**Các vấn đề hiện đại của công nghệ thông tin**

**Nhóm 7 : Dafny**

Contents

[A. Thành viên 2](#_Toc493075005)

[B. Phân công công việc 2](#_Toc493075006)

[C. Dafny Tutorial 3](#_Toc493075007)

[I. Guide 3](#_Toc493075008)

[1. Introduction 3](#_Toc493075009)

[2. Methods 4](#_Toc493075010)

[3. Pre- and Postconditions 5](#_Toc493075011)

[4. Assertions 7](#_Toc493075012)

[5. Functions 7](#_Toc493075013)

[6. Loop Invariants // Bất biến trong vòng lặp 8](#_Toc493075014)

[7. Termination\_ Tính dừng 9](#_Toc493075015)

[8. Arrays 10](#_Toc493075016)

[9. Quantifiers (định lượng) 11](#_Toc493075017)

[10. Predicates 12](#_Toc493075018)

[11. Framing 13](#_Toc493075019)

[12. Binary Search 14](#_Toc493075020)

[13. Conclusion 15](#_Toc493075021)

[II. Dafny – Termination\_Tính dừng 16](#_Toc493075022)

[III. Dafny - Sets \_ các tập 19](#_Toc493075023)

[IV. Dafny - Sequences 20](#_Toc493075024)

[V. Dafny – Collections 24](#_Toc493075025)

[1. Sets 24](#_Toc493075026)

[2. Sequences 26](#_Toc493075027)

[3. Multisets 29](#_Toc493075028)

[4. Maps 30](#_Toc493075029)

[VI. Dafny – Lemmas (Bổ đề) 30](#_Toc493075030)

[1. Introduction 30](#_Toc493075031)

[2. Searching for Zero 31](#_Toc493075032)

[3. Lemmas (Bổ đề) 32](#_Toc493075033)

[4. Counting (Đếm) 34](#_Toc493075034)

[5. Proving the Distributive Property (Chứng minh phân phối thuộc tính) 35](#_Toc493075035)

[6. Induction (Cảm ứng) 37](#_Toc493075036)

[7. Paths In a Directed Graph (đường dẫn trong một biểu đồ hướng dẫn) 38](#_Toc493075037)

[VII. Modules 43](#_Toc493075038)

[1. Introduction 43](#_Toc493075039)

[2. Declaring New Modules\_ Module mới 43](#_Toc493075040)

[3. Importing and Exporting Modules 44](#_Toc493075041)

[4. Opening Modules 48](#_Toc493075042)

[5. Module Abstraction 49](#_Toc493075043)

[6. Module Ordering and Dependencies 50](#_Toc493075044)

[7. Name Resolution 51](#_Toc493075045)

1. **Thành viên**

Thành viên nhóm 7:

1. Hoàng Thanh Hằng
2. Lê Thị Xuân
3. Phạm Thị Thu Hường
4. Phùng Thị Lương
5. **Phân công công việc**

|  |  |  |
| --- | --- | --- |
| **Tuần** | **Thành Viên** | **Công Việc** |
| **1** | Hoàng Thanh Hằng | Tìm hiểu phần Guide |
| Lê Thị Xuân | Tìm hiểu Termination |
| Phạm Thị Thu Hường | Tìm hiểu Sequences |
| Phùng Thị Lương | Tìm hiểu sets |
| **2** | Hoàng Thanh Hằng | Tìm hiểu 6-7 của Modules |
| Lê Thị Xuân | Tìm hiểu 1-5 của Modules, thuyết trình |
| Phạm Thị Thu Hường | Tìm hiểu LemmasCCollections |
| Phùng Thị Lương | Tìm hiểu CollectionsC |
| **3** | Hoàng Thanh Hằng |  |
| Lê Thị Xuân |  |
| Phạm Thị Thu Hường |  |
| Phùng Thị Lương |  |
| **4** | Hoàng Thanh Hằng |  |
| Lê Thị Xuân |  |
| Phạm Thị Thu Hường |  |
| Phùng Thị Lương |  |
| **5** | Hoàng Thanh Hằng |  |
| Lê Thị Xuân |  |
| Phạm Thị Thu Hường |  |
| Phùng Thị Lương |  |
| **6** | Hoàng Thanh Hằng |  |
| Lê Thị Xuân |  |
| Phạm Thị Thu Hường |  |
| Phùng Thị Lương |  |

1. **Dafny Tutorial**
2. **Guide**
3. [Introduction](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Dafny là một ngôn ngữ được thiết kế giúp cho việc viết code dễ dàng hơn. Điều này có nghĩa là sẽ không còn bất kỳ lỗi run time nào và cũng đồng thời giúp cho lập trình viên đảm bảo chính xác những gì họ dự định làm. Để làm được điều này, Dafny dựa trên các chú thích cao cấp về lí do và chứng minh tính đúng đắn của code. Hiệu quả của một đoạn mã có thể được đưa ra một cách trừu tượng, sử dụng tự nhiên, biểu hiện cấp cao của hành vi mong muốn, đó là dễ dàng hơn và ít bị lỗi để viết. Dafny sau đó tạo ra một bằng chứng cho thấy mã phù hợp với các chú thích (giả sử chúng là chính xác nhất, là hiển nhiên). Dafny nâng cao gánh nặng cho việc viết mã lỗi miễn phí bằng cách viết chú thích không có lỗi. Điều này thường dễ hơn viết code, bởi vì các chú thích ngắn hơn và trực tiếp hơn. Ví dụ, phần mô tả sau đây trong Dafny nói rằng mọi thành phần của mảng đều thỏa mãn một tính chất:

forall k: int :: 0 <= k < a.Length ==> 0 < a[k]

Đoạn code nói rằng đối với tất cả các số nguyên k là chỉ số vào mảng, giá trị tại chỉ mục đó lớn hơn không. Bằng cách viết các chú thích này, người ta tin tưởng rằng mã là chính xác. Hơn nữa, chính việc viết các chú thích có thể giúp người ta hiểu mã này đang làm gì ở mức sâu hơn.

Ngoài chứng minh tương ứng với chú thích do người dùng cung cấp, Dafny chứng minh rằng không có lỗi thời gian chạy, chẳng hạn như chỉ số ngoài phạm vi, dereferences null, chia cho số không, …

Dafny cũng chứng minh việc chấm dứt mã, ngoại trừ các vòng lặp được chỉ định đặc biệt.

Hãy bắt đầu viết một số chương trình Dafny

1. [Methods](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Dafny giống như một ngôn ngữ lập trình bắt buộc điển hình. Có các phương pháp, biến, kiểu, vòng lặp, nếu phát biểu, mảng, số nguyên, và nhiều hơn nữa. Một trong những đơn vị cơ bản của bất kỳ chương trình Dafny nào là Methods. Method là một đoạn mã bắt buộc để thực thi. Trong các ngôn ngữ khác, chúng có thể được gọi là thủ tục, hoặc các chức năng, nhưng trong Dafny, thuật ngữ "chức năng" được dành riêng cho một khái niệm khác mà chúng ta sẽ trình bày sau. Phương pháp được khai báo theo cách sau:

method Abs(x: int) returns (y: int)  
{  
 y:= 5  
}

Điều này tuyên bố một phương thức có tên gọi là "Abs" lấy một tham số số nguyên, được gọi là "x", và trả về một số nguyên duy nhất, được gọi là "y". Lưu ý rằng các loại được yêu cầu cho mỗi tham số và giá trị trả về và làm theo từng tên sau dấu hai chấm (:). Ngoài ra, các giá trị trả lại được đặt tên, và có thể có nhiều giá trị trả lại, như trong dưới đây:

method MultipleReturns(x: int, y: int) returns (more: int, less: int)  
{  
 more := x + y;  
 less := x - y;  
 // comments: are not strictly necessary.  
}

Dafny không sử dụng "=", mà là ": =". (Thực ra, như Dafny dùng "==" cho sự bình đẳng, không sử dụng một ký hiệu bằng một ký tự đại diện cho Dafny). Các câu đơn giản phải được theo sau bởi dấu chấm phẩy và khoảng trắng và các chú thích (// và / \*\* /) bị bỏ qua. Để trả lại một giá trị từ một phương thức, giá trị được gán cho một trong những giá trị trả về có tên trước khi một câu lệnh trả về. Trong thực tế, các giá trị trả lại hoạt động rất giống với các biến cục bộ, và có thể được gán cho nhiều hơn một lần. Tuy nhiên, các tham số đầu vào chỉ được đọc. Câu lệnh return được sử dụng khi người ta muốn quay trở lại trước khi kết thúc phần thân của phương thức. Câu lệnh return có thể chỉ là từ khóa trả về (trong đó giá trị hiện tại của các tham số ra được sử dụng), hoặc chúng có thể lấy một danh sách các giá trị để trả về. Ngoài ra còn có các câu lệnh phức hợp, chẳng hạn như câu lệnh if. Nếu câu lệnh yêu cầu dấu ngoặc quanh điều kiện boolean và hành động như mong muốn:

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

Một lưu ý là họ luôn cần căc dấu ngoặc nhọn {} cho các khối lệnh ở các nhánh, ngay cả khi nhánh chỉ chứa một lệnh duy nhất . Ở đây câu lệnh if kiểm tra xem x có nhỏ hơn không, sử dụng cú pháp của toán tử so khớp quen thuộc và trả về giá trị tuyệt đối nếu thích hợp. (Các toán tử so sánh khác là <=,>, <=,! = và ==, có ý nghĩa như trong các ngôn ngữ lập trình)

1. [Pre- and Postconditions](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Sức mạnh thực sự của Đại Pháp bắt nguồn từ khả năng chú thích các phương pháp này để xác định hành vi của họ. Ví dụ, một thuộc tính mà chúng ta quan sát với phương pháp Abs là kết quả luôn luôn lớn hơn hoặc bằng không, bất kể đầu vào. Chúng ta có thể đưa quan sát này vào nhận xét, nhưng sau đó chúng ta sẽ không có cách nào để biết liệu phương pháp có thực sự có thuộc tính này hay không. Hơn nữa, nếu có ai đó đến và thay đổi phương pháp, chúng tôi sẽ không được bảo đảm rằng bình luận đã được thay đổi để phù hợp. Với chú thích, chúng tôi có thể có Dafny chứng minh rằng tài sản mà chúng tôi tuyên bố về phương pháp là đúng. Có một số cách để đưa ra chú thích, nhưng một số phổ biến nhất, và cơ bản nhất là các phương pháp trước và sau hậu điều kiện.

tính của phương thức Abs, mà kết quả luôn luôn không âm, là một ví dụ của một hậu điều kiện: đó là cái gì đó đúng sau khi phương pháp trả về. Các hậu điều kiện, khai báo với từ khóa đảm bảo, được đưa ra như là một phần của khai báo của phương pháp, sau các giá trị trả lại (nếu có) và trước cơ thể phương pháp. Từ khóa được theo sau bởi biểu thức boolean. Giống như điều kiện và trong hầu hết các điều kiện, điều kiện hậu điều kiện luôn luôn là một biểu thức boolean: cái gì đó có thể đúng hoặc sai. Trong trường hợp của phương pháp Abs, một hậu điều kiện hợp lý là như sau:

method Abs(x: int) returns (y: int)  
 ensures 0 <= y  
{  
 ...  
}

Bạn có thể thấy tại sao giá trị trả lại được đặt tên. Điều này làm cho chúng dễ dàng tham khảo trong hậu điều kiện của một phương pháp. Khi biểu thức là đúng, chúng ta nói rằng các hậu điều kiện được nắm giữ. Điều kiện hậu điều kiện phải được giữ cho mỗi lần gọi hàm, và cho mỗi điểm trả về có thể xảy ra (bao gồm cả điểm tiềm ẩn ở cuối của hàm chức năng). Trong trường hợp này, thuộc tính duy nhất mà chúng ta đang diễn đạt là giá trị trả về luôn ít nhất bằng không.

* **Pre :** requires điều kiện (Điều kiện trước)
* **Post** :ensures điều kiện (Điều kiện sau)

Chúng ta có thể kết hợp hai điều kiện cùng với toán tử boolean và operator (&&), hoặc chúng ta có thể viết nhiều đảm bảo các thông số kỹ thuật

method MultipleReturns(x: int, y: int) returns (more: int, less: int)   
 ensures less < x  
 ensures x < more

* cách viết khác là : ensures less < x && x < more
* Ta có ví dụ :   
  {  
   more := x + y;  
   less := x - y;  
  }
* Cái này bị sai vì có thể sau hàm less không nhỏ hơn X cũng như more k lớn hơn x (VD y=0)
* để đúng , thêm preconditions: requires y>0 trước ensures less < x  
   là được

Giống như điều kiện hậu, nhiều điều kiện tiên quyết có thể được viết bằng toán tử boolean và (&&), hoặc bằng nhiều từ khóa. Với việc bổ sung các điều kiện này, Dafny bây giờ xác minh mã là chính xác, bởi vì giả định này là tất cả những gì cần thiết để đảm bảo mã trong cơ thể phương pháp là chính xác.

