Week 2: Languages. Formal Systems. Inference rules. Proofs.

Andrei Arusoaie

This material is heavily based on this book:

Boro Sitnikovski. *Introducing Software Verification with Dafny Language – Proving Program Correctness*, ISBN 978-1-4842-7977-9.

Why Do We Need Formal Languages?

Natural languages (English) are ambiguous:

"There exists a number such that it's greater than two and there exists a number such that it's greater than three."

Questions arise:

- Are these the same number or different numbers?
- Are we talking about positive numbers only?
- Are negative numbers allowed?

Computers require precision!

What is a Language?

A language consists of:

1. A finite set of **symbols**: $\{A, B, C, \ldots\}$
2. Rules to combine symbols into **strings**: ABBA, CAB, etc.
3. **Grammar** that defines the rules to construct *valid* strings

Examples:

- English: "Hi, how are you?" (valid)
- English: "hi, how" (not valid)
- Logic: $\forall x.x \in \mathbb{N} \land x \geq 0$ (valid)

Languages Shape Thinking

Language affects how we think:

Example: Numbers in different languages

- English: "ninety-six" $(90 + 6)$
- French: "quatre-vingt-seize" $(4 * 20 + 16)$
- Different grammars encode different construction rules

Insight:
Knowing multiple languages enriches not only our vocabulary but also exposes us to different ways of constructing and organizing ideas.

Different grammars = different ways of thinking

Choosing the Right Programming Language

Why does this matter?

- Different programming languages are better suited for specific tasks
- No single "best" language exists
- An implementation can be easier or harder depending on the language

In programming:

- C may be better for low-level systems programming
- Haskell may be better for expressing mathematical properties
- Python may be better for rapid prototyping
- SQL may be better for database queries

Choose the right (formal) language for the problem at hand!

From Languages to Formal Systems

Languages vs. Formal Systems:

- **Languages** allow us to transfer messages
- **Formal systems** allow us to transfer abstract ideas

Why formal systems?
Formal systems lie at the heart of mathematics and specify its foundations.

Key purpose: Enable reasoning about *form* rather than *content*

- Focus on structure, not meaning
- This abstraction makes them powerful tools
- Essential for proving software correctness

Before proving programs correct, we must understand what "proof" means!

What is a Formal System?

Definition: A model of abstract reasoning consisting of:

1. *Formal syntax*
   - Finite set of symbols
   - Grammar (a set of rules to combine the symbols, in order to obtain strings of symbols)
2. *Axioms*
   - Certain strings accepted as *valid* without proof
3. *Inference rules*
   - Used to construct new valid formulas (theorems)

Key idea: The grammar determines which strings are *syntactically* valid, while the inference rules determine which strings are *semantically* valid.

Example Formal System: Propositional Logic

1. **Alphabet (Symbols):**
   - Propositional atoms: $A = \{p, q, r, \ldots\}$
   - Logical connectives: $\neg, \wedge, \vee, \rightarrow$
   - Parentheses: $(,)$

2. **Grammar (BNF):**

$$\phi ::= a \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi),$$

where $a \in A$.

3. **Examples:**
   - Well-formed: $(p \wedge q)$, $((p \rightarrow q) \vee r)$, $\neg(p \wedge \neg q)$
   - NOT well-formed: $p \wedge q$, $p \wedge \wedge q$, $\rightarrow p$, pq

We may write $p \wedge q$ instead of $(p \wedge q)$ as a notational convention when no ambiguity arises.

Inference Rules

Definition: Rules that allow deriving new theorems from existing ones.

General form of inference rules:

$$\frac{\text{Premise}_1 \quad \ldots \quad \text{Premise}_n}{\text{Conclusion}}$$

Notation: In logic, we use $\vdash$ (turnstile) to say something "is provable"

- $\vdash \phi$ means "ϕ is a theorem"
- $\Gamma \vdash \phi$ means "ϕ is provable from assumptions Γ"

Example: Modus Ponens

$$\frac{\Gamma \vdash P \quad \Gamma \vdash P \to Q}{\Gamma \vdash Q}\ MP$$

If we know P is provable and $P \to Q$ is provable, we can prove Q.

