The Use and Anatomy of an Auto-Active Verifier

Rustan Leino

Lecture 4 - *July 3, 2026*

1 Desugaring Part 2

The `calc` keyword is used to establish relationships between expressions, especially when those relationships (such as equality or comparison) are established by invoking lemmas or unfolding definitions. One should think of `S0` in the following code block as a "hint" for proving `E0 == E1`.

Source Code : calc

```
calc {
E0
== { S0 }
E1
< { S1 }
E2
}
```

We could model this program in the IVL as follows:

IVL Code

```
S0
assert E0 == E1
S1
assert E1 < E2
assume E0 < E2
```

Here, the assertions state that the relationships between `E0` and `E1` and `E1` and `E2` hold, and by

Compiled By:
Namratha Gangamreddypalli, Jasper Geer, Max Gross, Abigail Pribisova,
Hans Sardar, and Konami Shu

transitivity, one can assume that `E0 < E2` in the rest of the program.

However, `calc` has an additional property that information learned from `S0` is forgotten in the following calculations. Therefore, one can really execute the above program using demonic choice and `assume false`, as follows:

IVL Code

```
S0
assert E0 == E1
assume false
[]
S1
assert E1 < E2
assume false
[]
assume E0 < E2
```

We `assume false` to forget information from prior branches. This decision allows us to represent each comparison judgment separately using demonic choice. The transitivity of the binary comparison operators `<` and `==` gives us the equality between `E0` and `E2`.

This is a simplification of how `calc` is actually represented. Additionally, both `E0, E1`, and `E2` need to be well-formed, giving the following IVL code:

IVL Code

```
S0; assert wf[E0]; assert wf[E1]
assert E0 == E1
assume false
[]
S1; assert wf[E2]
assert E1 < E2
assume false
[]
assume E0 < E2
```

This is a design choice - one could also encode the IVL to execute `S0, ..., SN` upfront and carry all of the information while establishing the comparison. One could also assume facts that were proved before and "forgotten" because of `assume false` in the IVL, like follows:

IVL Code

```
1 S0; assert wf[E0]; assert wf[E1]
2 assert E0 == E1
3 assume false
4 []
5 S1; assume wf[E1]; assert wf[E2]
6 assert E1 < E2
7 assume false
8 []
9 assume wf[E0] /\ wf[E2] /\ E0 < E2
```

However, to maintain soundness of Dafny, one would need a meta-proof to show that introducing these assumptions is legitimate.

2 Logic Engine

The logic engine of Dafny is a satisfiability modulo theories (SMT) solver. SMT solvers have been optimized to perform very well for these tasks. Since Dafny uses many different quantifiers, z3 is the most suitable SMT solver. However, there is nothing in the encoding that would only work in z3's logic. There are a few small verified SMT solvers, but verifying an SMT solver like z3 would involve formalizing and proving a lot of linear arithmetic. A benefit of other SMT solvers, like cvc5, is that they can sometimes provide a proof term that can be validated. However, it might also fail to provide a proof term for a proposition that holds. Most SMT solvers can provide a counterexample for failing propositions.

SMT solvers also differ in how they instantiate quantifiers. The SMT solver z3 uses match-based instantiation. Proof terms are generated, kept in a broader context, and checked if they match universally quantified variables. Vampire, on the other hand, implements a unification algorithm for quantifier instantiation. Another, more recent, option is to use AI. AI has a talent for looking through levels of complexity that a human cannot, and if you query it to produce a proof, this can be independently checked in another logic engine or proof assistant.

3 Lessons in Language Design

3.1 Learning from trying to verify Java and C#

Consider the following Java method:

```
String concat(String s, String t) {
    return s + t
}
```

In general, we might like to use methods like `concat` in specifications. This must be done with care; if we would also like it to be the case that specifications are erasable, then we must restrict our specification language to a "pure" subset which cannot produce observable side-effects.

This method may appear to be pure; however, it still produces observable side effects. Because equality on `String`s in Java is reference equality, and a new `String` is allocated each time `concat` is called, the following assertion will fail:

```
assert concat(s1, s2) == concat(s1, s2);
```

Dafny deals with many issues related to this example and others as follows:

First, strings and other immutable types should not be references with side effect-producing behaviour. This is remedied by including non-reference datatypes in Dafny. For example, a `string` in Dafny is represented by a sequence of chars, which is an immutable datatype. For one to get a string that is a reference, one would use an array of chars. This delineation allows one to reason safely about strings in specifications.

Second, because it is difficult to retrofit a language with a notion of purity, there is a clear distinction between expressions (which are pure by construction) and statements (which can produce side-effects). As a result, Dafny's *functions*, the bodies of which are expressions, are guaranteed to be side-effect free.

Third, `.equals(..)` methods, which compute structural equality between objects, are a consequence of an overly object-oriented mindset. If one desires object identity or mutation, references are the solution; otherwise, datatypes are the answer.

