Computing with an SMT solver

Nada Amin [0], K. Rustan M. Leino [1], and Tiark Rompf [0,2]

[0] EPFL, Lausanne, Switzerland
first.last@epfl.ch
[1] Microsoft Research, Redmond, WA, USA
leino@microsoft.com
[2] Oracle Labs, Lausanne, Switzerland
first.last@oracle.com

Manuscript KRML 237, 18 March 2014.

Abstract. Satisfiability modulo theories (SMT) solvers that support quantifier instantiations via matching triggers can be programmed to give practical support for user-defined theories. Care must be taken to avoid so-called matching loops, which may prevent termination of the solver. By design, such avoidance limits the extent to which the SMT solver is able to apply the definitions of user-defined functions. For some inputs to these functions, however, it is instead desireable to allow unadulterated use of the functions; in particular, if it is known that evaluation will terminate.

This paper describes the program verifier Dafny's SMT encoding of recursive user-defined functions. It then describes a novel encoding that, drawing on ideas from offline partial evaluation systems, lets the SMT solver evaluate "safe" function applications while guarding against matching loops for others.

0 Introduction

The collections of cooperating decision procedures in modern satisfiability modulo theories (SMT [2]) solvers provide a powerful reasoning engine. This power is harnessed in numerous applications where logical constraints are involved, including program verification, program analysis, program testing, program synthesis, constraint-based type inference, and theorem proving. While some of the theories supported (*e.g.*, the theory of uninterpreted functions) have complete decision procedures, the SMT solver may support other theories (*e.g.*, integer linear arithmetic) only by semi-decision procedures, either because of theoretical limitations or because of practical time or space compromises. It would be unreasonable to expect the SMT solver to provide support for all theories of interest. Luckily, many theories can be axiomatized in the input to the SMT solver, using logical quantifiers that give some interpretation to otherwise uninterpreted function symbols.

Quantifier support in an SMT solver was first implemented in Simplify [11], based on an idea from Greg Nelson's PhD thesis [25]. The idea is to give each universal quantifier a *matching pattern*, aka a *trigger*, that guides the instantiation of the quantifier. For example, consider the following fragment of input to an SMT solver:

```
Fib(0) = 0 ∧ Fib(1) = 1 ∧
∀ n: int {:Fib(n)} • 2 ≤ n ⟹ Fib(n) = Fib(n-2) + Fib(n-1)
```

where we have written `{:M}` to use the list of expressions `M` as the matching pattern for the enclosing quantifier. This instructs the SMT solver to instantiate the quantifier with `n:=E` whenever in its proof search the current set of ground terms includes a subexpression of the form `Fib(E)`. The approach of using triggers does not, in general, give a complete decision procedure, but the approach fits well into the SMT approach and has been used effectively in practice.

Understanding and making good use of matching patterns is a crucial part of the design of a system built on top of an SMT solver. Using triggers that are too liberal and hence allow too many instantiations can be a source of inefficiency in the proof search. A particular worry is that of non-termination among instantiations, a condition known as a *matching loop*. Therefore, it is necessary to use triggers to curb instantiations. On the other hand, using triggers that are too specific can be a source of incompleteness, since they may prevent instantiations that are needed in the proof. Both of these extremes are common mistakes.

In this paper, we explain and solve a problem with quantifiers that skirts the edge between the two extremes. While instantiations must in general be curbed, there are some instantiations where it is desireable to let the instantiations "run loose". For example, if `Fib(k)`, which matches the trigger, is a ground term in the proof search, then the resulting instantiation produces two new terms, `Fib(k-2)` and `Fib(k-1)`, and these terms also match the trigger. If nothing else is known about the term `k`, a neverending series of instantiations `n:=k-d`, one for each natural number `d`, could arise. We want to prevent the proof search from considering all of these, so some curbing is necessary. On the other hand, if the proof exploration produces a fact like `k=12`, then we would wish for the SMT solver to instantiate the quantifier enough times to figure out `Fib(k)=144`, as if it used the axiom to *compute* the value of `Fib(12)`. The problem we solve in this paper is to find an encoding that provides curbing in the general case and computation in the case where functions are involved on literals. We do the encoding as part of the input to the SMT solver, using appropriate matching patterns; no modification of the SMT solver itself is needed, assuming the SMT solver supports quantifiers via matching patterns in the first place.

