Using Dafny, an Automatic Program Verifier

Luke Herbert[1], K. Rustan M. Leino[2], and Jose Quaresma[1]

[1] Technical University of Denmark (lthhe,jncq)@imm.dtu.dk
[2] Microsoft Research leino@microsoft.com

Abstract. These lecture notes present Dafny, an automated program verification system that is based on the concept of dynamic frames and is capable of producing .NET executables. These notes overview the basic design, Dafny's history, and summarizes the environment configuration. The key language constructs, and various system limits, are illustrated through the development of a simple Dafny program. Further examples, linked to online demonstrations, illustrate Dafny's approach to loop invariants, termination, data abstraction, and heap-related specifications.

1 Preface

These lecture notes introduce the programming and verification language Dafny. They are primarily based on lectures given by Rustan Leino in 2011 at the 8th LASER Summer School, as transcribed by Luke Herbert and Jose Quaresma (who were students at that summer school). Other references to Dafny and influences on this tutorial include the Marktoberdorf Summer School lectures from 2008 [11] and 2011 [10], and the online Dafny tutorial [9].

Dafny is a state-of-the-art implementation of an automated verification system based around the idea of *dynamic frames* [6,7]. This is an approach to formal verification of program correctness that attempts to prove correctness of individual program parts locally, and from there infer the correctness of the whole program. The dynamic-frames approach makes it possible to reason about subparts even in the presence of data-abstraction [15]. Dafny is a usable implementation of this approach which has been used to verify several non-trivial algorithms.

This tutorial is a practical guide to using Dafny to write verifiable programs. While some description of the design of Dafny is given, this is not complete and serves mostly to overview Dafny use, and to point to more authoritative sources of information on Dafny internals.

This tutorial provides extensive code examples. Most code examples have web links to code pre-loaded in an online Dafny environment, which demonstrates many key Dafny features. You can follow the code examples by visiting the corresponding footnote link, to see what Dafny reports, and to further experiment with Dafny.

2 Dafny Background

The Dafny programming language is designed to support static verification of programs. It is imperative, sequential, supports generic classes and dynamic allocation, and, crucially, incorporates specification constructs. The specifications include pre- and post- conditions, frame specifications (read and write sets), loop invariants, and termination metrics. To further support specifications, the language also offers updatable ghost variables, recursive functions, and types like algebraic datatypes, sets, and sequences. Specifications and ghost constructs are used only during verification; the compiler omits them from the executable code. The Dafny compiler produces C# code, which is in turn compiled to MSIL bytecode for the .NET platform by the standard Microsoft C# compiler. However, the facilities for interfacing with other .NET code are minimal.

An overview of the entire Dafny system is given in figure 1. The Dafny verifier is run as part of the compiler. A programmer interacts with it much in the same way as with the static type checker; when the tool produces errors, the programmer responds by changing the program's type declarations, specifications, and statements. Dafny's program verifier works by translating a given Dafny program into the intermediate verification language Boogie 2 [1] in such a way that the correctness of the Boogie program implies the correctness of the Dafny program. Thus, the semantics of Dafny are defined in terms of Boogie (a technique applied by many automatic program verifiers). The Boogie tool is then used to generate first-order verification conditions that are passed to the Z3 SMT solver [4]. Any violations of these conditions are passed back as verification errors. In parallel with the verification of the code, a standard .NET executable is also emitted.

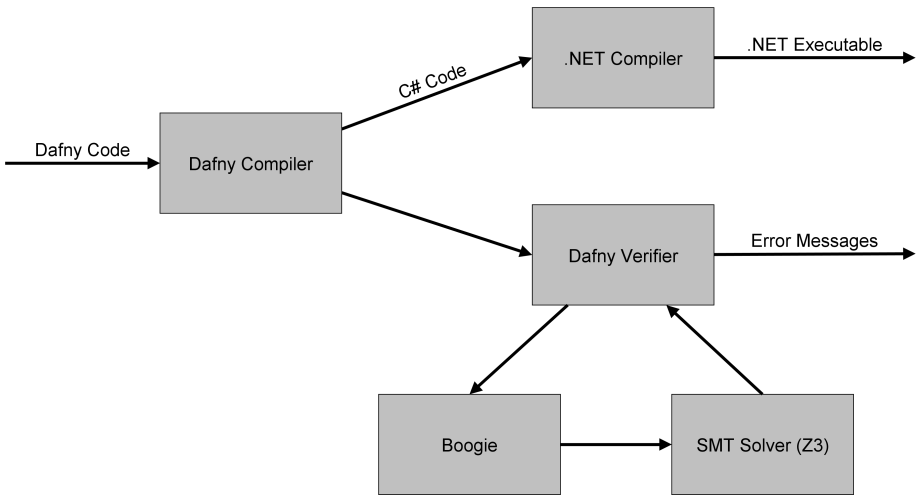

Fig. 1. The Dafny system

Dafny is a descendant of a series of program verifiers and extended static checkers. When the Dafny project started, the most recent of these to reach some maturity was the Spec# system [2], and a project to build the C program verifier VCC [3] was also in progress. Dafny started off as a research test bed for some specification ideas that were being considered for VCC. Dafny provided a more readable notation than the more primitive intermediate verification language Boogie, on which both Dafny and VCC rest. Dafny showed itself to be useful in verifying little algorithms. To handle more complicated algorithms, features were gradually added to Dafny. Dafny has grown from a programming notation to a full programming language, and from a language intended only for verification to a language that also compiles to executable code. Dafny has been used to verify a number of challenging algorithms, like the Schorr-Waite graph marking algorithm [12] and Floyd's "tortoise and hare" cycle detection algorithm. Dafny has also fared well in some program verification competitions [8]. Because the language was designed from scratch and with verification in mind from the start, it generally gives rise to cleaner programs than, say, programs verified with VCC, whose specifications are layered on top of an existing language. Moreover, because Dafny is type safe and uses unbounded integers, specifications written in Dafny are typically simpler than what is seen in, for example, VCC. For these reasons, Dafny stands out as a good choice for teaching concepts of program reasoning.