Không phải code nào cũng cần tiền điều kiện, nhưng nó cần có một chú thích , khẳng định để xác minh, như trong ví dụ sau

method Max(a: int, b:int) returns (c: int)

ensures a<=c && b<=c

{ if a>b

{return a;}

else

{return b;}

}

1. [Assertions](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Không giống tiền và hậu điều kiện, một khẳng định được đặt ở đâu đó ở giữa một phương pháp. Giống như hai chú thích trước đó, khẳng định có từ khoá, khẳng định, tiếp theo là biểu thức boolean và dấu chấm phẩy chấm dứt câu lệnh đơn giản. Một khẳng định nói rằng một biểu thức cụ thể luôn luôn giữ khi kiểm soát đạt đến phần đó của code. Ví dụ đoạn code sau

method Testing()  
{  
 assert 2 < 3;  
}

=> assert điều kiện để kiểm tra xem điều kiện test đã đúng chưa

VD : asset 2< 3 => đúng

Assert 2=3 => báo bị lỗi

**=> Theo kinh nghiệm cái này rất hữu ích , và nên có cho mỗi bài test**

1. [Functions](http://rise4fun.com/Dafny/tutorialcontent/Guide)

* Khai báo biến :

var x, y, z: bool := 1, 2, true;

var x: nat := 5; // nat là kiểu số nguyên không âm

* VD :

function abs(x: int): int  
{

if x < 0 then -x else x

}

=> hàm trả về trị tuyệt đối của x và dữ liệu trả về là kiểu int

Rõ ràng, điều kiện phải là một biểu thức boolean, và hai nhánh phải có cùng một loại. Bạn có thể tự hỏi tại sao mọi người sẽ bận tâm với các chức năng, nếu chúng bị hạn chế so với các phương pháp số kỹ thuật.

* **Gọi hàm :**

function max(a: int, b: int): int

{

if a>b then a else b

}

method Testing() {

assert max(1,2)== 2;

}

1. [Loop Invariants](http://rise4fun.com/Dafny/tutorialcontent/Guide) (Bất biến trong vòng lặp)

Khi mà chạy một vòng lặp, Dafny sẽ không biết rõ số lần vòng lặp chạy, mà Dafny cần phải xem xét tất cả các con đường thông qua 1 chường trình. Do đó, khi làm việc với một vòng lặp, ta cần cung cấp một đại lượng (điều kiện )bất biến (Không thay đổi trong toàn bộ quá trình lặp)

Từ khóa Khai báo invariant điều kiện

* VD

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

Khi bạn chỉ định một biến bất biến, Dafny chứng minh hai điều: invariant giữ khi nhập vòng lặp, và nó được bảo vệ bởi vòng lặp. Bằng cách bảo toàn, chúng ta có nghĩa là giả định rằng biến bất biến giữ ở đầu vòng lặp, chúng ta phải chứng minh rằng việc thực hiện cơ thể vòng lặp một lần làm cho bất biến giữ lại. Dafny chỉ có thể biết được khi phân tích cơ thể vòng lặp mà các biến bất biến nói, ngoài sự bảo vệ vòng lặp (điều kiện vòng lặp). Giống như Dafny sẽ không khám phá được các thuộc tính của một phương pháp riêng, nó sẽ không biết bất kỳ thuộc tính cơ bản nhất của một vòng lặp được bảo toàn trừ khi nó được nói qua một biến bất biến.

Một trong những vấn đề khi sử dụng các biến thể bất biến là dễ dàng quên rằng vòng lặp thực hiện tiến bộ, nghĩa là làm việc ở từng bước. Ví dụ, chúng ta có thể đã bỏ qua toàn bộ cơ thể của vòng lặp trong chương trình trước đó. Các biến bất biến sẽ đúng, bởi vì chúng vẫn còn đúng khi nhập vòng lặp, và vì vòng lặp không thay đổi bất cứ điều gì, chúng sẽ được bảo vệ bởi vòng lặp. Nhưng bước quan trọng từ vòng lặp đến hậu điều kiện sẽ không giữ. Chúng ta biết rằng nếu chúng ta thoát khỏi vòng lặp, thì chúng ta có thể giả định sự phủ định và các biến bất biến, nhưng điều này không nói gì về điều gì sẽ xảy ra nếu chúng ta không bao giờ thoát khỏi vòng lặp. Vì vậy, cần đảm bảo rằng vòng lặp kết thúc ở một thời điểm nào đó, điều này cho phép đảm bảo đúng đắn hơn nhiều

1. [Termination](http://rise4fun.com/Dafny/tutorialcontent/Guide)\_ Tính dừng

Dafny chứng minh tính dừng của code, tức là vòng lặp sẽ ko thực hiện mãi mãi, thông qua việc giảm giá trị của biến t (biến chú thích ).

Dafny chứng minh được tính dừng trong 2 trường hợp:

* Vòng lặp
* Đệ quy

Đối với biến chú thích t sẽ giảm sau mỗi vòng lặp hoặc đệ quy.

Đối với việc xác định các biểu thức giảm thì cần xác minh:

* Thực sự giảm giá trị
* Có giá trị nằm trong khoảng giới hạn nào đó.
* Trong nhiều trường hợp, 1 giá trị (số tự nhiên và số nguyên) giảm đi nhưng 1 số thứ khác vẫn sử dụng tốt. Đối với số nguyên ràng buộc mặc định được khởi tạo là 0.
* VD: Sử dụng hợp lí biểu thức giảm:

while 0 < i  
 invariant 0 <= i  
 decreases i  
 {  
 i := i - 1;  
 }

1. [Arrays](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Tất cả những gì chúng ta đã xem xét là tốt cho các chức năng đơn giản và các bài tập toán học ít, nhưng nó thực sự không hữu ích cho các chương trình thực sự. Cho đến nay chúng ta chỉ xem xét một số ít các giá trị tại một thời điểm trong các biến địa phương. Bây giờ chúng ta chuyển sự chú ý của chúng ta sang mảng dữ liệu. Mảng được xây dựng trong một phần của ngôn ngữ, với loại riêng của họ, mảng <T>, nơi T là một loại khác. Bây giờ chúng ta chỉ xem xét mảng các số nguyên, mảng <int>. Mảng có thể là null, và có một lĩnh vực xây dựng chiều dài, a.Length. Truy cập phần tử sử dụng cú pháp chuẩn và được lập chỉ mục từ số không, do đó, một [3] được trước bởi 3 phần tử a [0], a [1], và a [2] theo thứ tự đó. Tất cả các truy cập mảng phải được chứng minh là nằm trong giới hạn, là một phần của đảm bảo an toàn do lỗi của Dafny. Bởi vì kiểm tra giới hạn được chứng minh tại thời điểm xác minh, không cần phải kiểm tra thời gian chạy. Để tạo một mảng mới, nó phải được phân bổ bằng từ khoá mới, nhưng bây giờ chúng ta sẽ chỉ làm việc với các phương thức lấy một mảng được phân bổ trước đó làm đối số.

a: array<int> : mảng a có kiểu int ,chiều dài mảng có thể = null

Một trong những điều cơ bản nhất mà chúng ta có thể muốn làm với một mảng là tìm kiếm thông qua nó cho một khóa cụ thể và trả về chỉ mục của một nơi mà chúng ta có thể tìm thấy khóa nếu nó tồn tại.

* Ví dụ về tìm kiếm key trong mảng cho trước :

method Find(a: array<int>, key: int) returns (index: int)

requires a != null

ensures 0 <= index ==> index < a.Length && a[index] == key

ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key

{

index := 0;

while index < a.Length

invariant 0 <= index <= a.Length

invariant forall k :: 0 <= k < index ==> a[k] != key

{

if a[index] == key { return; }

index := index + 1;

}

index := -1;

}

1. [Quantifiers](http://rise4fun.com/Dafny/tutorialcontent/Guide) (định lượng)

Một định lượng trong Dafny thường có dạng của một biểu thức forall, còn được gọi là định lượng phổ. Như tên của nó cho thấy, biểu thức này là đúng nếu một số thuộc tính đúng cho tất cả các phần tử của một số tập. Bây giờ, chúng ta sẽ xem xét tập các số nguyên. Một ví dụ về định lượng phổ, trong một khẳng định, được đưa ra dưới đây:

assert forall k :: k < k + 1;

Định lượng sẽ giới thiệu một tên tạm thời cho mỗi phần tử của tập mà nó đang xem xét. Điều này được gọi là biến ràng buộc, trong trường hợp này là k. Biến ràng buộc có một loại, mà hầu như luôn luôn được suy luận hơn là được đưa ra một cách rõ ràng và thường int anyway. . Một cặp dấu hai chấm (: :) tách biệt ràng buộc và loại tùy chọn của nó khỏi thuộc tính định lượng (phải là kiểu bool). Trong trường hợp này, thuộc tính là thêm vào một số nguyên nào đó làm cho một số nguyên lớn hơn. Dafny có thể tự động xác minh tính năng này. Nói chung nó không phải là rất hữu ích để định lượng qua bộ vô hạn, chẳng hạn như tất cả các số nguyên. Thay vào đó, định lượng thường được sử dụng để định lượng trên tất cả các phần tử trong một mảng hoặc cấu trúc dữ liệu. Chúng ta làm điều này cho mảng bằng cách sử dụng toán tử ngụ ý để làm cho các thuộc tính định lượng đúng đắn cho các giá trị không phải là các chỉ số:

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

Điều này nói rằng k thực sự là một chỉ mục hợp lệ của mảng. Dafny thực tế có thể sử không chỉ để chứng minh rằng mảng được truy cập một cách an toàn mà còn giảm tập các số nguyên mà nó phải xem xét chỉ là các chỉ số có trong các mảng.

Với một định lượng sau, việc chứng minh một phần tử không ở trong mảng là đơn giản :

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

**BT: Tìm phần tử lớn nhất trong mảng (Chưa làm được)**

Một tìm kiếm tuyến tính không phải là rất hiệu quả, đặc biệt là khi nhiều truy vấn được làm bằng cùng một dữ liệu. Nếu mảng được sắp xếp, thì chúng ta có thể sử dụng thủ tục tìm kiếm nhị phân rất hiệu quả để tìm khoá. Nhưng để chúng tôi có thể chứng minh việc thực hiện chính xác của chúng tôi, chúng tôi cần một số cách để yêu cầu rằng mảng đầu vào thực sự được sắp xếp. Chúng ta có thể làm điều này trực tiếp với một bộ định lượng bên trong một mệnh đề đòi hỏi của phương pháp của chúng ta, nhưng một cách mô đun hơn để diễn tả điều này là thông qua một vị từ.

1. [Predicates](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Một predicate là một hàm trả về một boolean. Đó là một ý tưởng đơn giản nhưng mạnh mẽ xảy ra trong các chương trình của Dafny. Ví dụ, chúng ta định nghĩa vị từ được sắp xếp trên các mảng số nguyên như một hàm lấy mảng làm đối số, và trả về true nếu và chỉ nếu mảng đó được sắp xếp theo thứ tự ngày càng tăng. Việc sử dụng các vị từ làm cho mã của chúng ta ngắn hơn, vì chúng ta không cần phải viết ra quá dài. Nó cũng có thể làm cho mã của chúng ta dễ dàng hơn để đọc bằng cách cho một tài sản chung một cái tên.

Có một số cách chúng ta có thể viết các vị từ được sắp xếp, nhưng cách dễ nhất là sử dụng một định lượng trên các chỉ số của mảng. Chúng ta có thể viết một định lượng thể hiện thuộc tính, "nếu x là trước y trong mảng, thì x <= y", như là một định lượng trên hai biến ràng buộc:

forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]

Ở đây chúng ta có hai biến ràng buộc, j và k, cả hai số nguyên. Sự so sánh giữa hai số đảm bảo rằng cả 2 là chỉ số hợp lệ vào mảng, và j là trước k. S. Định lượng chỉ là một loại biểu thức có giá trị boolean ở Dafny, vì vậy chúng ta có thể viết các vị từ được sắp xếp như sau:

* **Ví dụ 1(Lỗi- sẽ được sửa ở Framing)**

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

Lưu ý rằng không có kiểu trả về, bởi vì các vị từ luôn trả về một boolean.Dafny từ chối mã này như đã cho, tuyên bố rằng vị từ không thể đọc mảng a . Khắc phục vấn đề này yêu cầu chú thích khác, đọc chú thích.

1. [Framing](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Vị từ được sắp xếp không thể truy cập mảng vì mảng không được bao gồm trong Frame của hàm. Frame đọc của một chức năng (hoặc vị từ) là tất cả các vị trí bộ nhớ mà chức năng được phép đọc. Lý do chúng ta có thể hạn chế một chức năng có thể đọc hay không là khi chúng ta ghi vào bộ nhớ, chúng ta có thể chắc chắn rằng các hàm không đọc phần bộ nhớ đó có cùng giá trị mà chúng đã làm trước đó. Ví dụ, chúng ta có thể có hai mảng, một trong số chúng, ta biết được sắp xếp. Nếu chúng ta không đặt chú thích cho thuộc tính được sắp xếp, thì khi chúng ta sửa đổi mảng chưa được phân loại, chúng ta không thể xác định liệu mảng khác đã ngừng sắp xếp hay không. Mặc dù chúng ta có thể cung cấp các biến đổi bất biến để bảo toàn nó trong trường hợp này, nó thậm chí còn phức tạp hơn khi thao tác các cấu trúc dữ liệu. Trong trường hợp này, Frame là rất cần thiết để làm cho quá trình xác minh khả thi.

