalized. Fig. 8 gives an example. It shows one missing step of the extensionality proof of Fig. 7, namely the one that establishes the precondition for the invocation of the induction hypothesis:

$$\forall\ \mathtt{i}\ \bullet\ \mathtt{0} \leq \mathtt{i} < \mathtt{length(xrest)} \implies \mathtt{ith(xrest,\ i)} = \mathtt{ith(yrest,\ i)}$$

The **forall** statement in the figure introduces a local immutable variable i, whose values range over $[0, \mathit{length}(\mathit{xrest}))$. This block will automatically *export* (*i.e.* make available to the following statements) the conclusion of the nested calculation, quantified over the range of i, which amounts precisely to the missing precondition in Fig. 7.

3 Encoding

Since calculation statements are an incremental addition to Dafny, largely reusing existing constructs such as expressions and statements, the poC implementation required relatively little effort. This means that a similar feature can be easily added to other auto-active verifiers, especially ones based on verification engines like Boogie [5] or Why3 [8].

We explain the encoding of calculation statements in terms of their desugaring into **assert** and **assume** statements. The simplest approach would be to treat a **calc** statement merely as syntactic sugar for asserting all of its steps, which is how one would write such a proof in the absence of support for calculations (see Fig. 4). This approach

```
ghost method Extensionality(xs: List, ys: List)
  requires length(xs) = length(ys); // (0)
  requires ∀ i • 0 ≤ i < length(xs) ⟹ ith(xs, i) = ith(ys, i); // (1)
  ensures xs = ys;
{
  match xs {
    case Nil ⇒
    case Cons(x, xrest) ⇒
      match ys { case Cons(y, yrest) ⇒
        calc {
          xs;
          Cons(x, xrest);
          calc { // nested calculation
            x;
            ith(xs, 0);
            // (1) with i := 0
            ith(ys, 0);
            y;
          }
          Cons(y, xrest);
          { /* Show (1) for xrest, yrest -- omitted in this figure */
            Extensionality(xrest, yrest); }
          Cons(y, yrest);
          ys;
} }   } }
```

Fig. 7. Extract from a proof of extensionality of lists. A nested calculation is used to transform a subterm of a line. The omitted step is fleshed out in Fig. 8.

is trivially sound (since it does not introduce any **assume**s), but it has some practical disadvantages. First, with this encoding, the SMT solver will accumulate all the facts it learns from each calculation step and try to apply them while proving the next step; thus in a large proof, the solver is more likely to get "confused" or "bogged down". Second, the conclusion of the calculation is not stated explicitly, so it might not be readily available after the proof.

Instead, we would like to make use of the fact that steps in a calculational proof are conceptually independent. Our goal is to ask the SMT solver to verify each step in isolation, and then simply postulate the conclusion of the calculation, without wasting the solver's time on putting the steps together and reasoning about transitivity. To this end, we introduce the encoding in Fig. 9. The construct **if** `(*)` denotes non-deterministic choice, and the conclusion relation `C⟨N⟩` is computed from the sequence `S⟨0⟩`, ..., `S⟨N-1⟩` of the step operators. Ending each branch of the conditional with **assume false** prevents the control flow from exiting it, effectively telling the SMT solver to forget everything it has learned from the branch.

To determine the conclusion relation `C⟨N⟩`, following the Isabelle/Isar approach [6], we define a set of *transitivity rules* of the form $R \circ S \subseteq T$ (in the terminology of Sec. 2,

```
calc {
  xs;
  Cons(x, xrest);
  //...
  Cons(y, xrest);
  { forall (i: nat | i < length(xrest)) {
      calc {
        ith(xrest, i);
        ith(xs, i + 1);
        // enclosing precondition
        ith(ys, i + 1);
        ith(yrest, i);
      } }
    Extensionality(xrest, yrest);
  }
  Cons(y, yrest);
  ys;
}
```

Fig. 8. A missing step of the proof in Fig. 7. We use Dafny's **forall** statement to introduce a variable i in the nested calculation and then generalize its conclusion.

this makes R and S chain-compatible). One example of such a rule is $a = b \wedge b < c \Rightarrow a < c$. Fig. 10 summarizes the transitivity rules used in poC.

We define C_i to be the conclusion relation after the first i steps: C_0 is equality, and transitivity rules are used to compute C_{i+1} from the previous conclusion relation C_i and the step operator S_i. It is easy to see that if the rules are sound (which is trivial to show for all chain-compatible operators in poC), and the individual steps of a calculation are valid, then each intermediate conclusion $line_0 \ C_i \ line_i$, and thus the final conclusion, is also valid.

