CHƯƠNG 4

Thử nghiệm chương trình

Như đã trình bày trong chương trước, người ta thường sử dụng các *kỹ thuật tĩnh* (static techniques) và *kỹ thuật động* (dynamic techniques) trong quá trình V&V để kiểm tra tính đúng đắn của một sản phẩm phần mềm.

Chương này sẽ trình bày một phương pháp tĩnh là *khảo sát* (inspection) chương trình, với vai trò như là một phép chứng minh phi hình thức và, một phương pháp động là thử nghiệm(testing) chương trình.

I. Khảo sát

Khảo sát (hay thanh tra) là những cuộc họp nhằm mục đích xác minh một sản phẩm. Phần lớn các phương pháp sản xuất phần mềm đều ấn định trước những cuộc họp như vậy. Tùy theo bản chất của sản phẩm cần khảo sát, người ta nói về khảo sát *thiết kế toàn thể* (global design), khảo sát *thiết kế chi tiết* (detailed design), và khảo sát mã nguồn.

Một kịch bản mẫu (typical scenario) cho một khảo sát mã nguồn như sau :

1. Cần đến 4 người gồm một chủ tịch, một người lập trình, một người thiết kế và một khảo sát (đều là những chuyên gia về Tin học, riêng khảo sát phải có kiến thức chuyên môn về lĩnh vực ứng dụng của sản phẩm).
2. Các thành viên nhận chương trình nguồn và các đặc tả trước cuộc họp ít ngày để đọc và chuẩn bị.
3. Cuộc họp kéo dài khoảng 1 giờ 30 đến khoảng 2 giờ.
4. Trong quá trình họp khảo sát :
   - Người lập trình đọc và giải thích chương trình của mình, có thể đọc từng dòng lệnh một và trả lời các câu hỏi được đặt ra.
   - Chương trình được phân tích căn cứ trên một danh sách các lỗi sai (errors) thông dụng do khảo sát cung cấp.
5. Cuộc họp không sửa lỗi tìm thấy mà chỉ ghi nhận qua biên bản mà thôi. Chính người lập trình sẽ tự sửa lỗi sau khi họp xong.
6. Nếu khi khảo sát tìm thấy trong chương trình, nhiều khiếm khuyết (failures), hoặc nhiều lỗi trầm trọng thì phải tiếp tục khảo sát lần sau, sau khi sửa lỗi.

Một số kịch bản coi trọng việc tìm lỗi sai và khuyến khích việc chạy demo trực tiếp mã chương trình (hand made) nguồn : khảo sát mang đến cuộc họp cách tiến hành và các dữ liệu liên quan để mọi người tiến hành thử nghiệm. Người ta còn gọi cách thử nghiệm như vậy là *walk throughs* (chạy suốt).

Một số kịch bản lại coi trọng việc chứng minh không hình thức : khảo sát đề nghị xác minh các tính chất cho phép thử nghiệm tính đúng đắn của sản phẩm. Người ta nói đây là việc khảo sát căn cứ trên việc xác minh.

Việc kiểm lại (review) khác với khảo sát vì rằng việc kiểm lại không đòi hỏi phải họp : Sản phẩm được giao cho những người không tham gia vào việc lập trình, họ có những khuynh hướng đánh giá độc lập.

Có thể nói phương pháp khảo sát có hiệu quả đáng kể : những số liệu tìm thấy trong các văn bản ghi nhận khoảng 50% sai sót được phát hiện khi khảo sát. Những con số dưới đây (lấy từ tạp chí IEEE[3] năm 1992 của Dyer M. từ bài báo "Verification Based Inspection") cho thấy các sai số tìm thấy khi phát triển dự án 5 phần mềm của hãng IBM :

Dự án	*Khảo sát thiết kế toàn bộ*	*Khảo sát thiết kế chi tiết*	*Khảo sát mã*	*Thử nghiệm đơn vị*	*Thử nghiệm hệ thống*
1			50	25	25
2	4	13	49	17	17
3	20	27	10	20	23
4	20	26	22	18	36
5	10	18	24	24	24

Một phương pháp khác, gọi là *phương pháp phòng sạch* (Clean-room Methodology), thay vì thử nghiệm (testing), khuyến khích việc khảo sát (inspection) bằng cách xác minh (verification) trong quá trình sản xuất phần mềm. Sự phát triển phần là liên tiếp làm mịn (raffinement) sản phẩm. Mỗi giai đoạn, người ta tiến hành chứng minh tính đúng đắn (prouving) một cách chặt chẽ, đồng thời với các cuộc khảo sát, sao cho sản phẩm phần mềm không chứa sai sót.

Việc thử nghiệm chỉ được tiến hành khi xác minh hậu nghiệm (a posteriori) nhờ các phương pháp thống kê, nhằm đạt được mục đích đặt ra lúc đầu. Phương pháp phòng sạch do H.Mills xây dựng tại IBM, đã được áp dụng để sản xuất các phần mềm cỡ lớn.

[3] IEEE, đọc là eye-triple-ee, tên viết tắt của Institute of Elechtrric and Engineers.

II. Các phương pháp thử nghiệm

Phương pháp thử nghiệm là cho chạy chương trình từ một số dữ liệu thử được chọn trước. Phép thử nghiệm dùng cho cả hai quá trình xác minh và hợp thức hóa V&V, với điều kiện rằng chương trình là chạy được.

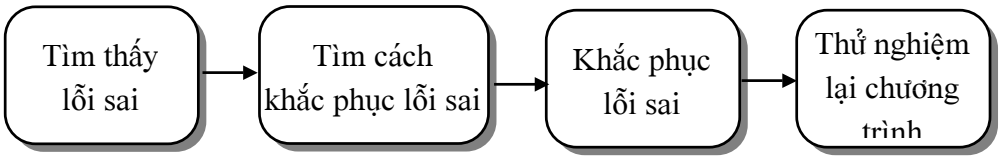
Việc thử nghiệm phân biệt :

1. Các phép chứng minh tính đúng đắn hay khảo sát mã nguồn mà không chạy chương trình, với quy trình "walkthroughs" bằng cách chạy demo (hand-made).
2. Chạy chương trình debugger để tìm sửa lỗi.

Các thử nghiệm và chạy debugger thường do các nhóm công tác khác nhau đảm nhiệm.

Để nâng cao hiệu quả, người ta thường phân công như sau : nhóm thử nghiệm là nhóm không lập trình, còn nhóm chạy debugger là nhóm tham gia lập trình ra sản phẩm. Quá trình debugger gồm nhiều giai đoạn :

1. Tìm thấy lỗi sai (Locate error)
2. Tìm cách khắc phục lỗi sai (Design error repair)
3. Khắc phục lỗi sai (Error repair)
4. Thử nghiệm lại chương trình (Re-teat program)



Hình 4.1. Quá trình debugger

Với những phương pháp lập trình truyền thống, quá trình V & V là khảo sát và thử nghiệm chương trình. Thực tế, việc thử nghiệm chiếm một phần đáng kể trong quá trình sản xuất phần mềm, chiếm khoảng từ 30% đến 50%, tùy theo bản chất của dự án Tin học.

II.1. Định nghĩa và mục đích thử nghiệm

Người ta đưa ra những định nghĩa sau đây :

- Một *thử nghiệm* là cho chạy (run) hay thực hiện (execution) một chương trình từ những dữ liệu được lựa chọn đặc biệt, nhằm để xác minh kết quả nhận được sau khi chạy là đúng đắn.
- Một *tập dữ liệu thử* là tập hợp hữu hạn các dữ liệu trong đó mỗi dữ liệu phục vụ cho một thử nghiệm.

- Mỗi *phép thử nghiệm* chỉ ra hoạt động từ việc thiết kế các tập dữ liệu thử, tiến hành thử nghiệm và đánh giá kết quả đến các giai đoạn khác nhau trong chu kỳ sống của phần mềm.
- *Người thử nghiệm* (hay *nhóm thử nghiệm*) có kiến thức chuyên môn Tin học có nhiệm vụ tiến hành phép thử nghiệm.
- Một *khiếm khuyết* (failure) xảy ra khi chương trình thực hiện cho ra kết quả không tương hợp với đặc tả của chương trình.
- Một lỗi sai (error) là một phần chương trình (lệnh) đã gây ra khiếm khuyết.

Người thử nghiệm có nhiệm vụ :

1. Tạo ra tập dữ liệu thử.
2. Triển khai các phép thử.
3. Lập báo cáo về kết quả thử nghiệm và lưu giữ.

Mục đích thử nghiệm là để :

1. Chứng minh rằng chương trình là đúng đắn

Để khẳng định tính đúng đắn của chương trình, cần tiến hành các thử nghiệm toàn bộ (exhaustive testing), đòi hỏi tập dữ liệu thử phải hữu hạn và có kích thước vừa phải sao cho đủ sức thuyết phục. Điều này trên thực tế rất khó thực hiện.

Sau đây là một tiêu chuẩn nổi tiếng của Dijkstra : "Các thử nghiệm cho phép chứng minh một chương trình là không đúng, bằng cách chỉ ra một phản ví dụ, tuy nhiên, không bao giờ có thể chứng minh được chương trình đó là đúng đắn".

2. Gây ra những khiếm khuyết của chương trình

Myers G. J. trong bài báo "The Art of Software Testing", Wiley 1979, đã định nghĩa thử nghiệm như sau :

"Phép thử nghiệm là cho chạy chương trình nhằm tìm ra những sai sót".

Từ đó, thường người ta nói về "thử nghiệm phá hủy" (destructive testings). Mục đích của những phép thử như vậy là tập trung tìm ra các lỗi sai từ những khiếm khuyết do người lập trình phạm phải. Người thử nghiệm tiến hành với mục đích nghịch (negative) : phép thử là thành công nếu tìm ra được khiếm khuyết, là thất bại trong trường hợp ngược lại. Việc thử nghiệm kiểu này thường được áp dụng trong quá trình viết chương trình.

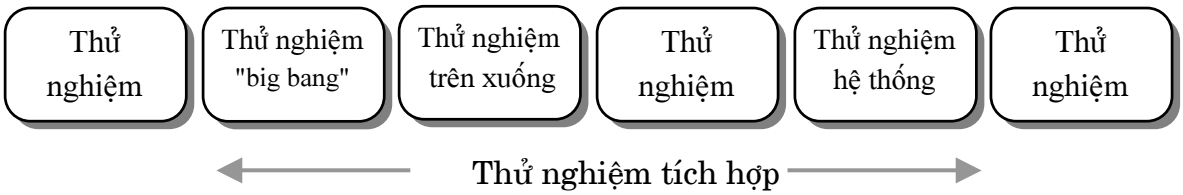
3. Đưa ra đánh giá tĩnh (static evaluation - static benchmark) **về chất lượng của chương trình.**

Người ta sử dụng phương pháp "thử nghiệm tĩnh" (static testing) cho mục đích này. Trong phương pháp phòng trắng, người ta chỉ tiến hành những phép thử tĩnh,

nhằm mục đích vừa đảm bảo công việc của người lập trình vừa đánh giá sự tin cậy của sản phẩm vận hành.

II.2. Thử nghiệm trong chu kỳ sống của phần mềm

Người ta phân biệt nhiều phương pháp thử nghiệm, tương ứng với các giai đoạn sản xuất phần mềm khác nhau.



Hình 4.2. Nhiều phương pháp thử nghiệm

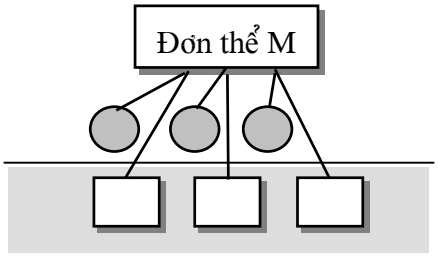
II.2.1. Thử nghiệm đơn thể

Thử nghiệm đơn thể (Module testing), hay thử nghiệm đơn vị (Unit testing) do người lập trình tự tiến hành. Phương pháp này hay được sử dụng trong lập trình cấu trúc (top-down programing). Các phương pháp thử nghiệm khác do người thử nghiệm tiến hành.

Giả sử gọi M là một đơn thể cần thử nghiệm riêng biệt. Khi đó, xảy ra hai trường hợp như sau :

Trường hợp 1 : những đơn thể do M gọi tới không có mặt lúc thử nghiệm.

Khi đó, những đơn thể do M gọi tới vắng mặt phải được thay thế bởi các chương trình cùng một giao diện với M. Các chương trình này thực hiện đúng chức năng mà chúng đại diện cho đơn thể vắng mặt và chúng được gọi là các trình stubs ("cuống").



Hình 4.3. Các đơn thể vắng mặt được thay bởi các trình stubs

Ví dụ, nếu đơn thể đang được thử nghiệm gọi một thủ tục sắp xếp ở đầu :

```
Procedure Sort (T: Array ; n: Integer);
```

người ta có thể sử dụng trình Stub :

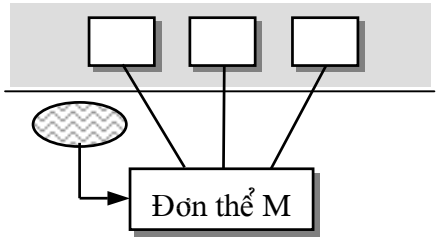
```
Procedure Sort (T: Array ; n: Integer) ;
   Writeln ('Dãy cần sắp xếp là : ') ;
   for i:= 1 to n do writeln (T[i]) ;
```

```
for i:= 1 to n do readln (T [i]) ;
```

Tiếp theo, người thử nghiệm sẽ tiến hành sắp xếp dãy đã nhập bằng tay để thay thế cho thủ tục sắp xếp vắng mặt.

Trường hợp 2 : những đơn thể gọi tới M không có mặt lúc thử nghiệm.

Khi đó, đơn thể gọi tới M nhưng vắng mặt phải được thay thế bởi một chương trình, được gọi là trình driver. Trình driver gọi M để M thực hiện trên các dữ liệu thuộc tập dữ liệu thử, sau đó ghi nhận các kết quả tính được bởi M để so sánh với các kết quả chờ đợi.



Hình 4.4. Dùng trình driver để gọi thực hiện M

Số lượng các trình stubs và các trình drivers cần thiết để tiến hành thử nghiệm các đơn thể phụ thuộc vào thứ tự các đơn thể được thử nghiệm.

II.2.2. Thử nghiệm tích hợp

Thử nghiệm tích hợp vừa nhằm tạo mối liên kết giữa các đơn thể, vừa được tiến hành đối với những đơn thể lớn hình thành hệ thống chương trình hoàn chỉnh. Có nhiều phương pháp thử nghiệm tích hợp.

1. Phương pháp "big bang"

Người ta xây dựng mối liên hệ giữa các đơn thể để tạo thành một phiên bản hệ thống hoàn chỉnh, sau đó thử nghiệm phiên bản này.

Như vậy người ta cần đến nhiều trình drivers, mỗi trình driver cho một đơn thể, một trình stubs cho tất cả các đơn thể của hệ thống, trừ đơn thể chính phải được thử nghiệm bằng phương pháp thử nghiệm đơn vị.

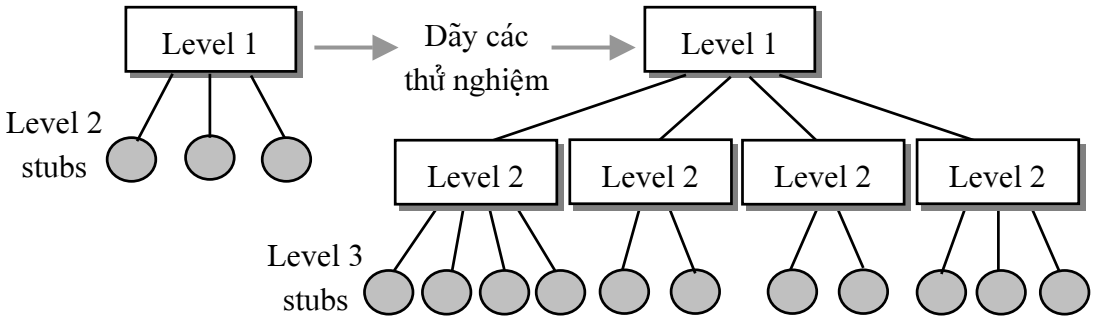
Phương pháp "big bang" nguy hiểm : tất cả các sai sót có thể đồng thời xuất hiện, việc xác định từng lỗi sai sẽ gặp khó khăn. Hơn nữa phương pháp này đòi hỏi một lượng tối đa các trình drivers và các trình stubs. Vì vậy thường người ta sử dụng các phương pháp tích hợp từ trên xuống, hay từ dưới lên.

2. Phương pháp thử nghiệm từ trên xuống

(Descendant hay Top-down Testing Method)

Bắt đầu thử nghiệm đơn thể chính, sau đó thử nghiệm chương trình nhận được từ sự liên kết giữa đơn thể chính và các đơn thể được gọi trực tiếp từ đơn thể chính, v.v...

Phương pháp này chỉ cần dùng một trình driver duy nhất cho đơn thể chính, nhưng cần một trình stub cho mỗi đơn thể còn lại.