3 Getting Started with Dafny

To get started, we suggest using the interactive version of Dafny on `http://rise4fun.com/Dafny`. While this does not provide all the capabilities of the full Visual Studio version of the tool, in particular with regard to debugging, it is able to thoroughly demonstrate the main capabilities of Dafny. Indeed, this entire tutorial can be completed using the `rise4fun` website.

To run Dafny using your own computer, download the binaries from `http://boogie.codeplex.com`. The binaries run on any .NET platform, which includes Windows as well as any system with the Mono .NET development framework.

The smoothest way to run Dafny is in Microsoft Visual Studio 2010, which brings the benefits of an integrated development environment where the program verifier runs in the background as the Dafny code is being developed. The installation currently requires downloading and building Dafny from source (also from `http://boogie.codeplex.com`) and dealing with some hardcoded file paths (see the instructions on that site).

We also suggest you load the `rise4fun` website and type in the following declaration of a simple method that swaps two variables.

```
method Swap(a: int, b: int) returns (x: int, y: int)
```

This method declaration states that the `Swap` method receives two values `a` and `b` as input and returns two values `x` and `y` as output. The intention is to let

the output values be the input values in swapped order, that is to say that x should have the value of b and y the value of a.

The central idea in Dafny is to express what the method is intended to do, and then have Dafny verify that this is indeed what the method actually does (an implementation of the concept of Hoare Logic [5]). Dafny can use an **ensures** declaration to express a postcondition for a method, informing Dafny that the variables should be swapped after execution of the method. This is applied to the example as follows:

```
method Swap(a: int, b: int) returns (x: int, y: int)
  ensures x == b && y == a;
```

After the signature of the method has been declared, we can write the body of the method which implements the swapping and use Dafny to verify that the method does indeed achieve what is intended. The suggested way to use Dafny is to first express what a method is intended to do, and then write the implementation of the method and let Dafny check if it is correct. For the preceding example[3], try to see if you can write a satisfactory method body.

4 The Basics of the Dafny Language

The swap example from section 3 on the previous page can be used to introduce the basic constructs of the Dafny language. That example used two of Dafny's basic constructs, namely **method** and **ensures**. Here is a body of imperative code for the method:

```
method Swap(a: int, b: int) returns (x: int, y: int)
  ensures x == b && y == a;
{
  x := b;
  y := a;
}
```

The method body is contained within the braces and it consists of a series of statements, such as assignments, conditionals, loops, and method calls. Assignments in Dafny are performed using := and simple statements are followed by a semi-colon.

The types of parameters, result values, and object fields must be declared explicitly, whereas the types of local variables (and of bound variables of quantified expressions, see section 7 on page 13) are usually inferred and can be omitted. Dafny's types include **bool** for booleans, **int** for unbounded integers, and **nat** for natural numbers (the non-negative integers, a subrange of **int**). User-defined classes and inductive datatypes are allowed. Dafny provides **set**<T> for an immutable finite set of values of type T, and **seq**<T> for an immutable finite sequence of values of type T. In addition, there are *array* types of one and more

[3] Interactive code sample: `http://rise4fun.com/Dafny/VjhK`

dimensions, written **array**<T>, **array2**<T>, **array3**<T>, and so on. Finally, the type **object** is a super-type of all reference types, implying that a value of **object** can be a reference to any class instance or array, or the special value **null**. However, it should be noted that Dafny has no inheritance support or other class subtypes.

The **ensures** keyword is used to specify a method's *postcondition*. A postcondition expresses a property that must hold after every invocation of the method through all possible return points. Postconditions form part of the method declaration and appear before the body block. In the Swap method, the postcondition says that the output values should have the inverse order from the input arguments.

To make the example more interesting, instead of using method in- and out-parameters, we can change the method to operate on two variables in the scope enclosing the method. These variables are declared using the keyword **var** outside the scope of the method. To be able to verify this new version of the Swap method, two important modifications are needed.

The first modification is due to the fact that the postcondition must be expressed in terms of the state of the variables before and after method execution. Dafny allows for this by making use of the **old** keyword, which when applied to a variable (**old**(variable)) operates as a function which refers to the value of the variable at the time the method was invoked.

The second modification required is to declare which variables the method is allowed to change, which is done using the keyword **modifies**. Although the declared variables may look like global variables, they are in fact fields of an implicit class. For now, just specify the Swap method with **modifies this**, which gives the method license to modify the fields of the object on which it is invoked, which here means it is allowed to modify the variables x and y. These modifications result in a new swap program:

```
var x: int;
var y: int;

method Swap()
  modifies this;
  ensures x == old(y) && y == old(x);
{
  x := y;
  y := x;
}
```

If you attempt to verify this code, Dafny reports that a postcondition might not hold. This is because the implementation of the method is now wrong. As an exercise, try to implement it correctly.

One possible solution using a temporary variable will look like this[4]:

```
var tmp := x;
```

[4] Interactive code sample: `http://rise4fun.com/Dafny/hpru`

```
x := y;
y := tmp;
```

The body of the Swap method can be expressed more succinctly by employing *parallel assignment*. This avoids the use of a temporary variable and performs the swap in a single line of code[5].

```
x,y := y,x;
```

If a Dafny program contains a unique parameter-less method called Main, then program execution will start there. It is not necessary to have a main method to do verification, only to produce a .NET executable. We will now create a Main method which will call our Swap method to perform a swap.

The main method sets the initial value for variables x and y, call the Swap method, and then check if the values of the variables were indeed swapped. This is confirmed by means of an *assertion* indicated by the **assert** keyword.

```
method Main()
  modifies this;
{
  x := 5;
  y := 10;
  Swap();
  assert x == 10 && y == 5;
}
```

An assertion statement forces Dafny to verify that the given boolean expression evaluates to true along all possible program paths to that point. You can think of the program as crashing if the asserted condition does not hold. Thus, only executions where the asserted condition holds ever get past the **assert** statement. You can observe a consequence of this in the following Warn method[6], where the verifier complains about the first **assert** statement but not the second—because the second assertion *does* hold in every execution that gets past the first assertion without crashing.

```
method Warn()
{
  var x;
  assert x<10;
  assert x<100;
}
```

It is okay to think of **assert** as possibly crashing the program at run time, but note that programs must pass the verifier before they are compiled, and the verifier will complain if it cannot prove the absence of such crashes.