The above argument comes with two caveats for users of traditional proof assistants. First, Dafny is an imperative language and its expressions can depend on mutable state. Obviously, verifying calculation steps in one state and postulating its conclusion in another state can cause trouble. We require that all hints be side-effect free, enforced by a simple syntactic check, and thus the state is preserved throughout the calculation statement, which implies that the encoding in Fig. 9 is sound. Second, unlike most proof languages, functions in Dafny (including some useful built-in functions like sequence indexing) can be partial (*i.e.* they can have preconditions). Thus, we have to make sure not only that the conclusion of the calculation holds, but also that it is well-defined. This is discussed in the next section.

3.0 Partial Lines

Each Dafny expression e has an associated *well-formedness* condition wf[e], and each Dafny statement is responsible for checking well-formedness of all its sub-expressions (in case of the **calc** statement, all the lines). We modify the encoding in Fig. 9 to assert

```
calc {
  line⟨0⟩;
S⟨0⟩ { Hint⟨0⟩ }
  line⟨1⟩;

 ...

  line⟨N-1⟩;
S⟨N-1⟩ { Hint⟨N-1⟩ }
  line⟨N⟩;

}
```

```
if (*) {
  Hint⟨0⟩
  assert line⟨0⟩ S⟨0⟩ line⟨1⟩;
  assume false;
} else if (*) {
 ...
} else if (*) {
  Hint⟨N-1⟩
  assert line⟨N-1⟩ S⟨N-1⟩ line⟨N⟩;
  assume false;
}
assume line⟨0⟩ C⟨N⟩ line⟨N⟩;
```

Fig. 9. A **calc** statement (on the left) and its **assert/assume** desugaring (on the right), where C⟨N⟩ denotes the conclusion relation for the full N steps.

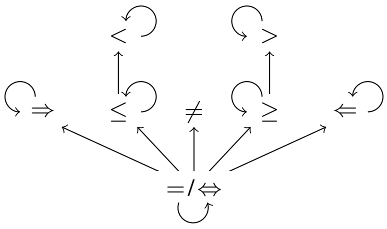

Fig. 10. Graphical representation of the transitivity rules for poC operators: For any two operators R and S such that S is reachable from R, add two rules: $R \circ S \subseteq S$ and $S \circ R \subseteq S$.

wf[line⟨0⟩] before the first branch, and for every step i assert wf[line⟨i+1⟩] after Hint⟨i⟩, since the hint might be useful in verifying well-formedness.

It gets more interesting with lines of type **bool**, because in Dafny some boolean operators, in particular implication, have short-circuiting (aka conditional) semantics, which is reflected in their well-formedness conditions:

$$wf[P \Rightarrow Q] \equiv wf[P] \wedge (P \Rightarrow wf[Q])$$

This allows one to write expressions like P(x) ⟹ Q(f(x)), where P is the precondition of function f.

It seems natural that implications should have the same effect in calculations. For example, the following calculation should be considered well-formed:

```
calc {
  P(x);
⟹ { /* Hint with a precondition P(x) */ }
  Q(f(x));
⟺
  R(f(x));
}
```

```
// Step "line⟨i⟩ ⟹ line⟨i+1⟩"
assume wf[line⟨i⟩];
assume line⟨i⟩;
Hint⟨i⟩
assert wf[line⟨i+1⟩];
assert line⟨i+1⟩;
assume false;
```

```
// Step "line⟨i⟩ S⟨i⟩ line⟨i+1⟩"
assume wf[line⟨i⟩];
Hint⟨i⟩
assert wf[line⟨i+1⟩];
assert line⟨i⟩ S⟨i⟩ line⟨i+1⟩;
assume false;
```

Fig. 11. Encoding of an implication-step (left) and any other step (right) that supports short-circuiting semantics. In addition, the `assert wf[line0];` is performed to check the well-formedness of `line0`.

The translation as defined so far will not accept this example. Moreover, the condition that each line be well-formed in the enclosing context is too strong in this case. Instead, we propose the following definition: a calculation is well-formed if each of its intermediate conclusions is well-formed.