Lệnh cho phép đọc : reads

Chữa ví dụ 1 ở mục 10 :

predicate sorted(a: array<int>)  
 requires a != null

reads a  
{  
 forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]  
}

vậy sẽ không bị lỗi không đọc được mảng a nữa

Chú thích ghi chú đọc mảng a không phải là một biểu thức boolean, giống như các chú thích khác chúng ta đã thấy, và có thể xuất hiện ở bất cứ đâu cùng với các điều kiện trước và sau hậu điều kiện. Thay vì thuộc tính cần phải đúng, nó chỉ định một tập hợp các vị trí bộ nhớ mà hàm được phép truy cập. Tên của một mảng, như trong ví dụ trên, là viết tắt của tất cả các phần tử của mảng đó. Một cũng có thể chỉ định các trường đối tượng và các đối tượng, nhưng chúng ta sẽ không quan tâm đến những chủ đề đó ở đây. Dafny sẽ kiểm tra xem bạn không đọc bất kỳ vị trí bộ nhớ nào không được nêu trong khung đọc. Điều này có nghĩa là các cuộc gọi hàm trong một hàm phải có khung đọc. Một điều cần lưu ý là các tham số cho các chức năng không phải là vị trí bộ nhớ không cần phải được khai báo.

Khung cũng ảnh hưởng đến các Methods. Như bạn có thể đoán, chúng không bắt buộc phải liệt kê những thứ họ đọc, vì chúng ta đã viết một phương thức truy xuất một mảng không có chú thích. Các phương pháp được phép đọc bất kỳ bộ nhớ nào mà họ thích, nhưng chúng được yêu cầu liệt kê những phần bộ nhớ mà chúng sửa đổi, với một sửa đổi chú thích. Họ gần như giống hệt với người họ hàng của họ đọc, ngoại trừ họ nói những gì có thể được thay đổi, chứ không phải là những gì giá trị của các chức năng phụ thuộc vào. Kết hợp với các lần đọc, các hạn chế sửa đổi cho phép Dafny chứng minh các thuộc tính của mã mà nếu không sẽ rất khó khăn hoặc không thể. Đọc và sửa đổi là một trong những công cụ cho phép Dafny có thể làm việc theo một phương thức cùng một lúc bởi vì họ hạn chế những thay đổi tùy ý về bộ nhớ đối với một điều mà Dafny có thể giải thích.

Lưu ý rằng khung chỉ áp dụng cho heap, hoặc bộ nhớ truy cập thông qua tài liệu tham khảo. Các biến địa phương không được lưu trữ trên đống, vì vậy chúng không thể được đề cập trong chú thích. Cũng lưu ý rằng các loại như bộ, chuỗi và đa lớp là các kiểu giá trị, và được đối xử như các số nguyên hoặc biến cục bộ. Mảng và các đối tượng là các loại tài liệu tham khảo, và chúng được lưu trữ trên heap (mặc dù như thường thì có một sự phân biệt tinh tế giữa bản thân tham khảo và giá trị mà nó trỏ đến.)

1. [Binary Search](http://rise4fun.com/Dafny/tutorialcontent/Guide)

Các vị từ thường được sử dụng để làm cho các chú thích khác rõ ràng hơn:

method BinarySearch(a: array<int>, key: int) returns (index: int)  
 requires a != null && sorted(a)  
 ensures ...  
{  
 ...  
}

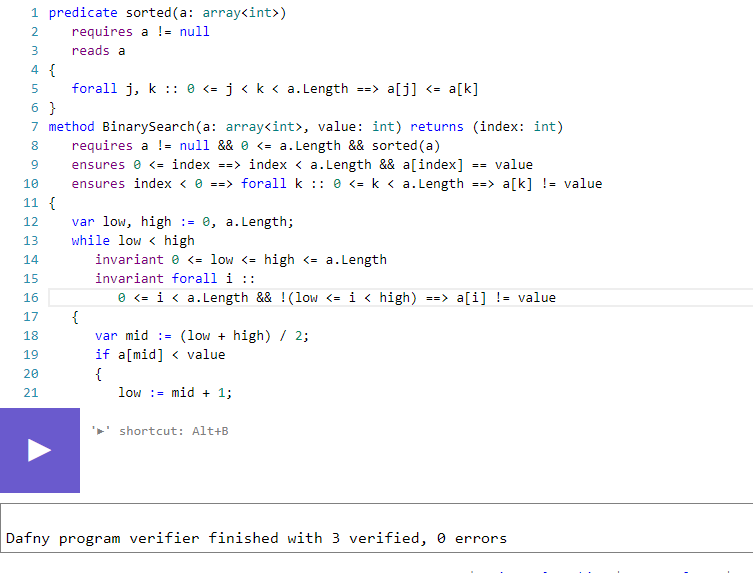
Ta có hậu điều kiện tương tự như khi làm cho tìm kiếm tuyến tính. Nhưng khác ở chỗ mảng ban đầu đã được sắp xếp. Dafny có thể mở gói function, bên trong method nó cũng biết.Ta có thể sử dụng nó để chứng minh tính chính xác của việc tìm kiếm. Phần thân của method được đưa ra như sau:

var low, high := 0, a.Length;  
while low < high  
 invariant 0 <= low <= high <= a.Length  
 invariant forall i ::  
 0 <= i < a.Length && !(low <= i < high) ==> a[i] != value  
 {  
 var mid := (low + high) / 2;  
 if a[mid] < value  
 {  
 low := mid + 1;  
 }  
 else if value < a[mid]  
 {  
 high := mid;  
 }  
 else  
 {  
 return mid;  
 }  
 }  
 return -1;

* Đầu tiên khai báo phạm vì tìm kiếm, các bất biến đầu tiên cho thấy phạm vi nằm trong mảng.
* Thứ 2 keyword sẽ không nằm ngoài phạm vi trên. Trong 2 nhánh (if của i) nhận thấy giá trị ở giữa không phải keyword, tiến hành di chuyển phạm vi để loại trừ nó và xác định bên phù hợp.Chúng ta cần bổ sung thêm một điểm khi di chuyển đầu cuối dưới của dải bởi vì nó bao gồm bên thấp.
* Nếu không +1 thì vọng lặp có thể không dừng. Để thoát khỏi điều này thì cần có kiểm tra sau 1 vòng lặp để xác định key nằm ở 1 chỉ số còn lại. Trong công thức trên thì do low==high nên vòng lặp thoát ra, có nghĩa là không có phần tử nào trong phạm vi tìm kiếm, không tìm thấy key. Điều này có thể được suy luận từ bất biến vòng lặp:

invariant forall i ::  
 0 <= i < a.Length && !(low <= i < high) ==> a[i] != value

* Khi low==high phủ nhận phần 1 . Bất biến nói rằng tất cả các yếu tố trong mảng không phải key và hậu điều kiện thứ 2 nắm giữ.



1. Conclusion

Vậy là chúng ta đã xem qua một lượt những gì mà Dafny có thể làm. Tuy nhiên, để thực sự tận dụng được sức mạnh của Dafny, bạn cần phải tiến sâu vào các chủ đề nâng cao: đối tượng, chuỗi và tập hợp, cấu trúc dữ liệu, lemmas, v.v ... Bây giờ bạn đã quen thuộc với những điều cơ bản của Đại Pháp, bạn có thể đọc các hướng dẫn về mỗi chủ đề này khi bạn giải trí. Mỗi hướng dẫn được thiết kế để có một hướng dẫn tương đối gọn để chủ đề của nó, mặc dù một số có lợi từ việc đọc những người khác trước. Các ví dụ cũng là một nơi tốt để tìm kiếm các chương trình mẫu Dafny. Cuối cùng, tham khảo có chứa các chi tiết gritty của Dafny cú pháp và ngữ nghĩa, cho khi bạn chỉ cần biết những gì các nhà khai thác tập hợp rời (it's !!, cho những người quan tâm).

Ngay cả khi bạn không sử dụng Dafny thường xuyên, ý tưởng viết ra chính xác mã nó là gì, và sử dụng mã này để chứng minh đúng mã là một kỹ năng hữu ích. Các biến thể bất biến, điều kiện trước và sau, và chú thích rất hữu ích trong gỡ lỗi mã, và cũng như là tài liệu cho các nhà phát triển trong tương lai. Khi sửa đổi hoặc thêm vào một codebase, họ xác nhận rằng sự đảm bảo của mã hiện tại không bị hỏng. Họ cũng đảm bảo rằng API được sử dụng đúng cách, bằng cách chính thức hóa hành vi và các yêu cầu và thực thi việc sử dụng đúng. Lý do từ bất biến, xem xét trước và sau hậu điều kiện, và bằng văn bản khẳng định để kiểm tra các giả định là tất cả các kỹ năng khoa học máy tính nói chung sẽ mang lại lợi ích cho bạn cho dù bạn làm việc ở ngôn ngữ nào.

1. **Dafny – Termination\_Tính dừng**

Dafny chứng minh tất cả chương trình đều có tính dừng.

Tính dừng được chứng mình trong 2 trường hợp:

* Vòng lặp
* Đệ quy

Kĩ thuật áp dụng: Giảm biến t (biến chú thích cho tính dừng của chương trình). Biến t là biến được quy định 1 giá trị và sẽ có giá trị nhỏ mỗi lần vòng lặp đi qua hoặc mỗi lần đệ quy. Đồng thời biến t có giá trị giới hạn để nó không giảm mãi. Vì tính dừng của biến t mà t chứng minh được chương trình có tính dừng. Dẫn đến để chứng minh tính dừng của chương trình, Dafny chứng minh biến t giảm sau mỗi lần lặp

Biến t: có nhiều loại giá trị thường sử dụng là các số nguyên, ràng buộc mặc định của số nguyên là 0, và việc chứng minh tính giảm của số nguyên khá dễ dàng.

* VD. Vòng lặp:

while i < n  
 invariant 0 <= i <= n  
{  
 // do something interesting  
 i := i + 1;  
}

Vòng lặp này có chấm dứt hay không?

Vì đây là 1 vòng lặp thông thường, Dafny có 1 quy tắc đặc biệt để đoán biến dừng. Dafny không thấy rõ biến t vì vậy nó đã đoán.

Nó thấy rằng điều kiện vòng lặp là sự so sánh giá trị hình thức A<B nó đoán được:

Giảm bớt B-A hay trong trường hợp này là n-i

Nếu thêm các chú thích này vào trong vòng lặp thì Dafny để tiếp tục kiểm chứng, đến khi t được giới hạn về 0.

Chúng ta có thể viết :

var i, n := 0, 11;  
while i < n  
 decreases n - i  
{  
 // do something interesting  
 i := i + 5;  
}

Ở vòng lặp cuối i=15 do đó giá trị dừng là -4. Do vòng lặp này là sai nên vòng lặp sẽ không thực hiện lại. Lưu ý phải bỏ bất biến vòng lặp như vậy i mới có thể vượt qua n.

Tính dừng của toàn bộ chương trình, không chỉ vòng lặp.

Kĩ thuật sử dụng:

* Tương tự trên
* Nó phân tích các hàm và phương pháp gọi nhau để tìm đệ quy.

VD hàm đệ quy:

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

Dafny chỉ ra rằng đối với hầu hết các chức năng mà là đệ quy thì chỉ cần gọi mình vơi giá trị nhỏ hơn của các thông số vì vậy mặc định đoán được thông số giảm. Có thể bổ sung cho việc giảm chú thích: decreases n để khai báo hàm.

* Đôi khi nó lại có lợi để có vòng lặp không dừng hoặc tính dừng không rõ.

VD: trong method sau:

method hail(N: nat)  
 decreases \*  
{  
 var n := N;  
 while 1 < n  
 decreases \*  
 {   
 n := if n % 2 == 0 then n / 2 else n \* 3 + 1;  
 }  
}

Chương trình này chấm dứt khi và chỉ khi các phỏng đoán Collatz là sự thật, song nó là vấn đề mở trong toán học nên Dafny không thực sự có thể chứng minh tính dừng của chương trình.

* Có thể mã hóa 1 cái gì đó giống 1 bộ xử lí dòng mà được thiết kế để chạy mãi mãi, như vậy Dafny sẽ cung cấp 1 "out" để chỉ thị cho Dafny không nên cố gắng chứng minh nữa. Nó được nêu ra ở method hail ở trên.
* 1 method có chứa 1 vòng lặp được đánh dấu bằng chính nó phải được đánh dấu bằng decreases \*

Dafny có thể sử dụng các giá trị khác số nguyên như biện pháp chấm dứt. Khi một dãy được chỉ định Dafny tự động sử dụng chiều dài như 1 biện pháp để chấm dứt.

Các tập hợp được coi là nhỏ hơn nếu một tập con là một tập hợp con của tập kia, do đó, mỗi tập phải được chứa trong tập trước.

Với các tập hợp, tập hợp trống rỗng nhỏ như bạn có thể đi và trình tự có độ dài số tự nhiên, nên cả 2 có giới hạn thấp hơn.

**VD:** Việc thực hiện hàm Ackermann sau đây sử dụng một cặp số nguyên để chứng minh chấm dứt:

function Ack(m: nat, n: nat): nat  
 decreases m, n  
{  
 if m == 0 then n + 1  
 else if n == 0 then Ack(m - 1, 1)  
 else Ack(m - 1, Ack(m, n - 1))  
}

Các mệnh đề giảm dần được viết ra rõ ràng, Dafny có thể dự đoán được nó. Một tuple sử dụng so sánh kích thước của các giá trị thành phần để xác định xem các biện pháp đã bị thu hẹp.

