The Dafny Programming Language and Static Verifier

Stefan Zetzsche

Amazon Web Services

June 29, 2023

1 Introduction

What is Dafny?

Live Demo

What is Dafny?

```
0  function fib(n: nat): nat {
1    if n ≤ 1 then
2      n
3    else
4      fib(n - 1) + fib(n - 2)
5  }
6
7  method ComputeFib(n: nat) returns (b: nat)
8    ensures b = fib(n)
9  {
10   var c := 1;
11   b := 0;
12   for i := 0 to n
13     invariant b = fib(i) ∧ c = fib(i + 1)
14   {
15     b, c := c, b + c;
16   }
17 }
```

Dafny and Rustan Leino

Dafny at Amazon: Authorization


```
0  permit(principal, action, resource)
1  when {
2    resource has owner ∧ resource.owner = principal
3  };
```

---

https://github.com/cedar-policy

https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing

Dafny at Amazon: Crypto Tools

Cryptography is hard to do safely and correctly.

---

https://docs.aws.amazon.com/aws-crypto-tools/index.html

https://github.com/aws/aws-encryption-sdk-dafny

https://aws.amazon.com/blogs/security/aws-security-profile-cryptography-edition-valerie-lambert-senior-software-development-engine

Table of Contents

2 Dafny as a Programming Language

Dafny as a Programming Language

Dafny is a mature language that allows you to:

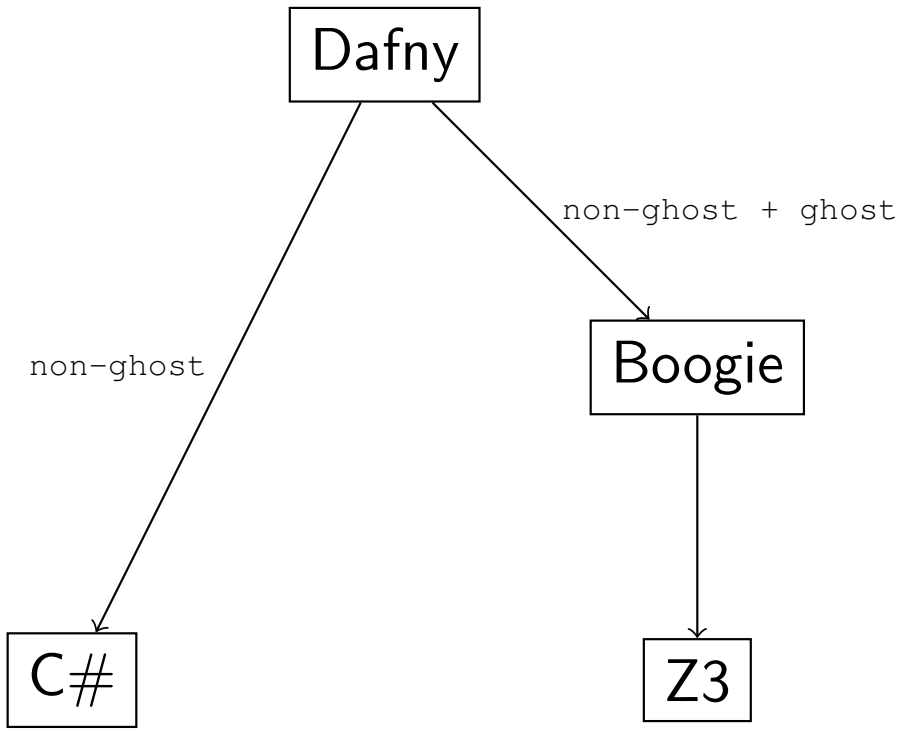
- write functional/imperative/OO programs
- compile programs
- execute programs
- interoperate with other languages