The final version of our encoding, which allows the example to go through, is given in Fig. 11. It differentiates between $\Longrightarrow$[1] and other step operators, which are not short-circuiting in Dafny. We can show that this encoding implements our definition of well-formed calculations. For example, in a calculation with only implication steps, the intermediate conclusion is $wf[line_0] \wedge (wf[line_0] \wedge line_0 \Rightarrow wf[line_i] \wedge line_i)$, which is well-formed according to Dafny rules.

4 Experiments and Discussion

4.0 Case Studies

To confirm the usefulness of poC, we tried it on five examples from various domains[2]:

- `Identity`, `SingleFixpoint` and `KnasterTarski` are mathematical theorems. The former two state properties of functions over natural numbers and the combination of function iteration and fixpoints, respectively; the latter is one part of the Knaster-Tarski fixpoint theorem.
- `List` is a set of properties of inductively defined lists, such as `ReverseTwice` and `Extensionality` discussed above. This case study is representative of reasoning about functional programs.
- `MajorityVote` is an algorithm, due to Boyer and Moore, that finds the majority choice among a sequence of votes. This example is representative of verifying imperative programs.

In total, our case studies comprise about 650 lines of Dafny.

To evaluate the benefits of calculations, we developed a second version of each example, where all proofs are at the same level of detail, but using the traditional approach

[1] $\Longleftarrow$ is handled by reversing the calculation and replacing it with $\Longrightarrow$.

[2] The examples are available online as `http://se.inf.ethz.ch/people/polikarpova/poc/`.

Table 0. Case studies and results.

		Traditional		Calculational		
	Methods	Tokens	Time(s)	Calculations	Tokens	Time(s)
`Identity`	5	527	2.0	4	460	2.0
`SingleFixpoint`	3	444	2.0	4	396	2.0
`KnasterTarski`	1	472	13.1	2	486	4.9
`List`	14	2445	27.4	15	1967	3.0
`MajorityVote`	4	1003	2.1	5	931	2.1
Total	27	4891	46.6	30	4240	14.0

with **`assert`** statements instead of calculations, along the lines of Fig. 4. Tab. 0 gives the comparison between the two versions of each example in terms of verbosity (expressed as the number of code tokens) and verification times (measured on a 2.7GHz i7 CPU, using a single core). We observe that calculations provide more concise proofs, reducing the number of tokens by around 10% on average. The effect of the more efficient encoding on verification times does not show consistently, but it is significant in a few larger proofs.

4.1 Comparison with Other Proof Notations

In this section, we provide a detailed comparison of poC with the proof formats used in two prominent interactive proof assistants: Coq and Isabelle/HOL. Other tools that support calculational proofs are briefly reviewed in related work (Sec. 5).

Proof languages are commonly divided into *procedural* (or imperative) and *declarative* (see *e.g.* [12]). In procedural languages, the proof—also called a *tactic script*—consists of commands, which make the assistant modify the proof state. Declarative languages, which aim at human-readable proofs, are instead oriented toward writing down the intermediate proof states, and letting the tool figure out the commands automatically whenever possible. In the following, we show the same lemma proven first in a procedural, and then in a declarative (more precisely, calculational) style.

Consider our motivational example from Sec. 1.1—list reversal is an involution—written as a Coq tactic script in Fig. 12. The commands **`induction`**, `simpl`, `auto`, and `rewrite` tell the proof assistant which tactics to apply, while `rev_unit` refers to a lemma (a special case of our `ReverseAppendDistrib` where one of the arguments is a singleton list). This script does not tell much to a non-expert user, except that it is a proof by induction and that it makes use of the lemma `rev_unit`.

In the Isabelle standard library, the same lemma looks as follows:

```
lemma rev_rev_ident [simp]: "rev (rev xs) = xs"
by (induct xs) auto
```

The perceived full automation is somewhat misleading: the secret is that all required auxiliary lemmas (in particular the counterpart of `ReverseAppendDistrib`) have been stated in the same theory and equipped with the `[simp]` attribute, which instructs the

```
Lemma rev_involutive : forall l:list A, rev (rev l) = l.
  Proof.
    induction l as [| a l IHl].
    simpl; auto.

    simpl.
    rewrite (rev_unit (rev l) a).
    rewrite IHl; auto.
  Qed.
```

Fig. 12. Proof that list reversal is an involution written as a Coq tactic script. The example is taken from the standard library.

simplifier to automatically apply them as rules. Introducing rewrite rules in this way is dangerous, and rarely works in user proofs. Moreover, this approach, though very concise, damages readability even more: the above script does not even contain the information about the required lemmas.

