The Dafny Programming Language and Static Verifier

Stefan Zetzsche
Amazon Web Services
October 4, 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 at Amazon: Authorization

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

https://github.com/cedar-policy
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/
Table of Contents

Introduction

Dafny as a Programming Language

Dafny as a Proof Assistant

Dafny for the Verification of Programs

Dafny at ILLC
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

- Dafny
  - non-ghost
  - non-ghost + ghost
- Boogie
- C#
- Z3
Compilation
Interoperate with {:extern}
2.1 Functional Programming
Functions, Constants, Predicates

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

const constantName: Type := expression;

predicate predicateName(param1: Type1, param2: Type2) {
    booleanExpression
}
```
Functions as In/Out Parameters

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

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

```python
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)
}
Immutables Collection Types

- Sequences
  \( \text{seq}(\text{length}, \ i \ => \ f(i)) \)

- Sets
  \( \text{set} \ x: T \mid p(x) :: f(x) \)

- Maps
  \( \text{map} \ x: T \mid p(x) :: f(x) \)

- Multisets
2.2 Imperative Programming
Methods

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

```plaintext
method Call() returns (o: int) {
    MethodName("Hello,", "World\n");
    o := FunctionName(42);
}
```
method IfElse() {
    if booleanExpression {
        // ...
    } else {
        // ...
    }
}
While Loop

```java
method WhileLoop() {
    while booleanExpression {
        // ...
    }
}
```
method ForLoop() {
    for variable := startExpression to stopExpression {
        // ...
    }
}
Arrays

```java
method Aliasing() {
    var A := new int[100];
    var B := A;
}
```
function Read(A: array<bool>): bool
reads A
{
    if A.Length == 0 then
        false
    else
        A[0]
}
method Modify(A: array<bool>, b: bool)
  modifies A
{
  if A.Length == 0 {
  } else {
    A[0] := b;
  }
}
2.3 Object-Oriented Programming
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);
}
Functions and Methods

class C {
  var mutableField: int

  function Get(): int
    reads this
    {
      mutableField
    }

  method Set(i: int)
    modifies this
    {
      mutableField := i;
    }
}
Inheritance

```scala
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
Type Symbols

```
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)
Axioms and Quantification

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

\[
\text{lemma} \ {::axiom}\ \text{Reflexive}(n:\ \text{NaturalNumber})
\]
\[
\quad \text{ensures} \ \text{Equal}(n, n)
\]

\[
\text{lemma} \ \text{AboutZero}()
\]
\[
\quad \text{ensures} \ \text{exists} \ n:\ \text{NaturalNumber} :: \ \text{Equal}(n, \text{Zero})
\]
\[
\{ \\
\quad \text{// ...} \\
\}
\]
Lemma

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

lemma AboutZero()
  ensures exists n: NaturalNumber :: Equal(n, Zero)
{
  Reflexive(Zero);
}
Second Order and Excluded Middle

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

```plaintext
lemma ThirdOrder()
    ensures forall P: (int -> bool) -> bool, p: int -> bool :: P(p) || !P(p)
{}
3.2 Structured Proofs
Proof Structure

```c

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
    }
  }
}
Disjunction

```
lemma ProofOfDisjunction1() {
  assert A || B by {
    assert A by {
      // Proof of A
    }
  }
}

lemma ProofOfDisjunction2() {
  assert A || B by {
    assert B by {
      // Proof of B
    }
  }
}
```
lemma ProofOfImplication() {
assert A ==> B by {
  if A {
    assert B by {
      // Proof of B
    }
  }
}
}
Equivalence

```plaintext
lemma ProofOfEquivalence() {
    assert A <=> B by {
        assert A ==> B by {
            // Proof of A ==> B;
        }
        assert B ==> A by {
            // Proof of B ==> A;
        }
    }
}
```
lemma ProofByContradiction() {
assert B by {
if !B {
assert false by {
// Proof of false;
}
}
}
}
Product

```c
lemma ProofOfProduct() {
  assert A ==> (B && C) by {
    assert A ==> B by {
      // Proof of A ==> B;
    }
    assert A ==> C by {
      // Proof of A ==> C;
    }
  }
}
```
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