Nếu phần tử đầu tiên, trong trường hợp này m, nhỏ hơn,không có vấn đề gì xảy ra với các giá trị khác. Họ có thể tăng, giảm, hoặc ở lại như nhau. Phần tử thứ hai chỉ được xem xét nếu phần tử đầu tiên không thay đổi. Sau đó, giá trị thứ hai cần giảm. Nếu không, thì phần tử thứ ba phải giảm, vv Cuối cùng, một trong các yếu tố phải giảm. Quá thời điểm đó, và các yếu tố khác lại được tự do để tăng hoặc làm bất cứ điều gì họ thích.

Trong chức năng Ack, có ba cuộc gọi đệ quy.

* Trong lần đầu tiên, m trở thành một nhỏ hơn, nhưng n tăng lên. Điều này là tốt vì n đến sau khi m trong tập này.
* Trong cuộc gọi thứ hai, m giảm, do đó, đối số thứ hai được cho phép là bất kỳ giá trị nào (điều đó là tốt, bởi vì Dafny không thực sự chứng minh bất cứ điều gì về kết quả của cuộc gọi đệ quy thứ ba). Người Dafny cần phải chứng minh rằng cuộc gọi thứ ba tuân thủ các biện pháp chấm dứt.
* Đối với cuộc gọi này, m vẫn giữ nguyên, nhưng n giảm, vì vậy tổng thể giảm cũng tốt.

Tính dừng áp dụng cho nhiều đôi bên cùng có đẹ quy hàm/ method.

* VD: cặp chẵn lẻ đệ quy được xác định:

predicate even(n: nat)  
 ensures even(n) <==> n % 2 == 0  
{  
 if n == 0 then true else odd(n-1)  
}  
predicate odd(n: nat)  
 ensures odd(n) <==> n % 2 != 0  
{  
 if n == 0 then false else even(n-1)  
}

Dafny chứng minh rằng họ chấm dứt bằng cách xem xét tất cả các con đường có thể thông qua hai chức năng.

1. **Dafny – Collections**

Loại giá trị là những loại đai diện một số thông tin mà không phụ thuộc vào trạng thái của heap. Những giá trị này có thể nhận diện toán học. chúng không thể được sửa đổi khi chúng được tạo ra. Ví dụ như sequences và sets. Bạn không thể thay đổi một set theo cách mà bạn sửa đổi chỉ mục trong một mảng. thay vào đó để thêm một phần tử vào set, bạn sẽ xây dựng dựa trên việc kết hợp set gốc và set singleton bao gồm phần tử mới. dĩ nhiên là những phần tử cũ vẫn tồn tại xung quanh. Việc thiếu sự phụ thuộc vào heap làm cho các loại giá trị đặc biệt hữu ích trong đặc tả.

Điều đó không có nghĩa là bạn không thể cập nhật những thứ với các loại giá trị trong nó. Biến cái mà bao gồm loại giá trị có thể được cập nhật để có giá trị mới của loại đó. Tất cả các loại giá trị có thể được lưu trữ trong các trường trên heap và được sử dụng trong real code cùng với các thông số kĩ thuật. các loại giá trị được xây dựng tron Dafny là sets, sequences, multisets, maps.

1. [Sets](http://rise4fun.com/Dafny/tutorialcontent/Collections)

Sets của các loại khác nhau tạo thành một trong những công cụ để kiểm thử cho Dafny. Sets đại diện cho một bộ sưu tập không có thứ tự của các phần tử, mà không lặp lại.

Sets có thể có bất kì loại nào, bao gồm cả object.

* Ví dụ về sets

var s1 := {}; // set rỗng

var s2 := { 1, 2, 3};

var s3, s4 := {1, 2} , {1, 4};

* Hai set bằng nhau nếu chúng có các phần tử giống nhau.
* Set mới có thể được tạo ra bằng cách sử dụng các phép toán:

+ Phép toán hội

assert s2 + s4 == {1, 2, 3, 4}

+ Phép toán hợp

assert s2 \* s3 == {1, 2}

+ Phép trừ

assert s2 – s3 == {3}.

Các toán tử so sánh

+ Tập hợp con

assert {1, 2} <= {1, 2, 3, 4}

+ Không có mối quan hệ

assert !({1, 2} <= {1, 4, 5})

+ Bằng nhau

assert {1, 2} == {1, 2}

+ Không bằng nhau

assert { 1, 2, 3} != {3, 4, 5}

* Set còn cung cấp các toán tử in và !in

assert 5 in {1, 2, 3, 4, 5}

// khẳng định 5 nằm trong tập {1, 2, 3, 4, 5}

* assert 1 !in {2, 3, 4, 5}

// khẳng định 1 không nằm trong tập {2, 3, 4, 5}

Một cách hữu ích để tạo sets là sử dụng bao hàm một set. Nó định nghĩa một set mới bao gồm điều kiện f(x) sao cho với mọi x thuộc loại T bất kì thỏa mãn điều kiện p(x).

Set x: T | p(x) :: f(x)

Ví dụ:

assert (set x | x in {0, 1, 2} :: x \* 2) == {0, 2, 4};

1. [Sequences](http://rise4fun.com/Dafny/tutorialcontent/Collections)

Sequences (chuỗi) là một kiểu được xây dựng trong Dafny, đại diện cho một danh sách có thứ tự. Chúng có thể được sử dụng để đại diện cho nhiều loại tập bao gồm danh sách, hàng đợi, ngăn xếp, v.v. Các Sequence là một loại giá trị bất biến: chúng không thể thay đổi khi được tạo ra. Theo đó, chúng tương tự các chuỗi (string) trong các ngôn ngữ Java và Python, ngoài ra chúng có thể là các chuỗi của các kiểu tùy ý chứ không chỉ là các chuỗi của các ký tự. Các loại Sequences được viết: seq<int> cho một chuỗi số nguyên (Lưu ý một lỗi đã phát hiện trong Dafny ngăn cản bạn tạo các chuỗi của naturals, nat. ). Ví dụ, hàm này lấy một dãy như một tham số:

predicate sorted(s: seq<int>)

{

forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j]

}

Chiều dài của một chuỗi được viết |s|. Các phần tử cụ thể của một dãy được truy cập bằng cách sử dụng cùng cú pháp dấu ngoặc vuông như các mảng. Cũng lưu ý rằng hàm không đòi hỏi một mệnh đề reads để truy xuất chuỗi. Đó là bởi vì chuỗi không được lưu trên heap; chúng là các giá trị, do đó, các chức năng không cần khai báo khi chúng truy cập chuỗi. Thuộc tính mạnh nhất của chuỗi là các chú thích và các chức năng có thể tạo ra và thao tác chúng. Ví dụ, một cách thể hiện thứ tự sắp xếp là đề quy: nếu phần tử đầu tiên nhỏ hơn phần tử còn lại, và phần còn lại được sắp xếp thì toàn bộ mảng được sắp xếp:

predicate sorted2(s: seq<int>)

{

0 < |s| ==> (forall i :: 0 < i < |s| ==> s[0] <= s[i]) && sorted2(s[1..])

}

Ký hiệu s[1..] đang cắt chuỗi. Nó có nghĩa là bắt đầu từ phần tử đầu tiên, lấy các phần tử cho tới khi kết thúc. Điều này không thay đổi s vì các chuỗi là không thay đổi. Thay vào đó, nó tạo một dãy mới các tất cả các phần tử giống thực tự trên, ngoại từ phần tử đầu tiên. Điều này tương tự như việc thêm các số nguyên trong đó các giá trị ban đầu không thay đổi, mà chỉ những giá trị mới tạo ra. Ký hiệu của sự phân chia là: s[i..j] với 0 <= i <= j <= |s|. Dafny sẽ thực thi các chỉ số này. Chuỗi kết quả sẽ có các phần tử j-i chính xác, và sẽ bắt đầu với phần tử s[i] và tiệp tục tuần tự qua chuỗi, nếu kết quả không rỗng. Điều này có nghĩa là phần tử tại chỉ số j bị loại trừ khỏi chuỗi cắt, nó phản ánh thời gian nghĩ để lập chỉ mục thông thường.

Các chuỗi cũng có thể được xây dựng từ các thành phần của chúng, sử dụng ký hiệu hiển thị: var s := [1, 2, 3]; Ở đây chúng ta có một biến nguyên trong một số đoạn code bắt buộc có chứa các phần tử 1, 2 và 3. Loại suy luận đã được sử dụng ở đây là chuỗi là một số nguyên. Ký hiệu này cho phép chúng ta xây dựng chuỗi rỗng và các chuỗi singleton:

[] //chuỗi rỗng, có thể là một chuỗi của bất cứ kiểu nào

[true] // một chuỗi singleton của kiểu seq<bool>

Chú thích chuỗi cắt và ký hiệu hiển thị có thể được sử dụng để kiểm tra thuộc tính của chuỗi:

* var s := [1, 2, 3, 4, 5];
* assert s[|s| - 1] == 5; //truy cập phần tử cuối cùng
* assert s[|s| - 1..|s|] == [5]; //cắt chuối đến phần tử cuối cùng, như một singleton
* assert s[1..] == [2, 3, 4, 5]; //mọi số trừ số đầu tiên
* assert s[..|s| - 1] == [1, 2, 3, 4]; //mọi số từ số cuối cùng
* Assert s == s[0..] == s[..|s|] == s[0..|s|] == s[..]; // cả chuỗi

Bởi đến nay các lệnh phổ biết nhất trên các chuỗi đang nhận được các phần tử đầu tiên và cuối cùng, và nhận được tất cả các phần tử khác trừ phần tử đầu tiên và cuối cùng, như thường được sử dụng trong chức năng đệ quy, chẳng hạn như sorted2 ở trên. Bên cạnh chức năng đang được truy cập hoặc cắt chuỗi, sequence có thể cũng để nối, sử dụng ký tự cộng (+):

* assert [1,2,3] == [1] + [2,3];
* assert s == s + [];
* assert forall i :: 0 <= i <= |s| ==> s == s[..i] + s[i..];

Lệnh assert thứ 2 thể hiện mối quan hệ giữa nối và cắt chuỗi. Vì toán tử cắt chuỗi là độc quyền trên một mặt và bao gồm mặt khác, các phần tử xuất hiện trong nối đúng một lần. Lưu ý rằng toán tử nối là kết hợp:

Assert forall a: seq<int>, b: seq<int>, c: seq<int> ::

(a+b) + c == a + (b + c)

Nhưng định lý Z3 sẽ không nhận ra điều này trừ khi nó được nhắc tới với một assert thực tế.

* Sequences cũng hỗ trợ toán tử in và !in, kiểm tra các ràng buộc trong một sequence:
* Assert 5 in s; //sử dụng s từ trước
* Assert 0 !in s;

Điều này cũng cho ta một phương tiện thay thế định lượng trong các phần tử của một sequence, khi chúng ta không quan tâm về chỉ số. Ví dụ, chúng ta có thể yêu cầu rằng một sequence chỉ chứa các phần tử có chỉ số trong sequence:

Var p := [2, 3, 1, 0];

Assert forall i :: i in p ==> 0 <= i < |s|;

Đây là thuộc tính của mỗi phần tử riêng của sequence. Nếu chúng ta muốn liên kết nhiều yếu tố với nhau, chúng ra sẽ cần phải định lượng trên chỉ số, như trong ví dụ đầu tiên.

Đôi khi chúng ta muốn cố thử câp nhật bản chất của sequence. Trong khi chúng ta không thể thay đổi thứ tự ban đầu, chúng ta có thể tạo một sequence mới với các yếu tố tương tự ở khắp mọi nơi ngoại trừ các phần tử cập nhật:

s[i := v] //thay thế chỉ số i bằng v trong seq s

Tất nhiên, các chỉ số i phải là chỉ số vào mảng. Cú pháp này là một phím tắt cho một toán tử có thể thực hiện với các toán tử thường xuyên cắt và nối. Bạn có thể điền vào đoạn code dưới đây để thực hiện điều này?

Function update(s: seq<int>, I: int, v: int): seq<int>

Requires 0 <= index < |s|

ensures update(s, I, v) == s[I := v]

{

//mở trong editor để xem đáp án.

}

Bạn cũng có thể tạo thành một sequence từ các phần tử của mảng. Điều này được thực hiện bằng cách sử dụng các ký hiệu "slice" tương tự như trên:

* var a := new int[3]; //3 phần tử mảng của các số nguyên

a[0], a[1], a[2] := 0, 3, -1;

* var s := a[..];
* assert s == [0, 3, -1];

Để có được một phần của mảng,các biên có thể được đưa ra cũng giống như một toán tử slicing thông thường:

* assert a[1..] == [3, -1];
* assert a[..1] == [0];
* assert a[1..2] == [3];

Vì sequence hỗ trợ in và !in, toán tử này cho chúng ta một cách dễ dàng để thể hiện " phần tử không trong mảng", chuyển:

forall k :: 0 <= k < a.length ==> elem != a[k]

Vào:

elem !in a[..]

