

Dafny

Modular proofs, mutable state, and abstraction

CMPUT 664 · Formal Verification · September 28, 2026

Today

- Extend our Hoare proofs to procedure calls.
- Understand Dafny's treatment of heap effects and abstraction.
- Connect ghost state, dynamic frames, and mathematical definitions.
- Follow how the verifier turns these into SMT obligations.

What we already know

For a specification $\{P\} S \{Q\}$, verification generates logical obligations.

- Assignments contribute substitutions.
- Branches contribute guarded obligations.
- Loops need an invariant that is initially true, preserved, and useful at exit.

What should happen when S calls another method?

A program built from procedures

```
1 procedure AdvanceTwice(n) returns r
2   a := Next(n);
3   r := Next(a)
4
5 procedure Client() returns r
6   requires true
7   ensures r = 6
8   r := AdvanceTwice(4)
```

```
1 procedure Next(x) returns y
2   y := x + 1
```

Our initial goal is just the client's postcondition:
 $r = 6$.

We can expand these calls and use the assignment rule we already know.

Backward proof with the calls expanded

Expanding `AdvanceTwice(4)` and its two calls to `Next` gives:

```
1  a := 4 + 1;  
2  r := a + 1;  
3  { r = 6 }           // required postcondition
```

Read the calculation upward through the program:

Position	Required condition
After <code>r := a + 1</code>	$r = 6$
Before <code>r := a + 1</code>	$a + 1 = 6$
Before <code>a := 4 + 1</code>	$(4 + 1) + 1 = 6 \equiv \text{true}$

The VC is $\text{true} \Rightarrow \text{true}$.

Multiple callers and an implementation change

Suppose several procedures call `Next`. With expansion, each caller's proof incorporates its body.

Original implementation

```
1 procedure Next(x) returns y
2   y := x + 1
```

Replacement implementation

```
1 procedure Next(x) returns y
2   y := x;
3   y := y + 1
```

The expanded code changes in each affected caller's proof, although the returned value is unchanged.

Can callers depend on the guaranteed result, so we can verify an implementation change independently?

Contracts as interfaces for separate proofs

We now choose reusable contracts for the helpers:

```
1 procedure AdvanceTwice(n) returns r
2   requires true
3   ensures r = n + 2
4   a := Next(n);
5   r := Next(a)
6
7 procedure Client() returns r
8   requires true
9   ensures r = 6
10  r := AdvanceTwice(4)
```

```
1 procedure Next(x) returns y
2   requires true
3   ensures y = x + 1
4   y := x + 1
```

A caller uses the contract. The callee's body must establish it.

If only **Next**'s body changes, its proof must be checked again. Caller proofs that use the unchanged contract remain valid.

The structure of the example

The declared contracts are claims we still need to prove.

1. Verify the individual procedures against their contracts.

For each body, assume its precondition and prove its postcondition:

Procedure	Assumed on entry	Required on return
<code>Next(x)</code>	true	$y = x + 1$
<code>AdvanceTwice(n)</code>	true	$r = n + 2$

2. Use the verified contracts to establish the overall client behavior.

Show that `Client`'s call to `AdvanceTwice(4)` establishes its postcondition $r = 6$.

How does a proof about a procedure's **body** justify using its contract at a **call**?

Hoare's rule of invocation makes that connection precise.

Hoare's rule of invocation

For a nonrecursive procedure p with no parameters and body B :

$$\frac{p \equiv B \quad \{P\} B \{Q\}}{\{P\} \text{ call } p \{Q\}}$$

First prove a property of the body. A call can then use that property.

This is a **partial-correctness** rule: if the call returns, Q holds.

Adapted to our notation from Hoare, *Procedures and Parameters* (1971), pp. 104–105.

Proof 1: the body of **Next**

The declared contract permits every integer input and ensures $y = x + 1$ on return.

```
1 procedure Next(x) returns y
2   { true }
3   y := x + 1;
4   { y = x + 1 }
```

The assignment rule gives

$$\text{wp}(y := x + 1, y = x + 1) \equiv x + 1 = x + 1.$$

Thus $\text{true} \Rightarrow \text{wp}(B, Q)$ is valid.