\[
\text{lemma UnitIsUnique}\langle T(!\text{new})\rangle(bop: (T, T) \rightarrow T, \text{unit1}: T, \text{unit2}: T) \\
\quad \text{requires } \forall x :: \text{bop}(x, \text{unit2}) = x \\
\quad \text{requires } \forall x :: \text{bop}(\text{unit1}, x) = x \\
\quad \text{ensures } \text{unit1} = \text{unit2} \\
\
\{ \\
\quad \text{calc} \{ \\
\quad \quad \text{unit1;} \\
\quad \quad = \text{bop}(\text{unit1}, \text{unit2}); \\
\quad \quad = \text{unit2}; \\
\quad \} \\
\} 
\]
Calculations 2

```c
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;
  }
}
```
lemma ProofUsingExistential<T>(p: T -> bool, q: T -> bool)
  requires A1: exists x :: p(x)
  requires A2: forall x :: p(x) ==> q(x)
  ensures exists x :: q(x)
{
  reveal A1;
  var c :| p(c);
  assert q(c) by {
    reveal A2;
  }
}
4 Dafny for the Verification of Programs
4.1 Independent Verification of Functional Programs
Conditional

```haskell
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

```plaintext
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
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

```c
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

```java
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))
}
```
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))
}
function SumFromZeroTo(n: int): int {
    if n <= 0 then
        0
    else
        n + SumFromZeroTo(n-1)
}
function SumFromZeroTo(n: int): int
decreases n
{
    if n <= 0 then
        0
    else
        n + SumFromZeroTo(n-1)
}
Termination 2a

```haskell
function SumFromTo(m: int, n: int): int {
    if m >= n then
        n
    else
        m + SumFromTo(m+1, n)
}
```
function SumFromTo(m: int, n: int): int
decreases n - m
{
    if m >= n then
        n
    else
        m + SumFromTo(m+1, n)
}
4.3 Verification of Imperative Programs
Hoare Logic (Total)

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

method S()
  requires P()
  ensures Q()
method Skip()
  requires P()
  ensures P()
{}
Assignment

\[
[P[E/x]] x := E \[P]\n\]

```c
method Assignment\langle T\rangle(x: T) returns (y: T)
    requires P(E(x))
    ensures P(y)
{
    y := E(x);
}
```
Composition

\[
\begin{align*}
\vdash & [P] S[Q] \quad [Q] T[R] \\
\vdash & [P] S; T[R]
\end{align*}
\]

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

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

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

\[ P_1 \rightarrow P_2 \ , \ [P_2] S [Q_2] \ , \ Q_2 \rightarrow Q_1 \]
\[ [P_1] S [Q_1] \]

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

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

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

\[
\begin{align*}
\{P \land B\} & \rightarrow S \{P\} \\
\{P\} & \text{ while } B \text{ do } S \{\neg B \land P\}
\end{align*}
\]

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

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

$$\{ P \land B \} \ S \ \{ P \}$$

$$\{ P \} \ \text{while} \ B \ \text{do} \ S \ \{ \neg B \land P \}$$

```java
method Times(n: nat, a: nat) returns (b: nat)
    ensures b == n * a
{
    b := 0;
    var i := 0;
    while i < n
        invariant b == i * a && i <= n
        { b := b + a;
          i := i + 1;
        }
}
5 Dafny at ILLC
5.1 Use Case Example
Big Step Semantics

Syntax

\[ c \in \text{cmd} ::= \text{Inc} \mid c_0; c_1 \mid c^* \]

Semantics

\[ \begin{align*}
& t = s + 1 \\
& s \xrightarrow{\text{Inc}} t \\
& s \xrightarrow{c_0} s', \quad s' \xrightarrow{c_1} t \\
& t = s \\
& s \xrightarrow{c^*} s', \quad s' \xrightarrow{c^*} t
\end{align*} \]
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 Opportunities
Open Source

https://github.com/dafny-lang/dafny
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 hotcp:
https://dafny24.hotcp.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 Zettche (stefanze@amazon.com) and Joseph Tassarotti (jt4767@nyu.edu).
Formal Reasoning at AWS

Formal Reasoning About the Security of Amazon Web Services

Byron Cook1,2

1 Amazon Web Services, Seattle, USA
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 40 significant services and features. In 2012, the number was nearly 100; in 2013, 200; in 2014, 138; 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 workflows.

Formal automated reasoning is one of the investors 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.

Related Work. In this work we discuss efforts to make formal verification applicable to new-core related to cloud security at AWS. For information on previous work within AWS to show functional correctness of some key distributed algorithms, see [16]. Other providers of cloud services also use formal verification to establish security properties, e.g. [33, 34].

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

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.
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/