КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА КІБЕРНЕТИКИ

РЕФЕРАТ

з дисципліни: «Формальні методи розробки програмних систем»
на тему: «Верифікація програм за допомогою Dafny»

Виконала:
студентка 1 курсу магістратури
групи ШІ
Кіндякова Діана Валеріївна

Верифікація програм за допомогою Dafny.

Dafny — мова програмування з функціональними та об'єктно-орієнтованими можливостями, яка підтримує автоматичну перевірку програм згідно зі специфікацією.

Завдяки використанню теорем, перед- і післяумов, інваріантів циклів, асерцій та інших засобів, вона дозволяє створювати програми з формально доведеною коректністю. Якщо програма проходить перевірку в Dafny, це гарантує її:

- повну коректність — вона завжди завершує виконання (не зациклиться і не аварійно завершиться);
- часткову коректність — якщо завершиться, то точно відповідатиме специфікації.

Dafny використовує потужний автоматичний доводник теорем — Z3, який здатен самостійно доводити правильність багатьох програм. Однак у складніших випадках йому може знадобитися додаткова допомога від програміста — у вигляді уточнених умов, лем або підказок.

Гарантії правильності

Однією з головних переваг Dafny є формальна перевірка правильності коду. Вона гарантує:

- відсутність помилок часу виконання, таких як вихід за межі масиву, ділення на нуль, звернення до null;
- виконання всіх передумов перед викликом методу;
- виконання післяумов після завершення методу;
- коректне завершення коду (тобто відсутність нескінченних циклів, якщо не зазначено інше).

Ці гарантії досягаються за рахунок верифікації на основі логічних формул, що автоматично генеруються з анотацій. Якщо анотації правильні, Dafny доводить, що реалізація коду відповідає їм. Водночас, Dafny не лише доводить правильність, а й допомагає виявити логічні суперечності або пропущені припущення, що особливо важливо для складних алгоритмів.

Основні конструкції мови Dafny

Мова Dafny поєднує імперативний стиль програмування із декларативними конструкціями для формалізації вимог до програм. Вона спеціально створена для полегшення формальної верифікації шляхом інтеграції специфікацій, таких як предусловія, постумови, інваріанти, а також підтримки логічних доведень.

1. Змінні.

У Dafny всі змінні мають **чітко визначений тип**. Наприклад:

- `int` — цілі числа (від'ємні та додатні),
- `nat` — натуральні числа (не від'ємні),
- `bool` — логічні значення (true або false),
- `array<T>` — масиви елементів типу T,
- `set<T>, seq<T>` — множини та послідовності.

Локальні змінні оголошуються в методах за допомогою ключового слова `var`:

```
var receipts: map<string, seq<Receipt>> // Зберігає квита
var totalEarnings: int // Загальна сума заробітку
```

Їх можна ініціалізувати з явним зазначенням типу або дозволити Dafny вивести тип автоматично. Це забезпечує гнучкість та зручність написання коду. Тип змінної можна не вказувати, якщо значення для ініціалізації вже визначає тип. Наприклад:

```
var x: int := 5;
var y := 10;
```

2. Асерції.

Асерції (`assert`) — це твердження, які вбудовуються в тіло методу, щоб перевірити, що певний логічний вираз завжди істинний у момент його досягнення під час виконання програми. Якщо це не так — верифікація не проходить. Асерції не впливають на виконання коду, але є способом перевірити, що Dafny відомо про змінні та їх значення на певному етапі виконання. Вони слугують точками інспекції, які дозволяють верифікувати очікувану поведінку або значення, виходячи з уже доведених властивостей (наприклад, після виклику методу з певною післяумовою). Наприклад:

```
var v := Abs(3);
assert 0 <= v;
assert v == 3;
```

3. Методи.

У Dafny методи(`method`) — це основна структурна одиниця для написання імперативного коду. Вони схожі на процедури або функції в інших мовах, але слово "`function`" у Dafny використовується окремо — для чистих функцій, що не змінюють стану. Кожен метод має чітко визначені вхідні параметри та вихідні значення з обов'язковим зазначенням типів. Dafny підтримує кілька вихідних значень, кожному з яких надається ім'я. Вхідні параметри є лише для читання, тоді як вихідні значення функціонують як локальні змінні, які можна перевизначати в тілі методу.