Hơn nữa, biên được bao gồm dễ dàng: forall k :: 0 <= k < i ==> elem != a[k]

Tương tự như:

elem !in a[..i]

1. [Multisets](http://rise4fun.com/Dafny/tutorialcontent/Collections)

Giống như sets, ngoại trừ chúng giữ dấu vết có bao nhiêu bản copy của mỗi phần tử chúng có. Điều này làm cho chúng đặc biệt hữu ích để lưu trữ tập hợp các phần tử trong mảng

Giống như sets, multisets không có thứ tự. Nhiều hoạt động được xác định trong sets cũng có sẵn trong multisets. Bạn có thể sử dụng in để kiểm tra xem một vài phần tử có trong multisets hay không. Phép (+) lấy tất cả các phần tử ở cả 2 multisets và những phần tử trùng nhau chỉ lấy một. Phép (-) ở multisets sẽ khác với sets khi giá trị của các phần tử trong multisets trùng nhau.

assert (multiset{1,1,1} - multiset{1,1}) == multiset{1};

* multiset disjoint (!!) hoạt động như mong đợi. Nó trả về true khi và chỉ khi hai multisets không có thành phần chung. Hai multiset bằng nhau nếu từng phần tử trong 2 multiset phải giống nhau.
* Multisets có thể được tạo ra từ sequences và sets bằng cách sử dụng multiset với ngoặc vuông.
* assert multiset([1,1]) == multiset{1,1};
* assert multiset({1,1}) == multiset{1};

Cả hai khẳng định đều chính xác vì multiset của sequence xem xét từng phần tử, trong khi sets chỉ có tối đa một trong mỗi phần tử. Dafny cho phép bạn viết {1,1} nó giống với {1} bởi vì những số trùng bị bỏ qua.

1. [Maps](http://rise4fun.com/Dafny/tutorialcontent/Collections)

* Maps trong Dafny đại diện cho mảng liên kết. Không giống như những loại trước, chúng có hai loại: key và value. Các value có thể được lấy ra, tìm kiếm dựa trên key. Một map có cấu trúc:

map<U,V> với U là key, V là value

* Mỗi map có một domain, đó là tất cả các key mà map có value. Để kiểm tra xem một key có nằm trong domain có thể sử dụng toán tử in. Ngoài ra 2 maps là tách rời (!!) nếu domain của chúng là những tập tách rời.
* Nếu m là một map, thì m[i:=j] là một map mới là kết quả của việc thêm i vào domain của m và sau đó gán key I với value j. Nếu i đã được gán một giá trị thì nó sẽ ghi đè lên trong map mới. hai map là bằng nhau nếu chúng có domain giống nhau và chúng ánh xạ key bằng nhau đến value bằng nhau. Lưu ý là domain của một map phải hữu hạn.
* Giống như sets, maps có bao gồm một map. Cú pháp cũng gần giống set

map i: T | p(i) :: f(i)

sự khác nhau giữa maps và sets là i ở đây là key và nó sẽ map với f(i).p(i). nó được sử dụng để xác định miền của map mới.

* ví dụ: map i | 0 <= I < 10 :: 2\*i

1. **Dafny – Lemmas (Bổ đề)**
2. [Introduction](http://rise4fun.com/Dafny/tutorialcontent/Lemmas)

* Đôi khi có những bước logic yêu cầu chứng minh một chương trình là đúng, nhưng chúng quá phức tạp đối với Dafny để khám phá và sử dụng. Khi điều này xáy ra, chúng ra có thể hỗ trợ cho Dafny bằng cách cung cấp một lemma.
* Lemmas là một định lí được sử dụng để chứng minh một kết quả khác chứ không phải mục tiêu của nó. Chúng cho phép Dafny tách bằng chứng thành 2 phần: chứng minh lemma, sau đó sử dụng nó để chứng minh kết quả cuối cùng, kết quả cuối cùng là sự chính xác của chương trình. Dafny và máy tính nói chung, rất phù hợp cho sự phân chia những tài liệu chi tiết và bao phủ tất cả các ca kiểm thử. Nhưng nó lại thiếu thông minh để xem các bước trung gian, cái làm cho việc chứng minh trở nên dễ dàng hơn.
* Bằng việc viết và sử dụng Lemmas, bạn có thể chỉ ra những bước thực hiện là gì và khi nào sử dụng chúng trong chương trình. Đặc biệt quan trọng là đối số quy nạp, nó là vấn đề khó nhất đối với những người chứng minh định lí.

1. [Searching for Zero](http://rise4fun.com/Dafny/tutorialcontent/Lemmas)

* Ví dụ: Tìm phần tử 0 trong mảng.

Điều thú vị là mảng mà ta tìm kiếm có hai thuộc tính đặc biệt: tất cả các phần tử trong mảng không âm và phần tử sau nhỏ hơn phần tử trước một đơn vị.

requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]

requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]

Giải thuật tìm phần tử 0 trong mảng

index := 0;

while index < a.Length

invariant 0 <= index

invariant forall k :: 0 <= k < index &amp;&amp; k < a.Length ==> a[k] != 0

{

if a[index] == 0 { return; }

index := index + a[index];

}

index := -1;

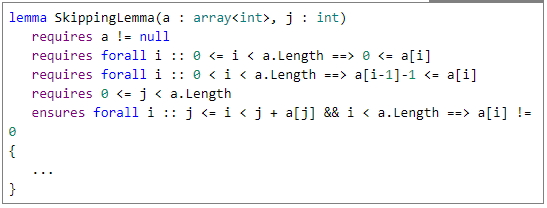
Mã này sẽ tính toán đúng kết quả, nhưng Dafny phàn nàn về sự bất biến của vòng lặp thứ hai. Dafny không tin rằng bỏ qua những thành phần này là hợp lí. Lí do vì điều kiện tiên quyết nói rằng mỗi phần tử sau giảm ít nhất một, nhưng nó không nói gì về mối quan hệ của các phần tử đứng xa nhau. Để thuyết phục Dafny về thực tế này, ta cần sử dụng một lemma.

1. [Lemmas](http://rise4fun.com/Dafny/tutorialcontent/Lemmas) (Bổ đề)

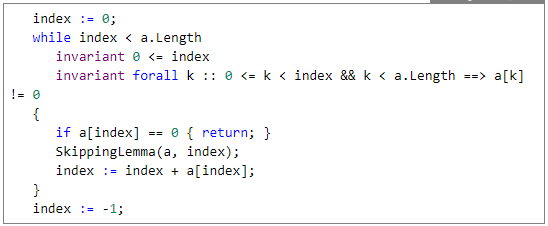
Một lemma thực sự chỉ là một phương thức ma. Thuộc tính mong muốn nêu bởi lemma (chính xác hơn, là kết luận của bổ đề) được tuyên bố như là điều kiện, cũng giống như cách bạn làm cho một phương thứcthông thường. Không giống như một phương thức, một lemma không bao giờ được phép thay đổi trạng thái. Vì một lemma là ảo, nó không cần phải được gọi tại thời gian chạy, do đó, trình biên dịch xóa nó trước khi xuất mã thực thi. Vì vậy, lemma là đại diện duy nhất cho hiệu quả của nó trên kiểm tra chương trình. Bạn có thể nghĩ rằng bổ đề như khẳng định chắc chắn, trong đó họ đang chỉ cần thiết để giúp chứng minh của chương trình sau cùng. Lemma điển hình có thể như:



Đối với vấn đề tìm kiếm zero, các thuộc tính mong muốn không có trong số các phần tử index cho đến index + a[index] có thể là số không. Chúng có các mảng và chỉ số bắt đầu từ dạng tham số, với các yêu cầu thông thường từ FindZero:

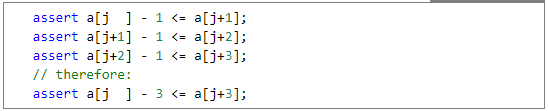


Postcondition là thuộc tính mong muốn mà chúng ta muốn. Giới hạn phụ i vì j + a[j] có thể là sự kết thúc của mảng. Chúng ta chỉ muốn nói về chỉ số trong phạm vi đó mà cũng là chỉ số mảng. Sau đó chúng ta thực hiện một bước rất quan trọng: Hãy kiểm tra lemma của chúng ta là đủ để chứng minh các vòng lặp bất biến. Bằng cách kiểm tra này trước khi điền vào lemma,chúng tôi đảm bảo rằng chúng đang cố gắng để chứng minh đúng. Các phương thức FindZero trở thành:



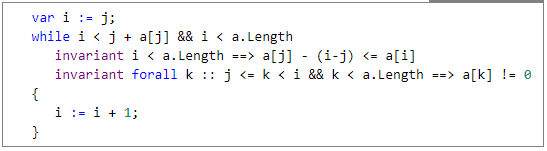
Bây giờ Dafny không phàn nàn về các phương thức FindZero, như lemma của postcondition cho thấy rằng vòng lặp bất biến được bảo tồn. Nó phản ánh về bổ đề chính nó, mà không phải là đáng ngạc nhiên cho rằng bên trong rỗng. Để có được Dafny chấp nhận bổ đề, chúng ta sẽ phải chứng minh postcondition đúng. Chúng ta làm điều này như làm tất cả mọi thứ trong Dafny: viết mã.

Chúng ta tôi đầu với các thuộc tính quan trọng của mảng, mà nó chỉ làm giảm chậm. Chúng tacó thể yêu cầu cho dù thuộc tính nhất định giữ bằng cách sử dụng assert. Ví dụ, chúng ta có thể thấy rằng Dafny biết:



Do đó chúng ta có thể thấy rằng Dafny có thể làm theo trong bước bất kỳ, và thậm chí có thể chuỗi chúng một cách thích hợp. Nhưng số lượng các bước chúng ta cần phải thực hiện không phải là hằng số: nó có thể phụ thuộc vào giá trị của a[j]. Nhưng chúng tôi đã có một cấu trúc để đối phó với một số biến của bước: trong khi vòng lặp!

Chúng tôi có thể sử dụng cấu trúc rất giống nhau ở đây để có được Dafny chuỗi các bước cùng nhau. Chúng tôi muốn lặp từ j đến j + a[j], theo dõi các ràng buộc thấp hơn khi chúng đi. Chúng tôi cũng theo dõi thực tế rằng tất cả các phần tử mà chúng tôi đã thấy cho đến nay không phải 0:



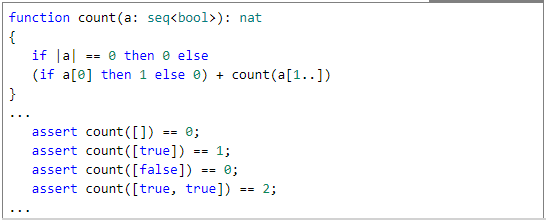
Biến đầu tiên cho ràng buộc trên các yếu tố hiện tại, nếu chúng tôi đã không chạy vào cuối mảng. Đối với mỗi chỉ số trong quá khứ j (trong đó có những i-j), mảng có thể là một số nhỏ hơn, do đó, giá trị này được trừ đi từ a[j]. Điều này chỉ nói rằng các phần tử hiện tại không thể bằng 0, vậy mà không có biến thứ hai, Dafny sẽ không thể biết rằng đã không có số 0. Dafny quên tất cả mọi thứ về vụ hành quyết của vòng ngoại trừ những gì được đưa ra trong invaraints, vì vậy chúng ta cần phải xây dựng một thực tế là có số không có bất cứ nơi nào cho đến nay.

Thật vậy! Vòng lặp chỉ ra số truy cập. Như chúng ta đã thấy trước khi Dafny là có thể tìm ra mỗi bước của riêng mình, vì vậy chúng tôi không cần phải làm bất cứ điều gì hơn nữa. Chúng tôi chỉ cần thiết để cung cấp cho nó cấu trúc của các bằng chứng mà nó cần thiết để thực hiện. Đôi khi các bước cá nhân là phức tạp, đủ rằng chúng cần một chút riêng của chúng subproofs, sử dụng hoặc là một loạt các câu khẳng định hoặc một bổ đề khác.

Khi làm việc với mảng, lặp đi lặp lại là một giải pháp tự nhiên cho nhiều vấn đề. Không có một số lần, Tuy nhiên, khi đệ quy được sử dụng để xác định các chức năng hoặc thuộc tính. Trong những trường hợp này, các bổ đề thường có cùng cấu trúc đệ quy. Để xem một ví dụ này, chúng tôi sẽ xem xét các vấn đề của đếm.

1. [Counting](http://rise4fun.com/Dafny/tutorialcontent/Lemmas) (Đếm)

Chúng ta sẽ đếm số trues trong một chuỗi bools, bằng cách sử dụng các chức năng count, được đưa ra dưới đây:



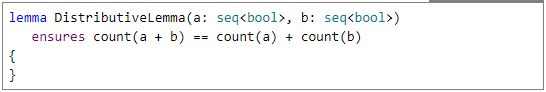
Mã này rất đơn giản, nhưng một điều cần chú ý là chức năng là định nghĩa đệ quy. Chức năng đệ quy như thế này là dễ bị đòi hỏi phải bổ đề. Đó là một thuộc tính mong muốn của số mà chúng tôi muốn để có thể sử dụng để xác minh một chương trình mà sử dụng chức năng này: nó phân phối trên bổ sung. Bằng cách này chúng có nghĩa là:



Ở đây, lần đầu tiên cộng(+) là trình tự nối, và thứ hai là số nguyên bổ sung. Rõ ràng, chúng tôi có thể phá vỡ bất kỳ thứ tự thành hai chuỗi a và b, tính chúng riêng biệt, và thêm các kết quả. Điều này là đúng, nhưng Dafny không thể chứng minh điều đó trực tiếp. Vấn đề là các chức năng không chia ra trình tự theo cách này. Các chức năng cần các yếu tố đầu tiên, tính số của mình, sau đó thêm nó vào phần còn lại của tiến trình. Nếu a là lâu dài, sau đó nó có thể là một trong khi trước khi quá trình cán này thực sự đạt đến count(b), do đó, Dafny không cố gắng để unwrap hơn là một vài cuộc gọi đệ quy. (Hai, để được chính xác. Xem giấy máy tính với một người giải quyết SMT Amin, Leino và Rompf, TAP 2014). Đây là một ví dụ về một tài sản mà yêu cầu một bổ đề để chứng minh.