Multi-Paradigms

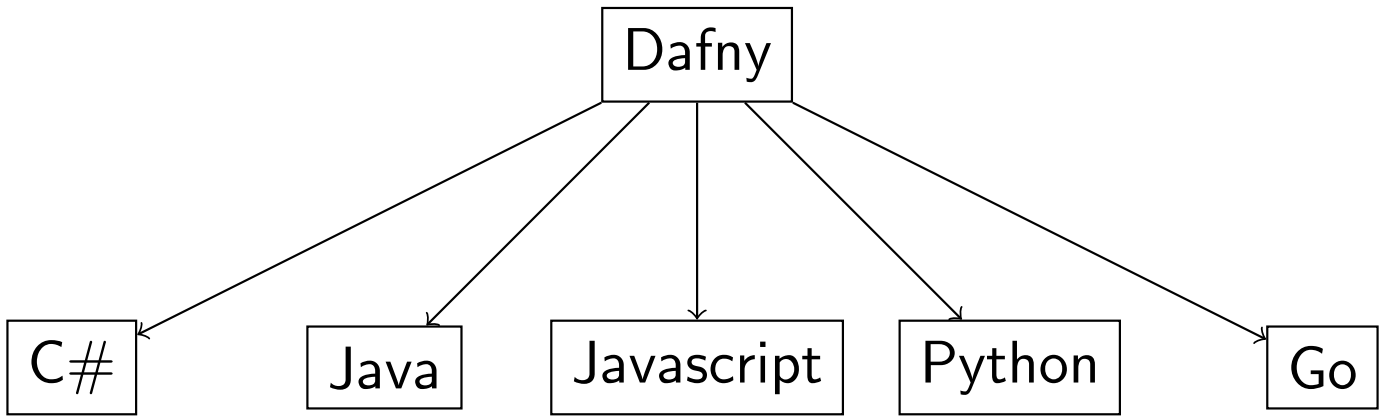
Dafny supports multi-paradigm concepts:

- inductive datatypes
- while-loops
- lambda expressions
- higher-order functions
- classes with mutable state
- polymorphism

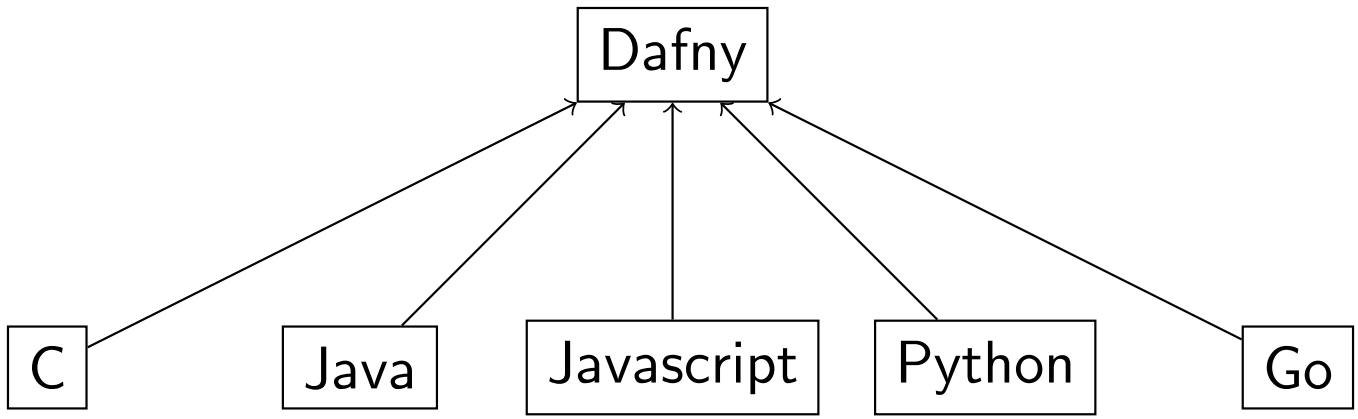
Pipeline



Compilation



Interoperate with {:extern}



2.1 Functional Programming

Functions, Constants, Predicates

```
0  function FunctionName(param1: Type1, param2: Type2): Type3 {
1    expression
2  }
3
4  const constantName: Type := expression;
5
6  predicate predicateName(param1: Type1, param2: Type2) {
7    booleanExpression
8  }
```

Functions as In/Out Parameters

```
0  function Apply(f: int -> int, n: int): int {
1    f(n)
2  }
3
4  function ApplyPartial(f: int -> int -> int, n: int): int -> int {
5    f(n)
6  }
```

Recursive Functions

```
0  function Factorial(n: nat): nat {
1    if n == 0 then 1 else n * Factorial(n-1)
2  }
```

Inductive Datatypes

```
0  datatype list = Nil | Cons(head: bool, tail: list)
1
2  function Conjunction(xs: list): bool {
3    match xs
4      case Nil ⇒ true
5      case Cons(head, tail) ⇒ head ∧ Conjunction(tail)
6  }
```