Методи використовують стандартні конструкції: умовні оператори "`if`", цикли, виклики інших методів, присвоєння. Особливістю є використання оператора := для присвоєння, а не =, == зарезервовано для перевірки на рівність

Ключовою особливістю методів у Dafny є можливість додавання специфікацій — таких як передумови (`requires`) і післяумови (`ensures`) :

```
method Abs(x: int) returns (y: int)
  requires x < 0
  ensures 0 <= y
  ensures y == -x
{
  return -x;
}
```

Які дозволяють точно визначати, які умови мають бути виконані до та після виконання методу відповідно. Вони описують поведінку методу на логічному рівні, і Dafny намагається формально довести, що реалізація методу їм відповідає. Наприклад:

```
method Max(a: int, b: int) returns (c: int)
  ensures c == a || c == b
  ensures c >= a && c >= b
{
  if a > b {
    return a;
  } else {
    return b;
  }
}
```

Однак важливо пам’ятати, що Dafny «забуває» реалізацію методу під час перевірки інших методів. Він використовує тільки специфікації методу — тобто те, що написано у `ensures`. Тобто, Dafny завжди передбачає, що метод може бути будь-яким, якщо він задовольняє свої `ensures`. Це вимагає суворої формалізації очікуваних властивостей.

4. Функції.

На відміну від методів, функції (`function`) не мають змінних і виконуються як чисті вирази. Dafny *не забуває* тіла функцій, тому вони можуть бути використані у специфікаціях (у `assert, ensures` тощо), на відміну від методів. Приклад функції:

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

5. Предикати

Предикат — це функція, яка повертає логічне значення (bool). Використовується для задання властивостей об’єктів або структур, наприклад:

Фреймінг

Dafny обмежує доступ до пам’яті в предикатах через анотацію `reads`, яка визначає, які частини пам’яті дозволено читати. Без цієї анотації Dafny не дозволяє звертатись до масиву в предикаті:

```
predicate sorted(a: array<int>)
  reads a
{
  forall j, k :: 0 <= j < k < a.Length ==> a[j] < a[k]
}
```

Підпис

6. Масиви.

Новий масив створюється за допомогою ключового слова new та задається як `array<T>`, де `T` — тип елементів. Наприклад, `array<int>` — це масив цілих

чисел. Існує також тип `array?<T>`, що охоплює як сам масив, так і значення `null`.

Масив має поле `Length`, що задає кількість елементів. Доступ до елементів здійснюється за індексами з нуля: `a[0], a[1], ..., a[n-1]`. Важливо, що усі доступи до елементів повинні бути доведені як безпечні (тобто в межах розмірів масиву). Приклад методу з використанням масиву:

```
method Find(a: array<int>, key: int) returns (index: int)
  ensures 0 <= index ==> index < a.Length && a[index] == key {
    // to do
  }
```

7. *Квантор forall*

Універсальний квантор (`forall`) дозволяє задати властивість, яка повинна виконуватись для всіх значень певної множини:

```
assert forall k :: 0 <= k < a.Length ==> a[k] != key;
```

А також спростити написання циклів:

```
forall i | 0 <= i < n - m {
  b[i] := a[m + i];
}
```

8. *Квантор exists*

Квантор існування (`exists`) дозволяє формально виразити твердження виду: "Існує хоча б один елемент, для якого виконується певна умова":

```
predicate P(n: int) {true}

method Main() {
    assert P(2);
    assert (exists n : int :: n > 1);
}
```

9. *Цикли*

Інваріанти циклів.

Для перевірки правильності циклів Dafny потребує інваріантів(`invariant`) — логічних виразів, які мають бути істинними перед початком циклу та після

кожної ітерації. Це дозволяє формально гарантувати, що цикл поводиться передбачувано:

```
var n := 2;
var i := 0;
while i < n
    invariant 0 <= i <= n
{
    i := i + 1;
}
```

Гарантування завершення циклу (termination).