We encountered this problem while using the Dafny program verifier [20], where occasionally it is necessary to compute or partially evaluate expressions that contain some literals. For example, we may wish for the SMT solver to compute `Fib(12)` above. As another example, given

```
∀ n: int, t: T •
  (n = 0 ⟹ iter(n, t) = t) ∧
  (n > 0 ⟹ iter(n, t) = iter(n-1, f(t)))
```

we may wish for the SMT solver to partially evaluate `iter(5,x)` as `f(f(f(f(f(x)))))`. The need for computation also arises when one wants to statically *test* the outcome of a given function. For example, one can use Dafny to define the formal semantics of a language, say some form of the lambda calculus, and one may then want to test that the

evaluation of a particular term reduces to the expected value. For instance, verifying the formula

```
n > 0 ⟹ reduces_to(Appl(Lambda(0, Var(0)), Const(29)), Const(29), n)
```

is a test that $(\lambda x_0.\ x_0)\ 29$ reduces to 29 in no more than n reduction steps.

Throughout the paper, we use Dafny as the context for explaining the problem and solution. What we say is likely to apply to any language or notation with user-defined functions that in some form are encoded as SMT input. In Sec. 1, we give a primer on matching patterns in an SMT solver. We describe how Dafny uses matching patterns to curb instantiation of user-defined functions in Sec. 2. This account of curbing represents the current encoding used in Dafny, which is more uniform and flexible than its previously described encodings [20]. In Sec. 3, we then give our encoding that allows literal arguments to be treated differently. We have implemented our encoding in Dafny and report on our experience in Sec. 4, through examples that show both full evaluation of functions and partial evaluation of functions.

1 A Primer on Matching Patterns

A simplified view of the operation of an SMT solver, suitable for our purposes, is the following. The solver is asked to check the validity of a conjecture `P`, often of the form `A` $\implies$ `Q` where `A` is a conjunction of axioms and `Q` is some proof goal. During the proof search, the *proof state* at any time is (some bookkeeping information and) a conjunction of atomic formulas, some of which consist only of ground terms and others of which are universal quantifiers. The ground terms are represented in an *E-graph* [25], a data structure that represents the congruence closure of a set of terms (that is, equalities between terms, with the built-in knowledge that two terms `f(x)` and `f(y)` are equal if the terms `x` and `y` are).

At opportune times, the SMT solver considers the quantifiers in the proof state and looks in the E-graph for ground terms that match the triggers of the quantifiers. The matching ground terms are used to instantiate the quantifiers. This yields more formulas and the proof search continues.

Logically, a universal quantifier holds for all values of the bound variables, but uninformed instantiations are not likely to be useful in the proof search. Therefore, matching patterns are used to limit the instantiations that can take place. Syntactically, a matching pattern is a set of terms whose free variables include all the bound variables of the quantifier. For example, a possible matching pattern for a quantifier $\forall$ `x,y` $\bullet$ ... (here and elsewhere, we omit types of bound variables when they are obvious or irrelevant for the example) is `{:f(x),g(x,y)}`. It says that the quantifier can be instantiated with `x,y:=E,F` in a proof state where the E-graph contains both the terms `f(E)` and `g(E,F)`. The terms given in a matching pattern are typically subterms of the body of the quantifier. Since the role of the matching pattern is to limit instantiations, terms that do not discriminate are not allowed; for example, `{:f(x),y}` is not a legal trigger for the quantifier above, since it places no constraints on the ground terms that could be used for `y`. Since matching is performed in the E-graph, which represents uninterpreted function symbols in the SMT solver, a matching pattern cannot use symbols that are interpreted

by some theory; for example, a matching pattern cannot make use of arithmetic inequalities like $\leq$[0]. One quantifier can contain several matching patterns; a match for any one of them can cause an instantiation.

As an example, suppose we want to define in the SMT input a function `ff` that applies a particular function `f` twice. Function `ff`'s defining axiom is the following:

$\forall$ `x` • `ff(x)` = `f(f(x))`

It is instructive to consider different choices of triggers for this quantifier.