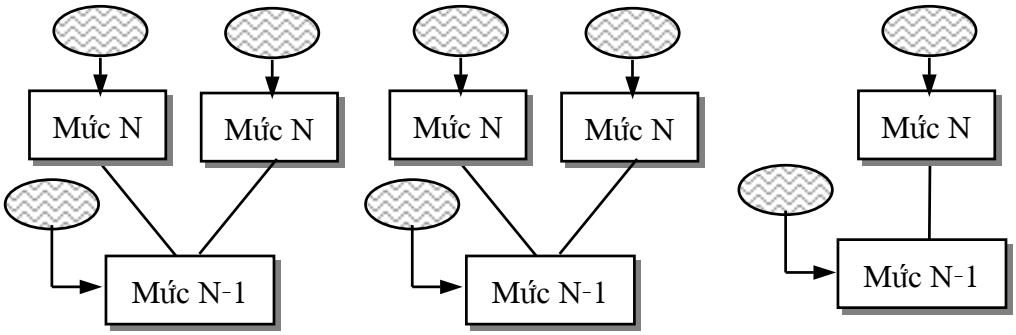


Hình 4.5. Phương pháp thử nghiệm từ trên xuống

3. Phương pháp thử nghiệm từ dưới lên

(Ascendant hay Bottom-Up Testing Method)

Bắt đầu thử nghiệm các đơn thể không gọi đến các đơn thể khác, sau đó các chương trình nhận được bởi sự liên kết giữa một đơn thể chỉ gọi đến các đơn thể đã được thử nghiệm với các đơn thể này, v.v . . Phương pháp này đòi hỏi mỗi đơn thể một trình driver, nhưng không cần trình stub.



Hình 4.6. Thử nghiệm từ dưới lên

Phương pháp tiến tỏ ra ưu điểm hơn phương pháp lùi, do các trình driversử dụng dễ viết hơn các trình stubs và có các công cụ để tạo sinh tự động các trình drivers. Mặt khác, thứ tự tích hợp thường bị ràng buộc bởi thứ tự có mặt của các đơn thể.

II.2.3. Thử nghiệm hệ thống

Vấn đề là thử nghiệm phần mềm hoàn chỉnh và phần cứng để đánh giá hiệu năng, độ an toàn, tính tương hợp với các đặc tả, v.v . . .

Những thử nghiệm này đòi hỏi nhiều thời gian. Người ta nói đến thử nghiệm chấp nhận (Acceptance testing), là những thử nghiệm phù hợp với sản phẩm cuối cùng qua hợp đồng đã ký với khách hàng (nhiều khi việc thử nghiệm này do khách hàng tiến hành), còn được gọi là thử nghiệm alpha và cuối cùng là thử nghiệm cài đặt (Setup Testing), là thử nghiệm đối với sản phẩm cuối cùng, tiến hành tại vị trí

của khách hàng (với các máy tính và hệ điều hành họ đang sử dụng). Người ta gọi các thử nghiệm cho phiên bản đầu tiên của phần mềm do khách hàng được lựa chọn đặc biệt tiến hành là thử nghiệm beta.

II.2.4. Thử nghiệm hồi quy

Người ta còn gọi các thử nghiệm tiến hành sau khi sửa lỗi là thử nghiệm hồi quy, hay thoái lui (regresgion testing) nhằm để xác minh nếu các sai sót khác không được xử lý khi sửa chữa. Kiển thử nghiệm này hay được dùng trong khi bảo trì. Để tiến hành hiệu quả các thử nghiệm này, cần lưu giữ lại những thử nghiệm đã làm trong quá trình sản xuất phần mềm, điều này giúp cho việc xác minh tự động các kết quả thử nghiệm thoái lui. Khó khăn gặp phải là trong số những thử nghiệm đã dắt dẫn, cần phải chọn những thử nghiệm nào chothử nghiệm thoái lui. Phương cách người ta hay làm là kết hợp mỗi lệnh của chương trình với tập hợp các thử nghiệm làm chạy chương trình.

II.3. Dẫn dắt các thử nghiệm

Việc dẫn dắt các thử nghiệm bao gồm :

- Xác định kích thước của tập dữ liệu thử (vấn đề kết thúc các TN).

- Lựa chọn các dữ liệu cần thử nghiệm.

- Xác định tính đúng đắn hay không đúng đắn của các kết quả nhận được sau khi thực hiện chương trình đối với các dữ liệu của tập dữ liệu thử (Vấn đề lời tiên tri - oracle).

Việc dẫn dắt các thử nghiệm kèm theo việc viết các chương trình bổ trợ như là các stubs và các drivers.

Vấn đề lời tiên tri

Một phép thử nghiệm (check program)

Một tập hợp hữu hạn các giá trị đưa vào.

Một tập hợp hữu hạn các cặp (Giá trị đưa vào, kết quả tương ứng).

Trong trường hợp 1, việc phát hiện ra các khiếm khuyết phải được làm bằng tay (by hand), từ đó dẫn đến một công việc xem xét kỹ lưỡng các kết quả mất thì giờ và mệt mỏi (làm hạn chế kích thước các tập dữ liệu thử). Có hai kiểu sai sót xảy ra khi xem xét :

- Một kết quả sai lại được xem như là đúng.

- Một kết quá đúng có thể được hiểu là sai.

Để tránh xác minh bằng tay, cần phải có những đặc tả khả thi, hay một phiên bản khác của chương trình (điều này có nguy cơ làm lan truyền sai sót từ phiên bản này sang phiên bản khác).

Trong trường hợp thứ hai, chính chương trình đang chạy tự phát hiện ra các khiếm khuyết, vấn đề là tìm ra được các giá trị đưa ra kết quả tương ứng với giá trị đưa vào. Điều này có thể làm " bằng tay " với một đặc tả khả thi, với một phiên bản của chương trình, với cùng những vấn đề đã gặp trong trường hợp đầu. Người ta cũng có thể vận dụng các phép thử cũ đã lưu giữ.

Chú ý rằng dùng chương trình xác minh tính đúng đắn của kết quả không luôn luôn đơn giản : nếu xảy ra có nhiều cái ra đúng tương ứng với mọt cái vào thì phải đặt kết qủa do chương trình tính ra dưới dạng quy tắc trước khi xác minh tính nhất quán với kết quả dự kiến trong tập dữ liệu thử. Điều này không phải luôn luôn làm được. Chẳng hạn làm sao có thể xác minh được rằng mã sinh ra bởi một trình biên dịch là đúng đắn, nếu chỉ thử nghiệm mã đó mà thôi ?

II.4. Thiết kế các phép thử phá hủy (Defect Testing)

II.4.1. Các phương pháp dựa trên chương trình

Các phương pháp này còn được gọi là phương pháp có cấu trúc (Structural Testing) hay thử nghiệm hộp trắng (white-box hay glass-box).

Mỗi chương trình tương ứng với một sơ đồ khối gồm các cấu trúc lựa chọn và các cấu trúc khối là một dãy tối đa các lệnh thực hiện (gồm các lệnh gán, lệnh gọi chương trình con, các lệnh vào-ra...) mà không có lệnh rẽ nhánh. Người ta gọi các *khối lệnh* là các đầu vào sơ đồ khối và các *quyết định* là các cung đi ra từ một cấu trúc lựa chọn.

a) Phủ các lệnh (các đỉnh)

Một phép thử là phủ (trùm) hết các lệnh của một chương trình nếu làm cho mỗi lệnh của nó được thực hiện. Đây là một tiêu chuẩn tối thiểu : Người ta không xét những thử nghiệm mà mỗi lệnh của chương trình không được thưc hiện ít nhất một lần.

Chú ý rằng tiêu chuẩn này không phải luôn luôn thỏa mãn bằng một chương trình có thể chứa các lệnh mà không thể được thực hiện.

b) Phủ các quyết định (các cung)

Một phép thử phủ các quyết định nếu trong khi thực hiện, mỗi cung của sơ đồ tổ chức của chương trình được duyệt qua ít nhất một lần : nghĩa là nếu mỗi phép chọn được thực hiện ít nhất một lần cho mỗi giá trị có thể (thuê hay fals e trong trường hợp ghép rẽ nhánh logic).

Như vậy, tiêu chuẩn này không phải luôn cần phải thỏa mãn.

Ví dụ : `if A > 0 then if A ≥ 0 then...else...`

c) Phủ các điều kiện

Ta xét một chương trình chứa cấu trúc rẽ nhánh logic gồm các lệnh not, end và or. Một phép thử phủ các điều kiện nếu việc thực hiện chương trình kéo theo sự tính giá trị của biểu thức này cho mọi giá trị logic có thê. Như vậy một biểu thức có hai toán hạng P, Q sẽ được tính toán với :

	A	B
	`true`	`true`
	`true`	`false`
	`false`	`true`
	`false`	`false`

Phép phủ các điều kiện cho phép củng cố phép phủ các quyết định. Ví dụ có thể phủ các quyết định bằng cách thực hiện phép lựa chọn P và Q với :

P = **`true`**, Q = **`true`** và P = **`false`**, Q = **`false`**,

điều này không cho phép phân biệt phép rẽ nhánh A or B.

d) Phủ các lộ trình thực hiện chương trình (path testing)

Một phép thử phủ các lộ trình chạy chương trình nếu gây ra việc thực thi mỗi lộ trình thực hiện chương trình. Không tồn tại phép thử như vậy nếu chương trình có vô hạn lộ trình thực hiện trong trường hợp tổng quát. Thông thường người ta xây dựng phép thử phủ các lộ trình thực hiện có số lượng ≤ một hằng đã cho.

e) Xác định dữ liệu cho phép phủ lộ trình thực hiện đặc biệt

Giả thiết rằng với mọi lệnh P của chương trình và mọi quyết định S, có thể tính ptpre (P, S), điều kiện đầu yếu nhất ứng với P và S. Người ta có thể với mọi lộ trình của chương trình, tính được một công thức E sao cho các dữ liệu của chương trình thỏa mãn E nếu và chỉ nếu việc thực hiện của chương trình đi theo lộ trình đã chọn.

Đặc biệt, E không là sai nếu và chỉ nếu lộ trình đã chọn là lộ trình thực thi. Như vậy chỉ cần tìm ra các dữ liệu làm thỏa mãn E để có phép thử phủ lộ trình đã chọn. Điều này có thể thực hiện bằng ta, hay chứng minh một cách sáng tạo công thức xE.

Phương pháp này được dùng để định nghĩa phép thử phủ các quyết định của một chương trình :

- Lựa chọn một tập hợp các lộ trình phủ các quyết định.
- Với mỗi lộ trình, tính điều kiện đầu yếu nhất tương ứng (hoặc một điều kiện đầu mạnh hơn).
- Tìm các dữ liệu thỏa mãn các điều kiện điều này.

f) Phủ các luồng dữ liệu

Với mỗi biến của chương trình, người ta gọi định nghĩa là một trường hợp của biến đó, một giá trị được gán cho biến (ví dụ : `x:=1, readln(x)` ...). Người ta gọi sử dụng là một trường hợp mà giá trị của biến được sử dụng (ví dụ : `y:= x+y` đối với biến x).

Trong các sử dụng, người ta phân biệt các sử dụng trong các lệnh không phải là lựa chọn, gọi là C- sử dụng, với C : calculus, các sử dụng trong các lệnh lựa chọn, gọi là P- sử dụng, với P : Predicate.

Một phép thử là phủ các C-sử dụng nếu với mỗi biến x, gây ra việc thực thi mới lộ trình giữa một định nghĩa x và một C-sử dụng đầu tiên cũa x.

Một phép thử là phủ các P-sử dụng nếu, với mỗi biến x gây ra việc thực thi mỗi lộ trình giữa một định nghĩa x, và một giá trị lựa chọn.

II.4.2. Các phương pháp dựa trên đặc tả

Những phương pháp này còn được gọi là thử nghiệm chức năng (funchương trìnhional testing), ha thử nghiệm này, người ta không chú ý đến chương trình, mà chỉ làm việc với đặc tả chức năng của chương trình. Người ta có thể thiết kế tập dữ liệu thử trước khi viết chương trình.

a) Các thử nghiệm toàn thể (Exhaustive Testing)

Người ta thử nghiệm chương trình với tất cả dữ liệu có thể về mặt lý thuyết, điều này chỉ làm được nếu tập hợp dữ liệu thử là hữu hạn. Thực tế, ngay cả khi tập hợp dữ liệu là hữu hạn thì thời gian thực hiện chương trình cho các thử nghiệm toàn thể là quá lớn trong phần lớn trường hợp.

Ví dụ :

1. Tính $\sqrt{x}$, với x nguyên giữa 0 và 2^{31}

Với thời gian một thử nghiệm là 1s, khi đó mất 2^{31} = 2147483648 s.

Một năm có 365 x 24 x 3600s = 31536000s.

Vậy thời gian một thực nghiệm toàn thể là ≈ 68 năm.

2. Thử nghiệm phép cộng các só nguyên giữa 0 và 2^{31}

Thời gian thử nghiệm một phép cộng là 1 μs. Số lượng dữ liệu là :

2^{31} x 2^{31} = 2^{62} ≈ 9.22 x 10^{18}

Thời gian thử nghiệm toàn thể là trên 292 471 năm.

b) Các thử nghiệm bởi các lớp tương đương (Equivalence partioning)

Nguyên lý : Phân hoạch tập hợp dữ liệu thành một số hữu hạn lớp và lựa một phân tử (hay một mẫu phân tử) trong mỗi lớp. Người ta đặt trong cùng một lớp các

dữ liệu được cho là phù hợp với chương trình theo cách đặc tả. Những dữ liệu này có thể cùng gây ra khiếm khuyết trong cùng tình huống.

Chú ý cần thử nghiệm các dữ liệu nằm ở phạm vi giáp ranh giữa các lớp tương đương để phát hiện các lỗi sai kiểu ≤ thường lẫn với <, v.v . . .

c) Thử nghiệm định hướng bởi cú pháp (Syntax Controlled Testing)

Khi dữ liệu là tập hợp các chuỗi ký tự (các ngôn ngữ lập trình), chúng được đặc tả bởi các ôtomat hữu hạn, hoặc bởi các văn phạm phi ngữ cảnh.

Ví dụ, nếu phần mềm được thử nghiệm có tính tương tác qua lại, như các hệ điều hành, thì tập hợp dãy các hành động có thể được định nghĩa bởi một ôtômat hữu hạn.

Người ta có thể định nghĩa các tập dữ liệu thử phủ các trạng thái đạt được, các cung, các lộ trình có độ dài bị chặn, v.v . . .

Khi tập hợp dữ liệu được định nghĩa bởi một văn phạm vi ngữ cảnh, người ta có thể xây dựng phép thử phủ các quy tắc của văn phạm (mỗi quy tắc được áp dụng ít nhất một lần để tiến hành một thử nghiệm).

Chú ý rằng lúc này, người ta chỉ có thể nhận được dữ liệu đúng, việc nhận được các dữ liệu sai bởi cùng phương pháp cần thiết phải viết một văn phạm sản sinh ra tập hợp các dữ liệu sai, điều này lại là một vấn đề hóc búa (vì rằng bù của một ngôn ngữ PNC chưa chắc đã là PNC).

d) Các thử nghiệm ngẫu nhiên (Random Testing)

Đây là tập dữ liệu thử sử dụng các dữ liệu lấy ngẫu nhiên, tuân theo luật xác suất, chẳng hạn luật đồng đều, dễ tiến hành nhưng thường là kém hiệu quả.

II.4.3. Kết luận

Hiện nay, người ta thường xây dựng phép thử nghiệm bằng cách phối hợp các thử nghiệm chức năng và thử nghiệm cấu trúc : người ta bắt đầu thử nghiệm chức năng trước (ngay khi đặc tả yêu cầu), sau đó hoàn thiện phép thử nghiệm bởi các tiêu chuẩn cấu trúc (bao bọc các lệnh, bao bọc các quyết định...) khi có được chương trình.

II.4.4. Các tiêu chuẩn kết thúc thử nghiệm

Vấn đề đặt ra là *khi nào thì kết thúc thử nghiệm* ? hay cụ thể hơn là *xác định phạm vi thử nghiệm như thế nào* ?

Nếu kết thúc thử nghiệm sớm thì có thể chưa tìm hết lỗi trong chương trình. Còn nếu kết thúc muộn quá thì lại nâng cao giá thành sản phẩm. Sau đây là một số tiêu chuẩn :

1. *Dừng khi không còn gây ra được khiếm khuyết.*

Thường thì một chương trình lớn bao giờ cũng có lỗi, tiêu chuẩn này tỏ ra phi thực tế, hơn nữa mâu thuẫn với mục đích của các thử nghiệm phá hủy.

2. *Dừng khi thời gian (hay kinh phí) gia hạn cho thử nghiệm đã hết.*

Để tiêu chuẩn này có hiệu lực thì phải định lượng được tập hợp các dữ liệu thử trước khi tiến hành thử nghiệm.

3. *Căn cứ vào kinh nghiệm của các dự án tương tự đã hoàn tất.*

Một phép thử nghiệm bao bọc các quyết định (hay 80% của các cung) không gây ra khiếm khuyết. Vấn đề : Lưa chọn tùy tiện của tiêu chuẩn.

4. *Thử nghiệm chừng 70 sai sót không được phát hiện hay sau một thời hạn 3 tháng không xảy ra.*

Vấn đề : Ước lượng số lượng sai sót trong chương trình, ước lượng tỷ lệ % các sai sót được phát hiện bởi thử nghiệm, ước lượng tỷ lệ % sai sót phạm phải trong các giai đoạn phát triển phần mềm và tại giai đoạn thử nghiệm mà những sai sót này được phát hiện.

5. *Thử nghiệm đến khi số lượng sai sót tìm thấy không còn giảm theo một cách có ý nghĩa nữa.*

Vấn đề : Làm sao ước lượng được số sai sót đã giảm theo cách có ý nghĩa ?

6. *Phương pháp các đột biến (Mutant method)*

Người ta thay đổi chương trình bằng cách đưa vào các lỗi. Các chương trình bị thay đổi được gọi là các "đột biến". Một phép thử là "tốt" nếu diệt được 100% (95%, v.v . . .) các "đột biến" đó.