You can now verify the complete swap program[7]. An alternative way to implement the Swap method is:

[5] Interactive code sample: `http://rise4fun.com/Dafny/Zw5s`
[6] Interactive code sample: `http://rise4fun.com/Dafny/AvCs`
[7] Interactive code sample: `http://rise4fun.com/Dafny/slYEo`

```
x := x + y;
y := x - y;
x := x - y;
```

You can check that the Swap method making use of this alternative method implementation[8] can also be verified by Dafny. Note that Dafny is able to reason about the arithmetic performed. While Dafny is able to reason about many common mathematical constructs that appear in programs, like linear arithmetic and boolean algebra, Dafny is not aware of all mathematical truths. For example, in the following program, which requires a complex mathematical proof [16], Dafny will not be certain that the postcondition will always hold. [9]

```
method Fermat(a: int, b: int, c: int) returns (ans: bool)
  ensures (ans == true);
{
  ans := true;
  if (0 < a && 0 < b && 0 < c && a*a*a + b*b*b == c*c*c) {
    ans := false;
  }
}
```

We can make the code more reusable if we change the swap program to use an object encapsulating the data instead of using global variables. We will create a class called Cell in Dafny in the following way:

```
class Cell {
  var data: int;
}
```

We created a class Cell with one integer variable called `data`. We can now change the signature of the Swap method to make use of Cell objects. This new version of the method will receive two Cell objects and swap their values. However, this introduces a new requirement that the method's input parameters should refer to proper Cell objects, that is, they can not have the **null** value. We express this requirement using a *precondition*, a boolean expression which is declared using the **requires** keyword. It is the responsibility of the caller to make sure the preconditions hold at the call site, and it is the responsibility of the callee (i.e. the method body) to make sure that the postconditions hold upon return from the method.

Using the Cell class, we now have the following Swap method, which includes pre- and post- conditions:

```
method Swap(x: Cell, y: Cell)
  requires x != null && y != null;
  ensures x.data == old(y.data) && y.data == old(x.data);
{
```

[8] Interactive code sample: `http://rise4fun.com/Dafny/OUQP`
[9] Interactive code sample: `http://rise4fun.com/Dafny/iwHS`

```
  x.data := x.data + y.data;
  y.data := x.data - y.data;
  x.data := x.data - y.data;
}
```

To accommodate the new Cell class, we will also update the method that calls the Swap method:

```
method Main()
{
  var c, d := new Cell, new Cell;
  c.data := 10;  d.data := 20;
  Swap(c, d);
  assert d.data == 10;
  assert c.data == 20;
}
```

In this version of the swap program[10], Dafny will report a new error. Namely that the Swap method performs an assignment which may update an object not in the enclosing context's modifies clause. This is due to the fact that, while methods are allowed to read whatever memory they like, they are required to declare which parts of memory they modify. The declaration is done with a **modifies** annotation, which lists the objects (or sets of objects) whose fields may be modified. In this case, the method changes the fields of the objects x and y, so we change our method declaration to look like this:

```
method Swap(x: Cell, y: Cell)
  requires x != null && y != null;
  modifies x, y;
  ensures x.data == old(y.data) && y.data == old(x.data);
{
  x.data := x.data + y.data;
  y.data := x.data - y.data;
  x.data := x.data - y.data;
}
```

That will solve the issue with modifying the cells[11]. Dafny now reports that the postcondition of our Swap method might not hold. This might be unexpected, since almost the same code for the Swap method was verified earlier. The only difference is that we are now using instances of the Cell class. This is the source of the problem—the variables defined in the Cell classes are referenced by pointers, which means the program now behaves differently. Specifically, when x.data and y.data refer to the same object, i.e., when the references x and y are the same, the value of this data field will always be 0 after we execute the Swap method.

To see if our reasoning is correct, we can use an *assumption*. An assumption is a boolean expression, denoted by the **assume** keyword, which from that point

[10] Interactive code sample: `http://rise4fun.com/Dafny/lAKHt`
[11] Interactive code sample: `http://rise4fun.com/Dafny/rG8`

onward is treated as a verification axiom. When using an assumption, the verifier only considers execution paths where the control flow either does not reach that assumption, or reaches the assumption and finds that its condition evaluates to true. Assumption statements are helpful to use temporarily when debugging a verification attempt, but they cannot be compiled and should not be left in the final program (the compiler will complain if they are).

Placing the assumption `assume x != y;` in the body of our erroneous `Swap` method implementation causes it to verify, thus confirming our understanding that the body is correct in this case[12]. The assumption has done its job, so let's remove it and think about how to proceed.

One of the key benefits of Dafny is now clear. The Dafny verification errors produced while writing the `Swap` method show some of the subtle properties of different implementations. In fact, Dafny has exposed a problem that will lead to a key design choice. One option is to keep the current implementation which is able to swap any two different cells, but which will return cells with value 0 when asked to swap the same variable. This option requires changing the specification, either by altering the postcondition[13] or by adding a precondition that requires `x` and `y` to refer to different objects[14]. Another option is to change the implementation to allow for swapping without restrictions[15].

5 Loop Invariants

So far, the code examples we have used have consisted of a finite number of control paths. When recursion or loops are involved, the number of control paths may be infinite. To reason about such control paths, it is necessary to provide annotations at various program points along the way. For recursive calls, you supply pre- and post- conditions, as we have already seen. For loops, you supply a *loop invariant*.

A loop invariant is a boolean expression that holds at the start of every iteration. The verifier checks that the loop invariant holds at the point where control flow reaches the loop and checks that it holds again at the end of every loop iteration. Thereby, it can assume the loop invariant to hold at the very top of each iteration (meaning at the point where the loop guard is about to be evaluated), which is how the verifier reasons about the code in the loop body and after the loop. In fact, the loop invariant is the only property the verifier remembers about the variables being modified in the loop from one iteration to another, so it is important to declare a loop invariant that says enough about these variables. This is similar to the way calls are handled, where pre- and post- conditions are the only properties that the verifier takes across the call boundaries.