Якщо тіло циклу не змінює змінні, інваріанти можуть бути правильними, але цикл ніколи не завершиться. Dafny вимагає, щоб цикл робив прогрес — тобто, щоб після кожної ітерації був ближчий до завершення Dafny доводить, що код завершується, тобто не зациклюється нескінченно, за допомогою "`decreases`" анотацій. Для багатьох речей Dafny здатна вгадати правильні анотації, але іноді це потрібно зробити явно. Є два місця, де потрібно гарантувати завершення: цикли та рекурсія. Обидві ці ситуації вимагають або явної анотації, або правильного припущення від Dafny:

```
method m()
{
  var i, n := 0, 20;
  while i < n
    invariant 0 <= i <= n
    decreases n - i
  {
    i := i + 1;
  }
}
```

10. Префікс ghost

У системі формальної верифікації Dafny важливою особливістю є можливість розділення логіки програми та логіки її доведення. Для цього передбачено спеціальний механізми — ghost-змінні, методи та функції які використовуються лише для специфікацій, інваріантів, допоміжних обчислень і не впливають на виконання програми під час її запуску. Оголошення Ghost-змінної:

```
ghost var Contents: seq<int>
```

Оголошення ghost-функції:

```
ghost function Sum(a: seq<int>): int
{
  if |a| == 0 then 0 else a[0] + Sum(a[1..])
}
```

Обмеження на використання

- Ghost-елементи не можна використовувати у виконуваному коді — наприклад, для обчислень, що впливають на результат методу.
- Ghost-змінні не можна модифікувати в контексті, де це впливає на логіку програми.
- Dafny гарантує, що ghost-елементи повністю усуваються при компіляції: результуючий виконуваний код не містить жодного посилання на них.

11. Класи.

Мова формальної верифікації Dafny підтримує класи, які дозволяють створювати динамічно виділені змінювані структури даних. Основою об’єктно-орієнтованого програмування в Dafny є посилання (вказівники), що дає гнучкість в реалізації, але ускладнює формальну специфікацію таких об'єктів. Водночас, саме в таких складних випадках формальна верифікація виявляється найбільш корисною.

Клас оголошується за допомогою ключового слова `class`. Клас може мати конструктор, що викликається при створенні нового об’єкта. Конструктор має назву constructor і може мати перед- та постумови.

Посилальні типи.

Класи в Dafny є посилальними типами (reference types), тобто об’єкти створюються в купі (heap), і змінні містять посилання на об’єкти. Це означає, що при передачі об'єкта як параметра передається посилання, а не копія.

modifies this

На відміну від функцій і предикатів, методи можуть змінювати стан об'єкта. Dafny вимагає, щоб методи явно вказували, які частини пам’яті вони змінюють, за допомогою анотації modifies. Якщо змінюється лише стан поточного об’єкта, досить написати modifies this. Приклад використання класу:

```
class ParkingMeter {

    var receipts: map<string, seq<Receipt>> // Зберігає квитанції: номер транспорту -> список (час початку, тривалість)
    var totalEarnings: int // Загальна сума заробітку
    var ratePerHour: int // Тариф за годину

    constructor(rate: int)
        ensures ratePerHour == rate
        ensures totalEarnings == 0
        ensures receipts == map[]
    {
        ratePerHour := rate;
        totalEarnings := 0;
        receipts := map[];
        print"Паркометр ініціалізовано з тарифом за годину: ", rate, "\n";
    }

    // Метод для видачі квитанції
    method IssueReceipt(terminalWorks : bool, vehicleNumber: string, duration: int,
                        paymentType:string, amountPaid : int, currentTime : int) returns (success: bool)
        requires duration > 0
        requires amountPaid >= 0
        modifies this
        ensures success ==> vehicleNumber in this.receipts && exists t:
                Receipt :: t in this.receipts[vehicleNumber] && t.startTime + t.duration > currentTime
    {…
    }

    // Метод для прийняття оплати
    method AcceptPayment(terminalWorks: bool, vehicleNumber: string, duration: int, …
    {…
    }

    // Метод для перевірки, чи було сплачено
    method VerifyPayment(vehicleNumber: string, currentTime: int) returns (isPaid: bool)
        ensures isPaid ==> vehicleNumber in this.receipts && exists t: …
    {…
    }
}
```

Інваріанти класу.