This proves the contract for every permitted input.

Formal parameters and actual arguments

The proved specification uses `Next`'s own names:

$$\{\text{true}\} \ y := \text{Next}(x) \ \{y = x + 1\}.$$

At `a := Next(n)`, instantiate **both** parts of the contract:

Formal name	Name at this call
Input x	Actual argument n
Result y	Fresh local result a

$$\{\text{true}\} \ a := \text{Next}(n) \ \{a = n + 1\}.$$

We now know the contract at this call site.

How do we turn that contract into commands for backward reasoning, without inspecting `Next`'s body?

What **havoc a** means

To model the call's result using its contract, first **allow any integer**, then **retain only results satisfying the postcondition**.

havoc a performs the first step: assign an arbitrary integer to **a**, forgetting its previous value. The proof covers every choice; other variables keep their values.

For a call with $n = 4$:

```
1 havoc a;           // a can now be any integer
2 assume a = 5;       // retain only the result allowed by the contract
```

With the weaker assumption **a > 4**, the possible results would be 5, 6, 7,

Backward reasoning must account for all choices:

$$\text{wp}(\mathbf{havoc\ } a, R) = \forall v. R[v/a].$$

Here v is a fresh integer and $R[v/a]$ replaces a by v in the desired condition.

Backward reasoning through a call

At $a := \text{Next}(n)$, substitute $x \mapsto n$ and $y \mapsto a$ in the contract:

Declared contract

```
1 procedure Next(x) returns y
2   requires true
3   ensures y = x + 1
```

Verification model of the call

```
1 assert true;
2 havoc a;
3 assume a = n + 1;
```

- **requires** $\rightarrow$ **assert**: the caller must prove the precondition; here it is trivially true.
- **ensures** $\rightarrow$ **assume**: the caller may use the guarantee. A separate proof checks that **Next**'s body establishes it.

These commands are verification pseudocode generated from the contract.

Working backward, **assume** gives $\Rightarrow$, **havoc** gives $\forall$, and **assert** gives $\wedge$:

$$\text{wp}_C(a := \text{Next}(n), R) = \text{true} \wedge \forall v. (v = n + 1 \Rightarrow R[v/a]).$$

Proof 2: backward through the last call

```
1 a := Next(n);  
2 r := Next(a);  
3 { r = n + 2 }      // required postcondition
```

Start with $r = n + 2$ and cross $r := \text{Next}(a)$:

$$\begin{aligned} & \text{wp}_C(r := \text{Next}(a), r = n + 2) \\ &= \text{true} \wedge \forall v. (v = a + 1 \Rightarrow v = n + 2) \\ &\equiv a + 1 = n + 2. \end{aligned}$$

This is the condition we need **between the two calls**. It becomes the target for the first call.

Proof 2: backward through the first call

Let $R(a, n) \equiv a + 1 = n + 2$. Cross $a := \text{Next}(n)$:

$$\begin{aligned}\text{wp}_C(a := \text{Next}(n), R) &= \text{true} \wedge \forall v. (v = n + 1 \Rightarrow R(v, n)) \\ &\equiv n + 2 = n + 2 \\ &\equiv \text{true}.\end{aligned}$$

```
1 { true }           // calculated precondition
2  a := Next(n);
3 { a + 1 = n + 2 }  // calculated intermediate condition
4  r := Next(a);
5 { r = n + 2 }      // required postcondition
```

The VC is $\text{true} \Rightarrow \text{true}$, now proved using the contract of `Next`.