---

[12] Interactive code sample: `http://rise4fun.com/Dafny/B78X`
[13] Interactive code sample: `http://rise4fun.com/Dafny/YWYs`
[14] Interactive code sample: `http://rise4fun.com/Dafny/HIBe`
[15] Interactive code sample: `http://rise4fun.com/Dafny/6OL`

To demonstrate the use of loop invariants, we will use a function. In Dafny, a function body has exactly one expression, whose type corresponds to the function return type. These constructs can only be used in annotations and their utility comes from the fact that they can be used to directly express program specifications. However, functions are not part of the compiled code; they are just used to aid program verification.

Let's start by creating a recursive Fibonacci function that returns the value of the n'th number of the zero-indexed Fibonacci sequence:

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

We can use this function in the loop invariant of an iterative version of the Fibonacci method, and by making use of this invariant, Dafny can reason about the loop in this method. This method will have the following signature:

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n);
```

The method receives a natural number (n) as input, and returns a natural number (x) which is the n'th Fibonacci number. The second line is the postcondition of the method and it tells us that the returned value is indeed the n'th number of the zero indexed Fibonacci sequence.

In the body of the method, we wish to build the required Fibonacci number iteratively. This can be done by using parallel assignment to both compute the next Fibonacci number and perform the needed housekeeping of the position in the sequence, in effect updating two numbers of the sequence at once. We will use x and y to keep track of those two consecutive numbers, with y corresponding to the newly computed number and x corresponding to number computed in the previous iteration. That is to say, after i iterations of the loop, x is the i'th number of the Fibonacci sequence and y is the i+1'th.

```
method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n);
{
  var i := 0;
  x := 0;
  var y := 1;
  while (i < n)
  {
    x, y := y, x+y;
    i := i + 1;
  }
}
```

You can now see what Dafny what reports[16].

[16] Interactive code sample: `http://rise4fun.com/Dafny/xeo`

As you might expect, Dafny cannot be certain that the postcondition holds. That is because there is no loop invariant that describes the values of x, y, and i through the iterations. Let us supply a loop invariant. We begin by describing the possible values of the loop index i. We know that i starts at 0, and it will increase until it is equal to n at which point the program will exit the loop. So this is the first invariant:

```
  invariant 0 <= i <= n;
```

We also need to describe x and y in the loop invariant, but for instructional purposes, let's explore what happens if we mention just x and not y. Remember, we intend the code to maintain x as the i'th value of the Fibonacci sequence, where i is the number of iterations performed. So, we write the following loop invariant:

```
  invariant x == Fib(i);
```

Dafny can now tell that the postcondition will hold: Starting from the very top of a loop iteration, the invariants about i and x hold. If the loop guard (i < n) happens not to hold, that is, if n <= i, then the first loop invariant lets the verifier conclude i == n and the second loop invariant lets the verifier conclude x == Fib(n), which is the desired postcondition.

However, Dafny will now tell us that the second loop invariant might not be preserved by the loop. This is because the next value of x depends on the previous value of y, about which our loop invariants do not yet say anything. So, we need to provide some information about the value of y in the loop. Remember our intention about y, which is to maintain it as the i+1'th Fibonacci number. Adding the corresponding loop invariant and running Dafny will now report that the program is verified successfully[17].

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

method ComputeFib(n: nat) returns (x: nat)
  ensures x == Fib(n);
{
  var i := 0;
  x := 0;
  var y := 1;
  while (i < n)
    invariant 0 <= i <= n;
    invariant x == Fib(i);
    invariant y == Fib(i+1);
  {
    x, y := y, x+y;
```

[17] Interactive code sample: `http://rise4fun.com/Dafny/l4ey`

```
    i := i + 1;
  }
}
```

Looking back at what we just did, you may see striking similarities between loop invariants and mathematical proofs that use induction. At any time, the loop invariant says what is true after all the loop iterations so far. When no loop iterations have taken place, you need to check that the loop invariant holds initially, which corresponds to the base case in typical proofs by induction. To prove that the loop invariant holds after $k+1$ iterations (for an arbitrary k), one gets to assume that it holds after k iterations, which corresponds to assuming the inductive hypothesis when doing the inductive step in typical proofs by induction.

So now we know that if we exit the loop in our ComputeFib method, the postcondition will hold. But what if we don't exit the loop? It's clearly desirable to have Dafny assure us that the program will definitely exit the loop.

6 Termination: Variant Functions

Dafny can prove termination by using *decreases* annotations. If we can label each loop iteration with a natural number and make sure that successive iterations strictly decrease that label, then it follows that at run time the program can only execute a finite number of loop iterations, and that is all the information needed to prove that the loop eventually terminates.

More generally, instead of a natural number, we can use any value as long as we choose a *well-founded* relation which induces strictly decreasing chains, i.e. it does not admit infinite descending chains. Dafny predefines a well-founded relation on each of its types, and it extends these to lexicographically ordered tuples, which then also form a well-founded relation. The tuple of expressions that labels a loop iteration is called a *variant* and is introduced using the keyword **decreases**. In the majority of cases, Dafny is able to infer the correct decreases annotations, but sometimes these have to be made explicit by the programmer. Usually, there is a loop variable that is being increased or decreased to control the number of iterations. When the loop condition is an inequality, it is normally the distance between the two variables that is decreasing. That is what happens in the Fibonacci sequence example, where we have `i < n` as the loop condition where **while** `(i < n)`. In this case, Dafny infers that what decreases in the loop is the difference between `n` and `i`.

Similarly, infinite recursion is avoided by labelling each recursive or mutually recursive method or function with a variant, also introduced with the keyword **decreases**.