Fourth, Dafny avoids the problem of leaking `this` from a constructor that occurs in Java and C#. This happens client code observes a half-built object before initialization finishes. However, this can lead to failing specifications even though the properly initialized value might have satisfied the specification. Dafny divides a constructor body into two phases, separated by a `new;` statement. Dafny only allows access to the object after every fields has been definitely assigned, marked by the `new;` statement.

Fifth, Dafny provides a built-in way to reason about unbounded integers, meaning that one does not need to constantly prove the absence of arithmetic overflows when writing specification and reasoning about integers.

3.2 Ghost Syntax

Dafny provides both functions, as described above, and `ghost` functions. Functions are compiled and can be called from executable code - additionally, they can be mentioned in specifications. `Ghost` functions are specification-only, and they are erased by the compiler. They can only be called from other ghost code or specifications. The reason to differentiate between the two is that `ghost` functions might quantify over infinite domains, which means that the function cannot realistically be executed.

3.3 Loops

Dafny strives to minimize the effort involved in generating loop invariants. Consider the following program initializing an array:

```
method Init<X>(a:array<int>, x:X)
  modifies a
{
  var i := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length
    invariant forall j :: 0 <= j < i ==> a[j] == x
  {
    a[i] := x;
    i := i + 1;
  }
}
```

Simply changing this method to do the same thing for a two-dimensional matrix more than doubles the number of invariants needed.

```
method Init<X>(arr: array2<X>, x: X)
  modifes arr
  ensures forall i, j :: 0 <= i < arr.Length0 &&
    0 <= j < arr.Length1 ==> arr[i, j] == x
{
var m := 0;
  while m < arr.Length0
    invariant 0 <= m <= arr.Length0
    invariant forall a, b :: 0 <= a < m && 0 <= b < arr.Length1 ==> arr[a, b] == x
  {
    var n 0;
    while n < arr.Length1
      invariant 0 <= n <= arr.Length1
      invariant forall a, b :: 0 <= a < m && 0 <= b < arr.Length1 ==> arr[a, b] == x
      invariant forall b :: 0 <= b < n ==> arr[m, b] == x
    {
```

```
      arr[m, n] := x;
      n := n + 1;
    }
  m := m + 1;
}
}
```

Therefore, Dafny adds aggregate statements, allowing one to specify attributes about an aggregate object in a singular statement. This simplifies the above method as follows:

```
method Init<X>(arr: array2<X>, x: X)
  modifies arr
  ensures forall i, j :: 0 <= i < arr.Length0 &&
    0 <= j < arr.Length1 ==> arr[i, j] == x
{
  forall i, j | 0 <= i < arr.Length0 && 0 <= j < arr.Length1
  {
    arr[i, j] := x;
  }
}
```

Another issue that one runs into is maintaining ghost state for mutable data structures. Consider a binary tree where each node stores a ghost field containing the set of its descendants:

```
class Node {
  var value: int
  var parent: Node
  var left: Node?
  var right: Node?

  ghost var Descendants: set<Node>
}
```

An iterative implementation for removing the minimum element is straightforward:

```
method Remove(root: Node) returns (r: Node)
{
  var p := root;
  while p.left != null {
    p := p.left;
  }
  p.parent.left := null;
  // update all Descendants
  return p;
}
```

However, updating the ghost state requires modifying the `Descendants` field of every ancestor of the removed node, making the loop cumbersome to specify.

Dafny instead encourages implementations whose structure mirrors the proof. A recursive implementation naturally updates the ghost state while the recursive calls return:

```
method RecursiveRemove(p: Node) returns (r: Node)
{
  if p.left == null {
    p.parent.left := null;
    return p;
  }

  r := RecursiveRemove(p.left);
  p.Descendants := p.Descendants - {r};
  return r;
}
```

For proofs and verification, this recursive implementation is more appropriate. However, an iterative implementation might be more efficient in practice. Therefore, Dafny offers tail-recursive compilation of programs, modulo ghost state and specification, providing the best of both worlds.

This motivates another language design principle, that some loops are difficult to specify so Dafny should Offer tail-recursive compilation, modulo ghosts.

Finally, in relation to termination of loops and recursive functions, Dafny makes the choice of providing a fixed well-founded order for all types, instead of allowing user-defined well-founded orders. However, the user can still define a mapping into this fixed order, allowing some flexibility with constructing domain-specific, intutitive orders.

3.4 Framing

Dafny also supports specifying and verifying mutable data structures in the heap. Rather than separation logic's separating conjunction, which frames implicitly, Dafny utilizes dynamic frames. Notably, the heap region a method may modify, or a function may read, is denoted explicitly by a value, usually a ghost set of objects, so reasoning about framing reduces to set membership and disjointness. These regions are referenced using `modifies` and `reads` clauses in specifications.