Proof 3: backward from the client's postcondition

```
1 procedure Client() returns r
2   requires true
3   ensures r = 6
4   r := AdvanceTwice(4)
```

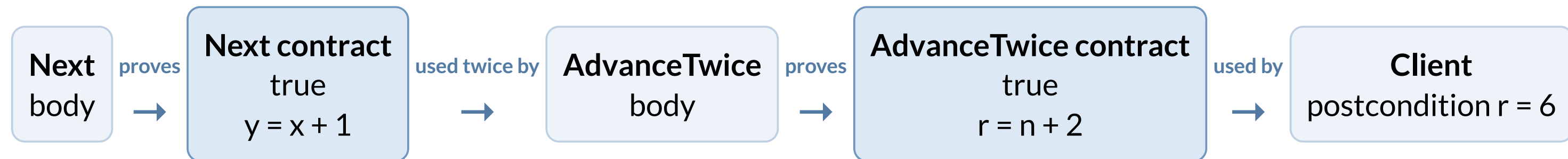
Use *AdvanceTwice*'s contract, starting with the target $r = 6$:

$$\begin{aligned} \text{wp}_C(r := \text{AdvanceTwice}(4), r = 6) \\ &= \text{true} \wedge \forall v. (v = 4 + 2 \Rightarrow v = 6) \\ &\equiv \text{true}. \end{aligned}$$

The client's verification condition is $\text{true} \Rightarrow \text{true}$.

Each body is analyzed backward from its own postcondition. At a call, the calculation uses the callee's contract.

How the proofs connect



A callee proves the guarantee that its callers use.

Every caller also proves the precondition at each call.

Modular verification checks these separate obligations and combines the results. A failed body proof leaves its advertised guarantee unestablished.

A weaker contract breaks the connection

Keep the bodies unchanged, but replace **Next**'s guarantee by $y > x$.

Backward from $r = n + 2$, the last call now requires

$$\forall v. (v > a \Rightarrow v = n + 2).$$

The inequality $v > a$ cannot guarantee the required equality.

For example, the two call summaries permit $n = 4, a = 6, r = 8$, violating $r = n + 2$.

Next still satisfies its weaker contract. Its current body would return 6 after two calls, but that fact is no longer available through the contract.

Procedure proofs and loop proofs

Boundary	What the surrounding proof uses	What must be checked separately
Procedure call	Contract, instantiated for this call	Body establishes the postcondition from the precondition
Loop	Invariant, plus the false guard at exit	Invariant holds initially and survives an iteration

Both summarize code so the surrounding proof can proceed.

Dafny expresses procedure contracts with `requires` and `ensures`. We will now see how it checks the resulting obligations.

Demo 1: checking the procedure proof

The program is the same `Client`, `AdvanceTwice`, and `Next` example.

The original Dafny paper

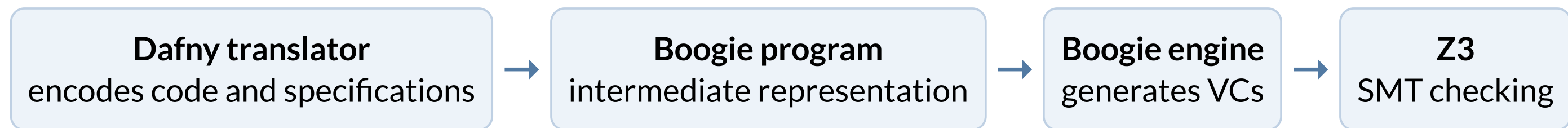
K. Rustan M. Leino, *Dafny: An Automatic Program Verifier for Functional Correctness* (2010)

The central question: how can an SMT-based verifier establish functional correctness of programs with mutable, linked data?

The paper combines a specification language, a logical encoding, and automated checking for programs with mutable data.

[Paper](#)

The verification pipeline



The programmer supplies specifications and proof structure. The tool translates them and attempts the obligations automatically.

VS Code displays results through the language server. The compiler can also produce executable code.