Vấn đề : Các sai sót đưa vào có phải là những sai sót thực tiễn (có thực)?

II.5. Các phép thử nghiệm thống kê

II.5.1. Mở đầu

Các phép thử nghiệm thống kê (Statistical Testing) nhằm để đo độ tin cậy (reliability) của phần mềm, nghĩa là đo xác suất chạy ổn định và đúng đắn trong nhứng điều kiện sử dụng cho trước. Các thử nghiệm phá hủy không cho phép đánh giá được tính tin cậy của một chương trình vì rằng các thử nghiệm phá hủy không tính đến các điều kiện sử dụng như phương pháp này.

Người ta gọi *khiếm khuyết* (failure) là những hiện tượng bất thường xảy ra làm hệ thống đang thực thi dẫn đến những hiệu quả không phù hợp với đặc tả ban đầu. Một khiếm khuyết có thể xảy ra do phần cứng hoặc do một sai sót trong chương trình. Sau đậy, người ta chỉ quan tâm đến những khiếm khuyết do lỗi phần mềm gây nên.

Trong những điều kiện sử dụng đã cho, sự xuất hiện thường xuyên các khiếm khuyết do các sai sót khác nhau gây ra là rất biến động : một số sai sót gây ra thường xuyên các khiếm khuyết, những sai sót khác thì rất hiếm, có thể không bao giờ xảy ra trên thực tế.

Việc thực thi một phần mềm với một dữ liệu cố định trước là một quá trình có tính xác định gây ra hoặc là một kết quả đúng, hoặc là một khiếm khuyết. Nếu người ta ở trong những điều kiện sử dụng chương trình, mỗi dữ liệu có thể được của chương trình sẽ cho một xác suất nào đó.

Tập hợp các dữ liệu cùng xác suất sử dụng như vậy được gọi là một mẫu sử dụng (use pattern) của chương trình. Từ một mặt cắt sử dụng đã cho, người ta định nghĩa xác suất một lần chạy cho một kết quả đúng và xác suất một khiếm khuyết, còn được gọi là tỷ suất khiếm khuyết.

Với một mô hình độc lập với thời gian, người ta định nghĩa độ tin cậy (reliability) của một chương trình như là xác suất của sự kiện " lần chạy sau của chương trình là đúng ", nghĩa là 1,xác suất của một khiếm khuyết.

Với một mô hình phụ thuộc thời gian, người ta định nghĩa độ tin cậy như là một xác suất của sự kiện "chương trình chạy đúng đắn trong thời gian t". Lúc này độ tin cậy là một hàm của thời gian.

Các mô hình phụ thuộc vào thời gian thường được sử dụng cho các phần mềm tương hỗ (như là các hệ điều hành). Tiếp theo đây, người ta sẽ chỉ khai triển các mô hình độc lập với thời gian.

Khi xuất hiện một khiếm khuyết, nếu là một khiếm khuyết về phần cứng, thì phải sửa chữa, nếu là một khiếm khuyết về phần mềm thì phải chạy trình sửa lội debugger.

Sửa chữa các hư hỏng thuộc về phần cứng là để thay thế những chi tiết hư hỏng, thiết lập lại sự vận hành ổn định của thiết bị như trước. Còn chạy trình debugger là để sửa các lỗi về thiết kế, tăng độ tin cậy của phần mềm.

Thường người ta sử dụng đại lượng liên quan đến độ tin cậy là số lần sử dụng trung bình cho đến khi xảy ra khiếm khuyết (đối với mô hình độc lập với thời gian), hoặc sử dụng sau một thời gian trung bình nào đó đến khi xảy ra khiếm khuyết (đối với mô hình phụ thuộc vào thời gian).

Đại lượng liên quan đến độ tin cậy MTTF (Mean Time To Failure : thời gian trung bình để xảy ra khiếm khuyết) được tính như sau :

Trong mô hình độc lập với thời gian :

Độ ổn định = xác suất một lần chạy đúng.

= 1 - xác suất một khiếm khuyết.

MTTF = số lần sử dụng trung bình cho đến khi xảy ra khiếm khuyết.

= 1 / xác suất một khiếm khuyết.

= 1/ (1 - Độ ổn định).

Tỷ suất khiếm khuyết là nghịch đảo của MTTF.

II.5.2. Ước lượng độ ổn định của một phần mềm

Để ước lượng độ ổn định hay khả năng vận hành thông suốt (reliability) của một phần mềm, người ta căn cứ vào kết quả của các phép thử nghiệm thống kê, nghĩa là các thử nghiệm ngẫu nhiên thùy theo mẫu sử dụng đã chọn.

a) Phương pháp trực tiếp

Giả thiết rằng trong khi tiến hành n phép thử, người ta gặp d khiếm khuyết. Ta có thể ước lượng độ ổn định của phần mềm đang xét bởi biểu thức :

$$1 - d / n$$

Phương pháp này chỉ có thể đưa ra một ước lượng tốt về độ ổn định nếu số các khiếm khuyết d là có nghĩa (chẳng hạn độ tin cậy là 1 nếu khi thử nghiệm không xảy ra khiếm khuyết nào, điều này không có nghĩa).

Hơn nữa, nếu sau khi thử nghiệm, mà chạy trình debugger, thì chương trình sẽ bị thay đổi và việc ước lượng sẽ chỉ còn hợp thức một cách có điều kiện khi giả thiết về chất lượng của quá trình debugger.

b) Phương pháp thử nghiệm giả thuyết (Hypothesis Testing)

Vấn đề là xây dựng một tập hợp các phép thử nghiệm mà kết quả được ấn định trước cho phép khẳng định hay bác bỏ độ ổn định của phần mềm đang xét có một giá trị R với một độ tin cậy x%. R và x thoả mãn :

$$0 < R < 1 \text{ và } 0 < x < 100$$

Các tham số R và x cũng như quy cách về kết quả được cố định trước. Người ta nói chương trình là được kiểm nghiệm nếu có độ ổn định R.

Cho c = x/100, ta có :

1 – c = xác suất cho một sản phẩm có độ ổn định thấp hơn độ ổn định R.

CHƯƠNG 5

Đặc tả phần mềm

I. Mở đầu đặc tả phần mềm

I.1. Khái niệm về đặc tả phần mềm

I.1.1. Đặc tả phần mềm là gì ?

Đặc tả (specification) được định nghĩa trong từ điển tiếng Việt (1997) : "*Mô tả thật chi tiết một bộ phận đặc biệt tiêu biểu để làm nổi bật bản chất của toàn thể*".

Theo Computer Dictionary của Microsoft Press® (1994), đặc tả là *sự mô tả chi tiết : Về mặt phần cứng, đặc tả cung cấp thông tin về các thành phần, khả năng và yếu tố kỹ thuật của máy tính. Về mặt phần mềm, đặc tả mô tả môi trường hoạt động và chức năng của chương trình.*

Theo IBM Dictionary of Computing (1994), đặc tả là (1) *một dạng thức văn bản chi tiết cung cấp các mô tả xác định về một hệ thống nhằm để phát triển hay hợp thức hoá*. (2) Trong lĩnh vực phát triển hệ thống, đặc tả là *mô tả cách thiết kế, cách bố trí thiết bị và cách xây dựng chương trình cho hệ thống.*

Như vậy, đặc tả là sự mô tả các đặc trưng nhằm diễn đạt các yêu cầu và các chức năng của một sản phẩm phần mềm cần thiết kế. Đặc tả liên quan đến các đối tượng, các khái niệm và các thủ tục nào đó cần đến khi phát triển chương trình. Đặc tả có các đặc trưng :

- Tính chính xác (Correctness)
- Tính trừu tượng (Abstraction)
- Tính chặt chẽ về mặt Toán học (Rigorousness)

I.1.2. Các phương pháp đặc tả

Người ta thường sử dụng 3 phương pháp đặc tả : đặc tả phi (không) hình thức, đặc tả hình thức và đặc tả hỗn hợp.

Đặc tả phi hình thức (informal specification) được diễn đạt bằng ngôn ngữ tự nhiên và toán học. Tuy phương pháp đặc tả này không chặt chẽ nhưng dễ hiểu và dễ diễn đạt. Ta thường sử dụng khi cần phát biểu các bài toán, các yêu cầu ban đầu.

Ví dụ 1 :

1. Tìm nghiệm của phương trình f(x) = 0 với f(x) là một đa thức có bậc cho trước sao cho với giá trị thực x thì f(x) có giá trị bằng 0.
2. Biến đổi mo một ma trận vuông A cấp n × n về dạng tam giác trên, nghĩa là ma trận A có các phần tử nằm phía trên đường chéo chính thì bằng 0.

Đặc tả hình thức (formal specification) được diễn đạt bằng ngôn ngữ đại số và logic toán, rất chặt chẽ, chính xác và không nhấp nhằng (non-ambiguous).

Ví dụ 2

1. Tìm nghiệm của phương trình f(x) = 0 với f(x) là một đa thức có bậc cho trước sao cho với giá trị thực x thì f(x) có giá trị bằng 0.
2. Biến đổi mo một ma trận vuông A cấp n × n về dạng tam giác trên, nghĩa là ma trận A có các phần tử nằm phía trên đường chéo chính thì bằng 0.

Các tính chất của đặc tả hình thức

- đặc tả mô tả những cái phải làm nhưng không phải mô tả làm như thế nào.
- Lập trình thể hiện tường minh việc lựa chọn cách khai triển : nghiên cứu thuật giải, cách viết công thức...
- Đặc tả cho phép diễn tả đầy đủ một vấn đề, giảm tối thiểu tính phức tạp của hệ thống đang xét.
- Đặc tả phải cho phép kiểm tra được quá trình phát triển phần mềm (chất lượng và tính tin cậy)

Đặc tả hình thức liên quan đến :

- Cấu trúc dữ liệu và các hàm (kiểu dữ liệu)

- Thời gian

- Thao tác

- Đơn thể hay đối tượng.

Tính đại số căn cứ trên việc định nghĩa các kiểu dữ liệu, tính hiệu quả đại số được xác định bởi các công cụ toán học, đại số và logic.

Đặc tả hỗn hợp (Mixing Specification) phối hợp giữa hai phương pháp : hình thức và phi hình thức. Thường mô tả phi hình thức nhằm làm giải thích rõ hơn, dễ hiểu hơn một khi mô tả hình thức quá phức tạp.

I.1.3. Các thí dụ minh họa

Mô tả các cấu trúc dữ liệu :

Cho ma trận vuông A cấp n × n, n ≥ 1 :

A = {a_{ij} | i = 1..n, j = 1..n} gồm các phần tử a_{ij} ở hàng i, cột j

Bốn đỉnh (góc) của ma trận A là a_{11}, a_{1n}, a_{nn} và a_{n1}

Đường chéo chính là vector d1 = {a_{ii} | i = 1..n}

Đường chéo phụ là vector d2 = {$a_{i, n-i+1}$ | i = 1, n}

Phần tử a_{ij} đối xứng với a_{ji} qua đường chéo chính d1

Phần tử a_{ij} đối xứng với a_{n-j+1} qua đường chéo phụ d2

Ma trận tam giác trên :

$A_0 = \{ a_{ij} \mid a_{ij} \neq 0, \forall i = 1..n, j = i..n \wedge a_{ij} = 0, \forall i = 2..n, j = 1..i - 1 \}$

Ma trận tam giác dưới :

$A^0 = \{ a_{ij} \mid a_{ij} \neq 0, \forall i = 1..n, j = 1..i \wedge a_{ij} = 0, \forall i = j..n - 1, j = 2..n \}$

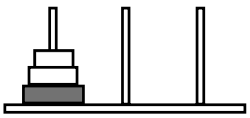
I.2. Đặc tả và lập trình

Trong những trường hợp có thể, người ta hướng đặc tả về một ngôn ngữ lập trình nào đó. Ví dụ về đặc tả đệ qui cho bài toán tháp Hà nội (Tower of Hanoi).

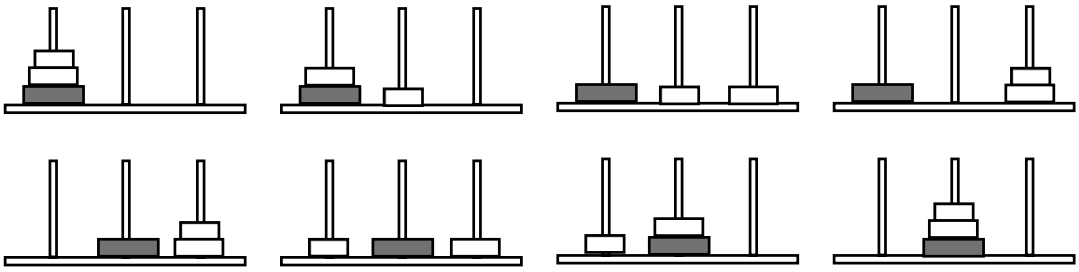
Cho chồng n đĩa n = 64 xếp thành hình tháp ở cột A (lớn nhất dưới cùng và nhỏ dần lên trên). Hãy chuyển chồng n đĩa này qua cột B theo nguyên tắc sau :

1. Mỗi lần chỉ di chuyển một đĩa từ cột này qua cột kia
2. Không đặt đĩa to lên đĩa nhỏ
3. Lấy vị trí cột C để đặt tạm các đĩa trung gian

Sau đây là bài toán Tháp Hà nội với n = 3 đĩa.



Hình 5.1. Chồng đĩa trước khi chuyển



Hình 5.2. Chồng đĩa sau khi chuyển (với7 lần xếp)

a) Cách giải phi hình thức

Chuyển n - 1 đĩa từ A qua C lấy B làm cột trung gian, sau đó chuyển đĩa dưới cùng từ A sang B. Tiếp tục chuyển n - 1 đĩa từ C qua B lấy A làm cột trung gian theo cách trên.

b) Cách giải hình thức bằng đặc tả

Gọi thủ tục chuyển n đĩa từ A qua B lấy C làm trung gian (n > 0) là :

Hà_nội (n, A, B, C)

và thủ tục chuyển một đĩa từ A qua B là :

Chuyển_một_đĩa(A, B).

Khi đó, ta có đặc tả :

```
Hà_nội (n, A, B, C) = if n > 0 then begin
                        Hà_nội (n - 1, A, C, B);
                        Chuyển_một_đĩa (A, B);
                        Hà_nội (n - 1, C, B, A)
                      End
```

ta dễ dàng viết các thao tác trên thành thủ tục như sau :

```
Procedure ChuyểnCột(n, A, B, C: TênCột);
  Begin
  if n>0 then begin
     ChuyểnCột(n-1, A, C, B);
     Chuyển_một_đĩa_từ_A_sang_C;
     ChuyểnCột(n-1, B, A, C);
     End
  End;
```

Thao tác *Chuyển_một_đĩa_từ_A_sang_C;* được viết thành lệnh :

```
Writeln('Chuyển một đĩa từ ', A:1, ' -> ', C:1);
```

Thêm biến đếm i để tính số bước chuyển đĩa, chương trình đầy đủ như sau :

```
Program HanoiTower;
Type TênCột = 1 .. 3;
Var i, N: Integer;
Procedure ChuyểnCột(n, A, B, C: TênCột);
  Begin
  if n>0 then begin
     ChuyểnCột(n-1, A, C, B);
     i:=i+1;
     Writeln(i:3,'Chuyển một đĩa từ ',A:1,' -> ',C:1);
     ChuyểnCột(n-1, B, A, C);
     End
  End;
Begin
  Write('Số đĩa cần chuyển : ');
```

```
    Readln(N);
    i:=0;
    ChuyểnCột;
    Readln
End.
```

Chạy chương trình trên sẽ cho kết quả như sau :

```
Số đĩa cần chuyển : 4
1.Chuyển một đĩa từ 1 -> 2
2.Chuyển một đĩa từ 1 -> 3
3.Chuyển một đĩa từ 2 -> 3
4.Chuyển một đĩa từ 1 -> 2
5.Chuyển một đĩa từ 3 -> 1
6.Chuyển một đĩa từ 3 -> 2
7.Chuyển một đĩa từ 1 -> 2
8.Chuyển một đĩa từ 1 -> 3
9.Chuyển một đĩa từ 2 -> 3
10.Chuyển một đĩa từ 2 -> 1
11.Chuyển một đĩa từ 3 -> 1
12.Chuyển một đĩa từ 2 -> 3
13.Chuyển một đĩa từ 1 -> 2
14.Chuyển một đĩa từ 1 -> 3
15.Chuyển một đĩa từ 2 -> 3
```

Trong trường hợp tổng quát n đĩa, số bước chuyển đĩa sẽ là :

$$2^0 + 2^1 + \ldots + 2^n = 2^n - 1 \text{ lần.}$$

Với n=64, giả sử thời gian để chuyển một đĩa là t giây, thì thời gian để chuyển hết 64 đĩa của bài toán Tháp Hà nội sẽ là :

$$(2^{64} - 1) \times t = 1.8446744074E+19 \times t \text{ giây.}$$

Một năm có $365 \times 24 \times 60 \times 60 = 31\ 536\ 000$ giây, giả sử $t = 10^{-2}$ giây thì số năm cần để chuyển 64 đĩa là :

$(1.8446744074E+19 / 31536000) \times 10^{-2} = 5.8494241735E+11 \approx 5.8$ tỷ năm !

Bài tập : 1, 2, 3, 4, 5 trang 140-141 (Nguyễn Xuân Huy).