Polymorphism

```
0  datatype list<T> = Nil | Cons(head: T, tail: list)
1
2  function Length<T>(xs: list<T>): nat {
3    match xs
4      case Nil ⇒ 0
5      case Cons(_, tail) ⇒ 1 + Length(tail)
6  }
```

Immutable Collection Types

- Sequences

```
seq(length, i ⇒ f(i))
```

- Sets

```
set x: T | p(x) • f(x)
```

- Maps

```
map x: T | p(x) • f(x)
```

- Multisets

2.2 Imperative Programming

Methods

```
0  method MethodName<T>(arg1: T, arg2: string) {
1    print(arg1);
2    print(arg2);
3  }
```

Methods

```
0  method Call() returns (o: int) {
1    MethodName("Hello,", "World\n");
2    o := FunctionName(42);
3  }
```

Conditional

```
0  method IfElse() {
1    if booleanExpression {
2      // ...
3    } else {
4      // ...
5    }
6  }
```

While Loop

```
0  method WhileLoop() {
1    while booleanExpression {
2      // ...
3    }
4  }
```

For Loop

```
0  method ForLoop() {
1    for variable := startExpression to stopExpression {
2      // ...
3    }
4  }
```

Arrays

```
0  method Aliasing() {
1    var A := new int[100];
2    var B := A;
3  }
```

Reading Arrays

```
0  function Read(A: array<bool>): bool
1    reads A
2  {
3    if A.Length = 0 then
4      false
5    else
6      A[0]
7  }
```

Modifying Arrays

```
0  method Modify(A: array<bool>, b: bool)
1    modifies A
2  {
3    if A.Length = 0 {
4    } else {
5      A[0] := b;
6    }
7  }
```

2.3 Object-Oriented Programming

Classes

```
0  class C {
1    var mutableField: int;
2    const immutableField: int
3
4    constructor(i: int, j: int) {
5      immutableField := i;
6      mutableField := j;
7    }
8  }
9
10 method M() {
11   var o := new C(0, 1);
12 }
```

Functions and Methods

```
0  class C {
1    var mutableField: int;
2
3    function Get(): int
4      reads this
5    {
6      mutableField
7    }
8
9    method Set(i: int)
10     modifies this
11   {
12     mutableField := i;
13   }
14 }
```

Inheritance

```
0  trait T {
1    method Print()
2  }
3
4  class C extends T {
5    method Print() {
6      print("Stefan");
7    }
8  }
9
10 class D extends T {
11   method Print() {
12     print("Zetzsche");
13   }
14 }
```

3 Dafny as a Proof Assistant

3.1 Formal Mathematics

Type Symbols

```
0  type NaturalNumber(00, =)
```

Constant Symbols

```
0  ghost const Zero: NaturalNumber
```

Function Symbols

```
0  ghost function Successor(n: NaturalNumber): NaturalNumber
```

Predicate Symbols

```
0  ghost predicate Equal(n: NaturalNumber, m: NaturalNumber)
```

Axioms

```
0  lemma {:axiom} Reflexive()
1   ensures ∀ n: NaturalNumber • Equal(n, n)
```

Axioms and Quantification

```
0  lemma {:axiom} Reflexive(n: NaturalNumber)
1    ensures Equal(n, n)
```

Lemma

```
0  lemma {:axiom} Reflexive(n: NaturalNumber)
1    ensures Equal(n, n)
2
3  lemma AboutZero()
4    ensures ∃ n: NaturalNumber • Equal(n, Zero)
5  {
6    // ...
7  }
```

Lemma

```
0  lemma {:axiom} Reflexive(n: NaturalNumber)
1    ensures Equal(n, n)
2
3  lemma AboutZero()
4    ensures ∃ n: NaturalNumber • Equal(n, Zero)
5  {
6    Reflexive(Zero);
7  }
```

Second Order and Excluded Middle

```
0  lemma SecondOrder()
1    ensures ∀ p: int -> bool • ∀ x: int • p(x) ∨ ¬p(x)
2  {}
```

Higher Order

```
0  lemma ThirdOrder()
1    ensures ∀ P: (int -> bool) -> bool, p: int -> bool • P(p) ∨ ¬P(p)
2  {}
```