Probably the best trigger for this quantifier is *{:ff(x)}*, because it will in effect make `ff` into something of a macro—as soon as an `ff` term arises among the ground terms, it will become equated with its definition. It is useful to think of quantifier instantiations as having a *direction*. The direction implied by the trigger *{:ff(x)}* is to go from a higher-level function `ff` to a more primitive function `f`. Such a direction is also what one would have in mind when designing effective input for a term rewriting system (*e.g.*, Maude [7]), but note that term rewriting systems and macros replace a source term with a target term, whereas instantiating a quantifier conjoins the instantiated quantifier body to the proof state.

Suppose there is a number of interesting properties that hold for values in the functional image of `ff`, but not necessarily for all values in the functional image of `f`. Then, we may want to produce `ff` terms whenever possible and to axiomatize the other properties in terms of `ff`. For this purpose, *{:f(f(x))}* may make a suitable trigger. Note that this trigger goes in the other direction from *{:ff(x)}*.

Let us consider what may happen if we choose the trigger to be *{:f(x)}*. Suppose a proof state contains the ground term `f(k)`. It will cause the quantifier to be instantiated for x:=k, giving us (an equality between) two new terms, `ff(k)` and `f(f(k))`. The existence of the latter among the ground terms now gives rise to the instantiation x:=`f(k)` and before we know it, the SMT solver will spend all its time instantiating the quantifier with longer and longer terms x:=`f(f(`. . .`f(k)`. . .`))`. This situation is known as a *matching loop* and is one that we want to avoid[1].

For a larger example that highlights typical trigger considerations and gives guidance on trigger design, see [22].

2 User-defined Functions and Curbing in Dafny

Dafny is a programming language that includes support for specifications and proofs. This built-in support makes the language suitable for reasoning about imperative and functional programs as well as some formalized mathematics. Dafny programs are translated into the Boogie intermediate verification language, which the Boogie verification engine then turns into input for the SMT solver Z3 [0,19]. For our purposes in this paper, the most relevant part of the Dafny language is its (possibly recursive) user-defined functions, and we consider their translation into axioms for the SMT solver.

[0] Because of this restriction, adding a new theory to the SMT solver comes at the considerable expense of not being able to match on its symbols.

[1] Boogie code for the matching loop example discussed here: `http://rise4fun.com/Boogie/mH23`.

Let us give some motivation by considering an example. Suppose we want to encode as SMT input a Dafny function that defines triangle numbers:

```
function Triangle(n: nat): nat
{
  if n = 0 then 0 else n + Triangle(n-1)
}
```

To encode this for the SMT solver, we introduce a function `Triangle` on the integers and supply an axiom like this:

```
∀ n: int • 0 ≤ n ⟹
  Triangle(n) = if n = 0 then 0 else n + Triangle(n-1)
```

How do we want this quantifier to be triggered? Whenever a proof search involves a term `Triangle(k)` for some subterm k, then it seems useful to instantiate the quantifier with n:=k, so we may consider the straightforward trigger *{:Triangle(n)}*. However, such a trigger would lead to a matching loop, because lacking any information about k, the SMT solver would explore both branches of the if expression, and the successive exploration of the else branch would lead to new instantiations of the quantifier[2].

To curb such instantiations, the Dafny verifier adds an extra parameter to the SMT encoding of the function. Borrowing a recent name from discussions about co-induction in the type-theory community, we will refer to this parameter as "fuel". The fuel parameter specifies how many unrollings of the function we want to allow the SMT solver to do. Note, the value of the function does not actually depend on the fuel parameter; it is used only to control the SMT solver's instantiations.

Since matching is performed in the E-graph, it is important that non-zero fuel values be recognizable structurally, with no theory reasoning and without interpreted symbols like `0` and `+`. Thus, we make use of Peano arithmetic (that is, unary arithmetic) and provide the following declarations in the SMT input:

```
type Fuel
function Z(): Fuel
function S(Fuel): Fuel
```

These declarations can be thought of as an inductive datatype like

```
datatype Fuel = Z | S(Fuel)
```

but we do not bother to say anything about `Z` and `S`, beyond fact that they are functions with the given signatures.

