

The Dafny Programming Language and Static Verifier

Stefan Zetsche

Amazon Web Services

October 13, 2023

1 Introduction

What is Dafny?

Live Demo

What is Dafny?

```
function Fib(n: nat): nat {  
  if n <= 1 then n else Fib(n - 1) + Fib(n - 2)  
}
```

```
method ComputeFib(n: nat) returns (b: nat)  
  ensures b == Fib(n)  
{  
  var c := 1;  
  b := 0;  
  for i := 0 to n  
    invariant b == Fib(i) && c == Fib(i + 1)  
    {  
      b, c := c, b + c;  
    }  
}
```

Dafny and Rustan Leino



Dafny Use Case: Cedar



```
permit(principal, action, resource)
when {
  resource has owner && resource.owner == principal
};
```

<https://github.com/cedar-policy>

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

Dafny Use Case: 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-engineer/](https://aws.amazon.com/blogs/security/aws-security-profile-cryptography-edition-valerie-lambert-senior-software-development-engineer/)

Dafny Use Case: VMC

A library for verified Monte Carlo algorithms

```
lemma Proposition(n: nat, i: nat)
  requires 0 <= i < n
  ensures
    var e := iset s: RNG | UniformModel(n)(s).0 == i;
    && e in event_space
    && mu(e) == 1.0 / (n as real)
```


POPL 2024

Wed 17 - Fri 19 January 2024 London, United Kingdom

Attending ▾ Tracks ▾ Organization ▾ 🔍 Search Series ▾

[🏠 POPL 2024 \(series\)](#) / [Dafny 2024 \(series\)](#) /

Dafny 2024

[About](#) [Call for Papers](#)

Call for Papers

We don't intend to publish the workshop's submissions. However, presentations may be recorded and the videos may be made publicly available.

Important Dates

- Submission: Wednesday, October 11, 2023 (AoE)
- Notification: Wednesday, November 15, 2023
- Workshop: Sunday, January 14, 2024

Submission Guidelines

To give a presentation at the workshop, please submit an anonymous extended abstract (2-6 pages, excluding references) via hotcrp:

<https://dafny24.hotcrp.com>

Please use the acmart two-column sigplan sub-format LaTeX style to prepare your submission:

<https://www.sigplan.org/Resources/Author/>

Contact

All questions about submission should be emailed to the program chairs Stefan Zetsche (stefanze@amazon.com) and Joseph Tassarotti (jt4767@nyu.edu).

Table of Contents

Introduction

Dafny as a Programming Language

Dafny as a Proof Assistant

Dafny for the Verification of Programs

Dafny at Oxford

2 Dafny as a Programming Language

Dafny as a Programming Language

Dafny is a mature language that allows you to:

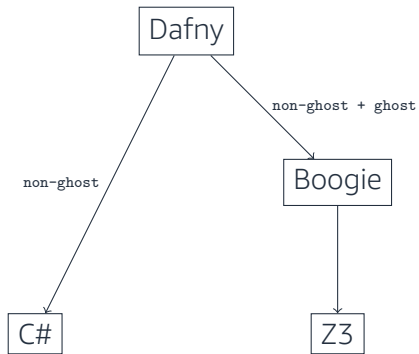
- write functional/imperative/OO programs
- compile programs
- execute programs
- interoperate with other languages

Multi-Paradigms

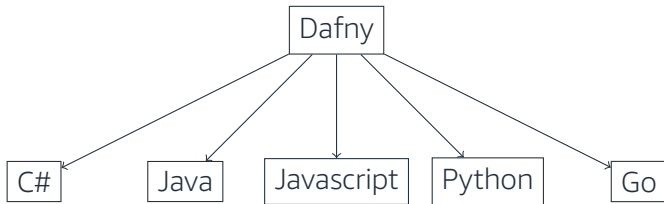
Dafny supports multi-paradigm concepts:

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

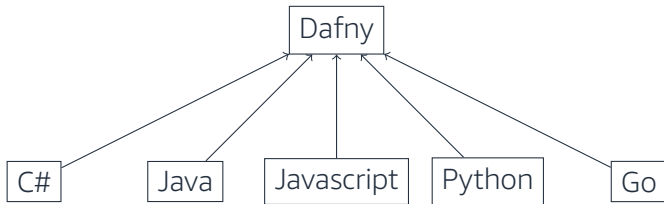
Pipeline



Compilation



Interoperate with `{:extern}`



2.1 Functional Programming

Functions, Constants, Predicates

```
function FunctionName(param1: Type1, param2: Type2): Type3 {  
    expression  
}
```

```
const constantName: Type := expression;
```

```
predicate predicateName(param1: Type1, param2: Type2) {  
    booleanExpression  
}
```