Prerequisites: Sequents

In Propositional Logic, the premisses and the conclusion of an inference rule are called **sequents** and have the form:

$$\Gamma \vdash \varphi,$$

where Γ is a set of formulae (axioms) and φ – a formula (conclusion).

$\Gamma \vdash \varphi$ is *correct* if whenever all formulae in Γ are true, then φ is also true.

Examples:

- If $\Gamma = \{p\}$, then $\Gamma \vdash p$. (Trivial: from p we can conclude p.)
- Γ can also be empty: $\emptyset \vdash \varphi$ means "φ holds without assumptions."
- Another example: $\{p, r\} \vdash p \wedge r$. Intuitively correct: if both p and r hold, then $p \wedge r$ holds. But how do we *prove* it?

Inference Rules for Propositional Logic/Natural Deduction

And-Introduction ($\wedge$-intro):

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{-intro}$$

And-Elimination ($\wedge$-elim):

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{-elim-L} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{-elim-R}$$

Or-Introduction ($\vee$-intro):

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\text{-intro-L} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\text{-intro-R}$$

Or-Elimination ($\vee$-elim):

$$\frac{\Gamma \vdash A \vee B \quad \Gamma \cup \{A\} \vdash C \quad \Gamma \cup \{B\} \vdash C}{\Gamma \vdash C} \vee\text{-elim}$$

Inference Rules for Propositional Logic/Natural Deduction

Hypothesis:

$$\frac{\cdot}{\Gamma \cup \{A\} \vdash A}\ \text{Hyp}$$

Implication Introduction ($\rightarrow$-intro/Deduction Theorem):

$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow\text{-intro}$$

Implication Elimination ($\rightarrow$-elim/Modus-Ponens):

$$\frac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \rightarrow\text{-elim}$$

Negation Introduction ($\neg$-intro) & Negation Elimination ($\neg$-elim):

$$\frac{\Gamma \cup \{A\} \vdash \bot}{\Gamma \vdash \neg A}\ \neg\text{-intro} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg\text{-elim}$$

Double Negation Elimination ($\neg\neg$-elim) & $\bot$-elim:

$$\frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \neg\neg\text{-elim} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ \bot\text{-elim}$$

What is a Proof?

Definition: In general, an argument showing that a conclusion logically follows from hypotheses.

To prove G from premises $\{g_1, g_2, \ldots, g_n\}$, we must show:

$$(g_1 \wedge g_2 \wedge \ldots \wedge g_n) \rightarrow G$$

A proof is valid if: whenever all premises are true, the conclusion is also true.

Example Proof: $\vdash (p \wedge q) \rightarrow (q \wedge p)$

Goal: Prove that $\vdash (p \wedge q) \rightarrow (q \wedge p)$ (N.b. $\vdash \varphi$ is a notation for $\emptyset \vdash \varphi$)

Example Proof: $\vdash (p \wedge q) \rightarrow (q \wedge p)$

Goal: Prove that $\vdash (p \wedge q) \rightarrow (q \wedge p)$ (N.b. $\vdash \varphi$ is a notation for $\emptyset \vdash \varphi$)

Proof using natural deduction:

1. $(p \wedge q) \vdash (p \wedge q)$ (Hyp)
2. $(p \wedge q) \vdash p$ ($\wedge$-elim-L, 1)
3. $(p \wedge q) \vdash q$ ($\wedge$-elim-R, 1)
4. $(p \wedge q) \vdash (q \wedge p)$ ($\wedge$-intro, 3, 2)
5. $\vdash (p \wedge q) \rightarrow (q \wedge p)$ ($\rightarrow$-intro, 1, 4)

$$\frac{\cdot}{\Gamma \cup \{A\} \vdash A}\ \text{Hyp} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim-L} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim-R}$$

$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow\text{-intro} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}$$

Example Proof: $\vdash (p \wedge q) \to (q \wedge p)$

Goal: Prove that $\vdash (p \wedge q) \to (q \wedge p)$ (N.b. $\vdash \varphi$ is a notation for $\emptyset \vdash \varphi$)

A proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{(p \wedge q) \vdash (p \wedge q)}\,\text{Hyp}}{(p \wedge q) \vdash q}\,\wedge\text{-elim-R} \quad \dfrac{\dfrac{\cdot}{(p \wedge q) \vdash (p \wedge q)}\,\text{Hyp}}{(p \wedge q) \vdash p}\,\wedge\text{-elim-L}}{(p \wedge q) \vdash (q \wedge p)}\,\wedge\text{-intro}}{\vdash ((p \wedge q) \to (q \wedge p))}\,\to\text{-intro}$$

$$\dfrac{\cdot}{\Gamma \cup \{A\} \vdash A}\,\text{Hyp} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge\text{-elim-L} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge\text{-elim-R}$$

$$\dfrac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \to B}\,\to\text{-intro} \qquad \dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge\text{-intro}$$

Why Formal Languages Matter

Natural languages limitations:

- Ambiguous and context-dependent
- No systematic way to verify correctness of reasoning
- Difficult to establish absolute trust in arguments

Formal languages enable trustworthy reasoning:

- They are precise, have *unambiguous* syntax and semantics
- Enable mechanical verification of proofs
- If reasoning follows the rules, the conclusions are guaranteed correct

Computers can operate on formal languages!

- Tools for formal software verification are possible (e.g., Dafny)
- Automated checking of proofs and reasoning

Dafny – intro

What is Dafny?

- A programming language with built-in verification
- Designed by Rustan Leino (Microsoft Research)
- Automatically proves program correctness
- Uses Z3 theorem prover under the hood

Key features:

- Preconditions (`requires`)
- Postconditions (`ensures`)
- Loop invariants (`invariant`)
- Algebraic datatypes
- Pattern matching

First Dafny Program

```
1 method Main() {
2   print "Hello, World!\n";
3 }
```

With verification:

```
1 method Main()
2   ensures 3 * 2 == 6
3 {
4   print "Hello, World!\n";
5 }
```

Dafny proves properties at compile time!

Basic Constructs: Methods and Functions

Method (can have side effects, runs at runtime):

```
method Abs(x: int) returns (y: int)
{
  if x < 0 {
    return -x;
  } else {
    return x;
  }
}
```

Function (pure, used in specifications):

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

Algebraic Data Types (ADTs)

ADTs define custom types with multiple variants (constructors).

Basic syntax:

```
datatype TypeName<T> =
  | Constructor1(field1: Type1)
  | Constructor2(field2: Type2, field3: Type3)
  | ...
```

Example: Boolean

```
datatype Bool = True | False
```

ADT Example: Lists

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

Creating lists:

```
// Empty list
const empytyList: List<int> := Nil

// [1, 2, 3]
const someList: List<int> :=
                    Cons(1, Cons(2, Cons(3, Nil)))
```

Note: Constructor parameters are *named*, making code more readable.

ADT Example: Binary Trees

```
datatype Tree<T> =
  Leaf
| Node(value: T, left: Tree<T>, right: Tree<T>)
```

```
// A simple tree:
//       5
//      / \
//     3   7
const myTree :=
  Node(
    5,
    Node(3, Leaf, Leaf),
    Node(7, Leaf, Leaf)
  )
```

ADT Example: Option Type

Represents a value that may or may not exist.

```
datatype Option<T> =
  | None
  | Some(value: T)
```

Usage:

```
function divide(x: int, y: int) : Option<int>
{
    if y == 0
    then None
    else Some(x / y)
}
```

Properly handle the special cases

Back to Functions

Pure functions use the `function` keyword:

- No side effects
- Deterministic (same input $\rightarrow$ same output)
- Can be used in specifications
- Must terminate

Syntax:

```
function functionName<T>(param: Type):
    ReturnType
{
  // function body
}
```

Function Example: Factorial

```
1  function factorial(n: nat): nat
2  {
3    if n == 0 then 1
4    else n * factorial(n - 1)
5  }
```

Key features:

- `nat` is the type of natural numbers (non-negative)
- Recursive definition
- Single expression body (no return statement)
- Dafny automatically verifies termination

Collatz

```
1  function collatz(n: nat) : nat
2  {
3      if n == 0 then 1
4      else if n % 2 == 0 then collatz(n / 2)
5      else collatz(3 * n + 1)
6  }
```