We can now encode the Dafny function `Triangle`. We declare it in the SMT input as follows:

```
function Triangle(fuel: Fuel, n: int): int
```

Next, we produce three axioms. The "synonym" axiom that says that the value of the fuel parameter is irrelevant:

[2] `http://rise4fun.com/Boogie/Agsl`

```
∀ fuel: Fuel, n: int • {:Triangle(S(fuel), n)}
  Triangle(S(fuel), n) = Triangle(fuel, n)
```

The "definition" axiom encodes the function body:

```
∀ fuel: Fuel, n: int • {:Triangle(S(fuel), n)} 0 ≤ n ⟹
  Triangle(S(fuel), n) = if n = 0 then 0 else n + Triangle(fuel, n-1)
```

Finally, the "consequence" axiom states properties that come from the signature and specification of the function:

```
∀ fuel: Fuel, n: int • {:Triangle(S(fuel), n)} 0 ≤ n ⟹
  0 ≤ Triangle(S(fuel), n)
```

This encoding provides curbing because the matching patterns will cause the quantifiers to be instantiated only when the fuel parameter is non-zero (more precisely, when it has the form `S` applied to something) and because recursive (and mutually recursive) calls in the right-hand side of the definition axiom use a smaller fuel value than the left-hand side.

The verifier translates other Dafny uses of the function with some default value for the fuel parameter. In Dafny, this default value is usually 1 (that is, `S(Z())`), but it is 2 in certain proof-obligation positions (like when a user-supplied assertion or postcondition needs to be verified). Although we currently do not provide it as a feature, the default fuel value could in principle be set by the user, globally or for particular functions or particular proof obligations.

We end this section with an example that both illustrates the technique and serves as a segue to the next section. Consider a lemma that proves, by induction, that `Triangle(n)` is at least twice as big as `n`, provided `n` is at least `3`. In Dafny, this is done as follows:[3]

```
lemma TriangleProperty(n: nat)
  ensures 3 ≤ n ⟹ 2*n ≤ Triangle(n);
{
  if n ≤ 3 {
    assert Triangle(3) = 6;  // the crucial property of the base case
  } else {
    TriangleProperty(n-1);  // invoke the induction hypothesis
  }
}
```

The "postcondition" of the lemma, given by the **ensures** clause, states the conclusion of the lemma. The body of the lemma is some code, where all control paths are verified to lead to the postcondition. The recursive call to `TriangleProperty` essentially obtains the inductive hypothesis, applied for `n-1`.

To detail the proof obligations for this lemma, we view the lemma as a pre/post-condition pair and write the following pseudo code (which is representative of what the intermediate form in Boogie will look like):

[3] We ignore the fact that Dafny has some support for automatic induction [21] and here give the proof explicitly.

```
assume 0 ≤ n;  // assume precondition of lemma
if n ≤ 3 {
  assert Triangle(S(S(Z())), 3) = 6;  // fuel = 2
} else {
  assert 0 ≤ n-1;  // check precondition of call
  // assume postcondition of call:
  assume 3 ≤ n-1 ⟹ 2*(n-1) ≤ Triangle(S(Z()), n-1);  // fuel = 1
}
// check postcondition of lemma:
assert 3 ≤ n ⟹ 2*n ≤ Triangle(S(S(Z())), n);  // fuel = 2
```

Note that the fuel argument is passed in as 2 in proof-obligation positions and as 1 elsewhere. The verification condition for this pseudo code is the following first-order formula, which is given to the SMT solver:

```
0 ≤ n ⟹
  (n ≤ 3 ⟹ Triangle(S(S(Z())), 3) = 6) ∧  // check then branch
  (3 < n ⟹ 0 ≤ n-1) ∧  // check else branch (trivial)
  // prove the postcondition from what is learnt in both branches:
  ( (n ≤ 3 ∧ Triangle(S(S(Z())), 3) = 6) ∨
    (3 < n ∧ 0 ≤ n-1 ∧ (3 ≤ n-1 ⟹ 2*(n-1) ≤ Triangle(S(Z()), n-1)))
     ⟹ 3 ≤ n ⟹ 2*n ≤ Triangle(S(S(Z())), n))
```

To prove the postcondition of the lemma, there are two cases. If $\texttt{n} = 3$, the postcondition follows from what is learnt from the then branch. If $4 \leq \texttt{n}$, the postcondition follows from the definition axiom and the induction hypothesis. In more detail, since the fuel parameter of the last call to `Triangle` has the form `S(...)`, the definition axiom is triggered and thus the final inequality becomes:

```
2*n ≤ n + Triangle(S(Z()), n-1)
```

This inequality follows from what is learnt from the else branch (that is, the induction hypothesis).