Functions as In/Out Parameters

```
function Apply(f: int -> int, n: int): int {  
    f(n)  
}
```

```
function ApplyPartial(f: int -> int -> int, n: int): int -> int {  
    f(n)  
}
```

Recursive Functions

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

Inductive Datatypes

```
datatype list = Nil | Cons(head: bool, tail: list)

function Conjunction(xs: list): bool {
  match xs
  case Nil => true
  case Cons(head, tail) => head && Conjunction(tail)
}
```

Polymorphism

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

```
function Length<T>(xs: list<T>): nat {  
  match xs  
    case Nil => 0  
    case Cons(_, tail) => 1 + Length(tail)  
}
```

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

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

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

Conditional

```
method IfElse() {  
    if booleanExpression {  
        // ...  
    } else {  
        // ...  
    }  
}
```

Loops

```
method Loops() {  
    while booleanExpression {  
        // ...  
    }  
  
    for variable := startExpression to stopExpression {  
        // ...  
    }  
}
```

Arrays

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

Modifying Arrays

```
method Modify(A: array<bool>, b: bool)
  modifies A
{
  if A.Length == 0 {
  } else {
    A[0] := b;
  }
}
```

2.3 Object-Oriented Programming

Classes

```
class C {  
    var mutableField: int  
    const immutableField: int  
  
    constructor(i: int, j: int) {  
        immutableField := i;  
        mutableField := j;  
    }  
}  
  
method M() {  
    var o := new C(0, 1);  
}
```

Get and Set

```
class C {  
    var mutableField: int  
  
    function Get(): int  
        reads this  
    {  
        mutableField  
    }  
  
    method Set(i: int)  
        modifies this  
    {  
        mutableField := i;  
    }  
}
```


Inheritance

```
trait T {  
  method Print()  
}
```

```
class C extends T {  
  method Print() {  
    print("Stefan");  
  }  
}
```

```
class D extends T {  
  method Print() {  
    print("Zetzsche");  
  }  
}
```

3 Dafny as a Proof Assistant

3.1 Formal Mathematics

Types, Constants, Functions, Predicates, Axioms, and Lemmas

Live Demo

Types, Constants, Functions, Predicates, Axioms, and Lemmas

```
type NaturalNumber

ghost const Zero: NaturalNumber

ghost function Successor(n: NaturalNumber): NaturalNumber

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

lemma {:axiom} Reflexive()
  ensures forall n: NaturalNumber :: Equal(n, n)

lemma {:axiom} ReflexiveAlternative(n: NaturalNumber)
  ensures Equal(n, n)

lemma AboutZero()
  ensures exists n: NaturalNumber :: Equal(n, Zero)
{ ReflexiveAlternative(Zero); }
```

Second Order and Excluded Middle

```
lemma SecondOrder()  
  ensures forall p: int -> bool :: forall x: int :: p(x) || !p(x)  
{}
```

Higher Order

```
lemma ThirdOrder()  
  ensures forall P: (int -> bool) -> bool, p: int -> bool :: P(p) || !P(p)  
{}
```

3.2 Structured Proofs

Proof Structure

```
lemma ProofStructure()  
  requires Assumptions  
  ensures Goal  
{  
  assert Goal by {  
    Assumptions  
  }  
}
```

Conjunction

```
lemma ProofOfConjunction() {  
  assert A && B by {  
    assert A by {  
      // Proof of A  
    }  
    assert B by {  
      // Proof of B  
    }  
  }  
}
```

Contradiction

```
lemma ProofByContradiction() {  
  assert B by {  
    if !B {  
      assert false by {  
        // Proof of false;  
      }  
    }  
  }  
}
```

Coproduct

```
lemma ProofOfCoproduct() {  
  assert (A || B) ==> C by {  
    assert A ==> C by {  
      // Proof of A ==> C;  
    }  
    assert B ==> C by {  
      // Proof of B ==> C;  
    }  
  }  
}
```

Calculations 1

```
lemma UnitIsUnique<T(!new)>(bop: (T, T) -> T, unit1: T, unit2: T)
  requires forall x :: bop(x, unit2) == x
  requires forall x :: bop(unit1, x) == x
  ensures unit1 == unit2
{
  calc {
    unit1;
    ==
    bop(unit1, unit2);
    ==
    unit2;
  }
}
```

Calculations 2

```
lemma UnitIsUnique<T(!new)>(bop: (T, T) -> T, unit1: T, unit2: T)
  requires A1: forall x :: bop(x, unit2) == x
  requires A2: forall x :: bop(unit1, x) == x
  ensures unit1 == unit2
{
  calc {
    unit1;
  == { reveal A1; }
    bop(unit1, unit2);
  == { reveal A2; }
    unit2;
  }
}
```