Now let us turn to declarative proofs. Both Isabelle and Coq have a built-in declarative language. The former one is called Isar ("Intelligible semi-automated reasoning") [30] and is well-established in the Isabelle community. The latter, named simply the Mathematical Proof Language [12], is more recent and not widely used. Both of them support some form of calculations.

Fig. 13 shows a proof of the same lemma in Isar, written in a calculational style. Isar supports calculations through a set of shortcuts [6]: `...` refers to the right-hand side of the most recently stated fact; **also** applies transitivity rules to the current intermediate conclusion and the new step to obtain the new conclusion; **finally** is like **also**, expect that it instructs the prover to use the new conclusion in proving what follows (in this case `?thesis`, which refers to the goal of the enclosing **proof** block).

The **by** clauses play the role of hints: they tell the prover which tactics and facts to use in establishing the step. For this example, we removed the magic `[simp]` attributes from the supporting lemmas, so we have to specify the lemma `rev_append` explicitly at the second step. We can use the lemma-rule to justify this calculation step because the step matches its statement *exactly*; otherwise, we would need additional, more sophisticated tactics to massage the formula into the right shape first (like in the third step, which appeals to the induction hypothesis).

The tactics used in this example are quite simple. Isabelle users also have more powerful tactics, such as `auto`, and even the metaprover Sledgehammer [9] at their disposal, so in practice Isar proofs can be written without excessive detail. However using this powerful automation still requires deep understanding of how the tool works. In particular, the **by** clauses include elements of the procedural proof language, such as the names of tactics[3], while poC hints are stated entirely in the language of the problem domain.

[3] Even what looks like punctuation (dash and dot) actually denotes tactics!

```
lemma rev_rev_ident: "rev (rev xs) = xs"
proof (induct xs)
  show "rev (rev []) = []" by simp
next
 fix x xs
 assume IH: "rev (rev xs) = xs"
 show "rev (rev (x # xs)) = x # xs"
 proof -
  have "rev (rev (x # xs)) = rev (rev xs @ [x])" by simp
  also have "... = rev [x] @ rev (rev xs)" by (rule rev_append)
  also have "... = rev [x] @ xs" by (simp, rule IH)
  also have "... = [x] @ xs" by simp
  also have "... = x # xs" by simp
  finally show ?thesis .
 qed
qed
```

Fig. 13. Proof that list reversal is an involution written in Isar in a calculational style. In Isabelle, `Cons` is written as `#` and `append` as `@`.

The support for calculations in Isar is very flexible: the step relations are not hard-coded, and are easily customizable through the addition of new transitivity rules. This comes at a price that sometimes Isabelle does not know which transitivity rule to apply, and you have to specify this explicitly.

Finally, we believe that the possibility to introduce custom syntax, such as infix operators for list functions above, increases proof readability and it would be nice to have it in Dafny as well.

In conclusion, Isar and Dafny in general (and their respective support for calculations in particular) have different target audiences: Isar is focused on flexibility and maximum control, oriented towards experts with a strong background in formal methods, while Dafny takes the “automation by default” approach, which provides for a lower entrance barrier.

Calculational proofs in Coq’s declarative language are not as flexible, in particular equality is the only supported step relation. Fig. 14 shows our running example. This notation is entirely declarative (and thus closer to poC): note that there is no mention of tactics inside the calculation and the trivial steps do not require justification. The downside is that the default justification tactic, though sufficient for this example, does not provide the same level of automation as SMT solvers.

4.2 Irrelevant Hints and Bogus Steps

One benefit of rule-based proof assistants, like Isabelle, is their ability to determine if the justification of a proof step does not make sense: they will either fail to apply a rule or the rule application will fail to reach the desired proof state.

This is not always the case in poC: one can provide hints that do not justify the step, as well as construct *bogus* steps, which just happen to be true by chance. Although these

```
Lemma rev_involutive {A: Type} (l:list A) : rev (rev l) = l.
proof.
  per induction on l.
    suppose it is nil.
      thus thesis.
    suppose it is (x :: xs) and IH:thesis for xs.
      have (rev (rev (x :: xs)) = rev (rev xs ++ [x])).
                               ~= (rev [x] ++ rev (rev xs))
                                 by (rev_unit (rev xs) x).
                               ~= (rev [x] ++ xs)
                                 by (IH).
                               ~= ([x] ++ xs).
                          thus ~= (x :: xs).
    end induction.
end proof.
Qed.
```

Fig. 14. Proof that list reversal is an involution written in Coq's Mathematical Proof Language in a calculational style. In Coq, `Cons` is written as `::` and `append` as `++`.

issues do not pose any threat to soundness, they are detrimental to readability, and thus important nevertheless. We have not implemented any solution to this problem, but we discuss some possibilities in this section.