II. Đặc tả cấu trúc dữ liệu

II.1. Cấu trúc dữ liệu cơ sở vectơ

II.1.1. Dẫn nhập

Cho một cuốn từ điển. Cần tra cứu một từ ở một trang nào đó bất kỳ :

Duyệt lần lượt các từ, từ đầu từ điển, cho đến khi gặp từ cần tra cứu, gọi là tìm tuần tự (giống tệp tuần tự)

Nếu từ điểm đã được sắp xếp ABC, có thể tìm ngẫu nhiên một từ, sau đó tùy theo từ đã gặp mà tìm phía trước hay phía sau từ đó từ cần tra cứu.

Có thể xem từ điển là một vectơ cho phép tìm kiếm ngẫu nhiên một từ.

Trong tin học, bộ nhớ máy tính cũng xem là một vectơ gồm các ô nhớ lưu trữ dữ liệu

II.1.2. Đặc tả hình thức

Cho một tập giá trị E và một số nguyên $n \in N$.

Một vectơ là một ánh xạ V từ khoảng $I \subset N$ vào E.

$V : I \rightarrow E$, I = [1..n], n là số phần tử của V, hay kích thước.

V có thể rỗng nếu n = 0

Ký hiệu vectơ bởi (V[1..n], E) hoặc E : V[1..n], hoặc V nếu không có sự hiểu nhầm.

Một phần tử của vectơ là cặp (i, V[i]) với $i \in [1..n]$, để đơn giản ta viết V[i].

Một vectơ có thể được biểu diễn bởi tập các phần tử của nó :

(V[1], V[2], ..., V[n]) hay $(x_1, x_2, ..., x_n)$ nếu x_i = V[i], là các giá trị (trực kiện)

Vectơ con : Ta gọi thu hẹp của V trên một khoảng liên tiếp của [1..n] là vectơ con của V[1..n] : V[i..j], j > i, rỗng nếu i > j

Ví dụ : V[1..5] = (7, 21, -33, 6, 8)

Các vectơ con : V[2..4] = (21, -33, 6)

V[1..3] = (7, 21, -33) v.v...

II.2. Truy nhập một phần tử của vectơ

Cho V[1..n]. Với $\forall i \in [1..n]$, phép truy nhập V[i] sẽ cho giá trị phần tử có chỉ số i của V. Kết quả không xác định nếu $i \notin [1..n]$

Ví dụ : V[1..5] = (7, 21, -33, 6, 8)

V[2] = 21, V[4] = 6 nhưng V[0], V[7]... không xác định.

Vectơ được sắp xếp thứ tự (SXTT)

Ta nói :

- Vectơ rỗng (n = 0) là vectơ được SXTT.

- Vectơ chỉ gồm 1 phần tử (n = 1) là vectơ được SXTT.

- Vectơ V[1..n], n > 1 là vectơ được SXTT nếu

$\forall i \in [1..n - 1], \forall [i] \leq \forall [i + 1]$

Có thể định nghĩa đệ qui 3 :

V[1..i] được SXTT, V[i] ≤ V[i + 1] ⇒ V[i..i + 1] được SXTT, với i ∈ [1..n - 1]

Một số ký hiệu khác :

a ∈ V[1..n] ⇔ ∃ j ∈ [1..n], a = V[j]

a ∉ V[1..n] ⇔ ∀ j ∈ [1..n], a ≠ V[j]

a < V[1..n] ⇔ ∀ j ∈ [1..n], a < V[j]

Ta cũng có cho các phép so sánh ≤ , >, ≥ , = và ≠ .

Để xét các thuật toán xử lý vectơ, ta sử dụng mô tả dữ liệu :

```
Const n = 100 ;
Type
    Vectơ = anay [ 1..n] of T ;
    {T là kiểu của các phần tử của vectơ}
```

II.3. Các thuật toán xử lý vectơ

Duyệt vectơ

Cho V[1..n], thuật toán duyệt vectơ được viết đệ quy như sau :

```
Procedure scan (V: vectơ; i, n: integer);
  Begin
  if i < = n then begin
     Operation (V[ i] );
     Scan (V, i + 1, n) { i := i + 1; nếu bỏ đệ qui }
  end
end;
```

II.3.1. Truy tìm tuần tự một phần tử của vectơ (sequential search)

a) Vectơ không được sắp xếp thứ tự

Lập luận giả sử đã xử lý i - 1 (1 < i ≤ n + 1) phần tử đầu của V và khẳng định rằng phần tử ∉ V[1..i - 1]

Xảy ra hai trường hợp :

i = n + 1 : phần tử ∉ V[1..n], kết thúc, phần tử ∉ V

i ≤ n : lại xảy ra hai trường hợp :

V[i] = phần tử : phần tử ∉ V[1..i], kết thúc, phần tử ∉ V

V[i] ≠ phần tử : phần tử ∉ V[1..i], tiếp tục i := i + 1

và cho phép khẳng định lại phần tử ∉ V[1..i - 1]

Ta viết thuật toán không đệ qui như sau :

```
function check(V: Vectơ; n: integer; phầntử: T): Boolean;
{ (n > 0) ⇒ (check, phầntử ∉ V)} ∨ (not check, phầntử ∉ V)}
Var i: integer;
```

```
begin
    i:= 1; { phântử ∉ V[ 1..i - 1] , i ≤ n}
    while (V[ i] <> phântử) and (i < 1) do
    { phântử ∉ V[ 1..i] , i < n}
        i := i + 1; { phântử ∉ V[ 1..i - 1] , i ≤ n}
    { ((V[ i] = phântử) ∨ (i = n), phântử ∉ V[ 1..i - 1] ,
    i ≤ n) ⇒ (V[ i] = phântử, phântử ∉ V)
    ∨ (V[ i] ≠ phântử, phântử ∉ V)}
    Check := (V[ i] = phântử)
    { (check, phântử ∉ V) ∨ (Not check, phântử ∉ V)}
end;
```

Ta có thể viết lại thuật toán dưới dạng đệ quy như sau :

```
function check(V:Vectơ, i,n:integer; phântử: T): Boolean;
{ n ≥ 0 ⇒ check, phântử ∈ V[ i..n] )
∨ (Not check, phântử ∉ V[ i..n] )}
begin
    if i > n then check := false
    else if V[ i] = phântử then check := true
    else check := check (V, i + 1, n, phântử)
end;
```

Khi gọi hàm, i có thể nhận giá trị bất kỳ, từ 1..n, đặc biệt i = 1

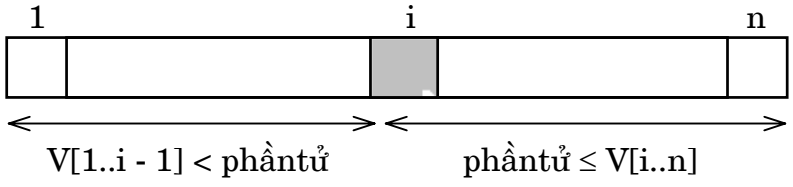
Trường hợp duyệt vectơ từ phải qua trái, ta không cần dùng biến i nữa :

```
function check (V:Vectơ; n: integer; phântử: T): Boolean;
{ n ≥ 0 ⇒ (check, phântử ∈ V) V (Not check, phântử ∉ V)}
begin
if n = 0 then check := false
else if V[ n] = phântử then check := true
    else check := check (V, n - 1, phântử)
end;
```

b) Vectơ được sắp xếp thứ tự

Ta cần tìm chỉ số i ∈ [1..n] sao cho thỏa mãn :

V[1..i - 1] < phântử ≤ V[i..n]



Hình 5.3. Vectơ được sắp xếp thứ tự

Vấn đề là kiểm tra đẳng thức phântử = V[i] không trong V đã được sắp xếp ?

Lập luận :

Giả sử đã xử lý i - 1 ($1 \le i \le n + 1$) phân tử đầu của V và V[1..i - 1] < phân tử đã được khẳng định : xảy ra hai trường hợp :

i = n + 1 : kết thúc V[1..n] < phân tử, phân tử $\notin$ V

i ≤ n : lại có hai trường hợp mới :

V[i] ≥ phân tử : kết thúc, đã tìm được i sao cho

V[i..i -1] < phân tử ≤ V[i..n]

chỉ còn phải kiểm tra phân tử = V[i] ?

V[i] <phân tử : có nghĩa V[1..i] < phân tử, tiếp tục thực hiện : i := i + 1 để có lại khẳng định V[1..i - 1] < phân tử

Ta có thuật toán như sau :

```
function checknum(V:vectơ; n:nguyên; phântử:T): Boolean;
{ V được SXTT, n > 0 ⇒ (checknum, phân tử ∈ V) ∨
(Not checknum, phầntử ∉ V)}
Var i: integer ;
begin
if phân tử > V[ n]  then
    checknum := false { not checknum, phântử ∉ V)}
    else begin { phântử ≤ V[ n]}
        i := 1 ; { V[ 1..i - 1] < phântử }
        while (V[ i] < phântử) do { V[ 1..i] < phântử }
           i := i + 1 ; { V[ 1..i] < phân tử}
           { V[ 1..i - 1] < phântử, V[ i] ≥ phântử }
           checknum := (V[ i] = phântử)
           { (checknum, phântử∈V) ∨ (¬checknum, phântử∉V)}
    end { (checknum, phântử∈V) ∨ (¬checknum, phântử∉V)}
end;
```

II.3.2. Tìm kiếm nhị phân (Binary search)

a) Phương án 1

Giả sử vectơ V[1..n] (n > 1) đã được sắp xếp thứ tự :

$\forall$ i $\in$ [1..n - 1], V[i] ≤ V[i +1]

Ta chia V thành 3 vectơ con V[1..m - 1], V[m..m] và V[m + 1..n] được sắp xếp thứ tự sao cho :

V[1..m - 1] ≤ V[m] ≤ V[m + 1..n]

Xảy ra 3 trường hợp :

$\in$ V[1..m - 1] nếu phân tử < V[m]

phân tử = V[m]

$\in$ V[m + 1..n] nếu phân tử > V[m]

lúc này ta trở lại bài toán đã xét : tìm phân tử trong vectơ V[1..m - 1] hoặc V[m +1..n]. Kết thúc nếu phân tử = V[m]

Một cách tổng quát, lần lượt xác định dãy các vectơ có V_1, V_2, ..., V_k sao cho mỗi V_i có kích thước nhỏ hơn kích thước của vectơ con trước đó V_{i-1}

Để ý rằng nếu chọn V_1 = V[1..n], V_2 = V[2..n], ..., V_k = V[k..n], ta đi đến phép tìm kiếm tuần tự đã xét ở trên.

Ta sẽ chọn m là vị trí giữa (nếu n lẻ) để cho V[1..m - 1] và V[m + 1..n] có kích thước bằng nhau, hoặc chọn m sao cho chúng hơn kém nhau một phân tử.

Khi đó kích thước của các vectơ thuộc dãy V_1, V_2, ..., V_k sẽ lần lượt được chia đôi tại mỗi bước : n, n/2, ..., $n/2^{k-1}$.

Như vậy, sẽ có tối đa [$\log_2 n$] vectơ con khác rỗng.

Ví dụ : nếu n = 9000, số vectơ con khác rỗng tối đa sẽ là 13, vì 2^{13} = 8192

Xây dựng thuật toán :

Sau một số bước, ta có vectơ con V[inf.. sup] sao cho :

V[1..inf - 1] < phầntử < V[sup + 1..n]

Xảy ra hai trường hợp :

- inf > sup (inf = sup + 1)

  (V[1..inf-1] < phầntử < V[sup+1..n], inf = sup+1) $\Rightarrow$ (phân tử $\notin$ V, kết thúc)

- inf $\leq$ sup : m = (inf + sup) div 2

khi đó ta có V[inf..m - 1] $\leq$ V[m] $\leq$ V[m + 1..sup]

Tồn tại 3 khả năng như sau :

- Phần tử = V[m] : kết thúc, phần tử $\in$ V
- Phần tử < V[m] : tiếp tục tìm kiếm trong V[inf..m - 1]
  lấy sup := m - 1 đê có lại khẳng định phần tử < V[sup + 1..m]
- Phần tử > V[m] : tiếp tục tìm kiếm trong V[m + 1..sup]
  lấy inf := m + 1 để có lại khẳng định V[1..inf - 1] < phần tử

Như vậy cả hai trường hợp : V[1..inf - 1] < phần tử < V[sup + 1..n]

Khởi đầu, lấu inf := 1 và sup := n

Ta có thuật toán như sau :

```
function binary (V:vectơ; n:integer; phântử:T): Boolean;
{ V được SXTT ⇒ (binary, phântử∈V)∨(not binary, phântử∉V)}
Var inf, sup, m : integer ;
OK : Boolean ;
begin
   OK : false ; { not OK, phân tử ∉ V}
```

```
    inf := 1 ; sup := n ;
    { V[ 1..inf - 1] < phântử < V[ sup+1..n] }
    while (inf ≤ sup) and (not OK) do begin
      m := (inf + sup) div 2 ;
      if V[ m] = phântử then OK := true { OK, phântử ∈ V}
      else { not OK}
         if V[ m] <phântử then inf:= m+1 { V[ 1..inf-1] <phântử}
         else sup := m - 1 ; { V[ sup + 1..n] > phântử}
         { (V[ 1..inf - 1] < phântử < V[ sup + 1..n] , not OK)
         ∨ (tìm thấy, phântử ∈ V)}
    end;
    { (inf = sup + 1) ∨ (tìm thấy),
    (¬ tìm thấy, V[ 1..inf - 1] < phântử < V[ sup + 1..n] ) ∨
    (tìm thấy, phântử ∈ V) ⇒
    (¬ tìm thấy, V[ 1..inf - 1] < phântử < V[ inf..n] ) ∨
    (tìm thấy, phântử ∈ V) ⇒
    (¬ tìm thấy, phântử ∉ V) ∨ (tìm thấy, phântử ∈ V)}
    Nhị phân := OK
end ;
```

Viết chương trình trên dưới dạng đệ quy :

```
function NhịPhân(V:Vectơ;inf,sup:integer;phântư:T):boolean;
{ (V được SXTT ⇒ (NhịPhân, phântử∈V) ∨ ¬ NhịPhân,phântử∉V)}
Var m : integer ;
begin
   if inf > sup then NhịPhân:= false
   else begin
     m := (inf + sup) div 2 ;
     if V[ m] = phântử then NhịPhân:= true
     else if V[ m] < phântử then
        NhịPhân:= NhịPhân(V, m+1, sup, phântử)
     else NhịPhân:= NhịPhân(V, inf, m - 1, phântử)
   end
end ;
```

Hàm này có thể được gọi với các giá trị inf, sup bất kỳ, thông thường được gọi bởi dòng lệnh :

```
NhịPhân (V, 1, n, phântử)
```

b) Phương án 2

Có thể tìm ra những phương án khác cho thuật toán tìm kiếm nhị phân. Chẳng hạn, thay vì kiểm tra đẳng thức V[m] = phântử, ta kiểm tra khẳng định :

```
V[ 1..inf - 1] < phântử ≤ V[ inf..n]
```

Ssau đó kiểm tra phântử = V[inf] để có câu trả lời.

Mặt khác, có thể thay đổi giá trị trả về của hàm tìm kiếm nhị phân bởi vị trí của phân tử trong vectơ, bằng 0 nếu phân tử ∉ V.

Nếu inf = 1, khẳng định có dạng V[1..0] < phầntử ≤ V[sup..n] và được viết gọn phầntử ≤ V[1..n].

```
function NhịPhân(V:vectơ;n:integer;phântử: T): integer ;
{ (V được SXTT, n > 0) ⇒ (m ∈ [ 1..n]
NhịPhân = m, V[ m] = phântử) V (NhịPhân = 0, phân tử ∉ V)}
Var m, inf, sup : integer ;
begin
   if phân tử > V[ n] then nhị phân := 0
   else begin
      inf := 1 ; sup := n ;
      { V[ 1..inf - 1] < phântử ≤ V[ sup..n]}
      while inf < sup do begin
         m := (inf + sup) div 2 ;
         if phầntữ ≤ V[ m] do sup := m { phâtử ≤ V[ sup..n]}
         else inf := m + 1 { V[ 1..inf - 1] < phân tử}
         { V[ 1..inf - 1] < phân tử ≤ V[ sup..n]}
      end ;
       { (inf = sup, V[ 1..inf - 1] < phân tử ≤ V[ inf..n]
       ⇒ (V[ 1..inf - 1] < phântử ≤ V[ inf..n]}
       if phântử = V[ inf] then NhịPhân:= inf
       else NhịPhân:= 0
    end
end ;
```

III. Đặc tả đại số : mô hình hóa phát triển phần mềm

(*Phần này chỉ phục vụ tham khảo*)

III.1. Mở đầu

Đặc tả đại số không mô tả các yếu tố liên quan đến thời gian thực thi cũng như trạng thái.

Ngôn ngữ đặc tả trạng thái liên quan đến :

- Ngữ nghĩa (Semantic)
- Cú pháp (syntax)
- Các thuộc tính (Properties)

Hình vẽ

Ngữ nghĩa của các đặc tả đại số liên quan đến :

- Dấu kí (signature) của một kiểu đại số trừu tượng
- Hạng (term) với các biến
- Phương trình và các tiên đề
- Các mô hình đặc biệt ...