As we just saw, we had more fuel than necessary to complete the proof of the postcondition in this example. But what about the proof of the then branch? The fuel supplied in its call to `Triangle` is enough for two instantiations of the definition axiom, which reduces the proof goal to:

```
3 + 2 + Triangle(Z(), 1) = 6
```

Since there is no fuel left, the SMT solver is unable to complete this proof (so a verification error will be reported to the user). In this next section, we describe how we extend the encoding to handle cases like this.

3 Enabling Computation

We enable computation by allowing unfolding steps that do not decrease the fuel parameter in chosen cases, picked at compile time. Dafny generates SMT input that allows unfolding for two kinds of function applications: (a) when *all* function arguments are

known to be constants, and (b) when all arguments that are part of the decreasing measure for termination (maintained internally by Dafny) are constants. In the first case, the result of the function application is known to be a constant as well. In the second case, the result is not necessarily a constant, but evaluation (via E-matching for the instantiations) is still guaranteed to terminate. Dafny propagates which expressions will definitely evaluate to constants and uses this information for further unfolding decisions; a technique known as *binding-time analysis* in the context of partial evaluation [17].

Our encoding for computation relies on an identity function, provided in the SMT input, to mark constant expressions as "literals":

```
function Lit<T>(x: T): T { x }
```

For each user-defined function, we add extra "computation" axiom(s) that trigger on literal argument(s) and that do not consume any fuel. For the `Triangle` function, we provide the following extra axiom:

```
∀ fuel: Fuel, n: int • {:Triangle(fuel, Lit(n))}
  0 ≤ n ⟹
  Triangle(fuel, Lit(n)) =
    if n = 0 then 0 else n + Triangle(fuel, Lit(n-1))
```

To enable computations, the Dafny compiler wraps all concrete values, such as `2`, with the `Lit` marker. The compiler also lifts simple operations on literal expressions: `Lit(x)+Lit(y)` becomes `Lit(x+y)`, since adding two constant values will produce a constant again. This lifting mechanism is also what enables recursive computations, since `Triangle(fuel, Lit(n)-Lit(1))` becomes `Triangle(fuel, Lit(n-1))`. Note that the variable `n` is wrapped as a literal expression `Lit(n)` because it is a formal parameter fixed as a literal in the trigger of the axiom.

For computations to be composable, we also `Lit`-wrap each function application on all literal arguments. Hence, `Triangle(fuel, Lit(3))` is tagged as a constant expression `Lit(Triangle(fuel, Lit(3)))`. This propagation of binding-time information is essential to enable computation on nested expressions, such as `Triangle(Triangle(3))` in Dafny.

Finally, a word of caution: we don't always want to compute. The SMT solver can prove `Fib(1000)`$\neq$`1000` on its own without computing `Fib(1000)`, but if we provide a "computation" axiom for `Fib` and give it too much importance, then the solver hangs instead. We resolve this tension by giving a low priority to the "computation" axioms. Also, as a small tweak that matters in practice, we also let if-then-else expressions act as a "barrier" for literals, so that we unwrap any top-level literals following the if, then or else expressions. This is why the computation axiom above does not return `Lit`-wrapped expressions.

4 Experience

We re-iterate the necessity of the fuel parameter with a complete Dafny example [4] which correctly verifies using the encoding described in Sec. 2 but enters a matching

[4] `http://rise4fun.com/Dafny/EHGl`

loop when the fuel parameter is ignored. The example proves the equivalence of the recursive and iterative definitions of the factorial function.

```
function Factorial(n: nat): nat
{
  if n = 0 then 1 else n*Factorial(n-1)
}
function FactorialIter(n: nat, acc: nat): nat
{
  if n = 0 then acc else FactorialIter(n-1, acc*n)
}
function Factorial'(n: nat): nat
{
  FactorialIter(n, 1)
}
lemma lemmaFactorialStep(n: nat, acc: nat)
  ensures acc*Factorial(n)==FactorialIter(n, acc);
{
}
lemma theoremFactorialEquiv(n: nat)
  ensures Factorial(n)==Factorial'(n);
{
  lemmaFactorialStep(n, 1);
}
```

This example demonstrates that curbing is sometimes essential when proving universally quantified theorems in Dafny. Now, we show that controlled relaxation of the curbing, described in Sec. 3, is also very important in practice.