To check if a hint is required to justify a calculation step, the tool might try to remove it and check if the program still verifies (within the given timeout). Unfortunately, this does not cover the cases when Dafny can do without the hint, but it would still be useful for a human reader: for example, since Dafny has an automatic induction heuristic, invoking the induction hypothesis will be considered irrelevant most of the time. Perhaps such a mode would be still useful; this remains a part of future work.

One might hope that most irrelevant hints are filtered out just because they are either not applicable in the current context (*i.e.* their precondition does not hold) or they fail to make the step go through. This highlights another benefit of stating the assumptions a lemma makes in its precondition, as opposed to adding them as antecedents to the postcondition: in the latter case, a useless lemma would be simply ignored.

Bogus steps are most likely to occur in calculations with boolean lines, where the truth value of some or all the lines is fixed[4]. Here is an example inspired by an actual error in an old version of one of our case studies. Recall the `Monotonicity` lemma from Fig. 6, whose postcondition is an implication $\mathtt{a} \leq \mathtt{b} \implies \mathtt{f(a)} \leq \mathtt{f(b)}$. When trying to prove this postcondition with a calculation, for the case of $\mathtt{a} < \mathtt{b}$, one might write:

```
if (a < b) {
  calc {
    a ≤ b;
    f(a) ≤ f(a + 1);
```

[4] In other cases, the probability that the values of the lines are related by pure chance is low.

```
      ...
      ⟹ f(a) ≤ f(b);
    }
  }
```

The first step of this calculation looks suspicious to say the least, but it holds because `a ≤ b` follows from the case split, and `f(a) ≤ f(a + 1)` is implied by the method's precondition, thus the step reduces to **`true`** $\Longleftrightarrow$ **`true`**. This calculation is logically valid, but it does not serve to explain the proof to a human reader, if anything, it only confuses them.

In this example, one can avoid the risk of bogus lines by choosing a weaker context (see the proof in Fig. 6). In general, it is a useful methodological rule to always perform a calculation in the weakest context required for it to go through.

Unfortunately, it is not always possible to avoid lines with fixed truth values: it is a common pattern to establish validity of a boolean formula by transforming it into the literal **`true`** by a chain of $\Longleftrightarrow$ or $\Longleftarrow$. Such calculations are dangerous: as soon as the formula is simple enough for Dafny to prove it, all subsequent steps (which might still be useful for a human reader) are potentially bogus. One approach could be to verify such steps out of (some parts of) their context, so that the validity of the lines cannot be established anymore, while the relation between the lines is still verifiable.

5 Related Work

It is hard to say who invented calculational proofs, but Dijkstra (who credits W. H. J. Feijen) definitely played an important role in popularizing them [14].

There are numerous dialects and extensions of this proof format. Back, Grundy and von Wright propose nested calculations [1], combining "the readability of calculational proof with the structuring facilities of natural deduction". They show how common proof paradigms (such as proof by contradiction, case split, induction) can be expressed using nested calculations, with the ultimate goal to make it possible to write a large proof as a single calculation, without the need to "escape" into natural language. A follow-up to this work is structured derivations [2]. Window inference [26, 29] is a related paradigm focusing on managing context during the transformation of subterms. As we showed in Sec. 2.1, poC naturally supports such structuring paradigms.

Attempts to mechanize calculational proofs go back to the Mizar system developed in the early 1990s [27], which supports iterated equality reasoning. There exists a number of implementations of window inference within various proof systems (*e.g.* [15]). The capabilities of those tools are constrained (for example, Mizar only supports equalities), and the level of automation is quite low.

The structured proof language Isar for the Isabelle proof assistant provides a very general support for calculations [6]. However, it is not purely declarative, and one still has to explicitly issue commands that modify the proof state.

At the other end of the spectrum is Math∫pad [4], where one can write proofs in common mathematical notation, not restricted to a particular formal language. It is a document preparation system oriented towards calculational construction of programs. Verhoeven and Backhouse [28] present an attempt to combine it with PVS in order to machine-check the proofs, however a fully automatic translation of the informal components of the notation is, of course, not possible.