Cú pháp của đặc tả đại số

Ví dụ :

Xây dựng kiểu string cho các xâu ký tự cùng các phép toán thông dụng trên xâu như sau :

- Tạo xâu rỗng mới (phép toán new)
- Ghép xâu (append)
- Thêm một ký tự vào xâu (add to)
- Lấy độ dài xâu
- Kiểm tra xâu rỗng (is empty)
- Kiểm tra hai xâu bằng nhau không (=)
- Trích ký tự đầu tiên của xâu (frist)

Để định nghĩa kiểu string, người ta còn sử dụng các kiểu sau :

- char : kiểu của ký tự
- nat : kiểu của số nguyên
- bool : kiểu giá trị logic

Tên các tập hợp và các phép toán trên tập hợp xác định một ký dấu (signature). Như vậy một dấu kí được xây dựng từ :

- Tên các kiểu đặc tả

- Tên các phép toán với việc chỉ rõ miền xác định (domain) và miền trị (range) như sau :

tên phép toán : miền xác định → miền trị

Ta xây dựng dấu kí từ kiểu string như sau

```
Adt String ;
Use char, Not, Bool ;
Sorts string ;
Operations
new : → string ;
append _ _ : String, string → string ;
add _ to _ : char, string → string ;
# _ : String → not ;
is empty ? _ string → bool ;
_ = _ : string, string → bool ;
frist _ : string → char ;
```

Tên xuất hiện trong một dấu kí gồm hai loại là có ích (internest) và bổ trợ (auxiliary) tùy theo vai trò của chúng. Ví dụ :

- String là có ích

- Char, not và bool là bổ trợ

Cú pháp (cp)

Cp đặc tả đại số sử dụng trong ví dụ trên được chia ra thành các khối : đầu, giao tiếp và thân của đặc tả. Mỗi khối gồm một số khai báo ngăn cách nhau bởi các từ khóa (có gạch chân)

Đối với khối giao tiếp (interface), người ta sử dụng các khái niệm tiền tố (prefix), trung tố (infix) và hậu tố (postfix) như sau :

Tiền tố : tên của phép toán được đặt trước dãy các tham biến

Ví dụ : appenend _ _ : string, string → string ;

Từ đó người ta có thể viết các hạng dưới dạng :

append x y hay

append (x y) hay

(append x y)

Trung tố : cho phép định nghĩa toán tử hay vị từ

Ví dụ _ = _ : string, string → bool ;

từ đó có thể viết các hạng dưới dạng :

x = y hay (x = y)

Hỗn hợp : cho phép viết các biểu thức bất kỳ một cách mềm dẻo như add_to_ : char, string → string

từ đó có thể viết các hạng dưới dạng :

add c to append (x y)

Trong nhiều trường hợp trên đây, các cặp dấu ngoặc dấu được dùng để phân cách các hạng với nhau

III.2. Phân loại các phép toán

Các phép toán được chia ra thành 2 loại :

Loại quan sát được (oprations)

Loại phát sinh (generator operations)

Loại quan sát được có các dạng sau :

Kiểu có ích [và kiểu bổ trợ] → Kiểu bổ trợ

Ví dụ : _ = _ : string, string → bool;

_ : string → not ;

is empty ? : string → bool ;

first _ : string → char ;

Loại phát sinh có dạng :

Kiểu có ích [và kiểu bổ trợ] → Kiểu có ích

Ví dụ :

new : _ → string ;

add_ to _ : char, string → string ;

Ở đây, phép toán new tạo ra một xâu rỗng, còn phép toán add _ to _ thêm một ký tự vào xâu.

Các tiên đề được xây dựng từ các phép toán dùng cho các kiểu bổ trợ giả sử được định nghĩa như sau :

true : → bool ;

false : → bool ;

not _ : bool → bool ;

_ and _ : bool, bool → bool ;

_ or _ : bool ; bool → bool ;

0 : → not ;

1 : → not ;

succ : not → not ;

_ + _ : not, not → not :

_ - _ : not, not → not :

_ * _ : not, not → not :

_ / _ : not, not → not :

_ = _ : not, not → bool;

a : → char ;

b : → char ;

...

_ = _ : char, char → bool ;

III.3. Hạng và biến

Trong đặc tả đại số, các biến được định kiểu và có thể nhận giá trị tùy ý tùy theo kiểu đã định nghĩa. Ví dụ : khai báo kiểu x : string ; y : string ; c : char ; định nghĩa các biến x, y, c để sử dụng trong các hạng sau đây :

add c to x = append (x y)

append (is empty ? (new), add x to x)

Hạng là một biểu thức nhận được từ việc tổ hợp liên tiếp các phép toán của singnature (dấu kí). Một hạng là hợp thức nếu hạng đó thỏa mãn các phép toán đã sử dụng (kiểu và vị trí). Qui tắc quy nạp được dùng để xây dựng tập hợp các hạng + có kiểu s được viết t : s được định nghĩa như sau :

$+ : s_1, s_2, ..., s_n \rightarrow s \wedge t_1 : s_1, t_2 : s_2, ..., t_n : s_n$

$(f\ t_1\ t_2\ ...\ t_n) : s$

trong đó sử dụng quy tắc khai báo kiểu biến

x : s

Từ đó, hạng hợp thức trong hai hạng từ ví dụ vừa xét là

add c to x = append (x y)

III.4. Phép thế các hạng

Phép thế (substitutions) là một phép toán trên các hạng cho phép thay thế các biến (có mặt) trong các hạng bởi các hạng khác. Tập hợp các biến FV xuất hiện trong một hạng được định nghĩa một cách đệ quy như sau :

$FV\ (ft_1t_2\ ...\ t_n) = FV\ (t_1) \cup FV\ (t_2) \cup ...\ FV\ (t_2) \cup ... \cup FV\ (t_n)$

FV (x) = {x}

Ví dụ : FV (append (is empty ? (new), add c to x)) = {x, c}

Phép thế trong một hạng t cho các thành phần chứa biến x bởi hạng u, ký hiệu t [u /x], được định nghĩa như sau :

Với $x \in FV(t)$ thì

$(f\ t_1\ t_2 \ldots t_n)\ [u/x] = (f\ t_1\ [u/x]\ t_2\ [u/x] \ldots t_n\ [u/x])$

y [u/ x] = u y = x

= y $y \neq x$

Ví dụ : append (is empty ? (new), (add c to x)) [(add c to y) / x]

= append (is empty ? (new), (new), (add c to (add c' to y)))

Mô tả các thuộc tính qua các phương trình

Các tiên đề sử dụng trong đặc tả được xây dựng theo logic vị trí bậc 1 dạng phương trình (pt)

Một phương trình hợp thức có vế trái và vế phải cùng kiểu hạng :

AX spec = $\{t = t' \mid t : s \wedge t' : s\}$

Trong ví dụ về đa kiểu string, phép toán is empty ? được định nghĩa theo phương trình :

is empty ? (new) = true ;

Có nghĩa một xâu vừa mới tạo ra là rỗng - sau đó, việc thêm một ký tự mới vào xâu sẽ cho kết quả là false :

is empty ? (add c to x) = false ;

Tính đệ quy của phương trình :

append (x, add c to y) = add c to (append (x, y)) ;

chỉ ra rằng việc ghép một xâu với xâu được tạo ra bằng cách thêm một ký tự vào xâu này thì cũng có giá trị như ghép hai xâu trước rồi sau đó thêm một ký tự vào xâu kết quả. Điều đó hợp lý vì ta có tính chất của phương trình : append (x, new) = x ;

nghĩa là ghép một xâu nào đó với xâu rỗng cũng cho ra kết quả chính xâu đó

Ta có các tiên đề về xâu ký tự như sau :

Axioms

is empty ? (new) = true ;

is empty ? (add c to x) = false ;

new = 0 ;

(add c to x) = x (x) = + 1;

append (x, new) = x ;

append (x, add c to y) = add c to append (x y) ;

(new = new) = true ;

add c to x = true ;

(add c to x = new) = false ;

(new = add c to x) = false ;

(add c to = add d to y) = (c = d) and (x = y) ;

where ...

... where

x, y : string ;

c, d : char ;

end string ;

Các tiên đề điều kiện

Các tiên đề điều kiện tích cực (positive conditional axions) là mở rộng của các phương trình, chúng là các mệnh đề Horm về tính bằng nhau, có dạng :

$t_1 = t_1' \wedge t_2 = t_2' \wedge ... \wedge t_n = t'_n \Rightarrow t = t'$

Ví dụ : is empty ? (x) = flase $\Rightarrow$ first (add c to x) = first (x) ;

is empty ? (x) = true $\Rightarrow$ first (add c to x) = c ;

III.5. Các thuộc tính của đặc tả

Đặc tả đặt ra hai vấn đề sau đây :

- Hợp thức hóa

- Lưỡng năng chứng bác (completude) của đặc tả

III.5.1. Mô hình lập trình (triển khai)

Các mô hình lập trình mô tả cách thức triển khai của đặc tả. Có nghĩa các chương trình trừu tượng sẽ kiểm chứng các thuộc tính đã trình bày trong đặc tả (thiết lập). Tập hợp các mô hình đặc tả với các phép toán kèm theo được ký hiệu Mod (spec).

Khái niệm lập trình dẫn đến quan hệ thỏa mãn ký hiệu Ư xác định tính triển khai đúng đắn của đặc tả. Ta có :

M $\in$ Mod (spec) $\Leftrightarrow$ ($\forall$t, t' : s và t = t' $\in$ Ax spec ta có M Ư t = t')

Với mọi tiên đề : t = t' của Ax spec

Mod (spec) Ư t = t' $\Leftrightarrow$ $\forall$M $\in$ Mod (spec),

M Ư t = t'

III.5.2. Mô hình đặc biệt

Những mô hình chấp nhận được bởi một đặc tả rất phong phú. Sau đây là một ví dụ về mô hình cho đặc tả kiểu Bool :

Hình vẽ

trong hai mô hình A và B ở trên, đặc tả kiểu Bool thõa mãn với các quy ước có giá trị là các dấu x, các phép toán biểu diễn bởi các quan hệ giữa miền xác định và miền trị.

Chú ý rằng A chứa các giá trị vô ích tương tự như việc sử dụng 1 byte cho kiểu Bool, còn B chứa vừa đủ (tối thiểu) giá trị cần thiết tương tự như sử dụng 1 bit cho kiểu Bool.

III.5.3. Mô hình đồng dư

Mô hình này là một thương đại số các hạng đồng dư định nghĩa bởi các quy tắc sau đây :

- t = t' là tiên đề khi đó t ~ t'

- Phản xạ : t ~ t

- Đối xứng : $t \sim t' \Rightarrow t' \sim t$

- Bắc cầu : $t = t' \wedge t' \sim t'' \Rightarrow t \sim t''$

- Khả thế : (cấu thành - substitutivite)

$t_1 \sim t_1' \wedge t_2 \sim t_2' \wedge \ldots \wedge t_n \sim t_n' \Rightarrow (ft_1t_2 \ldots t_n) \sim (ft_1't_2' \ldots t_n')$

- Thay thế : cho x là biến, u là hạng $t \sim t' \Rightarrow t\,[u/\,x] \sim t'\,[u/\,x]$

Quá trình khai triển một đặc tả

Khai triển một đặc tả là vấn đề khó khăn. Những định nghĩa về cú pháp các chức năng mong đợi không là khó khăn nhưng tính đúng đắn của chúng lại không kiểm chứng được dễ dàng.

III.6. Phép chứng minh trong đặc tả đại số

Mục đích phần này là chỉ ra cách chứng minh (chứng minh) các thuộc tính trong các đặc tả đại số. Một thuộc tính cần chứng minh có dạng một định lý, chẳng hạn dạng một phương trình.

Giả sử ta cần chứng minh thuộc tính sau đây trong đặc tả các số nguyên tự nhiên dương Not :

succ (0) = succ (succ (0)) = succ (succ (0)))

? succ (0) + succ (0)) = succ (succ (succ (0)))

Tiên đề : succ (x) + y = succ (x + y)

Quy tắc thay thế với s = {x = 0, y = succ (succ(0))}

succ (0) + succ (succ (0)) = succ (0 + succ(succ (0)))

Tiên đề : 0 + x = x và quy tắc thay thế với s = { x = succ (succ (0)))}

0 + succ (succ (0)) = succ (succ (0))

Quy tắc thay thế với phép succ trên (2)

succ (0 + succ (succ (0))) = succ (succ (0)))

Quy tắc bắc cầu cho (1) và (3)

5.

Định lý đã được chứng minh. Cần chú ý rằng thuộc tính này là hợp thức cho mọi quá trình đặc tả số tự nhiên Not.

III.6.1. Lý thuyếttương đương

Lý thuyết tương đương (của một đặc tả) được xây dựng từ các tiên đề của đặc tả, là tập hợp các định lý hợp thức qua các quy tắc sau đây :

- Phản xạ : t = t

- Đối xứng : t = t' $\Rightarrow$ t' = t

- Bắc cầu : t = t' $\wedge$ t' = t'' $\Rightarrow$ t' = t''

- Khả thế : t = t' $\wedge$ $t_2 = t_2'$ $\wedge$ ... $\wedge$ $t_n = t_n'$ $\Rightarrow$

(ft_1, t_2, ..., t_n) = (ft_1', t_2', ..., t_n')

- Phép thế : cho x là biến và u là hạng

$t_1 = t_1'$ $\wedge$ $t_2 = t_2'$ $\wedge$ ... $\wedge$ $t_n = t_n'$ $\Rightarrow$ t = t'

khi đó t_1 [u/ x] = t_1' [u/ x] $\wedge$ ... $\wedge$ t_n [u/ x] = t_n' [u/ x]

$\Rightarrow$ t [u/ x] = t' [u/ x]

- Phép cắt : $Cond_1 \wedge (u = u') \wedge cond_2 \Rightarrow t = t'$

và cond $\Rightarrow$ x = x', khi đó :

$cond_1 \wedge cond \wedge cond_2 \Rightarrow t = t'$

Các qui tắc của lý thuyết tương đương thể hiện các thuộc tính bằng nhau (phản xạ, đối xứng và bắc cầu), thuộc tính hàm (khả thế), các biến (phép thế) và thay thế các vế bằng nhau (phép cắt). Các quy tắc này xác định phép suy diễn Ê EQ, định lý sau đây minh họa kích chắc chắn và tính rõ của phép suy diễn

Định lý : lý thuyết tương đương

với một đặc tả spec, $\forall$t = t', Ax spec Ê EQ t = t'

$\Leftrightarrow$ Mod (Ax spec) Ư t = t'

Công thức Mod (Ax spec) Ư t = t' chỉ ra rằng phương trình là hợp thức trong mọi cách lập tỷình có thể. Tuy nhiên có thể xảy ra một số trường hợp đặc biệt khi mô hình không hợp lý, lúc đó có thể true = false.

Ta có thể chứng minh rằng thuộc tính

succ (succ (0)) - succ (succ (0)) = 0

là hợp thức (valid), những thuộc tính

succ (succ (succ (0))) - succ (succ (0)) = 0

Không là hợp thức trong đặc tả đang xét, vì rằng sau khi suy diễn, ta nhận được succ (0) = 0 là không hợp thức.

Ta có thể thấy rằng x - x = 0 không chứng minh được trong ngữ cảnh đang xét mặc dầu định lý này tỏ ra hiển nhiên trong đặc tả. Từ đó, ta có thể bổ sung thêm một số giả thiết cho mô hình để tăng khả năng chứng minh có thể.

III.6.2. Khái niệm về lý thuyết quy nạp

Như đã chỉ ra, ta cần thêm các định lý tổng quan hơn để có khả năng suy diễn từ các tiên đề của đặc tả, chẳng hạn như x + y = y + x là không có tính suy dần trong lý thuyết tương đương.

Ta sẽ thêm vào các qui tắc sử dụng trong lý thuyết tương đương một quy tắc mới như sau :

- Qui nạp : giả sử G là công thức sao cho x là một biến tự do, nếu với mọi t, G [t/ x] là suy diễn được thì G cũng suy diễn được cho t. Quy tắc này chỉ rõ rằng có thể kết luận rằngnếu việc chứng minh một định lý là hợp thức cho mọi trường hợp, định nghĩa bởi một hạng, bởi một biến thì định lý cũng hợp thức cho công thức được lượng hóa một cách phổ dụng trên biến này.

Tương tự đối với định lý tương đương, định lý sau đây cho kết quả thuyết phục cho việc suy diễn quy nạp đối với đặc tả hữu hạn.

Định lý 3.2 : Lý thuyết quy nạp

Với một đặc tả đại số spec

$\forall$t = t', Ax spec Ê Ind t = t'

$\Leftrightarrow$ Mod_{Gen} (Ax spec) Ư t = t'

Ta sẽ minh họa nguyên lý này bởi một ví dụ trên các giá trị logic xây dựng từ các phép toán true, false và not. Ta muốn chứng minh rằng :

not (not (b)) = b

- trường hợp cơ sở :

? not (not (true)) =not (false) = true ;

2. Not (not (false)) = not (true) = false ;

- Không quy nạp :

not (not (b)) = b suy ra not (not (not (b))) = not (b)

quy tắc khả thế với not cho not (not (b)) = b

not (not (not (b))) = not (b)

Nhờ quy tắc quy nạp mà thuộc tính mong muốn được chứng minh. Như vậy lý thuyết quy nạp cho phép chứng minh tính giao hoán của phép cộng trong Not qua x + y = y+ x việc chứng minh cần quy nạp hai lần trên x và y.

III.6.3. Chứng minh tự động bởi viết lại

Việc chứng minh bởi viết lại (demonstration by rewriding) là một kỹ thuật cho phép chứng minh tự động. Đó là quá trình ước lượng các hạng bằng cách viết lại một cách hệ thống các hạng thành các dạng chuẩn (dạng không thể ước lược được nữa) và phép chứng minh các thuộc tính. Nguyên lý sử dụng là hướng đến các phương trình đặc tả theo quy tắc viết lại và áp dụng liên tiếp các quy tắc này trên các hạng đã ước lược.