Інваріант класу зазвичай реалізується як булевий предикат Valid, який використовується як перед- і постумова у методах (лише постумова для конструктора). Наприклад:

```
class  MyClass {
  var x: int

   predicate Valid()
    reads this`x
  {
    x >= 0
  }

  constructor(init: int)
    requires init >= 0
    ensures Valid()
  {
    x := init;
  }

  method Inc()
    requires Valid()
    modifies this
    ensures Valid()
    {
        x := x + 1;
    }
}
```

Null-значення та перевірки

Змінні класів можуть мати значення null. Dafny вимагає явної перевірки на null, якщо не вказано, що змінна завжди не-null (наприклад, через ghost змінні або інваріанти):

```
method Use(p: MyClass?)
  requires p != null
{
  var xVal := p;
}
```

Автоматичні контракти

Атрибут `{:autocontracts}` дозволяє автоматично додавати специфікації до конструктора та методів: автоматично додає `Valid()` як перед- і постумову, а також додає `modifies` для відстеження змін. До того ж це зменшує кількість шаблонного коду, що підвищує зручність.

Комбінування ghost- і фізичних змінних

Звичайна практика в Dafny — описувати клас за допомогою двох наборів змінних фізичних (не-ghost), які реалізують структуру даних ефективно та ghost-змінних, які описують логічний вміст структури та її інваріанти.

Між цими наборами встановлюється зв’язок за допомогою інваріанта класу. Компільований код включає тільки фізичні змінні, а ghost-змінні та логічні конструкції використовуються лише під час перевірки. Приклад використання:

```
class {:autocontracts} SimpleQueue {
  ghost var Contents: seq<int>;
  var a: array?<int>;
  var m: int, n: int;

  ghost predicate Valid()
  reads this{
    a != null && a.Length > 0 && 0 <= m <= n <= a.Length && Contents == a[m..n]
  }

  constructor ()
    ensures Contents == [];
  {
    a, m, n, Contents := new int[10], 0, 0, [];
  }
```

```
  method Enqueue(d: int)
    ensures Contents == old(Contents) + [d];
  {
    if n == a.Length {
      var b := a;
      if m == 0 {
        b := new int[2 * a.Length];
      }
      forall i | 0 <= i < n - m {
        b[i] := a[m + i];
      }
      a, m, n := b, 0, n - m;
    }
    a[n], n, Contents := d, n + 1, Contents + [d];
  }

  method Dequeue() returns (d: int)
    requires Contents != [];
    ensures d == old(Contents)[0] && Contents == old(Contents)[1..];
  {
    d, m, Contents := a[m], m + 1, Contents[1..];
  }
}
```

12. Леми.

У мові формальної верифікації Dafny леми (lemma) є спеціальним видом процедур, призначених для доведення математичних властивостей, які не завжди очевидні для автоматичного доведення. Лема в Dafny — це логічне твердження, яке виконує роль допоміжного доведення: вона не виконує ніяких обчислень у програмі, але дозволяє верифікатору зробити правильні логічні висновки. Інколи вони також реалізуються за допомогою ghost-методів.

Нехай ми маємо деякий метод `ComputePow2(n: nat)`, який обчислює 2^n у логарифмічний час. Ми знаємо, що

$$2^n = (2^{n/2})^2$$

Це очевидне математичне твердження, але для верифікатора Dafny — це недоведене припущення, і тому йому потрібно це явно вказати.