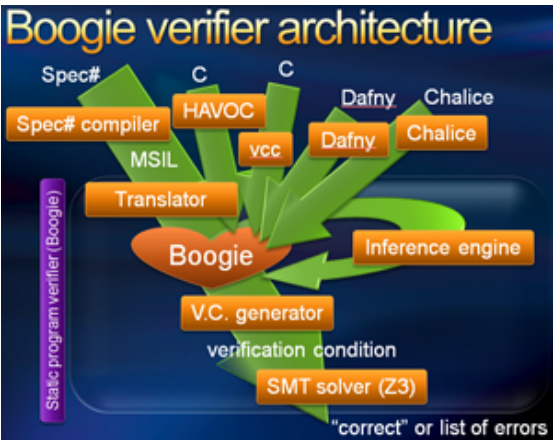
Boogie: a language and a verification engine

The **Boogie language** is an intermediate representation for verification: procedures, contracts, mutable variables, maps, and commands such as `assert`, `assume`, and `havoc`.

The **Boogie engine** processes that representation, generates verification conditions, and asks an SMT solver to check them.

Component	Responsibility
Dafny translator	Encode Dafny's semantics and required checks as a Boogie program
Boogie engine	Use contracts and invariants to summarize calls and loops; generate logical obligations
Z3	Check the resulting formulas

This separation lets different source languages reuse Boogie's verification machinery.



The Boogie proof task for **AdvanceTwice**

The call summaries we derived are the kind of commands Boogie uses during verification.

Procedure to verify

```
1 requires true
2 ensures r = n + 2
3
4 a := Next(n);
5 r := Next(a);
```

Next(x) guarantees $y = x + 1$.

Its implementation has a separate proof.

Boogie generates a VC requiring these assertions to hold on every modeled path.

This is a hand-simplified model; actual Dafny output includes additional declarations and checks.

Simplified verification model

```
1 assert true;           // first call's requires
2 havoc a;
3 assume a == n + 1;     // first call's ensures
4 assert true;           // second call's requires
5 havoc r;
6 assume r == a + 1;     // second call's ensures
7 assert r == n + 2;     // AdvanceTwice's ensures
```

The formula passed to Z3

For this model, backward calculation gives the verification condition

$$\forall n, a, r \in \mathbb{Z}. \quad (a = n + 1 \wedge r = a + 1) \Rightarrow r = n + 2.$$

Boogie asks Z3 whether its negation is satisfiable: can some integers satisfy

$$a = n + 1 \wedge r = a + 1 \wedge r \neq n + 2 ?$$

unsat: no such values exist, so this obligation is proved.

A satisfying model exposes a failing obligation; an unknown result or timeout leaves it unproved. Results are mapped back to the Dafny source.

Why heap reasoning matters to Dafny

The original Dafny paper applies this machinery to **functional correctness of programs that manipulate mutable, linked objects**.

A specification must describe both the **intended change and what the program preserves**. Shared objects make that harder than describing an integer result.

This motivates Dafny's **frames, ghost state, and representation specifications**. We will introduce them through a mutable counter, beginning with one cell.

The increment now updates an object

Keep the operation “add one,” but store the integer in a `Cell` object with a mutable field `value`.

Return an integer

```
1 procedure Next(x) returns y
2   y := x + 1
```

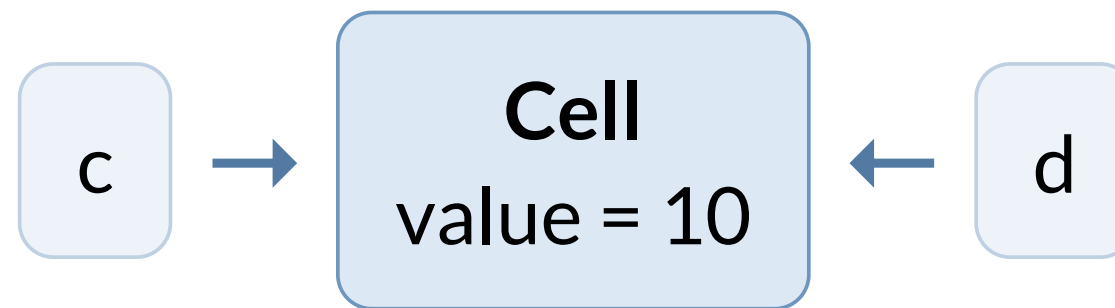
Update an existing cell

```
1 procedure Bump(c)
2   c.value := c.value + 1
```

The **heap** stores these objects. Different references can reach the same object, so we must ask: **what else can observe this update?**

A reference does not identify a separate object

Suppose the caller remembers `d.value`, then updates `c.value`.