Ví dụ : từ tiên đề not true = false ta sẽ dẫn đến quy tắc not true ??? false. Quy tắc này được dùng để chứng minh tính bằng nhau của dạng t = t'. Sự bằng nhau là hợp thức nếu hai vế của chúng được viết lại thành duy nhất một hạng không thể ước lược được nữa.

Định lý 3.3 Chứng minh bởi viết lại

Với một đặc tả spect, t, t'' là các hạng nếu

t ??? ... ??? to và t' ??? ... to thì :

Ax spec Ê EQ t = t'

Ở đây ta sử dụng ký hiệu t ???* t' cho dãy t ??? ...??? t hay t là một dạng chuẩn của hạng, nghĩa là một hạng không thể thu gọn.

Cần chú ý rằng đẳng thức tạo ra bởi viết lại không bắt buộc đồng nhất với đẳng thức nhận đuợc từ hệ thống suy diễn Ê EQ (không hòan toàn).

Để có thể thực hiện các phép chứng minh theo lý thuyết trước đây, ta cần nhận được một hệ thống viết lại hội tụ tương đương với hệ thống sinh bởi các tiêu đề.

Giải pháp đầu tiên là hướng tới các phương trình, Nếu hệ thống nhận được là đi đến đích (mọi hướng suy dẫn khác nhau có thể đều dẫn về cùng kết quả) và kết thúc (sau một số hữu hạn bước viết lại trước khi nhân được dạng chuẩn). Từ đó các thuộc tính chứng minh được tương đương với các phương trình xuất phát.

Chẳng hạn để đặc tả Bool, ta cần nhận được bằng cách hướng các tiên đề từ trái qua phải :

not (true) ??? false

not (false) ??? true

true and b ??? b

false and b ??? false

true or b ??? true

false or b ??? b

false xor b ??? not (b)

Ví dụ : sử dụng các quy tắc trên để viết lại hạng sau đây :

not (false or (true and false))

not (true and false) not (false or false)

not (false)

true

Tuy nhiên, nguyên lý hướng về viết lại không đủ để chứng minh mọi thuộc tính tương đương. Ta có thể minh họa điều đó trong đặc tả các số tự nhiên một cách đơn giản như sau :

Interface

Sort not ;

Operations

0 : → not ;

_ _ : not → not ;

_ + _ : not not → not ;

Body

Axions

a x 1 0 + x = x ;

a x 2 : x + (- x) = 0 ;

...

Từ đặc tả trên, ta có thể xây dựng các quy tắc :

0 + x ??? x

x + (- x) ??? 0

- 0 ??? 0

Cần chú ý rằng trong trường hợp này, việc hướng các quy tắc từ trái qua phải chưa đủ, vì nếu muốn chứng minh - 0 = 0 thì phải cần áp dụng tiên đề 1 từ phải qua trái, sau đó áp dụng tiên đề 2 từ trái qua phải, như vậy sẽ không tương ứng với việc lựa chọn định hóng ???.

Rõ ràng việc định hướng là một cơ chế chứng minh chưa đầy đủ, đặc biệt đối với các tiên đề về các phép tính sinh. Cơ chế này có thể đầy đủ trong nhiều tình huống thực tế.

Trong trường hợp các tiên đề không định hướng như phép giao hoán, các ký thuật đặc tả được phát triển để thực hiện viết lại (hệ viết lại kiểu modun kết hợp - giao hoán)

Các phép toán phát sinh (xây dựng)

Theo định nghĩa, một mô hình được phát sinh bởi một tập hợp con w các phép toán nếu mọi giá trị của mô hình này nhân được bởi một hạng được xây dựng từ các bộ sinh w. Định nghĩa này cho phép, theo dịnh lý quy nạp, chỉ xem xét các bộ sinh trong các chứng minh bằng cách chỉ chứng minh quy nạpchúng (bởi vì mọi hạng đạt được bởi các việc tổ hợp các bộ sinh)

III.6.4. Phân cấp trong đặc tả đại số

Các mô hình phân thể làm thỏa mãn các tiên đề và các hạn chế do các ràng buộc đơn thể chủ yếu dựa trên khả năng buộc đơn thể. Các ràng buộc đơn thể chủ yếu dựa trên khả năngkhông bị xáo trộn giữa các mô hình ở mức phân cấp thấp hơn khi sử dụng một đặc tả. Nguyên lý này cho phép sử dụng việc phân cấp các đặc tả trong các giai đoạn khai triển, đặc biệt khi làm mịn (refinement). Có nghĩa là các mô hình phải được lập trình độc lập với nhau.

Các kiểu xáo trộn có thể xuất hiện trong một đặc tả đại số là :

- "junk" (mảng) : các giá trị được thêm vào bởi việc dùng các đơn thể, ràng buộc về tính đầy đủ hạn chế kiểu xáo trộn này.

- "cofusion" (trộn lẫn) : các giá trị bị thay đổi (collapse), do ràng buộc về sự hiện hữu phân cấp làm ảnh hưởng đến đặc tả.

Lớp các mô hình phân cấp được ký hiệu bởi Hitol (spec) đó là những mô hình thỏa mãn quy tắc về sự hạn chế các mô hình trên các mô hình con bảo toàn được được ngữ nghĩa của chúng.

Tính rõ ràng (Completude suffisance)

Ta sẽ minh họa tình huống "junk" bằng một ví dụ

Adl Không_hoàn_toàn ;

In terface

Use Not, Bool ;

Operation

f : not → bool ;

Body

Axioms

f (succ (x)) = false ;

where

x : not ;

End ;

Ví dụ trên không rõ ràng khi thêm định nghĩa hàm f trên các đơn thể về not và bool. Các tiên đề về đơn thể của hàm f sẽ làm xáo trộn các kiểu đã định nghĩa. Ta thấy một giá trị mới f (0) kiểu sẽ không được xác định vì không có tiên đề nào chỉ ra f (0) là true hay false.

IV.Đặc tả hay cách cụ thể hóa sự trừu tượng

(Specification or How to Make Abstrations Real)

IV.1. Đặc tả phép thay đổi bộ nhớ

Giả sử ta cần đặc tả phép thay đổ nội dung một bộ nhớ. Để đơn giản hóa mà không làm mất tính được biểu diễn bởi sơ đồ sau :

1	0
2	0
3	0

bộ nhớ chỉ gồm 3 địa chỉ nhớ chứa 3 giá trị lúc đầu đều là 0 giả lệnh, làm thay đổi nội dung của bộ nhớ được ký hiệu bởi **chg** (x, y)

Ví dụ lệnh **chg** (2, 5) làm thay đổi địa chỉ thứ 2 từ giá trị 0 thành 5 :

1	0
2	0
3	0

chg (2, 5)

1	0
2	5
3	0

Như vậy, lệnh **chg** (x, y) đã thay đổi nội dung của địa chỉ x thành giá trị y và giữ nguyên nội dung của các địa chỉ còn lại.

Gọi q là bộ nhớ lúc đầu và q' là bộ nhớ nhận được sau khi thực hiện lệnh **chg** (x, y), ta có thể chuyển sơ đồ trên thành dạng điều kiện như sau :

q **chg** (x, y) q'	(1.1)

Ta nói lệnh **chg** (x, y) đã chuyển bộ nhớ q thành q'.

Định nghĩa ngữ nghĩa của lệnh **chg** chính là đặc tả, nhờ viết điều kiện tương đương với điều kiện (1.1)

Ví dụ sự kết hợp các điều kiện sau đây để đặc tả phép toán thay đổi **chg** :

$q'(x) = y$

$q'(a) = q(a)$ nếu $a \neq x$ (1.2)

Như vậy ta đã ngầm ẩn thừa nhận rằng q và q' chỉ định các hàm. Trong ví dụ này, với bộ nhớ 3 địa chỉ, miền xác định của hàm là tập hợp {1, 2, 3} tổng quát các địa chỉ và miền giá trị là các giá trị có thể lưu trữ được trong bộ nhớ đang xét, chẳng hạn là tập hợp các số nguyên $\{-2^{31}, ..., 2^{31}\}$ (1.3)

Một cách tổng quát, gọi A là tập hợp các địa chỉ, V là tập hợp các giá trị, ta có:

$q \in A \rightarrow V$ (1.4)

Như vậy một biểu thức dạng q (a) chỉ có nghĩa nếu $a \in A$, từ đó q (a) chỉ có nghĩa nếu $a \in A$, từ đó $q(a) \in V$. Do bộ nhớ q' nhận được từ q sau khi thực hiện lệnh **chg**, q' cũng thỏa mãn điều kiện (1.4)

$q' \in A \rightarrow V$

Tuy nhiên điều kiện (1.2) không phải luôn luôn có nghĩa vì rằng biểu thức q' (x) đòi hỏi $x \in A$. Ta thấy điều kiện (1.1) dẫn đến (1.2) nhưng ngược lại không hoàn tàon đúng.

Ta có thể hạn chế các điều kiện (1.2) để có được điều kiện tương đương như sau

$x \in A$

$y \in V$

$q'(x) = y$ (1.5)

$q'(a) = q(a)$ với $x \neq a$ và $x \in A$

Trong (1.5), hai điều kiện đầu được gọi là điều kiện đầu (preconditions), hai điều kiện sau được gọi là các điều kiện sau (postconditions)

Ta cần kiểm tra (1.5) là chấp nhận được, nghĩa là có điều kiện (1.4) là bất hiến (invariant). Muốn vậy ta cần chứng minh định lý sau đây :

((1.4) và (1.5)) kéo theo (1.4)' (1.6)

Định lý về tính chấp nhận được (plausibility)

Ta có thể nối hai điều kiện tương đương (1.1) và (1.5)

q **chg** (x, y) q'

$x \in A$

$y \in V$ (1.7)

q' (x) = y

q' (a) = q (a) với x ≠ a và a ∈ A

Người ta nói q xác định trạng thái (state) của hệ thống, một điều kiện như (1.4) là một bất biến của hệ thống, (1.7) là đặc tả lệnh làm chuyển (tiến triển) trạng thái của hệ thống. Ở đây ta sử dụng các biến có đánh dấu nháy để chỉ trạng thái của hệ thống sau khi chuyển đổi. Ta cũng nói một đặc tả là chấp nhận được (plausible) nếu đặc tả đó bảo toàn bất biến của hệ thống.

Nhận xét

Dễ dàng chứng minh định lý (1.6) nhưng cũng dễ dàng bước lại định lý bằng cách xét phản ví dụ sau :

1	0	1	0		
2	0	**chg** (2, 5)		2	5
3	0	3	0		
	9	4	0		

Rõ ràng ví dụ trên thỏa mãn các điều kiện (1.4) và (1.5) nhưng không thỏa mãn (1.4)'

Thực ra, lệnh **chg** (2, 5) thay đổi nội dung địa chỉ x (cho giá trị y) nhưng vẫn giữ nguyên các địa chỉ khác, nghĩa là kích thước bộ nhớ không thay đổi.

IV.2. Hàm

Trên đây, ta đã vận dụng quan điểm toán học về hàm, sau đây ta tiếp tục làm rõ một số khái niệm và tính chất của quan điểm này.

Cho hai tập hợp X và Y, biểu thức X → X biểu diễn tập hợp các hàm toàn phần (total) với miền xác định (nguồn) là X và miền trị (đích) là Y. Tương tự, biểu thức X → Y biểu diễn tập hợp các hàm bộ phận (partical) từ nguồn X vào đích Y. Sự khác nhau giữa chúng là ở chỗ, một hàm bộ phận không hoàn toàn xác định cho mọi giá trị của nguồn X.

Nếu f là một hàm bộ phận thì ký hiệu dom (f) (domain) là tập hợp con của X mà f xác định. Trong trường hợp hàm toàn phần, miềm xác định và nguồn là đồng nhất. Ví dụ hàm q ở mục trên là hàm toàn phần với nguồn X = {1, 2, 3} và đích Y = $\{-2^{31}, ..., 2^{31}\}$, ta có thể viết :

q ∈ {1, 2, 3} → $\{-2^{31}, ..., 2^{31}\}$

Tuy nhiên đây là một hàm bộ phận từ tập số nguyên Z vào chính nó vì các tập hợp nguồn và đích của q đều là tập hợp con của Z.

q ∈ Z → Z

lúc này dom (q) = {1, 2, 3}

Ta cũng có thể xây dựng tập hợp con của đích chứa các giá trị xác định từ tập hợp con của nguồn, gọi là ran (f) (range)

Khi một hàm được định nghĩa, ta có thể liệt kê các thành phần của hàm. Ví dụ, ta có :

1 0 0

2 0 chg (2, 5) 5

3 0 0

tương ứng với hai hàm :

q = { 1→ 0, 2 → 0, 3 → 0 }

q' = { 1 → 0, 2 → 5, 3 → 0 }

Để nhân được các dom và các ran từ các hàm, người ta tập hợp các phần tử đặt ở bên trái và bên phải của mũi tên tương ứng :

dom (q) = {1, 2, 3} dom (q') = 1, 2, 3}

ran (q) = { 0 } ran (q') = { 0, 5 }

Một cách tổng quan, ta có kết quả sau :

dom ({ x → y }) = { x }

ran ({ x → y }) = { y } (2.1)

f ∈ X → Y kéo theo dom (f) = X

Cho hàm f ∈ X → Y và một tập hợp con S ⊆ X, người ta ký hiệu f \ S là hàm nhân được bằng cách loại khỏi dom (f) các phần tử của S. Đây là phép hạn chế tương ứng với định nghĩa sau :

dom (f \ S) = dom (f) - S

(f \ S) (x) = f (x) với x ∈ dom (f) - S

HÌNH VẼ

Ta có : { 1 → 5, 2 → 8, 3 → 6 } \ { 1, 2} = { 3 → 6 } \ { 1, 2 } = { 3 → 6 }

Cho hai hàm có cùng nguồn và cùng đích nhưng có các dom rời nhau. Ký hiệu f ∪ g là hợp của hai hàm theo định nghĩa sau :

dom (f ∪ g) = dom (f) ∪ dom (g)

$$(f \cup g)(x) = \begin{cases} f(x) \text{ nếu } x \in \text{dom}(f) \\ g(x) \text{ nếu } x \in \text{dom}(g) \end{cases} \quad (2.4)$$

Bằng cách dùng hai phép toán trên đây, ta có thể định nghĩa sự chồng lên (overload) của một hàm bởi một hàm khác. Ta ký hiệu f + g tác động lên hai hàm f và g có cùng nguồn và cùng đích mà lần này, các dom không nhất thiết rời nhau :

Ta có : $f + g = (f \setminus dom(g)) \cup g$ (2.5)

Từ (2.3) ta có :

$$(f + g)(x) = \begin{cases} (f \setminus dom(g))(x) & \text{nếu } x \in dom(f \setminus dom(g)) \\ g(x) & \text{nếu } x \in dom(g) \end{cases} \quad (2.6)$$

hay

$$(f + g)(x) = \begin{cases} f(x) & \text{nếu } x \in dom(f) - dom(g) \\ g(x) & \text{nếu } x \in dom(g) \end{cases}$$

HÌNH VẼ

Ví dụ : $\{ 1 \to 0, 2 \to 0, 3 \to 0 \} + \{ 2 \to 5 \}$

$= (\{ 1 \to 0, 2 \to 0, 3 \to 0 \} \setminus \{ 2 \} \cup \{ 2 \to 0 \}$

$= \{ 1 \to 0, 3 \to 0 \} \cup \{ 2 \to 5 \}$

$= \{ 1 \to 0, 2 \to 5, 3 \to 0 \}$

Lý do cơ bản để đưa vào các phép toán \, ∪ và + thay vì sử dụng một cách hệ thống những định nghĩa của chúng (2.3), (2.4) và (2.7) trong việc hình thức hóa là ở chỗ tâ có thể chứng minh dễ dàng dãy các tính chất đại số sẽ sử dụng về sau.

Sau đây là một số tính chất :

$(f \cup g) \setminus S = (f \setminus G) \cup (g \setminus S)$

$(f + g) \setminus S = (f \setminus S) + (g \setminus S)$

$(f \setminus S) \setminus T = f \setminus (S \cup T)$

$f \cup g = g \cup f$

$(f \cup g) \cup h = f \cup (g \cup h)$

$dom(f \cup g) = dom(f) \cup dom(g)$ (2.8)

$ran(f \cup g) = ran(f) \cup ran(g)$

$(f + g) + h = f + (g + h)$

$dom(f + g = dom(f) \cup dom(g)$

Bây giờ ta có thể sửa chữa đặc tả đã nhận được ở cuốn mục trước (1.7). Đặc tả này rõ ràng đơn giản hơn :

$q\ chg\ (x, y)\ q'$

$x \in A$

$y \in V$

$q' = q + \{ x \to y \}$

Định lý về tính chấp nhận được (plausilility) bây giờ dễ dàng được chứng minh bởi phép tính hình thức đơn giản (simple formal calculus).

Theo (2.1) và (2.8), ta có :

dom (q') = dom (q) ∪ { x }

ran (q') ⊂ ran (q) ∪ { y }

Nhưng ta có giả thiết (1.4) vf giả thiết (2.8) và theo (2.2)

dom (q) = A và { x } ⊂ A

ran (q) ⊂ V và { y } V

Như vậy :

dom (q') = A

ran (q') ⊂ V

Từ đó theo (2.2) thì q' ∈ A → V

IV.3. Hợp thức hóa và phục hồi

Trong mục 1, ta đã đặc tả cách hoạt động của một bộ nhớ trong đó, ta có thể thay đổi nội dung của nó. Bây giờ, ta sẽ tiếp tục phát triển ví dụ này bằng cách đặt ra hai yêu cầu bổ sung ta muốn rằng những thay đổi trên bộ nhớ có đặc tính tạm thời, nghĩa là ta có thể thay đổi trở lại nhờ một phép toán thích hợp, mặt khác ta có hợp thức hóa (validation) bằng cách trả lại những thay đổi trước đó.