Рішення: ввести lemma-метод Lemma(n), який:

- вимагає, щоб n було парним
- доводить, що $pow2(n) = pow2(n/2)^2$
- виконує індуктивне доведення, бо вона викликає себе з n - 2

Dafny розглядає тіло `Lemma(n)` як доведення, де ви маєте переконати його в `ensures`. Це так зване proof by recursion = proof by induction. Приклад леми оголошеної через `lemma`

```
lemma Lemma(n: nat)
  requires n % 2 == 0
  ensures pow2(n) == pow2(n/2) * pow2(n/2)
{
  if n != 0 {
    Lemma(n - 2); // рекурсивне доведення
  }
}
```

Та приклад через ghost-метод:

```
ghost method Lemma(n: nat)
  requires n % 2 == 0
  ensures pow2(n) == pow2(n/2) * pow2(n/2)
{
  if n != 0 {
    Lemma(n - 2);
  }
}
```

Використання леми не залежить від того, як саме вона була задана:

```
function pow2(n: nat): nat {
  if n == 0 then 1 else 2 * pow2(n - 1)
}

method ComputePow2(n: nat) returns (p: nat)
  ensures p == pow2(n)
{
  if n == 0 {
    p := 1;
  } else if n % 2 == 0 {
    p := ComputePow2(n / 2);
    p := p * p;
    Lemma(n);
  } else {
    p := ComputePow2(n - 1);
    p := 2 * p;
  }
}
```

Нестабільність верифікації (Verification Variability)

Верифікація іноді проходить успішно, але після незначних змін — провалюється. Це явище називається *верифікаційною варіативністю*. Причини

цього рандомізовані рішення SMT-розв’язувача та алежність від порядку визначень у коді, назв, структури тощо.

Часто виникають труднощі саме з певними типами тверджень:

- Нелінійна арифметика.
- Взаємодія між цілими числами і бітовими векторами. Варто уникати змішаних операцій, коли це можливо.
- Кванторні формули. Використання forall та exists веде до складності перевірки.

1. Аналіз варіативності

Для того аби перевірити декілька разів із різними випадковими зернами (random seeds) та згенерувати CSV-звіт із часом/ресурсами для кожної партії тверджень можна використовувати команду:

```
$ dafny measure-complexity file.dfy --log-format csv --iterations 10
```

А для аналізу отриманого звіту:

```
$ dafny-reportgenerator summarize-csv-results --max-resource-cv-pct 20 --max-resource-count 200000 file.csv
```

Якщо коефіцієнт варіації перевищує 20%, це ознака потенційної нестабільності.

2. Ізоляція доведень

Коли Dafny перевіряє метод або функцію, він автоматично формує assertion batch — тобто групу тверджень (assert, requires, ensures, інваріанти тощо), які треба довести в межах одного запиту до SMT-розв’язувача (Z3). Якщо тверджень багато або вони складні, розв’язувач може зависнути на довгі хвилини? тайм-аутитись або непередбачувано поводитись (наприклад, після незначної зміни коду перестати доводити те, що доводив раніше).

Щоб уникнути цьогоі не доводити все одразу, верифікацію розбивають на менші частини (батчі). Це дозволяє обмежити кількість тверджень для кожної частини, перевяріти твердження в окремому контексті та легше знаходити місце, де саме щось не доводиться або гальмує.

Dafny надає три спеціальні атрибути для контролю розбиття:

1. `{:isolate_assertions}` – найпростіший і найпотужніший. Додається до методу/функції/леми та автоматично ізолює кожне твердження (`assert`, `requires`, `ensures`, інваріанти...) в окремий батч:

```
const TWO_TO_THE_32:  int := 0x1_00000000
newtype uint32 = x: int | 0 <= x < TWO_TO_THE_32

method {:isolate_assertions} ProveSomeArithmetic(x: uint32) {
  assert forall y :: y * y != 115249; // 115249 is prime
  assert (x as bv32) as uint32 <= x;
  assert x < 65535 ==> x * 2 == x + x;
}
```

Dafny створить 3 блоки тверджень – по одному на кожне `assert`.

2. `{:split_here}` – ручне розбиття на місці. Ставиться на assert-і, щоб розбити доведення "до цього моменту" і "після":

```
method Example(x: int) {
  assert x > 0;
  assert {:split_here} x + 1 > 0;
  assert x * 2 > 0;
}
```

Dafny створить 2 блоки: один до `{:split_here}`, другий — після.