Let us start by looking at an example of a simple recursive function that returns the sum of all the elements of a sequence of integers[18]. The decreases clause in the following example allows Sum to be calling itself with a sequence

[18] Interactive code sample: `http://rise4fun.com/Dafny/PAl`

whose length is decreased by one at each invocation. Since there is a lower bound on the size of a sequence, this implies ordering on successive calls is well-founded, and thus the recursion will eventually terminate.

```
function Sum(xs: seq<int>): int
  decreases xs;
{
  if xs == [] then 0 else xs[0] + Sum(xs[1..])
}
```

Now consider a more complex example, the Ackermann function[19]. As you can see, the decreases clause has the lexicographic tuple m,n that allows Dafny to prove termination. It proves this by using size comparisons of the component values to determine whether the measure has shrunk. In this case it uses two integers, but in general each component can be of different types. The comparison works lexicographically: if the first element, in this case m, is smaller, then it doesn't matter what happens to the other values. They could increase, decrease, or stay the same. The second element is only considered if the first element does not change. Then, the second value needs to decrease. If it doesn't, then the third element must decrease, etc. For proof of termination to be possible eventually, one of the elements must decrease, and the values of any subsequent elements are not taken into consideration.

```
function Ackermann(m: nat, n: nat): nat
  decreases m, n;
{
  if m == 0 then
    n + 1
  else if n == 0 then
    Ackermann(m - 1, 1)
  else
    Ackermann(m - 1, Ackermann(m, n - 1))
}
```

Looking more closely at the Ackermann function, there are three recursive calls. In the first, m becomes one smaller, but n increases. This makes the decreases clause valid since the first element of the tuple decreases (and it doesn't matter what happens to the ones after that). In the second call, m also decreases, so the second argument is again allowed to be any value. Dafny then needs to prove that the third call obeys the termination measure. For this call, m remains the same, but n decreases, so the overall measure decreases as well. Dafny is thus able to prove the termination of the Ackermann function.

7 Lemmas

Sometimes, intricate logical steps are required to prove program correctness, but they are too complex for Dafny to discover and use on its own. An example of

[19] Interactive code sample: `http://rise4fun.com/Dafny/hUYe`

this is the Fermat method shown in section 4 on page 7. When this happens, we can often give Dafny assistance by providing a lemma, which is a theorem used to prove another result.

Lemmas allow Dafny to break a proof into two: prove the lemma, then use it to prove the final result, i.e. the correctness of the program. Splitting up the proof in this way helps Dafny to see the intermediate steps that make the proof process easier. Lemmas are particularly useful for inductive arguments, which are some of the hardest problems for theorem provers.

The most common type of lemma is a *method lemma*. A method lemma is a method which has the desired property as a postcondition. The method does not change any state, and doesn't need to be called at run time. For this reason, it is declared to be a **ghost** method. It is present solely for its effect on the verification of the program, and to help the proof of the program. A typical method lemma has the following structure:

```
ghost method Lemma(...)
  ensures (desirable property);
{
  ...
}
```

Consider the following example; we will write a method that receives an array of integers as input and returns the index of the first zero that occurs in the array, or -1 if there are no zeros in the array. This is represented by the following two postconditions:

```
ensures index < 0  ==>
   (forall i :: 0 <= i < a.Length ==> a[i] != 0);
ensures 0 <= index ==>
   index < a.Length && a[index] == 0 &&
   (forall i :: 0 <= i < index ==> a[i] != 0);
```

Let's say that the array our method receives has two properties: its elements are non-negative and its values cannot decrease by more than one unit in consecutive positions. We will add these properties as preconditions to our method:

```
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i];
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i];
```

When writing the method, we can take advantage of this property, because if we know, for example, `a[j] == 3`, then we know that we will not find a zero before `a[j+3]`. Generalizing this, if `a[j]` is non-zero, we know we will not find a zero before `a[j+a[j]]` (due to the property that successive array values decrease by at most one) and, therefore, we can accordingly jump positions in the array while looking for the zeros. We thus have our FindZero method[20].

```
method FindZero(a: array<int>) returns (index: int)
  requires a != null;
```

[20] Interactive code sample: `http://rise4fun.com/Dafny/FVFT`

```
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i];
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i];
  ensures index < 0  ==>
     (forall i :: 0 <= i < a.Length ==> a[i] != 0);
  ensures 0 <= index ==> (index < a.Length && a[index] == 0 &&
     (forall i :: 0 <= i < index ==> a[i] != 0));
{
  index := 0;
  while (index < a.Length)
    invariant 0 <= index;
    invariant forall k :: 0 <= k < index && k < a.Length ==>
       a[k] != 0;
  {
    if (a[index] == 0) { return; }
    index := index + a[index];
  }
  index := -1;
}
```

If you check this code using Dafny, it will report that it cannot prove the loop invariant, since we are jumping several positions of the array. But we can write a lemma that allows Dafny to prove that loop invariant.

We want to prove that the elements of the array between a[j] and a[j+a[j]] cannot be zero. So we write a method that receives an array with the properties mentioned previously and an index j and proves that there are no zeros between a[j] and a[j+a[j]].

```
ghost method JumpingLemma(a : array<int>, j : int)
  requires a != null;
  requires forall i :: 0 <= i < a.Length ==> 0 <= a[i];
  requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i];
  requires 0 <= j < a.Length;
  ensures forall i :: j <= i < j + a[j] && i < a.Length ==>
     a[i] != 0;
{}
```

Before writing the body of this method, we can try to use it together with the FindZero method to see if our lemma helps Dafny prove the loop invariant. (It is not a problem that we didn't write the body yet, since when evaluating the FindZero method, it will only use the postconditions from the JumpingLemma method). The key change to the **while** loop in the FindZero method is shown below[21]:

```
while (index < a.Length)
  invariant 0 <= index;
  invariant forall k :: 0 <= k < index && k < a.Length ==>
     a[k] != 0;
  {
```

[21] Interactive code sample: `http://rise4fun.com/Dafny/7bN`

```
    if (a[index] == 0) { return; }
    JumpingLemma(a, index);
    index := index + a[index];
  }
```

As you can see, Dafny is now able to prove the loop invariant in the FindZero program. It still complains, but now because it is not able to prove the postcondition of our lemma, and it was not supposed to, since we didn't write the body of the method yet.