Ta gọi not (restart) và vld (validate) là những thao tác mới. Những yêu cầu bổ sung vừa nêu có thể biểu diễn hình thức bằng cách kết hợp các điều kiện sau đây :

q vld q_1

q_1 op q_2

...

q_{n-1} op q_n

q_n rst q'

Trong đó op (operations) là một thao tác dạng chung (x, y) hay rdm, dẫn đến điều kiện :

q' = q

với q và n bất kỳ. Vả lại, yêu cầu về tính "trong suốt" của phép toán hợp thức hóa dẫn đến điều kiện :

q' = q

Cách giải quyết hiên nhiên nhất mang tính ý niệm là làm tăng gấp đôi bộ nhớ. Như vậy trạng thái của hệ thống bây giờ được đặc trưng bởi hai biến p và q như sau :

p ∈ A → V

$q \in A \rightarrow V$ (3.1)

Bộ nhớ q đóngvai trò bộ nhớ trước đó, còn bộ nhớ p dùng để khôi phục trạng thái cũ khi cần khởi động lại. Sau đây là đặc tả của 3 thao tác cho hệ thống với mod thay thế chg :

(p, q) mod (x, y) ((p', q')

p' = p (3.2)

q chg (x, y) q'

Ta thấy thao tác thay đổi bộ nhớ xuất hiện như một sự mở rộng, ở mức đặc tả, của phép toán chg :

(p, q) rst (p', q')

q' = p

q' = q (3.3)

(p, q) vld (p', q')

p' = q

q' = q (3.4)

Dễ dàng chứng minh rằng cả ba phép toán này đều chấp nhận được, nghĩa là sau khi thực hiện chúng, ta có (3.1)'. Nói cách khác, p và q được thay thế bởi p' và q' trong (3.1), với giả thiết rằng (3.1) đã được kiểm chứng trước khi thực hiện chúng.

Rốt cuộc, ta phải chứng minh rằng việc kết hợp các điều kiện sau đây :

(p, q) vld (p_1, q_1)

(p_1, q_1) op (p_2, q_2)

...

(p_{n-1}, q_{n-1}) op (p_n, q_n)

(p_n, q_n) rst (p', q')

dẫn đến q' = q

Thật vậy, theo (3.4), ta có :

p_1 = q

và theo (3.2) và (3.4) do op là một trong hai phép toán mod (x, y) hoặc rst, ta có :

p_n = ... p_1

Cuối cùng, theo (3.3) :q' = p_n

Rõ ràng phép hợp thức hóa là trong suốt vì dẫn đến q' = q theo (3.4)

Sau đây là một ví dụ về các phép toán trên :

```
     q      p
1    0      0
2    0      0
3    0      0
mod (1, 1)
1    1      0
2    0      0
3    0      0
mod (2, 2)
1    1      0
2    2      0
3    0      0
mod (1, 3)
1    3      0
2    2      0
3    0      0
vld
1    3      3
2    2      2
3    0      0
mod (3, 1)        q      q
1    3      3
2    2      2
2    1      0
not
1    3      3
2    2      2
3    0      0
```

Phần tiếp theo sẽ làm mịn mô hình này, nghĩa là đưa vào các biến trạng thái mới để thể hiện các ràng buộc về phần cứng và phần mềm.

IV.4. Bắt đầu triển khai thực tiễn

Về mặt thực tiễn, có nhiều cách để triển khai hệ thống đã được đặc tả trong mục trước. Thật vậy, có nhiều yếu tố kỹ thuật có thể ảnh hưởng đến cách triển khai ; chẳng hạn kích thước không gian V các giá trị đóng vai trò quan trọng : giả sử rằng các phần tử của tập hợp A tương ứng với các địa chỉ của các trang trong một hệ thống có bộ nhớ phân trang (paging). Trong trượnghợp này, mỗi "giá trị" sẽ tương ứng với nôi dung của một trạng thái có kích thước diển hình là 1 bytes. Nếu những trang này dùng để thể hiện một bộ nhớ ảo 4 Mbytes, thì ta sẽ thấy rằng có 4096 trang và bởi vậy thời gian dùng để thực hiện các sao chép cần thiết cho các thao tác khởi động và hợp thức hóa có thể tỏ ra nặng nề.

Trong trường hợp vừa nêu (ta sẽ triển khai thực tiễn trong mục này), cách giải quyết là sử dụng cách gián tiếp : hai làbộ nhớ p và q tạo thành trạng thái của hệ thống đã xét trong mục trước, sẽ được thay thế bởi hai bảng (hai hàm) a và n (a: ancient, n : new) chứa các con trỏ tới bộ nhớ m. Hệ thống mới sẽ được đặc trưng bởi các thành phần sau :

$$\begin{aligned} a &\in A \rightarrow D \\ n &\in A \rightarrow D \\ m &\in D \rightarrow V \end{aligned} \qquad (4.1)$$

Trong đó D là tập hợp các địa chỉ của bộ nhớ m. Sau đây là sơ đồ minh họa hệ thống mới này :

	n	a	m	
1	2	2	1	2
2	4	6	2	5
3	5	5	3	5
	4	8		
	5	1		
	6	4		

Trong ví dụ trên, cũng trong các ví dụ trên về sau, ta tiếp tục sử dụng các giá trị V như trước.

Bằng cách kết hợp các bảng a và n với m, ta nhận được các bảng p và q của hệ thống cũ :

q	p	
1	5	5
2	8	4
3	1	1

Như vậy, ta có thể thấy rằng hai hệ thống không độc lập vớinhau : hai hệ thống mới hiện thực hóa hệ thống cũ và hệ thống cũ thể hiện sự thay đổi biến như sau

p (x) = m (a (x))

q (x) = m (n (x)) với x ∈ A (4.2)

Ta có thể kiểm chứng ngay được rằng những thay đổi của biến là có ý nghĩa (rõ ràng vì các hàm a, n và m là toàn phần) và chặt chẽ với bất biến (3.1) chỉ rõ rằng p và q dều thuộc tập hợp A → V

Phép biến đổi mod dặc tả bởi các điều kiện (3.2) sẽ được triển khai bởi một phép toán mới mod_1. Phép mod_1 sẽ ghi một giá trị mới, một địa chỉmà không thuộc vào miền trị (range) của n cũng như miền trị của a, nói cách khác, địa chỉ này chỉ phụ thuộc vào tập hợp :

ran (n) ∪ ran (a)

hay rõ hơn, thuộc tập hợp D - (ran (n) ∪ ran (0)), tập hợp được đặt tên là L (Liberty).

Nếu L ≠ ∅, một điều kiện trước dược đặt ra, thì ta có thể chọn một địa chỉ bất kỳ u, với đặc tả sau :

(a, n, m) mod_1 (x, y) (a', n', m')

x ∈ A

y ∈ V

L ≠ ∅

a' = a

n' = n + { x → u }

m' = m + { u → y } (4.3)

trong đó

L = ran (n) ∪ ran (a)

u ∈ L

Ta cần chứng minh rằng đặc tả (4.3) là chấp nhận được, nghĩa là :

((4.1) và (4.3)) kéo theo (4.1)' (4.4)

Mệnh đề (4.4) trên đây là hiển nhiên. Tiếp theo ta cần chứng minh phép mod_1 phù hợp (đúng) với phép mod đã đặc tả ở (3.2) như sau :

((4.2), (4.2)' và (4.3)) kéo theo (3.2) (4.5)

Mệnh đề này không hiển nhiên, tương ứng với sơ đồ giao hoán dưới đây :

HÌNH VẼ

Nói cách khác, nếu các giá trị của các biến (a, n, m) và (a', n', m') thỏa mãn đặc tả (4.3), thì khi trở về các biến cũ (p, q) và (p', q') (các biến đã bị thay đổi trở thành các biến (a, n, m) và (a', n', m') bởi các phép biến đổi (4.2) và 4.2)') các giá trị của các biến (p, q) và (p', q') thỏa mãn đặc tả (3.2).

Nói gọn lại, (4.3) hiện thực (3.2)

Các phép toán mới khởi tạo lại các hợp thức hóa mô tả trong các đặc tả sau đây rõ ràng chấp nhận được và phù hợp :

(a, n, m) rst_1 (a', n', m')

a' = a

n' = n (4.6)

m' = m

(a, n, m) vld_1 (a', n', m')

a' = n

n' = n (4.7)

m' = m

Nhìn vào các công thức, ta thấy đã không chép lại các giá trị nhưng chỉ có các địa chỉ có thể có theo một sự tiết kiệm đáng kể về thời gian nhưng không lớn lắm về không gian nhớ. Giả thiết với 4096 trang và địa chỉ của D là 2 bytes, mỗi bảng a và n sẽ chiến 8 Kbytes.

Hệ thống mới hoạt động qua ví dụ sau :

	n	a	m		
1	1	1	1	0	
2	2	2	2	0	
3	3	3	3	0	
	4	0			
	5	0			
	6	0			

mod_1 (1, 1)

1	4	1	1	0	L = { 4, 5, 6 }
2	2	2	2	0	u = 4 ∈ L
3	3	3	3	0	(chọn u = min (L))
	4	1			
	5	0			
	6	0			

IV.5. Phép hợp thành (cấu tạo)

Ở mục trước, ta đã sử dụng một cách phi hình thức phép hợp thành (composition opertion) của p và q theo a, m và n. Trong mục này, ta tiếp tục định nghĩa phép hợp thành một cách chặt chẽ hơn.

Phép hợp thành, ký hiệu là 0, tác động lên hai toán hạng là hai hàm, chẳng f và g, thuộc về các tập hợp $X \to Y$ và $Y \to Z$ tương ứng, sao cho hàm fog thuộc về tập $X \to Z$

Phép hợp thành có thể được định nghĩanhư sau :

$dom (fog) = \{ x \in dom (g) / g (x) \in dom (f) \}$

$(fog) (x) = f (g (x))$ với $x \in dom (fog)$ (5.1)

Ví dụ :

$\{ 2 \to 6, 5 \to 8 \} o \{ 1 \to 2, 4 \to 3, 7 \to 5 \} = \{ 1 \to 6, 7 \to 8 \}$

Từ định nghĩa trên có thể suy ra ngay rằng nếu các hàm f và g đều toàn phần (nói cách khác, nếu dom (g) = X và dom (f) = Y), thì hàm fog cũng là toàn phần. Một cách tổng quan hơn, nếu miền trị của g nằm trong miền xác định.của f thì hai hàm g và fog có cùng miền xác định. Ta có thể dễ dàng xây dựng một số luật kiểu đại số để nối liền phép hợp thành với các phép hội và hạn chế đã xét ở mục 2. Sau đây là một số luật như vậy :

$(f \cup g) o h = (f o h) \cup (g o h)$

$f o (g \cup h) = (fog) \cup (f o h)$ (5.2)

$S \cap rang (g) = \varnothing$ kéo theo $(f \setminus s) og = fog$ (5.3)

$dom (f) \cap ran (g) = \varnothing$ kéo theo $fog = \{ \}$ (5.4)

Biểu thức { } chỉ định hàm rỗng

$f o (g \setminus S) = (fog) \setminus S$ (5.5)

$\{ x \to y \} o \{ x \to u \} = \{ x \to y \}$ (5.6)

Từ các luật trên, ta có thể đơn giản hóa phép thay đổi biến đã định nghĩa ở (4.2) như sau :

$p = m o a$

$q = m o n$ (5.7)

Và ta có thể chứng minh dễ dàng định lý phù hợp (4.5) bằng cách sử dụng các tính chất trên. Thật vậy, ta cần chứng minh hai đẳng thức sau đây :

$m' o n' = (m o n) + \{ x \to y \}$

$m' o o' = m o a$

Nghĩa là :

$(m + \{ u \to y \}) o (n + \{ x \to u \}) = (m o n) + \{ x \to y \}$

$(m + \{ u \to y \}) \circ a = m \circ a$

với các giả thiết :

$u \notin ran(n)$ nghĩa là $\{ u \} \cap ran(n) = \varnothing$ (5.8)

$u \notin ran(a)$ nghĩa là $\{ u \} \cap ran(a) = \varnothing$ (5.9)

Ta có kết quả phụ như sau :

$(m \setminus \{ u \}) \circ (n \setminus \{ x \}) = ((m \setminus \{ u \}) \circ n) \setminus \{ x \}$ theo (5.5) $= (m \circ n) \setminus \{ x \}$

theo (5.3) và (5.8). Nhưng :

$\{ x \to y \} \circ (n \setminus \{ x \}) = \{ \}$

theo (5.4), (2.2) và (5.8). Và ta cũng có :

$(m \setminus \{ u \}) \circ \{ x \to u \} = \{ \}$

theo (5.4) và (2.2)

$\{ u \to y \} \circ \{ x \to u \} = \{ x \to y \}$

theo (5.6)

Như vậy theo (5.2) và (2.5) :

$(m + \{ u \to y \}) \circ (m + \{ x \to u \}) = (m \circ n) \setminus \{ x \} \cup \{ x \to y \} = (m \circ n) + \{ x \to y \}$

Mặt khác ta có :

$m \setminus \{ u \} \circ a = m \circ a$ theo (5.3)

$\{ u \to y \} \circ a = \{ \}$ theo (5.4)

Như vậy :

$(m + \{ u \to y \}) \circ a = m \circ a$ theo (2.4) và (5.3)

IV.6. Triển khai thứ hai

Mục này sẽ tối ưu cách triển khai đầu tiên đã trình bày trong mục 4 bằng cách xây dựng tập hợp L các địa chỉ tự do của D, tập hợp mà ta đã chọn tùy ý một phần tử u trong đặc tả phép toán mod_1 ở (4.3).

Ý tưởng thiết kế cách triển khai thứ hai này nằm ở chỗ giữ lại trạng thái của mỗi địa chỉ của D mà địa chỉ này có thể thuộc về một (và chỉ một mà thôi) trong 4 tập hợp rời nhau như sau :

RN _ RN

RA ∩ RN

RN _ RA

RA ∪ RN = L

trong đó RA = ran (a), RN = ran (n)

Tùy theo một địa chỉ d của D thuộc về một trong bốn tập hợp trên, ta nói trạng thái tương ứng sẽ là :

old (cũ) $d \in ran(a)$

common (chung) $d \in ran(a)$ và $d \in ran(n)$

new (mới) $d \in ran(a)$

free (tự do) $d \notin ran(a)$ và $d \notin ran(n)$

Khi một địa chỉ tự do được chọn, khi một thay đổi xảy ra, địa chỉ đó chuyển qua trạng thái mới ; về địa chỉ quá tải trong bảng n, nếu địa chỉ đó không phân chia bên trong bảng n (nghĩa là nếu hàm nlà đơn ánh và điều này được giả thiết là luôn đúng), khi đó, địa chỉ sẽ trở nên tự do nếu đạng ở trạng thái mới hoặc chuyển sang trạng thái cũ nếu đang ở trạng thái chung.

Khi một phép hợp thức hóa hay khởi động lại các địa chỉ tự do hay chung vẫn như cũ. Các địa chỉ mới chuyển thành tự do khi một sự khởi động lại và là trường hợp chung khi hợp thức hóa.

Cuối cùng, các địa chỉ cũ chuyển thành tự do khi hợp thức hóa và trở thành chung khi khởi động lại.Chú ý rằng các địa chỉ cũ không liên quan đến sự thay đổi. Sơ đồ dưới đây tóm tắt một cách phi hình thức những chuyển đổi khác nhau này.

Mục đích để hình thức hóa phương pháp này, ta đưa vào một biến mới s định nghĩa trạng thái của mỗi địa chỉ của D.

$s \in D \rightarrow \{fr, nw, cm, ol\}$ (6.1)

Ta có bất biến sau đây :

$RA - RN = adr(ol)$
$RA \cap RN = adr(cm)$ (6.2)
$RN - RA = adr(nw)$
$RA \cup RN = adr(fr)$

HÌNH VẼ

Trong đó :

$RA = ran(a)$
$RN = ran(n)$
$adr(z) = \{x \in D / s(x) = Z\}$

với $Z \in \{fr, nw, cm, ol\}$

Cuối cùng bất biến thứ ba chỉ rõ rằng cả hai hàm n và a đều đơn ánh, nghĩa là hai địa chỉ của A phân biệt sẽ luôn luôn tương ứng với các địa chỉ của D phân biệt qua các hàm này.

Tập hợp các hàm từ A vào D như vậy được ký hiệu

bởi A ⤆ D như vậy

$n \in A \setminus D$

$a \in A \setminus D$

Bây giờ sẽ là định nghĩa ba hàm chuyển tiếp lần lượt là f, g và h sử dụng khi thay đổi (cho các địa chỉ của D liên quan), khởi động lại thay cho hợp thức hóa (cho mọi địa chỉ của D) :

f = {fr → nw,

nw → fr, thay đổi

cm → ol}

g = {fr → fr,

nw → fr,

cm → cm,khởi động lại

ol → cm}

h = {fr → fr,

nw → cm,

cm → cm,hợp thức hóa

ol → fr}

Ta có đặc tả của 3 phép toán mới mod_2, rst_2, và vld_2 xuất hiện như là các mở rộng tương ứng từ mục 4 :

(a, n, m, s) mod_2 (x, y) (a', n', m', s')

(a, n, m) mod_1 (x, y) (a', n', m') (6.5)

s' = s + {u → f (s (u)), v → f (s) (v))}

xem (4.3)

trong đó :

L = { Z ∈ D | s (z) = fr }

u ∈ L

v = n (x)

(a, n, m, s) rst_2 (a', n', m', s') (6.6)

s' = gos xem (4.6)

(a, n, m, s) vld_2 (a', n', m', s') (6.7)

(a, n, m) vld_1 (a', n', m') xem (4.7)

s' = hos

Sau đây là một quá trình chuyển đổi của hệ thống

CHỪA

Chứng minh

Mặc dù trong mục trước ta đã kiểm chứng kỹ lưỡng đặc tả hệ thống và nhận được kết quả thõa mãn, nhưng chưa đảm bảo được tính đúng đắn của đặc tả trong mọi trường hợp.

