Program Verification with Danny (Part I)

Nuno Macedo

Dafny

- Programming language and tool for developing verified programs
- Developed by Microsoft Research
- Multi-paradigm language (imperative, functional, object-oriented)
- Two main components:
  - **Specifications** (verified statically)
  - **Implementations** (compiles into C#)

Dafny Specifications

- Specifications are declarative, either logic or functional programming
- Immutable instances (value types, in contrast to reference types)
- Constructs include
  - Preconditions (`requires`) and postconditions (`ensures`)
  - Loop variants (`decreases`) and invariants (`invariant`)
  - Assertions (`assert`)
  - Lemmas (`lemma`)
  - Functions (`function`) and predicates (`predicate`)
  - Ghost (`ghost`) variables and methods

Dafny Implementations

- Implementations are mostly imperative (but functional can be used)
- Composed of procedures (`method`)
- Strongly typed (often inferred)
- Some object-oriented features (classes, inheritance, …)
- Supports reference types (arrays, classes, …), mutable instances

Dafny Verification

- Methods are annotated with specifications
- Specifications continuously verified statically (compilation-time)
  - Automates part of the tableau proof method
  - Invalid specifications are errors: does not compile
- Uses solvers underneath, shows counter-examples to violations
- Not fully automatic (*undecidable*): may require additional annotations
- Compiles implementations into executable code
- Specifications are **not** part of the executable code

Dafny Examples

- Consider only value types
- No aliasing, assignment is copy
- No arrays, no class objects, no side-effects

Example: Integer Division

- Methods are executable code
- No annotations, but already errors flagged
- Dafny verifies total correctness
  - correctness + termination
- Cannot prove termination without restrictions on input

```
method div(n:nat, d:nat) returns (q:nat, r:nat)
{
    r := n;
    q := 0;
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
}
```

Return values part of state, may return multiple.
Variables declared with `var`, type (often) inferred.

Example: Integer Division

- With the `requires` precondition already proves termination
- Loop variant automatically inferred
- Can be made explicit with `decreases`
- Information from types (`nat`, ≥0) also used

```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
{
    decreases r - d Resolver
    No quick fixes available
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
}
```

Blue annotations are informative.
Code can be executed when there are no errors.

Example: Integer Division

- Can define basic tests as assertions
- These are not checked in runtime as in other languages
- Verified during compilation
- Invalid: methods are black-box for proofs
- Only the pre and postconditions are known

```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
{
    r := n;
    q := 0;
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

meth
{
    assert x == 3 && y == 3;
}
```

Ghost statement

assertion might not hold Verifier

View Problem   No quick fixes available

Counter-examples can be shown (not always helpful).

Example: Integer Division

- Postcondition added
- Assertion is now valid: consistent with postcondition
- But still not proved that code satisfied postcondition
- Loops require additional annotations

```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
ensures q*d + r == n && r < d
{
    r := n;
    q := 0;
    while (r >= d)
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
    assert x == 3 && y == 3;
}
```

Example: Integer Division

- Loop invariant: must hold before, in all iterations, and after
- Must imply the postcondition
- Creative step, cannot be automated by tools
- Variants often automatically detected, but not always

```
method div(n:nat, d:nat) returns (q:nat, r:nat)
requires d > 0
ensures q*d + r == n && r < d
{
    r := n;
    q := 0;
    while (r >= d)
    invariant q*d + r == n
    {
        q := q + 1;
        r := r - d;
    }
}

method Main()
{
    var x,y := div(15,4);
    assert x == 3 && y == 3;
}
```