3. `{:focus}` – для фокусування на конкретному блоку. Ставиться на `assert` або `assume`, щоби фокусуватись на одній частині доведення і відкласти решту:

```
method Example(x: int) {
  assert {:focus} x >= 0;
  assert x + 1 > 0;
}
```

Dafny перевірить фокусований блок окремо.

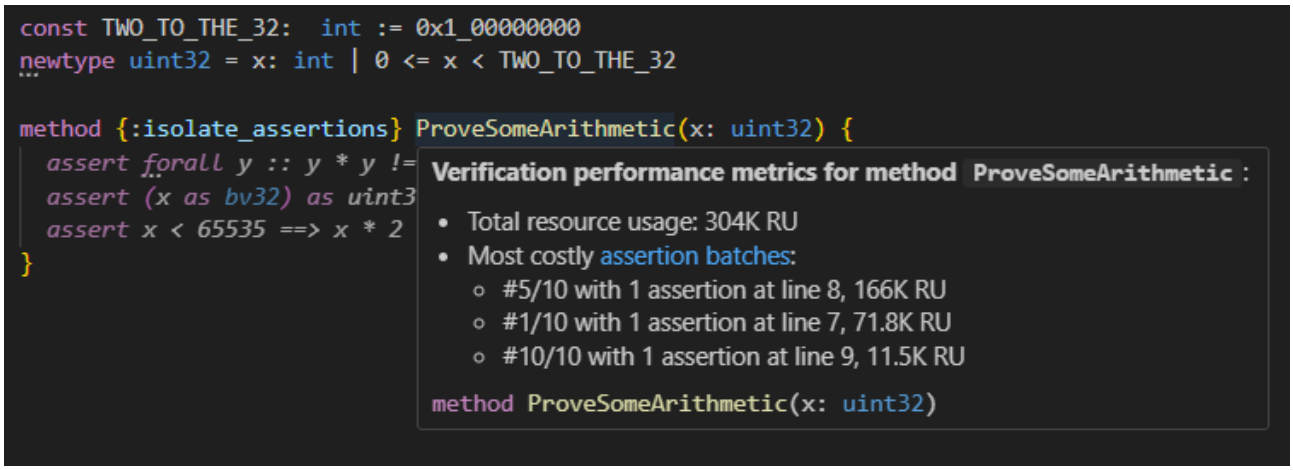
Аналіз складності: Resource Units (RU)

Після розбиття Dafny збирає метрики складності, наприклад:

- Duration: час виконання;
- Resource count: кількість ресурсів, витрачених Z3 (в умовних одиницях RU).

Це дозволяє побачити, який батч найдорожчий та зрозуміти, що саме гальмує верифікацію.

VS Code чи CLI може показувати це прямо в інтерфейсі (при наведенні на метод або assert):



3. Інші методи для покращення верифікації

Побудова доказів вручну

`--disable-nonlinear-arithmetic`

Перемикання арифметичного рушія Z3

`--boogie /proverOpt:O:smt.arith.solver=6`

Використання opaque-визначень

Оголошення функцій і предикатів як `opaque` означає, що їхні тіла приховані від верифікатора за замовчуванням, і доступні лише через явно оголошений контракт. Це зменшує обсяг інформації, яку Dafny аналізує при спробі доведення властивостей, знижуючи навантаження на SMT-солвер.

Якщо під час верифікації необхідно отримати доступ до прихованого визначення, використовується команда `reveal F();`, де `F` — ім'я функції або предиката. Такий підхід дозволяє зберігати вузьку область видимості кожного визначення й уникати перевантаження довільними деталями внутрішньої реалізації:

```
opaque predicate Valid()
opaque function ToString(): string
```

Функція `ToString()` вимагає, щоб об'єкт URL був валідним, але не потребує знання конкретної реалізації `Valid`, що дозволяє спростити перевірку.

