Using Dafny to write correct programs

K. Rustan M. Leino

Amazon Web Services

21 October 2024
TU Vienna
Vienna, Austria

Formal verification

- Sometimes, testing isn't enough
- Sometimes, there's no reason to have any bugs

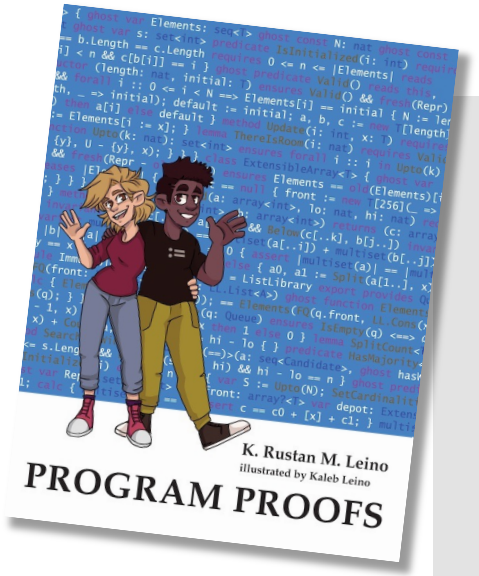
- Formal verification checks *every* behavior for *every* input and *every* environment
- Learning to reason formally is a great guide also for day-to-day *informal* reasoning

Language

- Dafny (`dafny.org`)
- Programming language
  - Java-like
  - Verification-aware (proof-oriented)
- Constructs for
  - Imperative programming
  - Functional programming
  - Specifications
  - Proofs
- VS Code integration

Some uses of Dafny

- Teaching
  - Over a decade, many universities
  - Book (MIT Press): `program-proofs.com`
- At AWS
  - AWS Encryption SDK
  - AWS Database Encryption SDK
  - AWS's authentication engine
    Talk at AWS re:Inforce 2024:
    https://youtu.be/oshxAJGrwMU?si=2HRf2XvNR-8RMIXZ
  - …
- …