Just like one would write tests in conventional languages, in Dafny, one can write out examples with their expected results, with the hope that they will be automatically verified. Thanks to our novel encoding that enables computation, this hope is now often materialized.

For example, in an implementation of the simply typed lambda calculus[5], we may wish to check that $\lambda(x : T).\lambda(f : T \to T).f(f(x))$ has type $T \to (T \to T) \to T$. This example now verifies automatically, but with the old encoding, which used curbing everywhere, it required 5 intermediate statements to verify (we use a direct encoding of variable names as numbers):

```
lemma example_typing()
  ensures has_type(map[], tabs(0, TBase,
                          tabs(1, TArrow(TBase, TBase),
                          tapp(tvar(1), tapp(tvar(1), tvar(0))))) =
          Some(TArrow(TBase, TArrow(TArrow(TBase, TBase), TBase)));
{
  var c := extend(1, TArrow(TBase, TBase), extend(0, TBase, map[]));
  assert find(c, 0) = Some(TBase);
```

[5] http://rise4fun.com/Dafny/IxqUu

```
  assert has_type(c, tvar(0)) = Some(TBase);
  assert has_type(c, tvar(1)) = Some(TArrow(TBase, TBase));
  assert has_type(c, tapp(tvar(1), tapp(tvar(1), tvar(0)))) =
         Some(TBase);
}
```

We now illustrate the utility of generating a computation axiom which triggers merely when decreasing formal parameters are literals. Even when the typing context is left abstract, Dafny automatically verifies type-checking of the example $\lambda(f : T \to T).f(f(x))$ provided a context with at least $(x : T)$:

```
lemma example_typing_m(m: map<int,ty>)
  requires 0 in m ∧ m[0]=TBase;
  ensures has_type(m, tabs(1, TArrow(TBase, TBase),
                      tapp(tvar(1), tapp(tvar(1), tvar(0))))) =
          Some(TArrow(TArrow(TBase, TBase), TBase));
{
}
```

Even though the context `m` is not a literal, computation is possible because only the term parameter is part of the decreasing measure of the `has_type` function, and that argument is a literal in this example application.

Here is another example [6], inspired by an exercise in the Coq textbook, Software Foundations [27]. This example shows that our encoding of computation plays well with function applications in complex expressions.

```
datatype Nat = O | S(Nat) // Peano numbers
function plus (n : Nat, m : Nat) : Nat
{
 // ...
}
function mult (n: Nat, m: Nat) : Nat
{
 // ... in terms of plus
}
function factorial(n: Nat): Nat
{
 // ... in terms of mult
}
function toNat(n: nat): Nat
{
  if n=0 then 0 else S(toNat(n-1))
}
lemma test_factorial1()
  ensures factorial(toNat(3))=toNat(6);
{
```

[6] http://rise4fun.com/Dafny/NRFA

```
}
lemma test_factorial2()
  ensures factorial(toNat(5))=mult(toNat(10),toNat(12));
{
}
```

With the previous encoding of user-defined functions that implements curbing without computation, proving `test_factorial2` would be very tedious and require many intermediate steps.

4.0 Limitations

Since we need to make the computation axioms low priority to avoid hanging computations, we also prevent some larger (tractable) computations. This is a matter of degree: we certainly don't want to allow `Fib(1000)` but what about `Fib(40)`?