Example: Integer Exponentiation

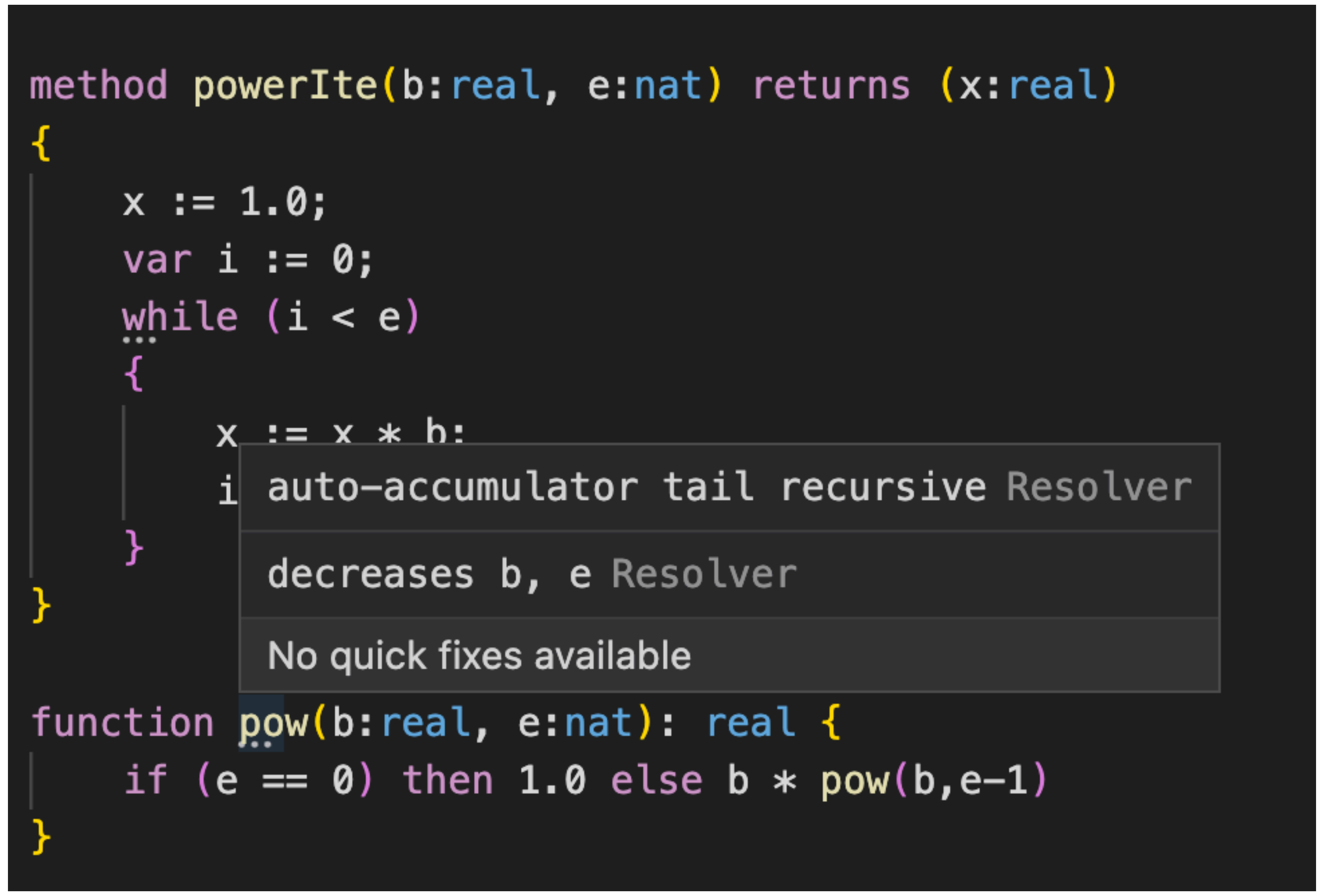
- Iterative, inefficient version (linear)
- How to specify the postcondition declaratively?
- Side-effect free functions

```
method powerIte(b:real, e:nat) returns (x:real)
{
    x := 1.0;
    var i := 0;
    while (i < e)
    {
        x := x * b;
        i := i + 1;
    }
}
```

Real constants require decimal part.

Example: Integer Exponentiation

- Functions: more abstract and declarative definition of behaviour
- Can be recursive (but no iteration)
- Pure, side-effect free, more prone to automatic analysis
    - Don’t need postconditions
- Performance doesn’t matter: will not compile
- Needs variant to prove termination, but often automatically inferred



Example: Integer Exponentiation

- Proved that the imperative implementation conforms to the functional specification

```
method powerIte(b:real, e:nat) returns (x:real)
ensures x == pow(b,e)
{
    x := 1.0;
    var i := 0;
    while (i < e)
    invariant x == pow(b,i) && 0 <= i <= e
    {
        x := x * b;
        i := i + 1;
    }
}

function pow(b:real, e:nat): real {
    if (e == 0) then 1.0 else b * pow(b,e-1)
}
```

Example: Integer Exponentiation

- What if we want a more efficient implementation?
- Recursive version, but could be iterative
- Logarithmic, based on $a^n \times a^m = a^{n+m}$
- Solvers don't know all theorems!
- Requires user input
- Not trivial to identify what lemma is needed (solving process is black box)

```
method powerRec(b:real, e:nat) returns (x:real)
ensures x == pow(b,e)
{
    if (e == 0) {
        return 1.0;
    }
    else if (e % 2 == 0) {
        var r := powerRec(b,e/2);
        return r * r;
    } else {
        var r := powerRec(b,(e-1)/2);
        return r * r * b;
    }
}
```

Example: Integer Exponentiation

- Sometimes hints can be provided as assertions
- Force the solver to calculate intermediary results
- In more complex cases, we need to define auxiliary lemmas
- The lemmas themselves must be proved correct: not always trivial!
- Here, instructed to prove with induction

```
lemma {:induction e1} powDist(b:real, e1:nat, e2: nat)
ensures pow(b,e1+e2) == pow(b,e1)*pow(b,e2)
{}

method powerRec(b:real, e:nat) returns (x:real)
ensures x == pow(b,e)
{
    if (e == 0) {
        return 1.0;
    }
    else if (e % 2 == 0) {
        powDist(b,e/2,e/2);
        var r := powerRec(b,e/2);
        return r * r;
    } else {
        powDist(b,(e-1)/2,(e-1)/2);
        var r := powerRec(b,(e-1)/2);
        return r * r * b;
    }
}
```