Calculations 3

```
lemma UnitIsUnique<T(!new)>(bop: (T, T) -> T, unit1: T, unit2: T)
  requires A1: forall x :: bop(x, unit2) == x
  requires A2: forall x :: bop(unit1, x) == x
  ensures unit1 == unit2
{
  assert unit1 == bop(unit1, unit2) by {
    reveal A1;
  }
  assert bop(unit1, unit2) == unit2 by {
    reveal A2;
  }
}
```

4 Dafny for the Verification of Programs

4.1 Independent Verification of Functional Programs

Conditional

```
function Abs(x: int): int {  
  if x < 0 then  
    -x  
  else  
    x  
}
```

```
lemma AbsPositive(x: int)  
  ensures Abs(x) >= 0  
{  
  if x < 0 {  
    assert -x > 0;  
  } else {  
    assert x >= 0;  
  }  
}
```

Recursion and Induction

```
function Length<T>(xs: list): nat {  
  match xs  
    case Nil => 0  
    case Cons(head, tail) => 1 + Length(tail)  
}
```

```
function Append<T>(xs: list, ys: list): list {  
  match xs  
    case Nil => ys  
    case Cons(head, tail) => Cons(head, Append(tail, ys))  
}
```

```
lemma AppendLength<T>(xs: list, ys: list)  
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)  
{  
  match xs  
    case Nil =>  
    case Cons(head, tail) => AppendLength(tail, ys);  
}
```

4.2 Dependent Verification of Functional Programs

Pre- and Postconditions 1

```
lemma AppendLength<T>(xs: list, ys: list)
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
  match xs
  case Nil =>
  case Cons(head, tail) => AppendLength(tail, ys);
}
```

Pre- and Postconditions 2

```
function Append<T>(xs: list, ys: list): list
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
  match xs
  case Nil => ys
  case Cons(head, tail) => Cons(head, Append(tail, ys))
}
```

Pre- and Postconditions 3

```
function Append<T>(xs: list, ys: list): list
  requires Assumption
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
  ensures Property
{
  assert Property by {
    // Proof of Property via Assumption
  }
  match xs
  case Nil => ys
  case Cons(head, tail) => Cons(head, Append(tail, ys))
}
```

Pre- and Postconditions 4

```
function Append<T>(xs: list, ys: list): list
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
  // && forall zs :: Append(Append(xs, ys), zs) == Append(xs, Append(ys, zs))
{
  match xs
  case Nil => ys
  case Cons(head, tail) => Cons(head, Append(tail, ys))
}
```


Termination 1a

```
function SumFromZeroTo(n: nat): nat {  
  if n == 0 then  
    0  
  else  
    n + SumFromZeroTo(n-1)  
}
```

Termination 1b

```
function SumFromZeroTo(n: nat): nat
  decreases n
{
  if n == 0 then
    0
  else
    n + SumFromZeroTo(n-1)
}
```

Termination 2a

```
function SumFromTo(m: nat, n: nat): nat
  requires m <= n
{
  if m == n then
    n
  else
    m + SumFromTo(m+1, n)
}
```

Termination 2b

```
function SumFromTo(m: nat, n: nat): nat
  requires m <= n
  decreases n - m
{
  if m == n then
    n
  else
    m + SumFromTo(m+1, n)
}
```

4.3 Verification of Imperative Programs

Total Hoare Logic

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

```
method S()  
  requires P()  
  ensures Q()
```

Composition

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

```
method S()  
  requires P()  
  ensures Q()
```

```
method T()  
  requires Q()  
  ensures R()
```

```
method Composition()  
  requires P()  
  ensures R()  
{  
  S(); T();  
}
```

Consequence

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

```
lemma Implications()  
  ensures P1() ==> P2()  
  ensures Q2() ==> Q1()
```

```
method S()  
  requires P2()  
  ensures Q2()
```

```
method Consequence()  
  requires P1()  
  ensures Q1()  
{  
  Implications();  
  S();  
}
```


Loops

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

```
method S()  
  requires P() && B()  
  ensures P()
```

```
method WhileLoop()  
  requires P()  
  ensures !B() && P()  
{  
  while B()  
    invariant P()  
    {  
      S();  
    }  
}
```

Dynamic Frames

Live Demo

Dynamic Frames

```
method Modify(A: array<nat>)  
  requires 0 < A.Length  
  modifies A  
  ensures A[0] == 42  
{  
  A[0] := 42;  
}
```

```
method Alias(A: array<nat>, B: array<nat>)  
  requires 0 < A.Length  
  requires A != B  
  modifies A  
  ensures forall i | 0 <= i < B.Length :: old(B[i]) == B[i]  
  ensures unchanged(B)  
  //ensures forall i | 0 <= i < A.Length :: old(A[i]) == A[i]  
{  
  Modify(A);  
}
```