3.2 Structured Proofs

Proof Structure

```
0  lemma ProofStructure()
1    requires Assumptions
2    ensures Goal
3  {
4    assert Goal by {
5      Assumptions
6    }
7  }
```

Conjunction

```
0  lemma ProofOfConjunction() {
1    assert A ∧ B by {
2      assume A;
3      assume B;
4    }
5  }
```

Disjunction

```
0   lemma ProofOfDisjunction1() {
1     assert A ∨ B by {
2       assume A;
3     }
4   }
5
6   lemma ProofOfDisjunction2() {
7     assert A ∨ B by {
8       assume B;
9     }
10  }
```

Implication

```
0  lemma ProofOfImplication() {
1    assert A ⟹ B by {
2      if A {
3        assume B;
4      }
5    }
6  }
```

Equivalence

```
0  lemma ProofOfEquivalence() {
1    assert A ⟺ B by {
2      assume A ⟹ B;
3      assume B ⟹ A;
4    }
5  }
```

Contradiction

```
0  lemma ProofByContradiction() {
1    assert B by {
2      assume ¬B ⟹ false;
3    }
4  }
```

Product

```
0  lemma ProofOfProduct() {
1    assert A ⟹ (B ∧ C) by {
2      assume A ⟹ B;
3      assume A ⟹ C;
4    }
5  }
```

Coproduct

```
0  lemma ProofOfCoproduct() {
1    assert (A ∨ B) ⟹ C by {
2      assume A ⟹ C;
3      assume B ⟹ C;
4    }
5  }
```

Calculations 1

```
0  lemma UnitIsUnique<T(!new)>(bop: (T, T) -> T, unit1: T, unit2: T)
1    requires ∀ x • bop(x, unit2) = x
2    requires ∀ x • bop(unit1, x) = x
3    ensures unit1 = unit2
4  {
5    calc {
6      unit1;
7    =
8      bop(unit1, unit2);
9    =
10     unit2;
11   }
12 }
```

Calculations 2

```
0  lemma UnitIsUnique<T(!new)>(bop: (T, T) -> T, unit1: T, unit2: T)
1    requires A1: ∀ x • bop(x, unit2) = x
2    requires A2: ∀ x • bop(unit1, x) = x
3    ensures unit1 = unit2
4  {
5    calc {
6      unit1;
7    = { reveal A1; }
8      bop(unit1, unit2);
9    = { reveal A2; }
10     unit2;
11   }
12 }
```

Calculations 3

```
0  lemma UnitIsUnique<T(!new)>(bop: (T, T) -> T, unit1: T, unit2: T)
1    requires A1: ∀ x • bop(x, unit2) = x
2    requires A2: ∀ x • bop(unit1, x) = x
3    ensures unit1 = unit2
4  {
5    assert unit1 = bop(unit1, unit2) by {
6      reveal A1;
7    }
8    assert bop(unit1, unit2) = unit2 by {
9      reveal A2;
10   }
11 }
```

Non-Deterministic Choice

```
0  lemma ProofUsingExistential<T>(p: T -> bool, q: T -> bool)
1    requires ∀ x • p(x) ⟹ q(x)
2    ensures ∃ x • q(x)
3  {
4    assume ∃ x • p(x);
5    var c :| p(c);
6  }
```

4 Dafny for the Verification of Programs

4.1 Independent Verification of Functional Programs

Conditional

```
0  function Abs(x: int): int {
1    if x < 0 then
2      -x
3    else
4      x
5  }
6
7  lemma AbsPositive(x: int)
8    ensures Abs(x) ≥ 0
9  {
10   if x < 0 {
11     assert -x > 0;
12   } else {
13     assert x ≥ 0;
14   }
15 }
```

Recursion and Induction

```
0  datatype list<T> = Nil | Cons(head: T, tail: list)
1
2  function Length<T>(xs: list): nat {
3    match xs
4      case Nil ⇒ 0
5      case Cons(head, tail) ⇒ 1 + Length(tail)
6  }
7
8  function Append<T>(xs: list, ys: list): list {
9    match xs
10     case Nil ⇒ ys
11     case Cons(head, tail) ⇒ Cons(head, Append(tail, ys))
12 }
13
14 lemma AppendLength<T>(xs: list, ys: list)
15   ensures Length(Append(xs, ys)) = Length(xs) + Length(ys)
16 {
17   match xs
18     case Nil ⇒
19     case Cons(head, tail) ⇒ AppendLength(tail, ys);
20 }
```