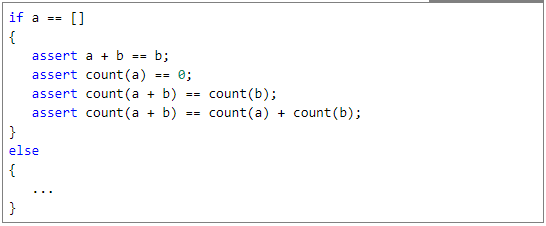
Trong trường hợp của chúng tôi, chúng tôi có hai lựa chọn cho bổ đề: chúng tôi có thể viết quantifier universal cùng chúng tôi đã có ở trên, hoặc chúng tôi có thể làm cho bổ đề cụ thể để một cặp trình tự a và b. Nó chỉ ra rằng khi chúng ta muốn phân phối bất động sản, chúng tôi không cần tài sản đầy đủ phổ quát. Chúng tôi là chỉ quan tâm đến một thực tế rằng count(a + b) == count(a) + count(b) cho hai cụ thể a và b mà được biết đến trong chương trình. Do đó khi chúng tôi gọi là bổ đề để có được tài sản, chúng tôi có thể nói điều đó hai chuỗi mà chúng tôi quan tâm. Nếu chúng tôi có trình tự khác nhau ở một nơi khác, chúng tôi có thể gọi các phương pháp với đối số khác nhau, giống như một phương pháp thông thường. Nó chỉ ra rằng chứng minh tài sản đầy đủ phổ quát, trong khi có thể, làm việc nhiều hơn so với chứng minh các trường hợp cụ thể, cụ thể, do đó, chúng tôi sẽ giải quyết trường hợp này đầu tiên.

Do đó bổ đề nên dùng như là đối số trình tự quan tâm, và postcondition là như sau:



1. [Proving the Distributive Property](http://rise4fun.com/Dafny/tutorialcontent/Lemmas) (Chứng minh phân phối thuộc tính)

Để viết bổ đề, chúng ta phải tìm ra một chiến lược để chứng minh nó. Như bạn có thể xác minh ở trên (không có ý định chơi chữ), bổ đề không hoạt động được nêu ra, như nếu không một bổ đề sẽ là không cần thiết. Để làm điều này, chúng tôi lưu ý rằng lý do Dafny không phải có thể chứng minh điều này ở nơi đầu tiên mà các count chức năng được xác định từ khi bắt đầu của chuỗi, trong khi các tài sản phân phối hoạt động vào giữa của một chuỗi. Do đó nếu chúng ta có thể tìm thấy một cách để làm việc từ phía trước trong tiến trình, sau đó Dafny có thể theo dõi bằng cách sử dụng định nghĩa của các chức năng trực tiếp. Vâng, các yếu tố đầu tiên của dãy là gì? Có một vài trường hợp, dựa trên đó (nếu có) của a và b là đầu có sản phẩm nào. Vì vậy bổ đề chúng tôi sẽ phải xem xét nhiều trường hợp, một đặc điểm chung của bổ đề. Chúng tôi nhận thấy rằng nếu a == [], sau đó a + b == b, bất kể những gì b là. Bổ đề xử lý trường hợp sử dụng cùng mã điều nào để xử lý các trường hợp: nếu báo cáo. Một chứng minh ngắn của các tài sản mong muốn được đưa ra sử dụng khẳng định dưới đây.



Chúng tôi có thể kiểm tra bổ đề của chúng tôi trong trường hợp này bằng cách thêm một đòi hỏi mệnh đề hạn chế a trường hợp này. Chúng tôi tìm thấy các mã xác minh. Điều này có nghĩa là nếu a == [], sau đó bổ đề của chúng tôi một cách chính xác sẽ chứng minh postcondition. Trong trường hợp này, chỉ khẳng định đầu tiên ở trên là cần thiết; Dafny được phần còn lại của các bước trên riêng của mình (thử nó!). Bây giờ chúng ta có thể xem xét trường hợp khác, khi 0 < |a|.

Mục tiêu của chúng tôi là liên quan đến count(a + b) để count(a) và count(b). Nếu a không phải là trình tự trống rỗng, sau đó khi chúng tôi sử dụng chúng tôi lừa của sau định nghĩa mở rộng count(a + b), chúng tôi nhận được:



Thông báo rằng chúng tôi nhận được count([a[0]]) và a[1..]. Những điều khoản này hai cũng sẽ xuất hiện nếu chúng ta mở rộng count(a). Cụ thể:



Cuối cùng, chúng tôi có thể thay thế này định nghĩa cho count(a) vào postcondition để có được:



Bây giờ điều này trông rất giống với những biểu hiện mà chúng tôi đã nhận sau khi mở rộng count(a + b). Khác biệt duy nhất là count(a[1..] + b) đã trở thành count(a[1..]) + count(b). Nhưng điều này là chính xác nơi chúng tôi đang cố gắng để chứng minh!

1. [Induction](http://rise4fun.com/Dafny/tutorialcontent/Lemmas) (Cảm ứng)

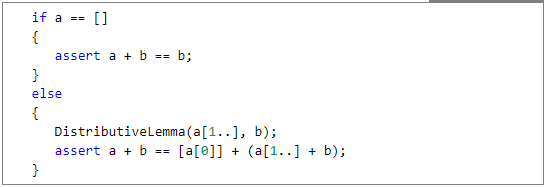
Các đối số, chúng tôi đang cố gắng để làm cho là quy nạp. Chúng ta có thể chứng minh mục đích của chúng tôi cho rằng một phiên bản nhỏ hơn của vấn đề là sự thật. Điều này là chính xác các khái niệm của cảm ứng: sử dụng một phiên bản nhỏ hơn của một vấn đề để chứng minh một một lớn hơn. Để làm điều này, chúng ta gọi là sở hữu đệ quy từ trong mã của chúng tôi. Nó là một phương pháp, do đó, nó có thể được kích hoạt bất cứ khi nào chúng ta cần nó.

Dafny sẽ giả định rằng các cuộc gọi đệ quy đáp ứng đặc điểm kỹ thuật. Đây là giả thiết quy nạp, tất cả các cuộc gọi đệ quy của bổ đề có giá trị. Điều này phụ thuộc chủ yếu vào một thực tế rằng Dafny cũng chứng minh việc chấm dứt. Điều này có nghĩa rằng cuối cùng, bổ đề sẽ không làm cho một đệ quy gọi. Trong trường hợp này, đây là chi nhánh đầu tiên của nếu tuyên bố. Nếu không có không có cuộc gọi đệ quy, sau đó bổ đề phải được chứng minh trực tiếp cho các trường hợp đó. Sau đó mỗi cuộc gọi trong ngăn xếp là hợp lý trong giả định bổ đề hoạt động cho các trường hợp nhỏ hơn. Nếu Dafny không chứng minh dãy chấm dứt, sau đó chuỗi có thể tiếp tục mãi mãi, và giả định cho mỗi cuộc gọi sẽ không được chứng minh.

Cảm ứng nói chung là tìm một cách để xây dựng các mục tiêu của bạn lên một bước tại một thời điểm. Xem cách khác, nó chứng minh mục tiêu của bạn trong điều khoản của một phiên bản nhỏ hơn. Bổ đề phân phối được chứng minh bởi deconstructing nối chuỗi một phần tại một thời gian cho đến khi các chuỗi đầu tiên là hoàn toàn đi. Trường hợp này đã được chứng minh là một trường hợp cơ sở, và sau đó toàn bộ chuỗi deconstructions được xác minh.

Chìa khóa để làm cho công việc này là Dafny không bao giờ có để xem xét toàn bộ chuỗi của cuộc gọi. Kiểm tra chấm dứt, nó có thể nhận được các chuỗi là hữu hạn. Sau đó, tất cả nó đã làm là kiểm tra một bước. Nếu một bước tùy ý là hợp lệ, sau đó toàn bộ chuỗi phải là tốt. Đây là cùng một logic Dafny sử dụng cho vòng: kiểm tra bất biến việc giữ ban đầu, và rằng một trong những bước tùy ý bảo tồn nó, và bạn đã kiểm tra toàn bộ vòng, bất kể bao nhiêu lần trong vòng lặp đi xung quanh. Sự giống nhau là bề ngoài nhiều hơn. Cả hai loại bổ đề (và cả hai loại lý do Dafny làm cho về chương trình của bạn) là quy nạp. Cũng không phải là đáng ngạc nhiên cho các mối quan hệ giữa lặp đi lặp lại và đệ quy như là hai phương tiện để đạt được điều tương tự.

Với điều này trong tâm trí, chúng tôi có thể hoàn thành bổ đề bằng cách gọi đệ quy bổ đề tại các chi nhánh khác của nếu tuyên bố:



Bây giờ bổ đề xác nhận. Nhưng những gì nếu chúng tôi muốn nhận mỗi cặp trình tự có liên quan theo cách này? Chúng tôi phải xem xét sử dụng một bổ đề trong Dafny để có thể làm điều này, chúng tôi sẽ khám phá với một ví dụ khác.

1. [Paths In a Directed Graph](http://rise4fun.com/Dafny/tutorialcontent/Lemmas) (đường dẫn trong một biểu đồ hướng dẫn)

Phần cuối cùng, nâng cao hơn, ví dụ hơn, chúng ta sẽ chứng minh thuộc tính về các đường dẫn trong một đồ thị. Về điều này, chúng ta sẽ có dịp để gọi một lemma (bổ đề) phổ biến trên tất cả các sequence (chuỗi) của các nút. Một đồ thị chuẩn bao gồm một số Nodes (nút) với một số liên kết tới các Nodes khác. Những liên kết này có hướng duy nhất, và những hạn chế duy nhất của chúng là một Node không thể liên kết với chính nó. Các nút được định nghĩa là:

Class Node

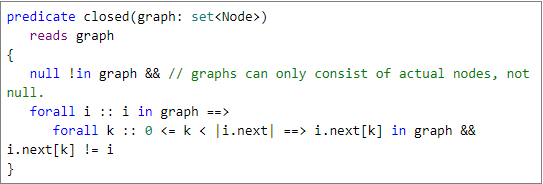
{

//một trường độc lập cho các nút liên kết với nhau

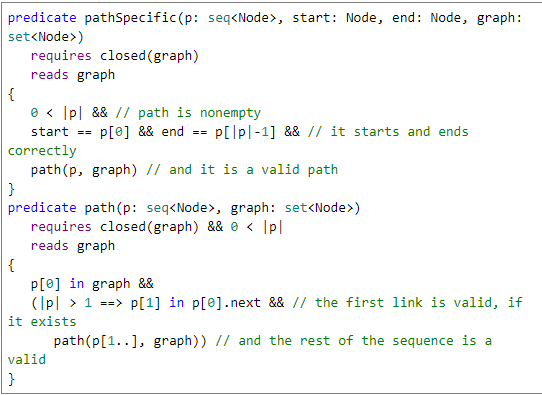
var next: seq<Node>

}

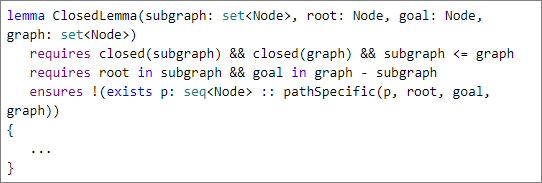
Chúng tôi đại diện cho một đồ thị như là một tập hợp của các nút null chỉ trỏ tới các nút khác trong đồ thị, và không để bản thân. Chúng tôi gọi là một tập hợp các nút đóng cửa:



Chúng đại diện cho một con đường như là một chuỗi các nút, nơi mỗi nút được liên kết bởi các nút trước đó trong đường dẫn nonempty. Chúng xác định hai predicates, một trong đó xác định một đường dẫn hợp lệ và khác để xác định liệu đường dẫn được đưa ra là một trong những hợp lệ giữa hai nút cụ thể trong biểu đồ:

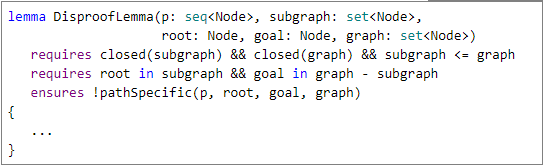


Bây giờ chúng tôi đang sẵn sàng để nhà nước bổ đề chúng tôi muốn chứng minh. Chúng ta xem xét một đồ thị và một tiểu biểu đồ: một tập hợp con của các nút biểu đồ cũng tạo thành một đồ thị. Đồ thị phụ này phải là đóng lại, tức là không chứa các liên kết bên ngoài của bản thân. Nếu chúng ta có tình hình như vậy, sau đó không thể có một đường dẫn hợp lệ từ một nút trong biểu đồ phụ một nút ở bên ngoài tiểu biểu đồ này. Chúng tôi sẽ gọi này thực tế bổ đề đóng cửa, mà chúng tôi nhà nước trong Dafny như sau:

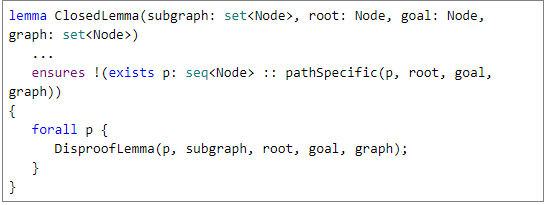


Nhà nước là điều kiện tiên quyết tất cả các yêu cầu: biểu đồ và đồ thị phụ là hợp lệ, nút gốc là trong biểu đồ phụ, nhưng mục tiêu không phải là, và tất cả mọi thứ được chứa trong các đồ thị chính. Postcondition nói rằng không là không có đường dẫn hợp lệ từ gốc đến mục tiêu. Ở đây chúng tôi chỉ chứng minh cho một cặp bắt đầu/kết thúc nút cụ thể.