5 Dafny at Oxford

5.1 Use Case Example

Big Step Semantics

Syntax

$$c \in \text{cmd} ::= \text{inc} \mid c_0; c_1 \mid c^*$$

Semantics

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

$\rightarrow \subseteq \text{state} \times \text{cmd} \times \text{state}, \quad \text{state} = \text{int}$

Big Step Semantics in Dafny

Live Demo

Big Step Semantics in Dafny

```
datatype cmd = Inc | Seq(cmd, cmd) | Repeat(cmd)

type state = int

least predicate BigStep(s: state, c: cmd, t: state) {
  match c
  case Inc =>
    t == s + 1
  case Seq(c0, c1) =>
    exists s' :: BigStep(s, c0, s') && BigStep(s', c1, t)
  case Repeat(c0) =>
    (t == s) || (exists s' :: BigStep(s, c0, s') && BigStep(s', Repeat(c0), t))
}

least lemma Increasing(s: state, c: cmd, t: state)
  requires BigStep(s, c, t)
  ensures s <= t
{}
```


5.2 Advanced Topics

Advanced Topics

- Verification of object-oriented programming
- Coinduction, extreme predicates, ordinals
- Subtypes
- Function-by-method
- Variance
- Opaqueness
- Testing, counter-examples

<https://leino.science/dafny-power-user>

<https://dafny.org/dafny/DafnyRef/DafnyRef>

5.3 Opportunities

Open Source

The screenshot shows the GitHub repository page for Dafny. The header includes navigation links for Product, Solutions, Open Source, and Pricing, along with a search bar and Sign In/Sign up buttons. The repository name 'Dafny' is displayed with its logo and a description: 'Dafny is a verification-aware programming language'. Below this, there are tabs for Overview, Repositories (16), Projects, Packages, and People (7). The 'Pinned' section shows the main repository and its statistics: 2,111 stars, 219 forks, 850 issues, and 102 pull requests, updated 1 hour ago. The 'Repositories' section lists other pinned repositories: 'blog' (The Dafny blog, updated 2 days ago), 'ide-vscode' (VSCode IDE integration for Dafny, updated 2 weeks ago), and 'libraries' (Libraries useful for Dafny programs, updated 3 weeks ago). The 'People' section shows a row of profile pictures of contributors. The 'Top languages' section shows a list of languages: Dafny, C#, JavaScript, Shell, and PHP.

Product Solutions Open Source Pricing Search Sign In Sign up

Dafny
Dafny is a verification-aware programming language

Overview Repositories 16 Projects Packages People 7

Pinned

dafny Public
Dafny is a verification-aware programming language
C# 2,111 stars 219 forks 850 issues 102 pull requests Updated 1 hour ago

Repositories

Find a repository... Type Language Sort

blog Public
The Dafny blog
JavaScript 0 stars 1 fork 0 issues 1 pull request Updated 2 days ago

ide-vscode Public
VSCode IDE integration for Dafny
TypeScript 16 stars 15 forks 52 issues 2 pull requests Updated 2 weeks ago

libraries Public
Libraries useful for Dafny programs
Dafny 34 stars 24 forks 30 issues 16 pull requests Updated 3 weeks ago

People

Top languages

Dafny C# JavaScript Shell PHP

<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 challenges 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 180; 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.

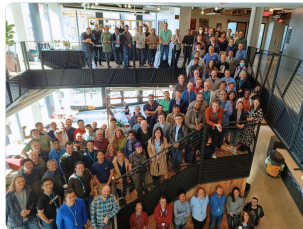
Related Work. In this work we discuss efforts to make formal verification applicable to use-cases related to cloud security at AWS. For information on previous work within AWS to show functional correctness of some key distributed algorithms, see [43]. Other providers of cloud services also use formal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

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



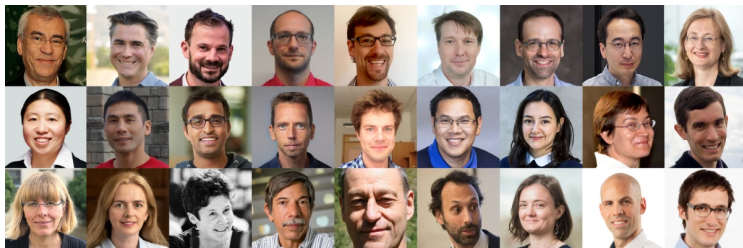
Amazon Science
@AmazonScience

Amazon researchers and engineers gathered for the annual Amazon Formal Reasoning Enthusiasts (FrE) workshop to discuss formal methods tools that improve quality of Amazon software and customer experience. 🙌



10:23 pm · 20 Oct 2022

Amazon Research Awards



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

<https://www.amazon.science/research-awards/call-for-proposals/automated-reasoning-call-for-proposals-fall-2023>

The End

<https://dafny.org/>