4.2 Dependent Verification of Functional Programs

Pre- and Postconditions 1

```
0  lemma AppendLength<T>(xs: list, ys: list)
1    ensures Length(Append(xs, ys)) = Length(xs) + Length(ys)
2  {
3    match xs
4      case Nil ⇒
5      case Cons(head, tail) ⇒ AppendLength(tail, ys);
6  }
```

Pre- and Postconditions 2

```
0  function Append<T>(xs: list, ys: list): list
1    ensures Length(Append(xs, ys)) = Length(xs) + Length(ys)
2  {
3    match xs
4      case Nil ⇒ ys
5      case Cons(head, tail) ⇒ Cons(head, Append(tail, ys))
6  }
```

Pre- and Postconditions 3

```
0  function Append<T>(xs: list, ys: list): list
1    requires Assumption
2    ensures Length(Append(xs, ys)) = Length(xs) + Length(ys)
3    ensures Property
4  {
5    assert Property by {
6      ProofOfPropertyViaAssumption();
7    }
8    match xs
9      case Nil ⇒ ys
10     case Cons(head, tail) ⇒ Cons(head, Append(tail, ys))
11 }
```

Pre- and Postconditions 4

```
0  function Append<T>(xs: list, ys: list): list
1    ensures Length(Append(xs, ys)) = Length(xs) + Length(ys)
2    // ∧ ∀ zs • Append(Append(xs, ys), zs) = Append(xs, Append(ys, zs))
3  {
4    match xs
5      case Nil ⇒ ys
6      case Cons(head, tail) ⇒ Cons(head, Append(tail, ys))
7  }
```

Termination 1

```
0  function Sum(n: int): int {
1    if n ≤ 0 then 0 else Sum(n-1)
2  }
```

Termination 2

```
0  function Sum(n: int): int
1    decreases n
2  {
3    if n ≤ 0 then 0 else Sum(n-1)
4  }
```

Termination 3

```
0  function SumUpTo(c: int := 0, n: int): int
1    decreases n - c
2  {
3    if c ≥ n then
4      n
5    else
6      c + SumUpTo(c + 1, n)
7  }
```

4.3 Verification of Imperative Programs

Hoare Logic

$$[P]S[Q] \qquad \text{iff} \qquad \text{wp}(S, Q) \Rightarrow P$$

```
0  method S()
1    requires P()
2    ensures Q()
```

Skip

$$\frac{}{[P]\ \mathsf{skip}\ [P]}$$

```
0  method Skip()
1    requires P()
2    ensures P()
3  {}
```

Assignment

$$\frac{}{[P[E/x]]\ x := E\ [P]}$$

```
0  method Assignment<T>(x: T) returns (y: T)
1    requires P(E(x))
2    ensures P(y)
3  {
4    y := E(x);
5  }
```

Assignment

$$\overline{[P[E/x]]\ x := E\ [P]}$$

```
0  function E(x: nat): nat {
1    x + 1
2  }
3
4  predicate P(x: nat) {
5    x % 2 = 0
6  }
7
8  method Assignment(x: nat) returns (y: nat)
9    requires P(E(x))
10   ensures P(y)
11 {
12   y := E(x);
13 }
```

Composition

$$\frac{[P]S[Q] \quad , \quad [Q]T[R]}{[P]\ S; T\ [R]}$$

```
0  method S()
1    requires P()
2    ensures Q()
3
4  method T()
5    requires Q()
6    ensures R()
7
8  method Composition()
9    requires P()
10   ensures R()
11 {
12   S();
13   T();
14 }
```

Consequence

$$\frac{P_1 \rightarrow P_2 \quad , \quad [P_2]\ S\ [Q_2] \quad , \quad Q_2 \rightarrow Q_1}{[P_1]\ S\ [Q_1]}$$

```
0   lemma Implications()
1     ensures P1() ⟹ P2()
2     ensures Q2() ⟹ Q1()
3
4   method S()
5     requires P2()
6     ensures Q2()
7
8   method Consequence()
9     requires P1()
10    ensures Q1()
11  {
12    Implications();
13    S();
14  }
```