Một cách để chứng minh không tồn tại của một cái gì đó là để chứng minh cho bất kỳ thứ tự các nút mà nó không thể là một đường dẫn hợp lệ. Chúng tôi có thể làm điều này với, bạn đoán nó, một bổ đề. Bổ đề này sẽ chứng minh cho bất cứ trình tự nhất định, mà nó không thể là một đường dẫn hợp lệ từ root để goal. Disproof một bổ đề đường dẫn sẽ như:

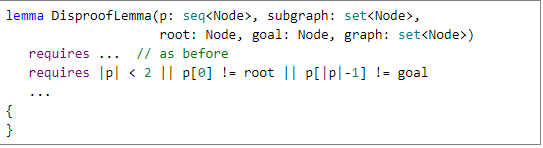


Các điều kiện tiên quyết là tương tự như ClosedLemma. Để sử dụng DisproofLemma ở ClosedLemma, chúng ta cần phải gọi một lần cho mỗi chuỗi các nút. Điều này có thể được thực hiện với của Dafny forall tuyên bố, mà tập hợp các hiệu ứng của cơ thể của nó cho tất cả các giá trị cho biến đó bị ràng buộc.

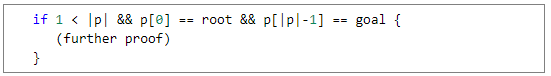


Như bạn thấy, điều này gây ra các ClosedLemma để xác minh, vì vậy chúng tôi kiểm tra bổ đề là thành công. Do đó DisproofLemma là đủ mạnh, và công việc của chúng tôi được giảm xuống chỉ cần chứng minh nó.

Không có một vài cách khác nhau mà một chuỗi các nút có thể là một đường dẫn không hợp lệ. Nếu đường dẫn là trống rỗng, sau đó nó không thể là một đường dẫn hợp lệ. Ngoài ra, các yếu tố đầu tiên của con đường phải root và các yếu tố cuối cùng cần goal. Bởi vì root in subgraph và goal !in subgraph, chúng ta phải có root != goal, vì vậy, tiến trình phải có ít nhất hai yếu tố. Để kiểm tra Dafny nhìn thấy điều này, chúng tôi có thể tạm thời đặt điều kiện tiên quyết bổ đề của chúng tôi như sau:

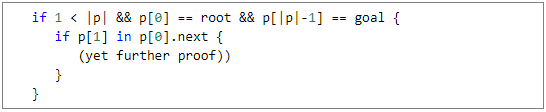


Lưu ý rằng điều này sẽ gây ra ClosedLemma để ngăn chặn xác minh, như bổ đề bây giờ chỉ hoạt động cho một số trình tự. Chúng tôi sẽ bỏ qua ClosedLemma cho đến khi chúng tôi đã hoàn thành DisproofLemma. Điều này để kiểm chứng, có nghĩa là Dafny là có thể chứng minh postcondition trong những trường hợp này. Do đó, chúng tôi chỉ cần chứng minh rằng đường dẫn không hợp lệ khi những điều kiện này không giữ. Chúng tôi có thể sử dụng một if tuyên bố để thể hiện điều này:

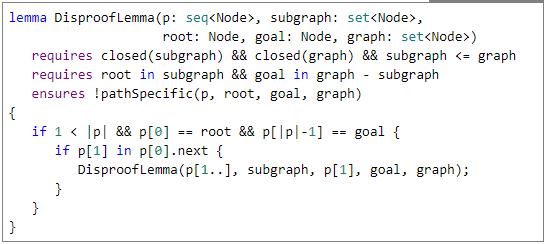


Nếu đường là ít nhất hai yếu tố lâu dài, các yếu tố đầu tiên là root, và cuối cùng là goal, sau đó chúng tôi có một bằng chứng hơn nữa. Nếu những điều kiện này không được đáp ứng (có nghĩa là, nếu bảo vệ của các if tuyên bố là sai, và kiểm soát tiếp tục trong các chi nhánh khác tiềm ẩn) Dafny sẽ chứng minh postcondition ngày của riêng mình (nâng cao nhận xét: bạn có thể kiểm tra này bằng cách thêm các tuyên bố tạm thời assume false; bên trong các chi nhánh sau đó của các if). Bây giờ chúng ta chỉ cần điền thêm bằng chứng một phần. Làm như vậy, chúng ta có thể giả định các điều kiện bảo vệ của các if tuyên bố. Chúng tôi bây giờ có thể sử dụng quy nạp trick tương tự như trên.

Nếu tiến trình bắt đầu lúc root và kết thúc lúc goal, nó không thể hợp lệ vì trình tự phải tại một số điểm có một nút mà không phải là trong danh sách trước đó các nút tiếp theo. Khi chúng tôi có được bất kỳ thứ tự cụ thể như thế này, chúng tôi có thể phá vỡ nó thành hai trường hợp: trình tự không hợp lệ trong các liên kết từ nút đầu tiên vào thứ hai, hoặc là bị hỏng, một nơi nào đó xuống dòng. Giống như trong ví dụ đếm, Dafny có thể thấy rằng nếu liên kết nút đầu tiên thứ hai là không hợp lệ, sau đó trình tự không thể là một đường dẫn vì điều này gương định nghĩa của path. Vì vậy, chúng tôi chỉ có công việc phải làm gì nếu các liên kết đầu tiên là hợp lệ. Chúng tôi có thể bày tỏ điều này với nhau if nội dung:



Ở đây có sự cảm ứng. Chúng ta biết rằng p[0] == root và p[1] in p[0].next. Chúng tôi cũng biết từ những điều kiện tiên quyết mà root in subgraph. Vì vậy, bởi vì closed(subgraph), chúng ta biết rằng p[1] in subgraph. Đây là những điều kiện cùng chúng tôi bắt đầu với! Những gì chúng tôi có ở đây là một phiên bản nhỏ hơn của cùng một vấn đề. Chúng tôi có thể chỉ cần gọi đệ quy DisproofLemma để chứng minh rằng p[1..] không phải là một con đường. Điều này có nghĩa là, một định nghĩa của path, mà p không thể là một con đường, và postcondition thứ hai là hài lòng. Điều này có thể là implmented như là:



Bây giờ DisproofLemma để kiểm chứng, và với việc loại bỏ các điều kiện tiên quyết kiểm tra, chúng tôi thấy rằng ClosedLemma xác minh là tốt. Do đó chúng tôi đã chứng minh rằng không thể có một đường đi từ bên trong một đồ thị tiểu đóng cửa với bên ngoài.

Các forall tuyên bố là hữu ích khi một bổ đề cần phải một số chặn lần instantiated. Ví dụ cho thấy một phiên bản đơn giản của các forall tuyên bố. Đối với phiên bản tiên tiến hơn, xem, ví dụ, Well-founded chức năng và cực Predicates trong Dafny: A hướng dẫn bởi Leino, IWIL-2015, hoặc ví dụ trong Dafny kiểm tra mật.

Luôn luôn nhớ để kiểm tra bổ đề của bạn là đủ để chứng minh những gì bạn cần. Không có gì là khó chịu hơn so với chi tiêu một thời gian làm cho một bổ đề kiểm tra, chỉ để tìm hiểu, bạn cần một cái gì đó mạnh mẽ hơn. Điều này cũng cho phép bạn tránh việc tạo ra một bổ đề với một điều kiện tiên quyết là hạn chế như vậy mà bạn không thể gọi nó là nơi mà bạn cần.

1. [**Modules**](http://rise4fun.com/Dafny/tutorial/Modules)
2. [Introduction](http://rise4fun.com/Dafny/tutorialcontent/Modules)

* Cơ cấu một chương trình bằng cách phá vỡ nó thành nhiều phần là một phần quan trọng của việc tạo ra các chương trình lớn. Trong Dafny, điều này được thực hiện thông qua *các module* .
* Cung cấp một cách để nhóm lại với nhau liên quan đến các loại, các class,method ,function và các module khác nhau, cũng như kiểm soát phạm vi của tuyên bố.
* Module có thể nhập khẩu lẫn nhau để tái sử dụng mã, và nó có thể trừu tượng trên mô-đun để tách riêng một thực hiện từ một giao diện.

1. [Declaring New Modules](http://rise4fun.com/Dafny/tutorialcontent/Modules)\_ Module mới

* Một mô-đun mới được khai báo với từ khóa:

module Mod {  
 ...  
}

* Thân mô-đun có thể bao gồm bất cứ điều gì mà bạn có thể đặt ở mục cấp đầu, bao gồm classes, datatypes, types, methods, functions, etc.

module Mod {  
 class C {  
 var f: int;  
 method m()   
 }  
 datatype Option = A(int) | B(int)  
 type T  
 method m()  
 function f(): int  
}

* Module lồng nhau:

module Mod {  
 module Helpers {  
class C {  
 method doIt()  
 var f: int;  
 }  
 }  
}

* Sau đó, bạn có thể tham khảo các các thành viên của module Helpers trong module Mod :

module Mod {  
 module Helpers { ... }  
 method m() {  
 var x := new Helpers.C;  
 x.doIt();  
 x.f := 4;  
 }  
}

* Medthod and function được xác định ở cấp module có sẵn như các lớp, chỉ với tên mô đun đặt trước chúng. Chúng cũng có sẵn trong các phương pháp và chức năng của các lớp học trong cùng một module:

module Mod {  
 module Helpers {  
 function method addOne(n: nat): nat {  
 n + 1  
 }  
 }  
 method m() {  
 var x := 5;  
 x := Helpers.addOne(x); // x is now 6  
 }  
}.

* + Theo mặc định, định nghĩa các hàm (và các vị từ) được hiển thị bên ngoài mô-đun chúng được định nghĩa. Add assert x == 6;đến cuối m () sẽ xác minh.

1. [Importing and Exporting Modules](http://rise4fun.com/Dafny/tutorialcontent/Modules)

* Import
* Tạo modules con mới rất hữu ích, nhưng đôi khi bạn muốn tham khảo những thứ từ một mô-đun hiện có, chẳng hạn như thư viện. Trong trường hợp này, bạn có thể nhập một mô đun khác. Điều này được thực hiện thông qua từ khóa import, và có một vài hình thức khác nhau, mỗi hình thức có một ý nghĩa khác nhau.
* Loại đơn giản nhất là nhập khẩu cụ thể và có dạng import A = B. Tuyên bố này tạo ra một tham chiếu đến mô đun B (phải tồn tại), và liên kết nó với tên mới A:

module Helpers {  
 ...  
}  
module Mod {  
 import A = Helpers  
 method m() {  
 assert A.addOne(5) == 6;  
 }  
}

* Ở đây sử dụng A thay vì Helpers với method m(). Để sử dụng các thành phần của 1 module thì nó phải được khai báo trong đó hoặc được import.
* Dafny không hỗ trợ 2 module trùng tên cùng 1 lúc, vì vậy để đảm bảo ko xung đột thì A# Helpers.
* Export sets
* Import sẽ cho phép truy cập vào tất cả các khai báo (và các định nghĩa của chúng) từ mô-đun được Import.Để kiểm soát điều này chính xác hơn chúng ta có thể sử dụng các bộ Export.
* Mỗi bộ Export có thể có một danh sách các tờ khai từ mô-đun hiện tại, được đưa ra như provides( cung cấp) hoặc reveals( tiết lộ). Export không có tên được coi là Export mặc định cho mô đun đó và được sử dụng khi không có tập hợp được đặt tên rõ ràng.

module Helpers {  
 export Spec provides addOne, addOne\_result  
 export Body reveals addOne  
 export extends Spec  
 function method addOne(n: nat): nat   
 {  
 n + 1  
 }  
 lemma addOne\_result(n : nat)  
 ensures addOne(n) == n + 1  
 { }  
}

* Theo ví dụ trên thì có 3 bộ export:
* Spec: được đưa ra dưới dạng provides: chỉ cho phép truy cập vào function addOne, nhưng ko cho truy cập vào định nghĩa của nó
* Body: đưa ra dưới dang reveals cho phép truy cập vào thân function addOne
* Spec: Export mặc định, là mở rộng của bộ Spec, nó chỉ đơn giản cung cấp cho tất cả các tuyên bố Export mà Spec làm.

module Mod1 {  
 import A = Helpers`Body  
 method m() {  
 assert A.addOne(5) == 6; // succeeds, we have access to addOne's body  
 }  
 method m2() {  
 //A.addOne\_result(5); // error, addOne\_result is not exported from Body  
 assert A.addOne(5) == 6;  
 }  
}  
module Mod2 {  
 import A = Helpers`Spec  
 method m() {  
 assert A.addOne(5) == 6; // fails, we don't have addOne's body  
 }  
 method m2() {  
 A.addOne\_result(5);  
 assert A.addOne(5) == 6; // succeeds due to result from addOne\_result  
 }  
}  
module Mod3 {  
 import A = Helpers  
 method m() {  
 assert A.addOne(5) == 6; // fails, we don't have addOne's body  
 }  
}