- Dafny: `cannot prove termination; try supplying a decreases clause`
- In 2019, mathematician Terence Tao proved that the Collatz conjecture is "almost" true for "almost" all starting numbers
- `https://terrytao.wordpress.com/2019/09/10/almost-all-collatz-orbits-attain-almost-bounded-values/`

Pattern Matching with `match`

Pattern matching deconstructs ADTs:

```
1 function length<T>(xs: List<T>): nat
2 {
3   match xs
4   case Nil => 0
5   case Cons(head, tail) => 1 + length(tail)
6 }
```

- `match` expression examines the structure of `xs` as in the definition of `List<T>`
- Each `case` handles exactly one constructor
- Binds constructor fields to variables
- Must be exhaustive (all cases covered)!

More Pattern Matching Examples

List append:

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

Tree height:

```
function height<T>(t: Tree<T>): nat
{
  match t
  case Leaf => 0
  case Node(_, left, right) =>
    1 + max(height(left), height(right))
}
```

Higher-Order Functions

Functions can take other functions as parameters!

```
1  function mymap<A, B>(f: A -> B, xs: List<A>):
       List<B>
2  {
3      match xs
4      case Nil => Nil
5      case Cons(head, tail) =>
6          Cons(f(head), mymap(f, tail))
7  }
8
9  function increment(x: int): int { x + 1 }
10
11 method testMap() returns(res: List<int>)
12   ensures res == Cons(2, Cons(3, Cons(4, Nil)))
13 {
14     var list := Cons(1, Cons(2, Cons(3, Nil)));
15     res := mymap(increment, list);
16 }
```

Higher-Order Functions

```
1  function filter<T>(pred: T -> bool, xs: List<T>)
       : List<T>
2  {
3    match xs
4    case Nil => Nil
5    case Cons(head, tail) =>
6      if pred(head)
7      then Cons(head, filter(pred, tail))
8      else filter(pred, tail)
9  }
10 function isEven(x: int): bool { x % 2 == 0 }
11 method testFilter() returns(result: List<int>)
12     ensures result == Cons(2, Cons(4, Nil))
13 {
14     var list := Cons(1, Cons(2, Cons(3, Cons(4,
           Nil))));
15     result := filter(isEven, list);
16 }
```

Function Specifications

Dafny's superpower: **formal specifications**

```
1  function factorial(n: nat): nat
2    requires n >= 0
3    ensures factorial(n) >= 1
4    ensures n > 0 ==> factorial(n) >= n
5  {
6    if n == 0 then 1
7    else n * factorial(n - 1)
8  }
```

- `ensures`: postconditions (what's true after execution)
- `requires`: preconditions (assumptions about inputs)
- Dafny **automatically verifies** these properties!

Specification Example: List Length

```
1  function length<T>(xs: List<T>): nat
2    ensures length(xs) >= 0
3  {
4    match xs
5    case Nil => 0
6    case Cons(_, tail) => 1 + length(tail)
7  }
8
9  function append<T>(xs: List<T>, ys: List<T>):
       List<T>
10   ensures length(append(xs, ys)) ==
11           length(xs) + length(ys)
12 {
13   match xs
14   case Nil => ys
15   case Cons(head, tail) =>
16             Cons(head, append(tail, ys))
17 }
```

Termination: The `decreases` Clause

Dafny requires proof that functions terminate.

```
function gcd(a: nat, b: nat): nat
  requires b > 0
  decreases a
{
  if a == 0 then b
  else gcd(b % a, a)
}
```

- `decreases n` says the parameter `n` gets smaller
- Dafny verifies the measure decreases in each recursive call
- Usually inferred automatically for simple cases
- Explicit `decreases` needed for complex recursion

Propositional Logic in Dafny

Direct encoding of our formal language:

```
datatype Atom = P | Q | R
datatype Prop =
  | Var(Atom)                          // Constants
  | Not(Prop)                          // Negation
  | And(Prop, Prop)                    // Conjunction
  | Or(Prop, Prop)                     // Disjunction
  | Imp(Prop, Prop)                    // Implication
```

Grammar (BNF):

$$\phi ::= a \mid \neg\phi \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi),$$

where $a \in A = \{p, q, r, \ldots\}$.