When constructing the body, we want to iterate the array's index from j to j + a[j] (making sure that it is still smaller than the size of the array) and prove that each of those elements is non-zero. The following code will achieve this:

```
var i := j;
while(i < j + a[j] && i < a.Length)
  invariant i < a.Length ==> a[j] - (i-j) <= a[i];
  invariant forall k :: j <= k < i && k < a.Length ==>
     a[k] != 0;
{
  i := i + 1;
}
```

The first invariant represents the fact that the value of a[j] subtracted by the distance between j and i is smaller or equal than a[i]. This intuitively expresses the fact that, for example, the value of a[j+3] is not less than a[j]−3. This property can be inferred from the properties of the arrays, step by step. The other invariant states that the values of the array between a[j] and a[i] (which corresponds to the current iteration) are non-zero. Using this code as the method body will allow Dafny to prove the postconditions of the lemma. In turn, the postcondition of the lemma method allows the verifier to prove that the loop invariant in FindZero is maintained, which completes the proof of correctness of FindZero[22].

Note how the lemma and its proof were themselves expressed as a Dafny method. Essentially, because Dafny already knows how to reason about calls and loops, which are related to mathematical induction, these programming features can also be used to prove lemmas by induction. This is powerful and useful, and the power is accessible via ordinary constructs used in programming and program verification.

8 Abstraction

So far, the programs we have seen would be considered examples of programming in the small. For such programs, it is common that method specifications serve to hide the algorithmic details used in the method's implementation. When a program is larger and contains reusable components, it is desirable to hide more

[22] Interactive code sample: `http://rise4fun.com/Dafny/bqu`

implementation details than we have seen until now. In particular, it becomes desirable to hide the details of how data is represented. That is the subject of this section and the next.

To motivate and demonstrate abstraction in Dafny programs, let us introduce a class Counter, which behaves like a simple counter, with the following specification:

```
class Counter
{
  var Value: int;

  constructor Init()
    modifies this;
    ensures Value == 0;

  method GetValue() returns (x: int)
    ensures x == Value;

  method Inc()
    modifies this;
    ensures Value == old(Value) + 1;

  method Dec()
    modifies this;
    ensures Value == old(Value) - 1;
}
```

We have not shown it here, but it is easy to write implementations for these methods and to create a Main method to test the class and its specifications[23].

The variable Value is a simple way to explain the operation of the class to clients. However, suppose the implementer of the class wants to represent Value in an alternative way? To illustrate this point, suppose the Counter implementation counts the number of increment and decrement operations; then Value can be represented as the difference between those two counts. We declare two more variables:

```
var incs: int;
var decs: int;
```

These variables can be initialized in the constructor and updated appropriately in the Inc and Dec methods.

We would now like to change the implementation of GetValue to compute the return value in terms of incs and decs, rather than Value. That is, we want the implementation to do x := incs - decs, but we want the specification to still show the postcondition as x == Value, which is simpler. To verify this postcondition, it now becomes necessary to make explicit the relationship between the more abstract view of the class (Value) and the more concrete view of the implementation (incs and decs).

[23] Interactive code sample: `http://rise4fun.com/Dafny/K9iD`

One way to make this relationship explicit would be to add Value == incs − decs to the postcondition of the constructor and to the pre- and post- conditions of the methods. This would not be satisfactory, for two reasons. First, if the relation ever were to change, we would have to update the program text in many places. Second, and more importantly, the details of this relation are to remain private with the class implementation; clients of the class should not have to be concerned with the details. So, we instead write the relation in the body of a function that we shall name Valid:

```
function Valid(): bool
  reads this;
{
  Value == incs − decs
}
```

In this declaration, we see a new annotation: **reads**. Just like methods have to declare which parts of the object store they may update, functions have to declare which parts of the object store they depend on. We will have more to say about these so-called *frame* issues in the next section.

Finally, we need to use and enforce this relationship, which we do by adding Valid() as postcondition of the constructor and as a pre- and post- condition of all methods which ensures that if the relationship holds before invocation it must also do so after execution. Since we now no longer need Value in the compiled code, we can mark it as ghost.

By applying the suggested changes, we can now verify the program[24]. Because Value is a ghost variable, it will not be present in the compiled code and hence it is necessary to have the GetValue method so that we can have access to the actual value of the Counter instance we will be using.

```
class Counter
{
  // public variable
  ghost var Value: int;
  // private variables
  var incs: int;
  var decs: int;

  function Valid(): bool
    reads this;
  {
    Value == incs − decs
  }

  constructor Init()
    modifies this;
    ensures Valid();
    ensures Value == 0;
```

[24] Interactive code sample: `http://rise4fun.com/Dafny/or9`

```
  {
    incs, decs, Value := 0, 0, 0;
  }

  method GetValue() returns (x: int)
    requires Valid();
    ensures x == Value;
  {
    x := incs - decs;
  }

  method Inc()
    requires Valid();
    modifies this;
    ensures Valid();
    ensures Value == old(Value) + 1;
  {
    incs, Value := incs + 1, Value + 1;
  }

  method Dec()
    requires Valid();
    modifies this;
    ensures Valid();
    ensures Value == old(Value) - 1;
  {
    decs, Value := decs + 1, Value - 1;
  }
}

method Main()
{
  var c := new Counter.Init();
  c.Inc();  c.Inc();
  c.Dec();
  c.Inc();
  assert c.Value == 2;
}
```

9 Dynamic Frames

In the previous section, we presented a way to specify the behavior of a class in terms of a ghost variable. We related the value of the ghost variable to the values of implementation variables in a function Valid, which we mentioned in method pre- and post- conditions (essentially as a class invariant [14]). This specification idiom also applies when a class implementation uses more complicated data structures, except that we need to extend the idiom to better deal with framing, that is, **reads** and **modifies** clauses.