"Programs as proofs" is a paradigm commonly used in auto-active program verifiers (see *e.g.* [17]), which allows the reuse of common programming language constructs familiar to programmers for structuring proofs. Pushing auto-active verification into areas previously exclusive to interactive proof assistants is currently an active research topic, with recent advances in automatic induction [21] and even co-induction.

6 Conclusions and Future Work

We have presented a tool for verifying calculational proofs, which stands out due to the combination of a high degree of automation and a programmer-friendly proof language oriented towards to the problem domain rather than the mechanics of the prover.

We believe that poC is not just a useful feature in a program verifier, but also an important step towards a full-fledged auto-active proof assistant. We envision such a tool to be very close to Dafny; in particular, reusing the programs-as-proofs paradigm, but perhaps putting more emphasis on proof concepts rather than on, say, method declarations. Towards this goal, we would need to extend Dafny with support for more general mathematical constructs, such as infinite sets. In order to provide soundness guarantees for an SMT-based proof assistant, a promising approach is generating proof certificates [0] that can be checked by a system with a small trusted core, such as Coq. This seems to be a fertile opportunity for combining the automation found in poC with the rigor inherent in Coq.

Acknowledgments We thank Carlo A. Furia, Ioannis T. Kassios, and Scott West for comments on a draft of this paper, as well as Valentin Wüstholz for suggesting one of our case studies.

References

0. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *CPP*, pages 135–150, 2011.
1. R. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9(5–6):469–483, 1997.
2. R.-J. Back. Structured derivations: a unified proof style for teaching mathematics. *Formal Aspects of Computing*, 22(5):629–661, 2010.
3. R. Backhouse. Special issue on the calculational method. *Information Processing Letters*, 53(3):121–172, 1995.
4. R. Backhouse, R. Verhoeven, and O. Weber. Math∫pad: A system for on-line preparation of mathematical documents. *Software - Concepts and Tools*, 18(2):80–, 1997.
5. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
6. G. Bauer and M. Wenzel. Calculational reasoning revisited: an Isabelle/Isar experience. In *TPHOLs 2001*, volume 2152 of *LNCS*, pages 75–90. Springer, 2001.
7. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

8. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *BOOGIE 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.
9. S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In *IJCAR*, pages 107–121, 2010.
10. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. HipSpec: Automating inductive proofs of program properties. In *Workshop on Automated Theory eXploration: ATX 2012*, 2012.
11. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
12. P. Corbineau. A declarative language for the Coq proof assistant. In *TYPES*, pages 69–84, 2007.
13. E. W. Dijkstra. EWD1300: The notational conventions I adopted, and why. *Formal Asp. Comput.*, 14(2):99–107, 2002.
14. E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Texts and monographs in computer science. Springer, 1990.
15. J. Grundy. Transformational hierarchical reasoning. *Comput. J.*, 39(4):291–302, 1996.
16. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
17. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VS-Tools workshop at VSTTE 2010*, 2010.
18. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
19. J. Koenig and K. R. M. Leino. Getting started with Dafny: A guide. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security: Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 152–181. IOS Press, 2012. Summer School Marktoberdorf 2011 lecture notes.
20. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
21. K. R. M. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, 2012.
22. K. R. M. Leino and M. Moskal. Usable auto-active verification. In T. Ball, L. Zuck, and N. Shankar, editors, *Usable Verification workshop*. `http://fm.csl.sri.com/UV10/`, 2010.
23. P. Manolios and J. S. Moore. On the desirability of mechanizing calculational proofs. *Inf. Process. Lett.*, 77(2-4):173–179, 2001.
24. T. Nipkow. *Programming and Proving in Isabelle/HOL*. `http://isabelle.informatik.tu-muenchen.de/`, 2012.
25. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE-11*, volume 607 of *LNAI*, pages 748–752. Springer, 1992.
26. P. J. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *J. Log. Comput.*, 3(1):47–61, 1993.
27. P. Rudnicki. An overview of the MIZAR project. In *University of Technology, Bastad*, pages 311–332, 1992.
28. R. Verhoeven and R. Backhouse. Interfacing program construction and verification. In *World Congress on Formal Methods*, pages 1128–1146, 1999.
29. J. von Wright. Extending window inference. In *TPHOLs'98*, volume 1479 of *LNCS*, pages 17–32. Springer, 1998.
30. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.