* có thể sử dụng bộ Export để kiểm soát những định nghĩa kiểu nào có sẵn.Tất cả các khai báo kiểu (ví dụ loại mới, loại, loại dữ liệu, v.v.) có thể được xuất khẩu như provides or reveals.

module Helpers {  
 export provides f, T  
 export Body reveals f, T  
 type T = int  
 function f(): T { 0 }  
}  
module Mod {  
 import A = Helpers  
 function g(): A.T { 0 } // error, T is not known to be int, or even numeric  
 function h(): A.T { A.f() } // okay  
}

* Một khi xuất khẩu đã được nhập khẩu cho thấy một opaque type( loại mờ, ko rõ) trước đây, tất cả các sử dụng hiện tại của nó được biết là inner type.

module Mod2 {  
 import M = Mod  
 import A = Helpers`Body  
 function j(): int  
 ensures j() == 0 //succeeds  
 { M.h() }  
}

* ký hiệu đặc biệt "\*" có thể được đưa ra sau khi provides or reveals để chỉ ra rằng tất cả các khai báo phải được cung cấp hoặc tiết lộ.

module A {  
 export All reveals \* // reveals T, f, g  
 export Spec provides \* // provides T, f, g  
 export Some provides \* reveals g // provides T, f reveals g  
 type T = int  
 function f(): T { 0 }  
 function g(): int { 2 }  
}

* có thể cung cấp nhiều Export cùng một lúc để tạo ra một tập hợp Import.

module A {  
 export Justf reveals f  
 export JustT reveals T  
 type T = int  
 function f(): int { 0 }  
}  
module B {  
 import A`{Justf,JustT}  
 function g(): A.T { A.f() }  
}

* Export Consistency\_ Tính nhất quán trong Export
* trình bày một cái nhìn nhất quán của một module: bất cứ thứ gì xuất hiện trong một tuyên bố Export chính nó phải được exported.Xem lại ví dụ trước, chúng ta không thể tạo một tập xuất xuất hiện cho thấy f mà không tiết lộ T. Đây là lý do đơn giản mà chúng ta sẽ tạo ra một ràng buộc kiểu 0: T mà không thể giải quyết nếu T là mờ đục. Tương tự như vậy, chúng ta không thể tạo một tập xuất mà cung cấp hoặc tiết lộ f nếu chúng ta không ít nhất cũng cung cấp T.

module Helpers {  
 export provides f, T // good  
 export Body reveals f, T // good  
 export BadSpec reveals f, provides T // bad  
 export BadSpec2 provides f // bad  
 type T = int  
 function f(): T { 0 }  
}

* Vì chúng ta có thể xác định các module có chứa cả tờ khai import, export, chúng ta cần phải xuất các tờ khai từ các module ngoài để tạo một bộ xuất khẩu nhất quán. Tuyên bố từ các module ngoài không thể được đưa vào Export một cách trực tiếp, tuy nhiên việc import cung cấp chúng có thể.

module Mod {  
 export Try1 reveals h // error  
 export Try2 reveals h, provides A.f, A.T // error, can't provide these directly  
 export reveals h, provides A // good  
 import A = Helpers  
 function h(): A.T { A.f() }  
}

* Khi Import Mod chúng ta bây giờ cũng có được quyền truy cập đủ điều kiện vào những gì được cung cấp trong ImportA. Chúng tôi cũng có thể chọn để trực tiếp nhập khẩu này, để cung cấp cho họ một cái tên ngắn hơn.  
  module Mod2 {  
   import M = Mod  
   import MA = M.A  
   function j(): M.A.T { M.h() }  
   function k(): MA.T { j() }  
  }

1. [Opening Modules](http://rise4fun.com/Dafny/tutorialcontent/Modules)

Đôi khi tiền tố các thành phần của module mà bạn nhập với tên thật tẻ nhạt và xấu, ngay cả khi bạn chọn một cái tên ngắn khi nhập nó. Trong trường hợp này, bạn có thể Import Module nhập"opened", khiến tất cả các thành viên của nó khả dụng mà không cần thêm tên module

module Mod {  
 import opened Helpers  
 method m() {  
 assert addOne(5) == 6;  
 }  
}

Khi open 1 module thì các thành phần mới sẽ có mức độ ưu tiên thấp hơn nên chúng sẽ bị ẩn bởi local definitions (định nghĩa cục bộ). Điều này có nghĩa là nếu bạn định nghĩa 1 hàm địa phương gọi addOne(), thì hàm từ Helpers sẽ không có sẵn dưới tên đó nữa.

Khi các module được mở thì ràng buộc ban đầu với tên đó vẫn còn vì vậy bạn có thể sử dụng tên đã bị ràng buộc để nhận các thứ bị ẩn đi.

module Mod {  
 import opened Helpers  
 function addOne(n: nat): nat {  
 n + 2  
 }  
 method m() {  
 assert addOne(5) == 6; // this is now false,  
 // as this is the function just defined  
 assert Helpers.addOne(5) == 6; // this is still true  
 }  
}

* Nếu 2 module cùng được mở và các thành phần có cùng tên thì phải có tiền tố modul thì mới có thể gọi được. Từ khóa opend có thể được sử dụng với bất kì loại khai báo import nào kể cả module trừu tượng.

1. [Module Abstraction](http://rise4fun.com/Dafny/tutorialcontent/Modules)

Sử dụng 1 Import abstract module khi sử dụng cách thực hiện cụ thể không hiệu quả, cần 1 module thực hiện 1 số interface.

Trong Dafny viết import A:B có nghĩa là ràng buộc tên A nhưng thay vì nhận được chính xác module B thì nhận được bất kì module là sàng lọc của B. Thông thường module B có thể có các định nghĩa trừu tượng, các class vớiphương thức bodyless, hoặc không thích hợp để sử dụng trực tiếp. Bất kì sàng lọc nào của B có thể được sử dụng an toàn.

Mở đầu với:

module Interface {  
 function method addSome(n: nat): nat   
 ensures addSome(n) > n  
}  
module Mod {  
 import A : Interface  
 method m() {  
 assert 6 <= A.addSome(5);  
 }  
}

Nhận thấy: có thể biết chính xác hơn nếu biết rằng addSome thực sự tăng chính xác 1. Các module dưới đây có hành vi này. Hơn nữa hậu điều kiện được làm mạnh hơn nên chắc chắn đây là 1 sàng lọc của interface module.

module Implementation refines Interface {  
 function method addSome(n: nat): nat   
 ensures addSome(n) == n + 1  
 {  
 n + 1  
 }  
}

* Sau đó có thể thay thế Implementation cho A trong 1 module mới,

module Mod2 refines Mod {  
 import A = Implementation  
 ...  
}

Khi tinh chỉnh abstract import thành 1 khối cụ thể, module cụ thể phải là 1 sàng lọc rõ ràng về 1 trừu tượng.( đã được khai báo với refines)

1. [Module Ordering and Dependencies](http://rise4fun.com/Dafny/tutorialcontent/Modules) (Tổ chức và sự phụ thuộc)

Dafny không phải quá chú trọng về thứ tự các mô-đun xuất hiện, nhưng họ phải tuân theo một số quy tắc để được hình thành tốt. Theo nguyên tắc chung, cần phải có cách để đặt các mô-đun trong một chương trình sao cho mỗi cái chỉ đề cập đến những điều được định nghĩa trước khi nó xuất hiện trong văn bản nguồn. Điều đó không có nghĩa là các mô-đun phải được đưa ra theo thứ tự đó. Dafny sẽ tìm ra lệnh đó cho bạn, giả sử bạn không thực hiện bất kỳ tài liệu tham khảo thông tư nào. Ví dụ, điều này là khá rõ ràng vô nghĩa:

import A = B  
import B = A

Bạn có thể import tuyên bố ở cấp cuối, và bạn có thể nhập các mô-đun xác định cùng cấp:

import A = B  
method m() {  
 A.whatever();  
}  
module B { method whatever() {} }

Trong trường hợp này, mọi thứ đều được định nghĩa bởi vì chúng ta có thể đặt B đầu tiên, tiếp theo là import A, và cuối cùng là m (). Nếu không có lệnh, thì Dafny sẽ đưa ra một lỗi, phàn nàn về sự phụ thuộc theo chu kỳ.

Lưu ý rằng khi sắp xếp lại mô-đun và import, chúng phải được giữ trong cùng một mô-đun có chứa, cho phép loại bỏ một số cấu trúc mô bệnh lý. Ngoài ra, hàng nhập khẩu và mô-đun luôn luôn được coi là lần đầu tiên, ngay cả ở cấp cao. Điều này có nghĩa là những điều sau đây không được hình thành:

method doIt() { }  
module M {  
 method m() {  
 doIt();  
 }  
}

Bởi vì mô đun M phải được khai báo đầu tiên trước bất kỳ khai báo nào, chẳng hạn như các phương pháp. Để xác định các hàm toàn cầu như thế này, bạn có thể đặt chúng trong một mô đun (gọi là Globals) và mở nó vào bất kỳ mô đun nào cần chức năng của nó. Cuối cùng, nếu bạn import thông qua một đường dẫn, chẳng hạn như nhập khẩu A = B.C, thì điều này tạo ra sự phụ thuộc của A trên B, vì chúng ta cần phải biết B là gì (nó là trừu tượng hay gì).

1. [Name Resolution](http://rise4fun.com/Dafny/tutorialcontent/Modules)

(Todo: Những điều sau đây đã thay đổi trong Dafny. Mô tả ở đây nên được thay đổi để phản ánh các quy tắc mới.)

Khi Dafny nhìn thấy một cái gì đó giống như A.B.C, làm thế nào nó biết những gì từng phần đề cập đến? Quy trình Dafny sử dụng để xác định trình tự nhận dạng nào như thế này đề cập đến việc phân giải tên. Mặc dù các quy tắc có vẻ phức tạp, thường thì họ làm những gì bạn mong đợi. Dafny đầu tiên tìm kiếm nhận dạng ban đầu. Tùy thuộc vào những gì nhận diện đầu tiên đề cập đến, phần còn lại của định danh được tra cứu trong ngữ cảnh thích hợp. Các quy tắc đầy đủ như sau:

Các biến địa phương, các tham số và các biến ràng buộc. Đây là những thứ như x, y, và i trong var x ;, ... returns (y: int), và forall i :: ....

Các lớp, kiểu dữ liệu, và tên mô đun (được cung cấp đây không phải là chỉ một phần của định danh). Lớp cho phép các thành viên tĩnh của chúng được truy cập theo cách này, và các loại dữ liệu cho phép các nhà thầu của họ được truy cập. Các mô-đun cho phép bất kỳ thành viên nào của họ được gọi như thế này

Tên người xây dựng (nếu không rõ ràng). Bất kỳ loại dữ liệu nào cũng không cần chứng chỉ (do đó, tên kiểu dữ liệu tự nó không cần tiền tố), và cũng có một constructor được đặt tên duy nhất, có thể được chỉ dẫn bằng tên của nó. Vì vậy, nếu datatype List = Nhược điểm (Danh sách) | Nil là loại dữ liệu duy nhất tuyên bố các nhà xây dựng Cons và Nil, sau đó bạn có thể viết Cons (Nil). Nếu tên constructor không phải là duy nhất, thì bạn cần phải thêm tiền tố vào tên của kiểu dữ liệu (ví dụ List.Cons (List.Nil))). Điều này được thực hiện cho mỗi nhà xây dựng, không phải cho mỗi loại dữ liệu.

Các trường, các chức năng và các phương thức của lớp hiện tại (nếu trong một ngữ cảnh tĩnh, thì chỉ cho phép các phương thức và hàm tĩnh). Bạn có thể tham khảo các trường của lớp hiện tại hoặc như this.f hoặc f, giả định rằng tất nhiên là f đã không được ẩn bởi một trong những ở trên. Bạn có thể luôn luôn tiền tố này nếu cần, mà không thể được ẩn. (Lưu ý, một trường có tên là một chuỗi số phải luôn có một số tiền tố.)

Các hàm và phương thức tĩnh trong mô-đun bao bọc. Lưu ý, điều này chỉ đề cập đến các hàm và phương thức khai báo ở cấp mô đun, chứ không phải các thành viên tĩnh của một lớp được đặt tên.

Môđun đã mở được xử lý ở mỗi cấp độ, sau khi các tờ khai trong mô đun hiện tại. Mô đun đã mở chỉ ảnh hưởng đến các bước 2, 3 và 5. Nếu một tên mơ hồ được tìm thấy, một lỗi được tạo ra, thay vì tiếp tục xuống danh sách. Sau khi nhận dạng đầu tiên, các quy tắc về cơ bản là giống nhau, ngoại trừ trong bối cảnh mới. Ví dụ: nếu định danh đầu tiên là một mô-đun, thì định danh tiếp theo sẽ xem xét mô-đun đó. Mô-đun mở chỉ được áp dụng trong mô-đun được mở ra. Khi tìm kiếm trong mô-đun khác, chỉ những điều được tuyên bố rõ ràng trong mô-đun đó được xem xét.