If `c == d`, changing `c.value` also changes what `d.value` denotes.

A claim that the second value is unchanged needs information about the heap and possible sharing.

Frames: what may a method change?

`Bump(c)` should increment `c.value`. A caller also needs to know whether an unrelated cell `d` keeps its value.

A **write frame** specifies the part of the heap a method is permitted to modify. Pre-existing locations outside that frame stay unchanged.

For our example, choose the frame to contain only the object referenced by `c`:

Object	Permission during the call
<code>c</code>	Its fields may change
Any other pre-existing object	Its fields must stay unchanged

If `c != d`, the frame protects `d.value`. If `c == d`, both names refer to an object inside the frame.

Dafny declares a method's write frame with **`modifies`**.

A write frame in Dafny

```
1 method Bump(c: Cell)
2   modifies c
3   ensures c.value == old(c.value) + 1
4 {
5   c.value := c.value + 1;
6 }
```

modifies `c` gives permission to update the fields of the object referenced by `c`. It does not require every field to change.

ensures specifies the required result. Here `old(c.value)` means the field value at method entry.

The heap in the logical model

Think of a heap as a map $H[\text{object}, \text{field}]$.

For this call, the result specification gives

$$H'[c, \text{value}] = H[c, \text{value}] + 1.$$

The object-level frame gives, for pre-existing objects,

$$o \neq c \quad \Rightarrow \quad H'[o, f] = H[o, f].$$

Ghost state: information kept for the proof

So far, we have specified `Bump`'s update to `c` and its write frame. We have not yet written the caller's claim about `d`.

We want `d.value` after the call to equal its value before the call. To compare them, we record the earlier integer in a **ghost variable**: storage used only for the proof.

The programmer writes this in the caller:

```
1 ghost var saved := d.value; // remember d's value before the call
2 Bump(c);
3 assert d.value == saved;    // state the preservation claim
```

The verifier checks it: with `modifies c` and the caller's precondition `c != d`, the frame preserves `d.value`, proving the assertion. If the references may alias, the proof fails.

Dafny does not invent this snapshot or assertion. It checks the supplied proof annotations and erases ghost state from compiled code.

Demo 2: which facts survive a call?

```
1 ghost var saved := d.value;  
2 Bump(c);  
3 assert d.value == saved;
```

Functions inside contracts

So far, we have written formulas directly in contracts. As specifications grow, **functions let us name and reuse mathematical expressions.**

Return briefly to **Next**: its postcondition was $y == x + 1$.

Define the expected value

```
1 ghost function NextValue(x: int): int
2 {
3   x + 1
4 }
```

Use it in the contract

```
1 method Next(x: int) returns (y: int)
2   ensures y == NextValue(x)
3 {
4   y := x + 1;
5 }
```

The function defines a value without changing program state. Here Dafny uses $\text{NextValue}(x) == x + 1$ to prove the method's postcondition. **ghost** makes this helper available only for specifications and proofs.

This example just names arithmetic. What if a specification function reads an object's fields?

Read frames: what does a function depend on?

For our cell, a function can name the observed value. We can then write **Bump**'s postcondition using that name:

```
1 ghost function Value(c: Cell): int
2   reads c
3 {
4   c.value
5 }
```

```
1 method Bump(c: Cell)
2   modifies c
3   ensures Value(c) == old(Value(c)) + 1
4 {
5   c.value := c.value + 1;
6 }
```

Unlike **NextValue(x)**, **Value(c)** depends on the heap. The same reference can produce a different value after its fields change.