Opaque контрактні умови

Предикати в `requires, ensures, assert` та `invariant` можуть також бути приховані від верифікатора шляхом додавання міток. Їх можна пізніше вибірково розкривати:

```
method OpaquePrecondition(x: int) returns (r: int)
  requires P: x > 0
  ensures r > 0
{
  r := x + 1;
  assert r > 0 by { reveal P; }
}
```

Контроль інлайнінгу рекурсивних функцій

За замовчуванням Dafny інлайнить тіла рекурсивних функцій до глибини 2 в `assert`-висловлюваннях і до глибини 1 в `assume`. Це може бути змінено за допомогою анотації `{:fuel n}`, що дозволяє явно контролювати глибину розгортання рекурсії.

Вимкнення припущень

За замовчуванням `assert` не лише перевіряє істинність твердження, але й додає його в контекст як припущення. Якщо це небажано, це можна вимкнути:

```
assert {:subsumption 0} P;
```

Це дозволяє уникнути накопичення непотрібних тверджень, що можуть ускладнити подальше доведення.

Уникнення кванторів

Хоча використання кванторів `(forall, exists)` дозволяє формулювати узагальнені твердження, вони значно ускладнюють роботу SMT-солвера. Краще доводити параметризовані версії тверджень:

```
lemma LemmaMulEquality(x: int, y: int, z: int)
  requires x == y
  ensures x * z == y * z {

}
```

Замість:

```
ensures forall x, y, z :: x == y ==> x * z == y * z
```

Такі леми легше доводити, і вони сприяють стабільнішій поведінці верифікатора.

Ручна інсталяція кванторів і використання тригерів

Тригери — це механізм, який використовується у SMT-, щоб керувати тим, коли та як кванторні твердження (з `forall` чи `exists`) застосовуються у доведеннях.

Коли ми пишемо кванторне твердження, наприклад з `forall:`

```
ensures forall x, y, z :: x == y ==> x * z == y * z
```

SMT-розв’язувач не може "перевірити" це одразу для всіх значень x, y, z. Замість цього він намагається застосовувати це правило до конкретних випадків, коли з’являються вирази, схожі на "тригери". У випадках, коли використання кванторів є необхідним, можна вручну задавати тригери за допомогою анотацій `{:trigger ...}` для керування тим, коли формула повинна інстанціюватися. Проте це слід вважати крайнім заходом — зазвичай Dafny самостійно вибирає адекватні тригери.

Декомпозиція визначень

Розбиття великих функцій, методів і предикатів на менші частини — це загальна рекомендація для покращення верифікації:

- Коротші функції легше перевіряти.
- Допоміжні леми спрощують контракт основної функції.
- Абстраговані визначення можуть бути повторно використані.

Ghost-функції довжиною понад 5 рядків часто вказують на надлишкову складність.

Підсумок.

Інструмент Dafny є потужним середовищем для формальної верифікації програм, що дозволяє розробникам гарантувати правильність коду ще до його виконання. Завдяки підтримці специфікацій, автоматичному доведенню тверджень та ефективній взаємодії зі SMT-розв'язувачем Z3, Dafny забезпечує високий рівень надійності. Особливо важливим є механізм ізоляції доведень , який допомагає локалізувати складні місця, покращує продуктивність перевірки та робить верифікацію масштабованою. Таким чином, Dafny є сучасним та практичним інструментом для створення коректного програмного забезпечення в критично важливих галузях.

Джерела.

1. Соннекс, В.; Дросопулу, С. «Перевірене програмування на Dafny: вступний курс з використання Dafny для написання програм з повністю верифікованими специфікаціями» [Електронний ресурс]. – Режим доступу: https://www.doc.ic.ac.uk/~scd/Dafny_Material/Lectures.pdf
2. «Початок роботи з Dafny: посібник» [Електронний ресурс]. – Режим доступу: https://dafny.org/dafny/OnlineTutorial/guide
3. «Розробка верифікованих програм із використанням Dafny» К. Растан М. Лейно. Дослідницький центр Microsoft. – Редмонд, Вашингтон, США, 2013. – Режим доступу: https://homepage.cs.uiowa.edu/~tinelli/classes/181/Fall15/Papers/Lein13.pdf.
4. Оптимізація верифікації Dafny [Електронний ресурс]. Документація Dafny. – Режим доступу: [https://dafny.org/latest/VerificationOptimization/] – Дата звернення: 03 травня 2025 р.