Propositional Logic in Dafny

Direct encoding of our formal language:

```
datatype Prop =
  | P | Q | R                          // Constants
  | Not(Prop)                          // Negation
  | And(Prop, Prop)                    // Conjunction
  | Or(Prop, Prop)                     // Disjunction
  | Imp(Prop, Prop)                    // Implication
```

Examples of encoding formulas:

- $(p \wedge q)$ becomes `And(P, Q)`
- $(\neg p \rightarrow q)$ becomes `Imp(Not(P), Q)`
- $((p \vee q) \wedge r)$ becomes `And(Or(P, Q), R)`

Valuations of Propositional Variables

Def: A *valuation* is a function that assigns truth values to atoms:

$$\alpha : A \to \{\text{true}, \text{false}\}$$

Example valuation α_1:

$$\alpha_1(p) = \text{true}, \quad \alpha_1(q) = \text{false}, \quad \alpha_1(r) = \text{true} \quad \alpha_1(\_) = \text{true}$$

Encoding in Dafny:

```
1  function alpha1(a: Atom): bool
2  {
3      match a
4      case P => true
5      case Q => false
6      case R => true
7  }
```

Semantics of Propositional Formulas

Def: Given a valuation α, we define $[\![\varphi]\!]_\alpha$ inductively:

$$
\begin{aligned}
[\![a]\!]_\alpha &= \alpha(a) \\
[\![\neg\varphi]\!]_\alpha &=! [\![\varphi]\!]_\alpha \\
[\![\varphi \wedge \psi]\!]_\alpha &= [\![\varphi]\!]_\alpha \,\&\&\, [\![\psi]\!]_\alpha \\
[\![\varphi \vee \psi]\!]_\alpha &= [\![\varphi]\!]_\alpha \,||\, [\![\psi]\!]_\alpha \\
[\![\varphi \to \psi]\!]_\alpha &=! [\![\varphi]\!]_\alpha \,||\, [\![\psi]\!]_\alpha
\end{aligned}
$$

Encoding in Dafny:

```
function eval(phi: Prop, alpha: Atom -> bool): bool
{
  match phi
  case Var(p) => alpha(p)
  case Not(p) => !eval(p, alpha)
  case And(p, q) => eval(p, alpha) && eval(q, alpha)
  case Or(p, q) => eval(p, alpha) || eval(q, alpha)
  case Imp(p, q) => !eval(p, alpha) || eval(q, alpha)
}
```

Testing Evaluation

Example: Evaluate $((p \wedge q) \rightarrow r)$ under α_1

- Formula: $((p \wedge q) \rightarrow r)$
- Valuation: $\alpha_1(p) = \text{true}, \alpha_1(q) = \text{false}, \alpha_1(r) = \text{true}$
- Details:
  $[\![((p \wedge q) \rightarrow r)]\!]_{\alpha_1} = \ldots = !([\![p]\!]_{\alpha_1} \,\&\&\, [\![q]\!]_{\alpha_1}) \,||\, [\![r]\!]_{\alpha_1} =$
  $!(\text{true} \,\&\&\, \text{false}) \,||\, \text{true} = !\text{false} \,||\, \text{true} = \text{true}$

Dafny implementation:

```
1  method testEval() returns (b: bool)
2  {
3      var phi := Imp(And(Var(P), Var(Q)), Var(R));
4      b := eval(phi, alpha1);
5      assert b == true;
6  }
```

Note: Dafny automatically verifies the assertion holds!

Validity in Propositional Logic

Def: A formula φ is *valid* (*tautology*) if it evaluates to true under all valuations:

$$\models \varphi \quad \text{iff} \quad \textit{for all } \alpha : [\![\varphi]\!]_\alpha = \text{true}$$

Example: The formula $p \to p$ is valid, i.e., $\models p \to p$.

Dafny:

```
1 method testPimpliesP()
2   ensures forall alpha ::
3       eval(Imp(Var(P), Var(P)), alpha) == true
4 {}
```

Dafny proves the postcondition automatically, no computation needed— just pure verification.