Let us continue with the Counter example from the previous section, but use our previous Cell class, from section 4 on page 7, for the incs and decs variables instead of integers:

```
var incs: Cell;
var decs: Cell;
```

We let the constructor initialize these fields by allocating new Cell objects:

```
incs, decs := new Cell, new Cell;
incs.data, decs.data, Value := 0, 0, 0;
```

In the other methods, we replace incs and decs with incs.data and decs.data, respectively. We apply that replacement in function Valid, too; in addition, we need Valid to express that the two Cell references are non-null and distinct:

```
incs != null && decs != null && incs != decs &&
Value == incs.data - decs.data
```

If we try to verify the resulting program[25], we get complaints about violating the declared frames—function Valid is reading more than its **reads** clause allows, and methods Inc and Dec modify more than their **modifies** clauses allow. A quick fix to this problem is to change these three frame specifications to also list the object referenced by incs and decs[26]. This quick fix is unsatisfactory, because it exposes the implementation fields incs and decs in public specifications. We need a way to abstract over these fields in the specifications.

A solution to this problem is *dynamic frames* [6,7]. In Dafny, a dynamic frame is simply an expression that denotes a set of objects and that is used in **reads** and **modifies** clauses. The frame is dynamic in the sense that the expression may evaluate to different sets of objects, depending on the program state. We will now describe the standard idiom for using dynamic frames in Dafny.

We start by introducing a variable Repr, which will stand for the set of objects in the object's *representation*. In the case of Counter, that set of objects is {**this**, incs, decs}. The type of Repr is **set<object>**, and since the field will be used only in specifications, not at run time, we declare it as **ghost**.

```
ghost var Repr: set<object>;
```

We change the constructor to initialize Repr to the desired set, and we change the frames of methods Inc and Dec to:

```
modifies Repr;
```

which says that the method is allowed to modify an object in the set Repr. More precisely, this **modifies** clause gives the method license to modify the fields of any object o that, at the time the method is invoked, is in the set denoted by Repr.

Note that the frame for the constructor Init is still just **this**. This is desirable and is part of the standard idiom. First, at the time the constructor is called,

[25] Interactive code sample: `http://rise4fun.com/Dafny/H065`
[26] Interactive code sample: `http://rise4fun.com/Dafny/dqu`

the field Repr has an unknown value, so it would not make sense to list it in the **modifies** clause. Second, the data fields to which Init assigns belong to objects that were created after Init was invoked, and Dafny allows any such newly allocated objects to be modified, without any need to mention them in the **modifies** clause. Third, the only other modifications performed by Init are to fields of **this**, as permitted by **modifies this**.

We have a few more things to address before we are done. If we tried to verify the program as it stands now[27], the verifier would complain that methods Inc and Dec do not respect their frames, which are specified by **modifies** Repr. The reason for this is that the preconditions of these methods do not say anything about the contents of Repr. To address this problem, we want to change the definition of Valid to say something about Repr. So, let us change the definition of Valid to this:

```
function Valid(): bool
  reads this, Repr;
{
  this in Repr && null !in Repr &&
  incs in Repr && decs in Repr &&
  incs != decs &&
  Value == incs.data - decs.data
}
```

We did several changes here. First, validity now implies that **this** is in Repr. Second, by convention, we exclude **null** from Repr (this isn't strictly necessary, but we strongly recommend it—in our experience, the verification errors that can arise from allowing **null** in Repr may not make it evident that the problem is somehow related to **null** and can therefore be confusing). Third, we list incs and decs as being contained in Repr. Fourth, since Repr does not contain **null**, it follows that incs and decs are non-null, so we don't need to mention those properties explicitly. Finally, we changed the **reads** clause, which requires some further explanation.

Why does the frame of Valid explicitly list **this**? Why doesn't just **reads** Repr suffice?

It may seem that **reads** Repr would suffice, since we intend **this** to be included in Repr. However, when checking that the body of Valid adheres to the reads frame, the verifier does not know anything about Repr. It may help to consider how the body of Valid would be evaluated from an arbitrary state at run time. The first conjunct (**this in** Repr) can evaluate to either **false** or **true**. If it evaluates to **false**, the function can return **false** without evaluating the other conjuncts. If it evaluates to **true**, then **this in** Repr holds. In either case, note that the remaining conjuncts are evaluated only if **this in** Repr holds, which means that **reads** Repr implies the fields of **this** can be read. But to read the first conjunct itself, which includes the field **this**.Repr, the verifier needs to know that **this** is allowed by the frame. To break this bootstrapping circularity, we simply list both **this** and Repr in the **reads** clause.

[27] Interactive code sample: `http://rise4fun.com/Dafny/aAwn`

By changing Valid as described above, our class verifies[28], but we're not quite finished yet, because our postcondition specifications are not strong enough to be useful for clients. Consider some client code, like the Main method we used to test the class in the previous section[29]:

```
method Main()
{
  var c := new Counter.Init();
  c.Inc();  c.Inc();
  c.Dec();
  c.Inc();
  assert c.Value == 2;
}
```

This code gives rise to several frame violation errors. The problem is that the postconditions of Init and the other methods do not say enough about the objects in Repr. For example, the specifications would allow a method to change incs to point to a Cell object in use by another Counter object[30]:

```
method ShareMe(cnt: Counter)
  requires Valid() && cnt != null && cnt.Valid();
  modifies Repr;
  ensures Valid() && Value == old(Value);
{
  if (incs.data == cnt.incs.data) {
    incs.data := cnt.incs.data;
  }
}
```

Calling this method could lead to two Counter objects sharing the same representation. Dafny allows such specifications and implementations, which are sometimes useful. The situation is fine, because the possibility of sharing does not escape the verifier. Indeed, this is why the verifier complains about the Main client above.