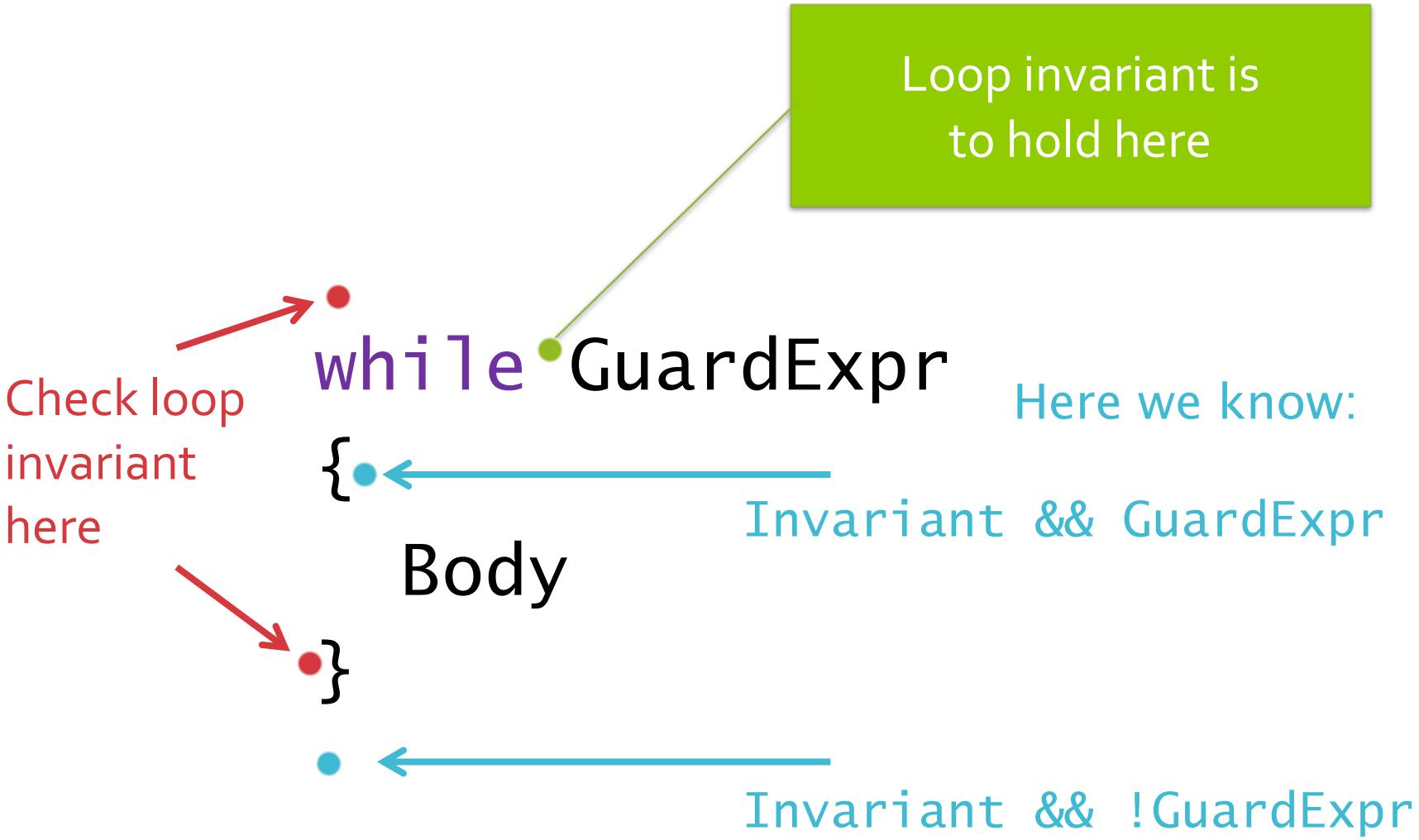
Methods and pre-/post-conditions

```
method Split(c: int) returns (a: int, b: int)
  requires 0 <= c < 10_000
  ensures 0 <= a < 100 && 0 <= b < 100
  ensures c == 100 * a + b
{
  a := c / 100;
  b := c % 100;
}

method Combine(a: int, b: int) returns (c: int)
  requires 0 <= a < 100 && 0 <= b < 100
  ensures 0 <= c < 10_000
  ensures c == 100 * a + b
{
  if a == 0 {
    c := b;
  } else if a == 1 {
    c := 100 + b;
  } else {
    c := 100 * a + b;
  }
}

method Caller(c: int)
  requires 0 <= c
{
  if c < 10_000 {
    var a, b := Split(c);
    var d := Combine(a, b);
    assert c == d;
  }
}
```

Reasoning about loops



A simple loop

```
method Square(n: nat) returns (s: nat)
  ensures s == n * n
{
  s := 0;
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant s == i * i
  {
    ?
    i := i + 1;
  }
}
```

Figuring out what's needed

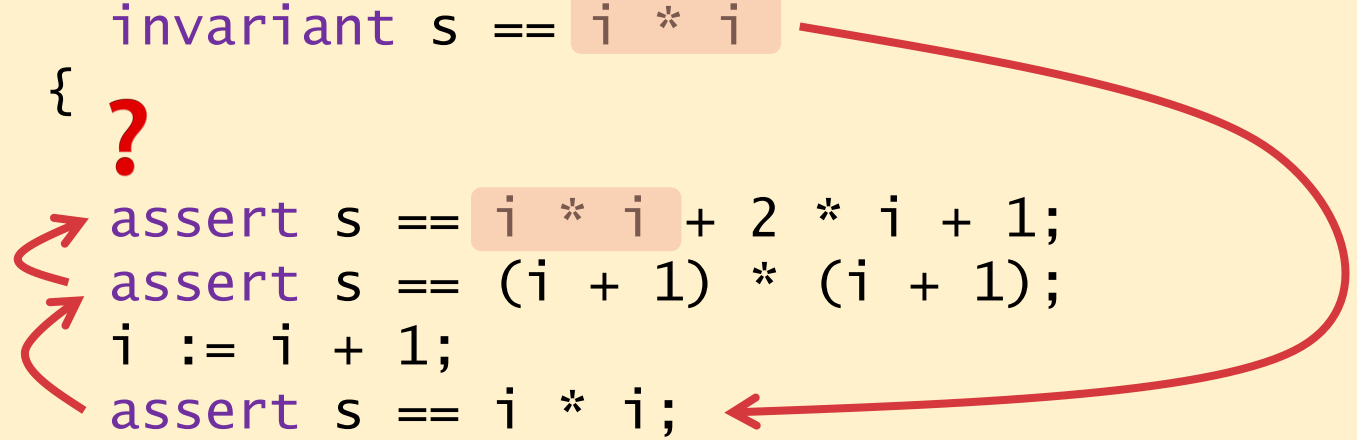
```
assert Condition[x := E];
x := E;
assert Condition;
```