We write **reads c** to declare which objects' fields the function may read. Dafny checks the definition against this read frame.

Demo 2b: a function observes the heap

```
1 ghost var beforeC := Value(c);  
2 ghost var beforeD := Value(d);  
3 Bump(c);  
4 assert Value(c) == beforeC + 1;  
5 assert Value(d) == beforeD;
```

With $c \neq d$, both assertions verify: the postcondition explains the change to `Value(c)`; the frames explain why `Value(d)` is preserved.

`beforeC` is a ghost variable that remembers a value before one call. **We can also keep ghost state across calls to describe what an object represents.**

Ghost state as an abstract model

Suppose a queue contains **A**, then **B**, then **C**, in that order.

What the program stores

```
1 Array: [B, C, empty, A]
```

What the proof records

```
1 Model: [A, B, C]
```

The next item is **A**, in the last slot. After **A**, we wrap to the start of the array to reach **B** and **C**.

This ghost sequence records the order in which the items should leave the queue.

The contract can say: removing an item returns the first element of **Model** and removes that element from **Model**.

The implementation proof must show that the array and its indices still represent the model. This keeps storage details out of the description of queue behavior.

Next, a small counter will show how code keeps a model consistent with its stored state.

A minimal model: complete counter example

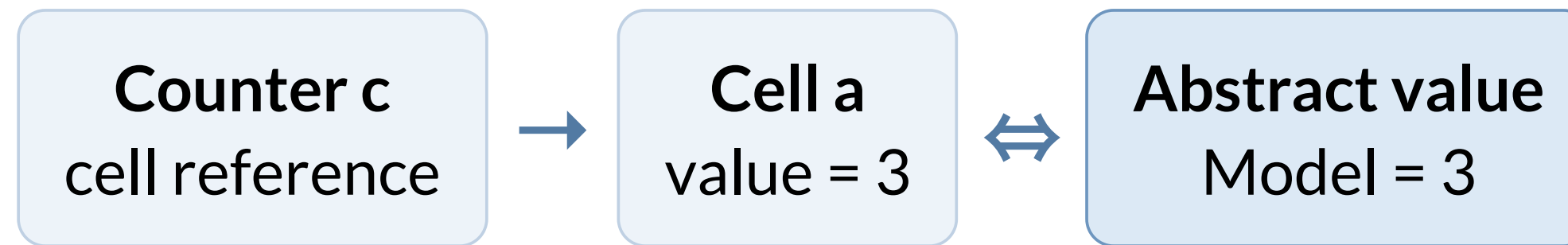
```
1 class Cell {  
2   var value: int  
3 }
```

```
1 method Client(c: Counter)  
2   requires c.cell.value == c.Model  
3   modifies c, c.cell  
4 {  
5   ghost var before := c.Model;  
6   c.Increment();  
7   assert c.Model == before + 1;  
8   assert c.cell.value == c.Model;  
9 }
```

```
1 class Counter {  
2   var cell: Cell  
3   ghost var Model: int  
4   constructor()  
5     ensures cell.value == Model  
6     ensures Model == 0  
7   {  
8     var a := new Cell;  
9     a.value := 0;  
10    cell := a;  
11    Model := 0;  
12  }  
13  method Increment()  
14    requires cell.value == Model  
15    modifies this, cell  
16    ensures cell.value == Model  
17    ensures Model == old(Model) + 1  
18  {  
19    cell.value := cell.value + 1;  
20    Model := Model + 1;  
21  }  
22 }
```

How the code maintains the model

`cell.value` is the count used by the program. `Model` is a proof-only count, erased from the executable.



1. **Initialize:** the `Counter` constructor sets `cell.value` and `Model` to zero.
2. **Assume agreement:** `Increment` requires `cell.value == Model` on entry.
3. **Update both:** its body increments the stored count and the ghost count.
4. **Prove the result:** its postconditions restore agreement and show that `Model` increased by one. The client can therefore conclude that the stored count increased too.