We chose to make unfolding decisions at compile time, when generating SMT input in Dafny, as opposed to delegating to the SMT solver to make such decisions on-the-fly, at run time – in particular, we chose not to provide the SMT solver with any axioms that would create fresh applications of the `Lit` marker. Clearly, such a static binding-time analysis is approximate by nature and cannot always deduce that an expression will evaluate to a constant. Hence, we definitely miss some easily computable expressions, as we show next. We extend the previous sample code with an example that Dafny cannot auto-verify:

```
function returnFst(a: nat, b: nat): nat
{
  if b=0 then a else returnFst(a, b-1)
}

lemma test_factorial_indirect(n: nat)
  ensures factorial(toNat(returnFst(5, n)))=mult(toNat(10),toNat(12));
{
}
```

Note that we use a convoluted definition of `returnFst`. Otherwise, the function would be inlined by the Dafny compiler, and the example reduced to `test_factorial2`. The problem here is that the Dafny compiler does not detect that `returnFst(5, n)` is in fact equivalent to `5`, and hence fails to recognize at compile time that the argument to factorial is indeed a literal. Interestingly, it is enough guidance for verification to provide this fact:

```
lemma eqReturnFst(a: nat, b: nat)
  ensures returnFst(a, b)=a;
{
}
lemma test_factorial_indirect_ok(n: nat)
  ensures factorial(toNat(returnFst(5, n)))=mult(toNat(10),toNat(12));
{
```

```
    eqReturnFst(5, n);
  }
```

We leave a closer investigation of online techniques to future work, where the SMT solver would make unfolding decisions on the fly. The benefit would be an increase in precision, but in general, ensuring termination is harder in an online setting. We conjecture that the additional information about termination measures that is available in Dafny could be put to use here as well, possibly at the expense of a more involved encoding.

5 Related Work

Partial Evaluation Partial evaluation [17] denotes a class of program transformations that aim to pre-compute parts of a program that do not depend on dynamic input values. The result of partial evaluation is a residual program, where expressions that only depend on statically known values are evaluated to constants. Partial evaluation is usually applied to improve performance, as the residual program performs less work. In our case, we are interested in the simplification aspect: in a verification context, evaluating an application of a user-defined function means that we can directly reason about the result value and need not reason about the function definition.

Partial evaluation comes in two flavors: online [4,28,29] and offline [6,8,14,15]. In an online setting, decisions whether to evaluate or to residualize an expression are made on the fly. If a function is called with only static arguments, the function will be evaluated. If a subset of the arguments is static, a specialized function may be generated. A well-known problem in online partial evaluation is that it is difficult to ensure termination (and even a terminating computation might take a long time). The second flavor of partial evaluation is offline. Here, a binding-time analysis first classifies each expression as static or dynamic. A second pass then evaluates all expressions classified as static.

In our case, evaluation is a special case of proof search in an SMT solver. Conceptually, evaluation corresponds to unfolding of functions and simplifying. On the SMT level, unfolding means instantiation of the corresponding quantifier. Without further directions, the SMT solver will make decisions online, whether or not to unfold quantifier definitions, based on heuristics like a global instantiation depth. Since the SMT solver knows nothing about user-defined functions apart from their axiomatization, it cannot know whether a particular function will terminate or not, and if unfolding is profitable. On the Dafny level, however, this kind of information is readily available. Our encoding serves the purpose of communicating this information to the solver. In essence, we implement a classic offline partial evaluation scheme. We perform a simple binding-time analysis to identify static expressions within Dafny. We tag those expressions with a `Lit(.)` marker, and we emit axioms that direct the SMT solver to unfold functions if they are called with `Lit(.)` arguments. But we also get some of the effects of an online scheme, because the SMT solver may end up combining results from our simple binding-time analysis. For example, given the Dafny program snippet

```
var y := 12;
assert y ≤ k ∧ k < y + m ∧ m = 1 ⟹ Fib(k) = 144;
```

our simple binding-time analysis will classify only `12` as "static". But after the SMT solver's theory reasoning concludes `k` = `y` = `12`, that "static" classification is in effect transferred to `k`, and thus the term `Fib(k)` will be fully evaluated.

A different approach is taken by the Leon verification system [18]. Instead of mapping user-defined functions to quantifiers and invoking the SMT solver only once, Leon invokes the solver interactively, while successively unfolding function definitions in the solver input. This is an example of a practical online approach, but crucially one that circumvents the brittle solver heuristics.

Computation in Proof Assistants Many interactive proof assistants rely on computations. For example, uses of Coq [5] and PVS [26] routinely need computation as part of type checking. These computations are not set up using trigger-based quantifiers, but are instead based on custom tactics or other heuristics or mechanisms.