Loops (Partial)

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S\ \{\neg B \wedge P\}}$$

```
0   method S()
1     requires P() ∧ B()
2     ensures P()
3
4   method WhileLoop()
5     requires P()
6     ensures ¬B() ∧ P()
7   {
8     while B()
9       invariant P()
10    {
11      S();
12    }
13  }
```

Loops (Partial)

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \text{while } B \text{ do } S\ \{\neg B \wedge P\}}$$

```
0  method Times(n: nat, a: nat) returns (b: nat)
1    ensures b = n * a
2  {
3    b := 0 ;
4    var i := 0;
5    while i < n
6      invariant b == i * a && i <= n
7    {
8      b := b + a;
9      i := i + 1;
10   }
11 }
```

5 Dafny at PPLV

5.1 Use Case Example

Big Step Semantics

Syntax

$$c \in \mathsf{cmd} ::= \mathsf{Inc} \mid c_0; c_1 \mid c^*$$

Semantics

$$\frac{t = s + 1}{s \xrightarrow{\mathsf{Inc}} t} \qquad \frac{s \xrightarrow{c_0} s' \quad , \quad s' \xrightarrow{c_1} t}{s \xrightarrow{c_0;c_1} t} \qquad \frac{t = s}{s \xrightarrow{c^*} t} \qquad \frac{s \xrightarrow{c} s' \quad , \quad s' \xrightarrow{c^*} t}{s \xrightarrow{c^*} t}$$

$\rightarrow \subseteq \mathsf{state} \times \mathsf{cmd} \times \mathsf{state}, \qquad \mathsf{state} = \mathsf{int}$