To correct the problem, we show the final part of the standard dynamics-frame idiom in Dafny. To the constructor, we add the postcondition:

```
ensures fresh(Repr - {this});
```

which says that, upon return from Init, all objects other than **this** in Repr are ones that were allocated after Init was invoked. In other words, Init sets Repr to some set of newly allocated objects, except it may also possibly contain **this**. This specification is abstract enough to not explicitly mention the private Counter implementation, and yet strong enough to allow a client to follow up the call to Init by a call to a method (like Inc or Dec) declared with **modifies** Repr—after all, any object newly allocated in Init is also newly allocated in the caller, Main,

[28] Interactive code sample: `http://rise4fun.com/Dafny/MsRQ`
[29] Interactive code sample: `http://rise4fun.com/Dafny/uT0k`
[30] Interactive code sample: `http://rise4fun.com/Dafny/7RSR`

so the caller is allowed to modify the objects that, upon return from Init, are contained in Repr.

We add a similar postcondition to the mutating methods Inc or Dec:

```
ensures fresh(Repr - old(Repr));
```

This postcondition says that any objects added to Repr are newly allocated. The program now verifies[31]:

```
class Cell {
  var data: int;
}

class Counter
{
  // public variable
  ghost var Value: int;
  ghost var Repr: set<object>;
  // private variables
  var incs: Cell;
  var decs: Cell;

  function Valid(): bool
    reads this, Repr;
  {
    this in Repr && null !in Repr &&
    incs in Repr && decs in Repr &&
    incs != decs &&
    Value == incs.data - decs.data
  }

  constructor Init()
    modifies this;
    ensures Valid() && fresh(Repr - {this});
    ensures Value == 0;
  {
    incs, decs := new Cell, new Cell;
    incs.data, decs.data, Value := 0, 0, 0;
    Repr := {this};
    Repr := Repr + {incs, decs};
  }

  method GetValue() returns (x: int)
    requires Valid();
    ensures x == Value;
  {
    x := incs.data - decs.data;
  }
```

[31] Interactive code sample: `http://rise4fun.com/Dafny/fGVu`

```
  method Inc()
    requires Valid();
    modifies Repr;
    ensures Valid() && fresh(Repr - old(Repr));
    ensures Value == old(Value) + 1;
  {
    incs.data, Value := incs.data + 1, Value + 1;
  }

  method Dec()
    requires Valid();
    modifies Repr;
    ensures Valid() && fresh(Repr - old(Repr));
    ensures Value == old(Value) - 1;
  {
    decs.data, Value := decs.data + 1, Value - 1;
  }
}

method Main()
{
  var c := new Counter.Init();
  c.Inc();  c.Inc();
  c.Dec();
  c.Inc();
  assert c.Value == 2;
}
```

Our explanation of the standard dynamic-frames idiom may have been long, but the idiom is simple to follow: First, declare the ghost variable Repr, and let Valid describe the contents of Repr as we did above. Then, declare constructors (like Init) to include the following specification:

```
modifies this;
ensures Valid() && fresh(Repr - {this});
```

declare mutating methods (like Inc and Dec) to include the following specification:

```
requires Valid();
modifies Repr;
ensures Valid() && fresh(Repr - old(Repr));
```

and declare query methods (like GetValue) to include the following specification:

```
requires Valid();
```

These idiomatic specifications may be verbose, but they lend themselves to modular specifications and can flexibly be adapted to allow various forms of sharing of data structures.

10 Conclusion

Dafny is a general-purpose specification language and verifier. Through programmer-provided specifications and other annotations, it makes possible the verification of the functional correctness of a program. The symbol manipulation in the proofs themselves is performed in a mostly automatic fashion, hidden from the user of the tool.

In these lecture notes, we have introduced Dafny through a series of simple examples. We have demonstrated a key strength of Dafny, namely abstraction and dynamic frames, which allow us to scale to larger programs (for further information on this see, for example, [13]). By learning and using Dafny, you can construct programs that behave as specified. The concepts you learn also apply when programming in other languages, even if you don't have a verification tool and instead do the reasoning informally.

Acknowledgments

We are indebted to the reviewers, who gave a lot of themselves to take us through various drafts of these lecture notes. Thank you!

References

1. Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005* (Sept. 2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds., vol. 4111 of *Lecture Notes in Computer Science*, Springer, pp. 364–387.
2. Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. Specification and verification: The Spec# experience. *Communications of the ACM 54*, 6 (June 2011), 81–91.
3. Cohen, E., Dahlweid, M., Hillebrand, M. A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, Aug. 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 23–42.
4. de Moura, L., and Bjørner, N. *Z3: An Efficient SMT Solver*, vol. 4963 of *Lecture Notes in Computer Science*. Springer Berlin, Berlin, Heidelberg, Apr. 2008, ch. 24, pp. 337–340.
5. Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.
6. Kassios, I. T. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods* (Aug. 2006), J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085 of *Lecture Notes in Computer Science*, Springer, pp. 268–283.
7. Kassios, I. T. The dynamic frames theory. *Formal Aspects of Computing 23* (2011), 267–288.

8. Klebanov, V., Müller, P., Shankar, N., Leavens, G. T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K. R. M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., and Weiss, B. The 1st verified software competition: Experience report. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods* (June 2011), M. Butler and W. Schulte, Eds., vol. 6664 of *Lecture Notes in Computer Science*, Springer, pp. 154–168.
9. Koenig, J. Dafny guide, 2011.
10. Koenig, J., and Leino, K. R. M. Getting started with Dafny: A guide. IOS Press, 2012. Summer School Marktoberdorf 2011 lecture notes, to appear.
11. Leino, K. R. M. Specification and verification of object-oriented software. In *Engineering Methods and Tools for Software Safety and Security*, M. Broy, W. Sitou, and T. Hoare, Eds., vol. 22 of *NATO Science for Peace and Security Series D: Information and Communication Security*. IOS Press, 2009, pp. 231–266. Summer School Marktoberdorf 2008 lecture notes.
12. Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning* (Berlin, Heidelberg, Apr. 2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 348–370.
13. Leino, K. R. M., and Monahan, R. Dafny meets the verification benchmarks challenge. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010* (Aug. 2010), G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, Eds., vol. 6217 of *Lecture Notes in Computer Science*, Springer, pp. 112–126.
14. Meyer, B. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.
15. Weiss, B. *Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.
16. Wiles, A. Modular elliptic curves and Fermat's last theorem. *The Annals of Mathematics 141*, 3 (May 1995), 443–551.