Satisfiability in Propositional Logic

Def: A formula φ is *satisfiable* if there exists at least one valuation that makes it true:

$$\varphi \text{ is satisfiable} \quad \text{iff} \quad \textit{there exists } \alpha : [\![\varphi]\!]_\alpha = \text{true}$$

Example: $p \wedge q$ is satisfiable with α_1: $\alpha_1(x) = \text{true}$, for any $x \in A$.

Dafny:

```
1  method testSat()
2    ensures exists alpha ::
3          eval(And(Var(P), Var(Q)), alpha) == true
4  {
5    var alpha1 := (a : Atom) => true;
6    assert
7      eval(And(Var(P), Var(Q)), alpha1) == true;
8  }
```

Note: We have to provide a *witness* valuation α_1, then assert it satisfies the formula. Dafny uses the witness to verify the existential claim.

Equivalence in Propositional Logic

Definition: Two formulas φ and ψ are *logically equivalent* if they have the same truth value under all valuations:

$$\varphi \equiv \psi \quad \text{iff} \quad \textit{for all } \alpha : [\![\varphi]\!]_\alpha = [\![\psi]\!]_\alpha$$

Example: The formulas $p \to q$ and $\neg p \vee q$ are logically equivalent.

Verification in Dafny:

```
method testEquiv()
  ensures forall alpha ::
    eval(Imp(Var(P), Var(Q)), alpha) ==
    eval(Or(Not(Var(P)), Var(Q)), alpha)
{}
```

Note: Dafny automatically verifies this classical equivalence.

Predicates in Dafny

Predicates are boolean-valued functions that can be used in specs.

Key features:

- Declared with `predicate` keyword
- Return type is always `bool` (implicit)
- `ghost` means used only for verification!
- Can use quantifiers (`forall`, `exists`)

Syntax:

```
1 ghost predicate predicateName(params) {
2   // boolean expression
3 }
```

We can reuse complex logical properties via predicates; improves readability as well.

Defining Logical Properties as Predicates

We can encode our semantic definitions as (reusable) predicates:

```
1  ghost predicate satisfiable(phi: Prop) {
2    exists alpha: Atom -> bool :: eval(phi, alpha)
3  }
4
5  ghost predicate tautology(phi: Prop) {
6    forall alpha :: eval(phi, alpha)
7  }
8
9  ghost predicate equivalent(phi: Prop, psi: Prop)
10 {
11   forall alpha ::
12     eval(phi, alpha) == eval(psi, alpha)
13 }
```

Lemmas in Dafny

Lemmas are theorems that Dafny proves automatically.

Syntax:

```
lemma lemmaName()
  ensures postcondition
{
  // proof (optional)
}
```

Differences w.r.t. methods:

- Ghost code (verification only, not compiled)
- Prove mathematical properties
- Can be called in other proofs to establish facts
- Body can be empty if Dafny can prove it automatically

Using Predicates in Lemmas: Satisfiability

Lemma: The formula $p \wedge q$ is satisfiable.

```
1  lemma TestSatisfiable()
2    ensures satisfiable(And(Var(P), Var(Q)))
3  {
4    var alpha1 := (a : Atom) => true;
5    assert
6      eval(And(Var(P), Var(Q)), alpha1) == true;
7  }
```

Proof:

1. Dafny needs a witness valuation: we provide α_1 (which maps all propositional variables to true)
2. Assert the formula evaluates to true under α'
3. Dafny verifies this satisfies the existential in the predicate

Using Predicates in Lemmas: Validity & Equivalence

Lemma: The formula $p \rightarrow p$ is a tautology.

```
1 lemma testValidity()
2   ensures tautology(Imp(Var(P), Var(P)))
3 { }
```

Lemma: Implication translates into a disjunction.

```
1 lemma testEquivalence()
2   ensures equivalent(Imp(Var(P), Var(Q)),
3                      Or(Not(Var(P)), Var(Q)))
4 { }
```

Overview

1. **Formal languages** eliminate ambiguity
2. **Formal systems** = Language + Axioms + Inference Rules
3. **Proofs** are systematic applications of inference rules
4. **Dafny** enables formal systems for verification
5. Dafny automatically verifies correctness using SMT solving
6. Sometimes, in Dafny, we can help the prover (e.g., providing witness for existential statements)

Questions?