Big Step Semantics in Dafny

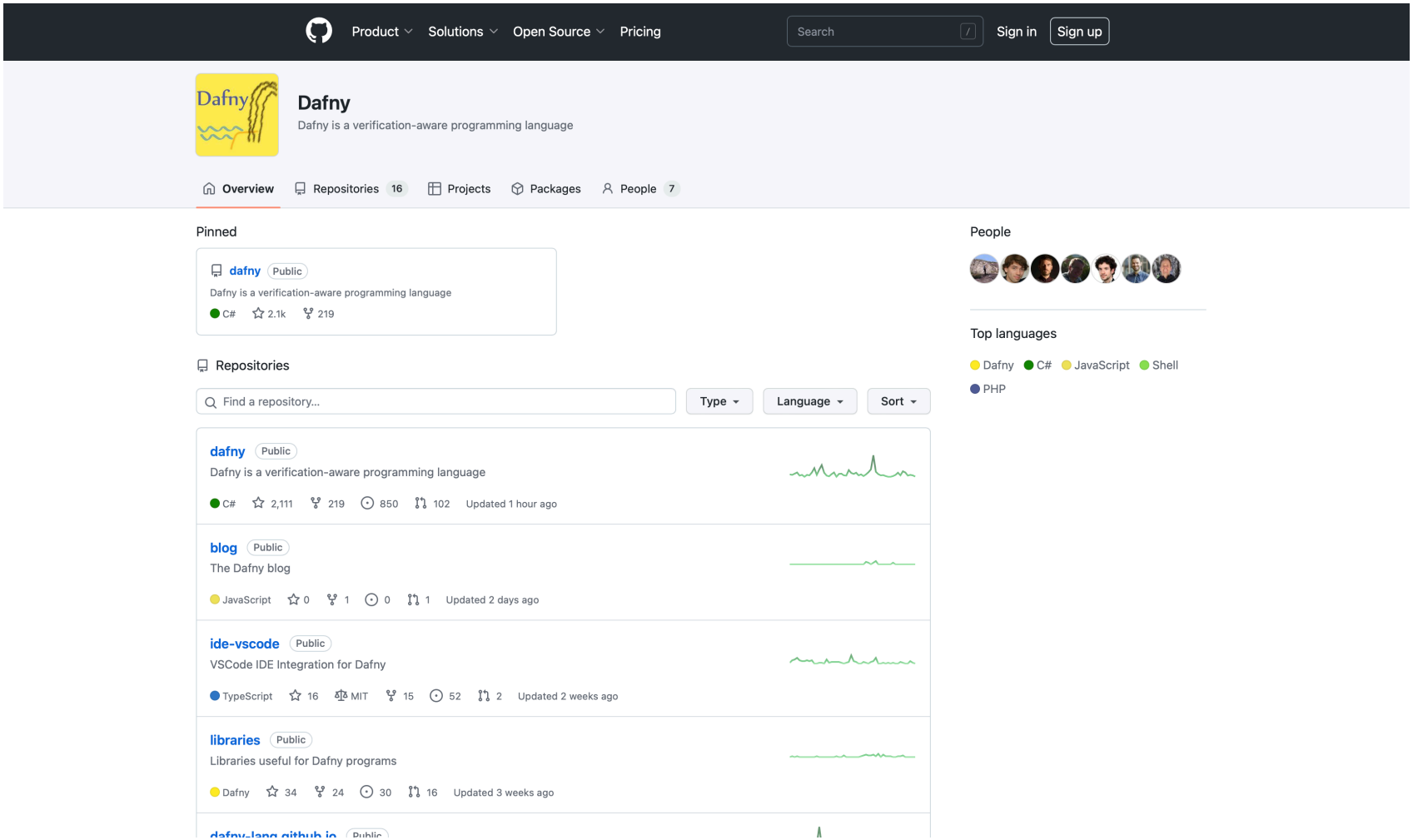
Live Demo

Big Step Semantics in Dafny

```
0  datatype cmd = Inc | Seq(cmd, cmd) | Repeat(cmd)
1
2  type state = int
3
4  least predicate BigStep(s: state, c: cmd, t: state) {
5    match c
6      case Inc ⇒
7        t = s + 1
8      case Seq(c0, c1) ⇒
9        ∃ s' • BigStep(s, c0, s') ∧ BigStep(s', c1, t)
10     case Repeat(c0) ⇒
11       (t = s) ∨
12       (∃ s' • BigStep(s, c0, s') ∧ BigStep(s', Repeat(c0), t))
13 }
14
15 least lemma Increasing(s: state, c: cmd, t: state)
16   requires BigStep(s, c, t)
17   ensures s ≤ t
18 {}
```

5.2 Opportunities

Open Source



`https://github.com/dafny-lang/dafny`

Formal Reasoning at AWS

Formal Reasoning About the Security of Amazon Web Services

Byron Cook[1,2(✉)]

[1] Amazon Web Services, Seattle, USA
byron@amazon.com
[2] University College London, London, UK

Abstract. We report on the development and use of formal verification tools within Amazon Web Services (AWS) to increase the security assurance of its cloud infrastructure and to help customers secure themselves. We also discuss some remaining challenges that could inspire future research in the community.

1 Introduction

Amazon Web Services (AWS) is a provider of *cloud services*, meaning on-demand access to IT resources via the Internet. AWS adoption is widespread, with over a million active customers in 190 countries, and $5.1 billion in revenue during the last quarter of 2017. Adoption is also rapidly growing, with revenue regularly increasing between 40–45% year-over-year.

The challenge for AWS in the coming years will be to accelerate the development of its functionality while simultaneously increasing the level of security offered to customers. In 2011, AWS released over 80 significant services and features. In 2012, the number was nearly 160; in 2013, 280; in 2014, 516; in 2015, 722; in 2016, 1,017. Last year the number was 1,430. At the same time, AWS is increasingly being used for a broad range of security-critical computational workloads.

Formal automated reasoning is one of the investments that AWS is making in order to facilitate continued simultaneous growth in both functionality and security. The goal of this paper is to convey information to the formal verification research community about this industrial application of the community's results. Toward that goal we describe work within AWS that uses formal verification to raise the level of security assurance of its products. We also discuss the use of formal reasoning tools by externally-facing products that help customers secure themselves. We close with a discussion about areas where we see that future research could contribute further impact.

Related Work. In this work we discuss efforts to make formal verification applicable to use-cases related to cloud security at AWS. For information on previous work within AWS to show functional correctness of some key distributed algorithms, see [43]. Other providers of cloud services also use formal verification to establish security properties, *e.g.* [23,34].

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 38–47, 2018.
https://doi.org/10.1007/978-3-319-96145-3_3

https://link.springer.com/chapter/10.1007/978-3-319-96145-3_3

Amazon Research Awards

https://www.amazon.science/research-awards/program-updates/79-amazon-research-awards-recipients-announced