The counter makes the agreement rule easy to see. In the queue example, that rule would connect the array and its indices to the logical sequence.

Demo 3: checking the ghost model

```
1 method Increment()  
2   requires cell.value == Model  
3   modifies this, cell  
4   ensures cell.value == Model  
5   ensures Model == old(Model) + 1
```

One set can name the objects we may update

Our counter has two objects: the counter `c` and its backing cell `a`.

Inside the counter, `this` refers to `c` and `cell` refers to `a`. The method frame

```
1 modifies this, cell
```

therefore permits changes to the fields of **those two objects**.

We can also keep their references in a ghost set named `Repr`:

$$\text{Repr} = \{c, a\} = \{\text{this}, \text{cell}\}.$$

Then **`modifies Repr`** names the same objects. `Repr` is just a programmer-chosen name for the set.

For the counter as written, `modifies this, cell` is enough. A set gives us a way to name a collection of objects together.

What changes in a dynamic frame?

The programmer updates **Repr**; Dafny does not update it automatically.

Suppose **cell** points to **a** and **Repr = {this, a}**. To replace the cell, we write this code inside the counter:

```
1 var b := new Cell;  
2 b.value := cell.value;  
3 cell := b;           // change the backing cell  
4 Repr := {this, b};    // explicitly update the ghost set
```

If we omit the last assignment, **Repr** still contains **the old cell a**, even though **cell** now points to **b**.

After both assignments, a later call with **modifies Repr** permits updates to the counter and **b**.

A ghost history for the counter

`Model = 3` records the current count. Suppose we also want the proof to remember the counts produced by earlier increments: **3, 2, 1**, newest first.

That needs a structured ghost value. Define an immutable list:

```
1 datatype List<T> = Nil | Cons(head: T, tail: List<T>)
```

`Nil` is empty; `Cons` adds one value at the front. For example:

```
1 ghost var history := Cons(3, Cons(2, Cons(1, Nil)));  
2 ghost var updated := Cons(4, history);
```

These are mathematical values, so constructing `updated` preserves `history`.

We now want to state: **recording one more count increases the history length by one.**

A function describes the ghost history

To express that property, define the length of the list:

```
1 ghost function Length<T>(xs: List<T>): nat
2   decreases xs
3 {
4   match xs
5   case Nil => 0
6   case Cons(_, tail) => 1 + Length(tail)
7 }
```

The desired property is

$$\text{Length}(\text{Cons}(v, h)) = 1 + \text{Length}(h).$$

The nonempty case counts one entry, then recurses on its tail. `decreases xs` requires progress to a smaller list.

Before using this definition in a proof, Dafny must check that the recursion is valid.

Why the recursive call must make progress

Keep the same `Length` definition, but accidentally recurse on the **whole list**:

```
1 case Cons(_, tail) => 1 + Length(xs) // should use tail
```

For a nonempty list, accepting this as an equation would give

$$\text{Length}(xs) = 1 + \text{Length}(xs),$$

which is impossible over integers.

Dafny rejects the recursive call: `xs` is not smaller than itself. The original call on `tail` makes progress toward `Nil`.

Termination matters even for a ghost function: its definition must be sound before the verifier can use it as a mathematical fact.

Demo 4: a job counter with a ghost history

What Dafny adds to our course toolbox

We already had contracts, Hoare rules, loop invariants, and SMT validity checking.

Now we can connect those ideas to programs with a heap:

- frames describe effects and dependencies
- ghost state expresses abstract models and their relation to concrete data
- mathematical values and functions describe abstract behavior
- well-founded definitions and suitable encodings support automated checking

Assignment 2 uses a smaller part of this model

The replication assignment represents states as immutable values.

Its main work is choosing the right contracts and prefix invariants, then using modular verification to compose them.

It does not require implementing **Repr** sets or a mutable object representation.

The Assignment 2 README includes the Dafny syntax reference and setup instructions.