Quantifiers and Triggers The SMT solver Simplify [11] was the first to give comprehensive support of trigger-based quantifiers. As DPLL(T)-style architectures became popular, experimental SMT solvers [13,23] and mature SMT solvers [1,3] also added support for quantifiers. For our purposes, the efficient implementation of matching in the SMT solver Z3 [10] took quantifier support to a heightened level [9].

More information about how triggers work can be found in the descriptions of Simplify [11] and Z3 [10], as well as in Michał Moskal's PhD thesis [24]. Dross *et al.* have studied a logical semantics for triggers [12]. Leino and Monahan [22] convey the artform of typical trigger design through a particular example.

Curbing by Constructor Cases In the current version of Dafny, curbing is achieved as we have described in Sec. 2. In a previous version, functions whose body consisted of a **`match`** expression would get translated "Haskell style" into one axiom per constructor **`case`**. Inspired by VeriFast [16], this translation attempts to curb instantiations by including the name of a constructor in each trigger, which means that the axioms will not be applied unless it is already known which case applies. However, we have found that we no longer need that translation for curbing, and in fact our current curbing allows more examples to be verified.

6 Conclusions

Proper attention to the design of matching triggers is crucial for any tool that wants to harness the power of an SMT solver with trigger-based quantifiers. The Dafny program verifier simultaneously uses two techniques to encode the user-defined functions that it has to reason about. While one technique curbs instantiations and thus limits the use of function definitions, the other technique is designed to give unadulterated use of the function definitions. The first technique is useful because many inductive program proofs need only one unfolding of functions, and the curbing prevents matching loops in the SMT solver. The other technique is useful because it allows the SMT solver to perform computations and partial evaluations, and axioms are set up in such a way that they apply only when the function arguments are literals. The two techniques come

together automatically in Dafny, and there is no need for users do anything special to obtain the benefits.

Using our design, we have profited from the use of computation in Dafny. For example, we have transcribed into Dafny examples from two chapters of the Coq-based *Software Foundations* book by Pierce *et al.* [27], which uses many examples to test the given definitions. Our design and implementation of the two simultaneous techniques in Dafny now make it possible to benefit from computation while doing proofs within the comfort of the automation provided by an SMT-based program verifier.

Acknowledgments We thank Nik Swamy for useful comments on an earlier draft of this paper.

References

0. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
1. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *LNCS*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.
2. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In Aarti Gupta and Daniel Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
3. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
4. Andrew A. Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
6. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Sci. Comput. Program.*, 17(1-3):3–34, 1991.
7. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
8. Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *PEPM*, pages 145–154. ACM, 1993.
9. Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *CADE-21*, volume 4603 of *LNCS*, pages 183–198, 2007.
10. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, March–April 2008.
11. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

12. Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012*, EasyChair 2013 EPiC Series, pages 22–31, 2012.
13. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *15th Computer-Aided Verification conference (CAV)*, July 2003.
14. Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation: A case study. *Structured Programming*, 12(3):123–144, 1991.
15. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *J. Funct. Program.*, 1(1):21–69, 1991.
16. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.
17. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
18. Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *OOPSLA*, pages 407–426. ACM, 2013.
19. K. Rustan M. Leino. Specification and verification of object-oriented software. In Manfred Broy, Wassiou Sitou, and Tony Hoare, editors, *Engineering Methods and Tools for Software Safety and Security*, volume 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 231–266. IOS Press, 2009. Summer School Marktoberdorf 2008 lecture notes.
20. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, April 2010.
21. K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation — 13th International Conference, VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, January 2012.
22. K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 615–622. ACM, March 2009.
23. K. Rustan M. Leino, Madan Musuvathi, and Xinming Ou. A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, volume 3440 of *LNCS*, pages 334–348. Springer, April 2005.
24. Michał Jan Moskal. *Satisfiability Modulo Software*. PhD thesis, Institute of Computer Science, University of Wrocław, 2009.
25. Charles Gregory Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox PARC, June 1981.
26. Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.
27. Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2013. `http://www.cis.upenn.edu/~bcpierce/sf`.
28. Dan Sahlin. The mixtus approach to automatic partial evaluation of full prolog. In Saumya K. Debray and Manuel V. Hermenegildo, editors, *NACLP*, pages 377–398. MIT Press, 1990.
29. Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *FPCA*, volume 523 of *LNCS*, pages 165–191. Springer, 1991.