?

the `Condition`, but with `x` replaced by `E`

A simple loop

```
method Square(n: nat) returns (s: nat)
  ensures s == n * n
{
  s := 0;
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant s == i * i
  {
    ?
    assert s == i * i + 2 * i + 1;
    assert s == (i + 1) * (i + 1);
    i := i + 1;
    assert s == i * i;
  }
}
```



A simple loop

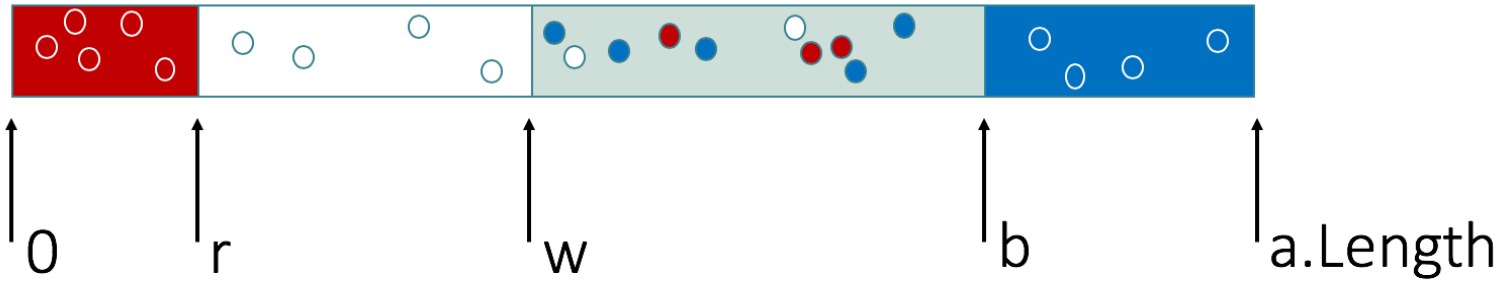
```
method Square(n: nat) returns (s: nat)
  ensures s == n * n
{
  s := 0;
  var i := 0;
  while i < n
    invariant 0 <= i <= n
    invariant s == i * i
  {
    s := s + 2 * i + 1;
    i := i + 1;
  }
}
```

Linear Fibonacci

```
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)
{
  x := 0;
  var y, i := 1, 0;
  while i < n
    invariant 0 <= i <= n
    invariant x == Fib(i) && y == Fib(i + 1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
}
```

Dutch National Flag



Dutch National Flag

```
datatype Color = Red | White | Blue

function InOrder(c: Color, d: Color): bool {
  c == Red || c == d || d == Blue
}

method DutchFlag(a: array<Color>)
  modifies a
  ensures forall i, j :: 0 <= i < j < a.Length ==> InOrder(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b := 0, 0, a.Length;
  while w != b
    invariant 0 <= r <= w <= b <= a.Length
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant forall i :: r <= i < w ==> a[i] == White
    invariant forall i :: b <= i < a.Length ==> a[i] == Blue
    invariant multiset(a[..]) == old(multiset(a[..]))
  {
    if a[w] == Red {
      a[r], a[w] := a[w], a[r];
      r, w := r + 1, w + 1;
    } else if a[w] == White {
      w := w + 1;
    } else {
      b := b - 1;
      a[w], a[b] := a[b], a[w];
    }
  }
}
```

Conclusion

Use Dafny to

- Write verified programs
- Write a verified part of a larger program written in some other language
- Formalize your models
- Learn, and teach!

Program safely!