Để đi đến một kết quả tổng quát, ta cần phải chứng minh không phải cho một trường hợp đặc biệt nào đó mà phải cho các dữ liệu tượng trưng thỏa mãn những giả thiết nào đó, các phép toán đã đặc tả là phù hợp và chấp nhận được.

Việc chứng minh tính phù hợp của các phép toán đặc tả ở (6.5), (6.6) và (6.7) so với các phép toán đặc tả ở (4.3), (4.6) và (4.7) là hiển nhiên vì rằng trong cách lập các công thức thì các phép toán (4.3), (4.6) và (4.7) một cách tương ứng.

Trái lại, việc chứng minh tính chấp nhận được phức tạp hơn. Trước hết ta cần chứng minh ba nhóm định lý bất biến sau đây :

((6.1) và (6.5)) kéo theo (6.1)' (7.1)

((6.2) và (6.5)) kéo theo (6.2)' (7.2)

((6.3) và (6.5)) kéo theo (6.3)' (7.3)

((6.1) và (6.6)) kéo theo (6.1)' (7.4)

((6.2) và (6.6)) kéo theo (6.2)' (7.5)

((6.3) và (6.6)) kéo theo (6.3)' (7.6)

((6.1) và (6.7)) kéo theo (6.1)' (7.7)

((6.2) và (6.7)) kéo theo (6.2)' (7.8)

((6.3) và (6.7)) kéo theo (6.3)' (7.9)

Đối với 3 định lý ở nhóm 1, ta có thể dẫn đến các giả thiết cho các điều kiện sau đây :

$a \in A \cup D$

$n \in A \cup D$

$u \notin RN$ (7.10)

$u \notin RA$

$v \in RN$

Trong đó u và được định nghĩa ở (6.5) và RA, RN được định nghĩa ở (6.2)

chứng minh (7.1)

Một khó khăn nhỏ là hàm chuyển tiếp f được định nghĩa ở (6.4) là hàm bộ phận. Cần chứng minh rằng s' được định nghĩa đúng, nghĩa là các biểu thức f (s (u)) và f (s (v)) trong (6.5) có nghĩa, nói cách khác ta có :

s (u) ∈ dom (f)

s (v) ∈ dom (f)

điều này hiển nhiên vì rằng theo (7.10), ta có :

s (u) = fr

s (v) = { nw, cm}

và theo (6.4) ta có

dom (f) = {fr, nw, cm}

Để chứng minh (7.2) và (7.3) ta cần kết quả sau đây liên quan đến sự quá tải của một hàm đơn ánh thừa nhận mà không chứng minh :

f ∈ X ⇀ Y kéo theo f' ∈ X ⇀ Y

x ∈ dom (f) ran (f') = r'

y ∉ Y - ran (f) y ≠ f (x)

Trong đó :

f' = f + {x → y}

r' = (ran (f) - {f (x)}) ∪ {y}

chứng minh (7.2) : Theo (7.10), (7.11) và (6.5) ta có

RA' = RA (vì rằng a' = a theo (4.3))

RN' = (RN {V}) ∪ {u} theo 7.11

u ≠ v theo 7.10

HÌNH VẼ

Xảy ra hai trường hợp :

1/ v ∈ RA, nghĩa là s (v) = cm

Khi đó :

RA' - RN' = (RA - RN) ∪ {v}
RA' ∩ RN' = (RA ∩ RN) - {v}
RN' - RA' = (RN - RA) ∪ {u}
RA' ∪ RN' = (RA ∪ RN) - {u}

HÌNH VẼ

2/ v ∉ RA, nghĩa là s (v) = nw

Khi đó :

RA' - RN' = RA - RN

$RA' \cap RN' = RA \cap RN$

$RN' - RA' = ((RN - RA) - \{v\}) \cup \{v\}$

$RN' \cup RA' = ((RN \cup RA) - \{u\} \cup \{v\}$

HÌNH VẼ

Như vậy các chuyển tiếp từ s (u) và s (v) như sau

$fr \rightarrow nw$ với u

$cm \rightarrow ol$ với v trong trường hợp 1/

$nw \rightarrow fr$ với v trong trường hợp 2/

Các chuyển tiếp này tương ứng với các chuyển tiếp đã chỉ ra bởi hàm g định nghĩa ở (6.4)

IV.7. Triển khai thực hiện lần thứ ba

Lần này, ta giả thiết rằng xảy ra các sai sót cần phải dự phòng nhờ hệ thống hợp thức hóa và khởi động lại.

Giả sử các bảng a, n, m và s được cài đặt trên các thiết bị nhớ khác như sau :

HÌNH VẼ

Từ cách tổ chức này, ta muốn dự phòng các sai sót tác động lên bộ nhớ trong bằng cách khởi động lại từ đĩa. Ở đây, ta đã cài đặt các bảng s và s trong bộ nhớ trong với mục đích tăng tính hiệu quả của phép thay đổi bộ nhớ là nhanh nhất có thể.

Với mục đích trên, việc khởi động lại làm thay đổi bảng s từ chính nó (thực tế là s' = gos theo (6.6)) không còn có tác dụng nữa vì rằng ta giả thiết rằng bộ nhớ trung tâm là s sẽ không còn nữa.

Một giải pháp là gấp đôi bảng s lên đĩa cho mỗi lần hợp thức hóa. Xây dựng bảng mới t được tương ứng với bất biến như sau :

$t \in D \rightarrow \{fr, cm\}$ (8.1)

Chú ý rằng ta không cần mọi giá trị các trạng thái địa chỉ đĩa vì rằng t chỉ dùng để tái sinh lại bảng s khi khởi động lại, từ đó ta có bất biến bổ sung như sau :

$\{x \in D \mid t(x) = cm\} = ran(a)$ (8.2)

Ta có các đặc tả mới như sau :

$(a, n, m, s, t)\ mod_3\ (x, y)\ (a', n', m', s', t')$

$(a, n, m, s)\ mod_2\ (x, y)\ (a', n', m', s')$ (8.3)

t' = t xem (6.5)

$(a, n, m, s, t)\ rst_3\ (a', n', m', s', t')$

$(a, n, m)\ rst_1\ (a', n', m')$(8.4)

s' = t xem (4.6)

t' = t

(a, n, m, s, t) vld_3 (a', n', m', s', t') (8.5)

(a, n, m, s) vld_2 (a', n', m', s') xem (6.7)

t' = s'

Dễ dàng chứng minh rằng cả ba đặc tả trên là phù hợp và chấp nhận được. Mặt khác ta thấy rằng phép khởi động lại tái sinh bộ nhớ trong từ đĩa và chỉ từ đĩa mà thôi.

Triển khai lần thứ tư và lần thứ năm

Ta sẽ mã hóa các giá trị fr, nw, cm và ol như hàm đơn ánh k như sau :

k = {(0, 0) → fr,
(0, 1) → ol,
(1, 0) → cm, (9.1)
(1, 1) → nw}

Sau đó ta biểu diễn các hàm s và t nhờ ba chuỗi bit như sau :

b ∈ D → {0, 1} để biểu diễn s

c ∈ D → {0, 1}

d → {0, 1}để biểu diễn t (9.2)

Cuối cùng, thay đổi các biến tương ứng với các điều kiện sau :

s (x) = k (b (x), c (x))
t (x) = k (d (x), 0) với x ∈ D (9.3)

Giả sử là phép bù (đảo ngược bít) như sau

0 = 1,1 = 0 (9.4)

Ta thấy có thể mã hóa hàm f đã định nghĩa ở (6.4) nhờ hai phép bù và hàm h cũng đã dịnh nghĩa ở (6.4) nhờ phép sao chép và đặt lề 0 tương ứng với hàm :

Z ∈ D → {0} (9.5)

Ta có 3 đặc tả mới như sau :

(a, n, m, b, c, d) mod_4 (x, y) (a', n', m', b', c', d')
(a, n, m) vld_1 (a', n',m')
b' = b (9.8)
c' = z xem (4.7)
d' = b

Bây giờ ta chỉ cần tóm tắt lại những gì đã làm cho đến lúc này, nghĩa là một mặt, sao chép lại các đặc tả (4.3), (4.6) và (4.7) vào bên trong của (9.6), (9.7) và (9.8), mặt khác, nhóm các bất biến (4.1), (6.1), (6.2), (6.3), (8.1), (8.2) và (9.2)

Điều này làm được bằng cách khử các biến trở thành dư thừa (chứa s và t) bởi các phép thay đổi biến (9.3).

Khi sao chép, ta thấy rằng đặc tả (4.3) chứa điều kiện trước $L \neq \varnothing$ không dễ gì tính được. Để khắc phục nhược điểm này ta đưa vào một biến mới w là một số nguyên

$w \in N$ (9.9)

w chứa các phần tử của tập hợp L (cardinality)

$w = | RA \cup RN |$

Ta thừa nhận ngầm rằng các tập hợp D và A đều là hữu hạn. Khi hợp thức hóa và khởi động lại, bộ đếm w được khởi tạo giá trị $|D| - |A|$ (vì rằng a và n đều đơn ánh) là một hằng số dương của hệ thống. Khi có sự thay đổi, bộ đếm w tăng lên khi và chỉ khi địa chỉ v, quá tải trong n, đang ở trạng thái cm, nghĩa là nếu b (v) = 1 và nếu c (v) = 0

Với sự mở rộng mới này, bất biến của hệ thống lúc này sẽ là :

$a \in A \rightarrow D$ (6.3)
$n \in A \rightarrow D$ (6.3)
$m \in D \rightarrow V$ (4.1)
$b \in D \rightarrow \{0, 1\}$ (9.2)
$c \in D \rightarrow \{0, 1\}$ (9.2)
$d \in D \rightarrow \{0, 1\}$ (9.2)
$d \in D \rightarrow \{0, 1\}$ (9.2)
$w \in N$ (9.9)
$RA - RN = \{x \in D / b (x) = 0 \text{ và } c (x) = 1\}$ (6.2)
$RA \cap RN = \{x \in D / b (x) = 1 \text{ và } c (x) = 0\}$ (6.2)
$RN - RA = \{x \in D / b (x) = 1 \text{ và } c (x) = 1\}$ (6.2)
$RA \cup RA \{x \in D / b (x) = 0 \text{ và } c (x) = 0\}$ (6.2)
$RA = \{x \in D / d (x) = 1\}$(9.2)
$W = | RA \cup RN |$ (9.10)

Trong đó

RA = ran (a)
RN = ran (n)

Sau đây là các đặc tả được tóm tắt bằng cách thay thế các danh sách dài các biến bởi hai biến trạng thái state và trạng thái có dấu nháy (') state'

$state\ mod_5\ (x, y)\ state'$
$x \in A$
$y \in V$
$w \neq 0$
$a' = a$
$n' = n + \{x \rightarrow u\}$

$m' = m + \{u \to y\}$
$b' = b + \{u \to b\ (u)\ v \to b\ (v)\}$ (9.12)
$c' = c + \{u \to c\ (u), v \to c\ (v)\}$
$d' = d$
$(b\ (v) = 1\ \&\ c\ (v) = 0) \Rightarrow w' = w - 1$
Trong đó
$u \in \{Z \in D \mid b\ (z) = 0\ \&\ c\ (z) = 0\}$
$v = n\ (x)$
state rst_5 state'
$a' = a$
$n' = a$
$m' = m$
$b' = d$ (9.13)
$c' = z$
$d' = d$
$w' = \mid D \mid - \mid A \mid$
state vdl_5 state'
$a' = n$
$n' = n$
$m' = m$
$b' = b$ (9.14)
$c' = z$
$d' = b$
$w' = \mid D \mid - \mid A \mid$

Trong đó $Z \in D \to \{0\}$

Sau đây là quá trình biến đổi cụ thể

IV.8. Đặc tả làm gì ?

Sau đây ta sẽ trả lời câu hỏi về mục đích (cho ai, cho cái gì) của các công việc mà ta đã làm. Vai trò đầu tiên của một đặc tả là cho phép mở ra các tranh luận về đề tài đặc tả đề tài đặc tả để cập đến. Thực tế, khác với một chương trình, một đặc không phải viết ra để máy tính có thể hiểu được mà để cho những NSD có thể hiểu và tin tưởng vào tính đúng đắn của nội dung đã đặc tả.

Từ đặc tả lúc đầu ở mục 3 cho đến các đặc tả tiếp theo ở các mục 4, 6, 8 và 8, ta đã sử dụng các ràng buộc mỗi lúc một mang tính thực tiễn. Ta đã khẳng định được tính đúng đắn của đặc tả bởi các chứng minh định lý phù hợp và chấp nhận được : bảo toàn tính bất biến.

Bây giờ vấn đề là sử dụng các đặc tả để lập trình. Trong mục này, ta sẽ lập trình cho các trường hợp đặc tả (9.12), (9.13) và (9.14), bằng cách sử dụng ngôn

ngữ giả Pascal. Ta đưa vào các quy tắc để tiết lập mối liên hệ giữa đặc tả và lập trình.

Quy tắc 1 :

Khi một đặc tả chứa các điều kiện trước, chương trình tương ứng sẽ là một hàm. Theo nghĩa của Pascal, mỗi giá trị sai của điều kiện trước sẽ trả về biến trạng thái state một giá trị phân biệt.

Chương trình tương ứng với đặc tả (9.12) là như sau :

```
if not (x in A) then
state := bad - x
else if not (y in v) then
state := bad - y
else if w = 0 then
state := no more place
else begin
State := OK ;
Modification {gọi thủ tục}
end ;
```

Quy tắc 2 :

Khi một đặc tả chứa các biến phụ, ta có thể mở mọi thủ tục chứa các biến này như là các biến cục bộ. Thủ tục này bắt đầu bởi các lệnh khởi động

Thủ tục Modification như sau :

```
procedure Modification ;
var u, v : D
begin
    u := 1 ; { chọn u là địa chỉ bé nhất của L}
    while (b (u) ≠ 0) or (c (u) ≠ 0) do
    u := u + 1 ;  (10.2)
    v := n (x)
    { tiếp tục thân thủ tục
end ;
```

Quy tắc 3 :

Các điều kiện sau khác nhau nếu có dạng a' = ... (trong đó dấu ba chấm ... chỉ định một biểu thức khởi động chứa các biến có đánh dấu nháy), thì có thể được chuyển thành phép gán qua các quy tắc bổ trợ như sau :

- Loại bỏ các điều kiện sau dạng đẳng thức.

Ví dụ : d' = d

- Thay thế dấu = bởi dấu gán bằng :=

- Thực hiện phép tối ưu khi một hàm là quá tải

- Loại bỏ các dấu nháy '

- Thay thế một điều kiện sau bởi cấu trúc điều kiện if... else :

Các điều kiện sau không bị loại bỏ của đặc tả (9.12) như sau :

```
if not (x in A) then
      state := bad - x
   else if not (y in V) then
      state := bad - y(10.1)
   else if w = 0 then
      state := no-more-place
   else begin
      state := O.K ;
      Modication { gọi giá trị thủ tục}
      end ;
```

Quy tắc 2 :

Khi một đặc tả chứa các biến phụ, ta có thể mở một thủ tục chứa các biến này như là các biến cục bộ. Thủ tục này bắt đầu bởi các lệnh khởi động.

Thủ tục Modication như sau :

```
Procedure Modication ;
var u, v : D
begin
   u := 1 ; { chọn u là địa chỉ bé nhấtcủa L}
   uhile (b (u) ≠ 0) or (c (u) ≠ 0) do
   u := u + 1 ;  (10.2)
   v := n (x)
   { tiếp tục thân thủ tục
end ;
```

Quy tắc 3 :

Các điều kiện sau khác nhau nếu đều có dạng a' = ... (trong đó dấu chấm ... chỉ dịnh một biểu thức không chứa các biến có đánh dấu nháy), thì có thể được chuyển thành phép gán qua các quy tắc bổ trợ như sau :

- Loại bỏ các điều kiện sau dạng đẳng thức.

Ví dụ : d' = d

- Thay thế dấu = bởi dấu gán bằng :=

- thực hiện phép tối ưu khi một hàm là quá tải.

- Loại bỏ các dấu nháy '

- Thay thế một điềukiện sau bởi cấu trúc điều kiện if ... else :

Các điều kiện sau không loại bỏ của đặc tả (9.12) như sau :

n (x) := u ;

m (u) := y ; (10.3)

b (u) := b (u) ; b (v) := b (v) ;

c (u) := c (u) ; c (v) = c (v) ;

if (b (v) = 1) and (c (v) = 0) then w := w - 1

Quy tắc 4 :

Một cách hệ thống các hệ chương trình đã viết được bởi các quy tắc đảm bảo tính "song song" đưa vào từ đặc tả. Từ các đoạn chương trình (10.1), (10.2) và (10.3) ta nhận được công thức đấy đủ hơn như sau :

Một số đề thi

I. Đặc tả (Specification)

1. Cho ma trận vuông A cấp $n \times n$. Viết đặc tả thể hiện : ***a)*** Mỗi phần tử trên đường chéo chính là phần tử lớn nhất trên cùng hàng đi qua phần tử đó. ***b)*** Mỗi phần tử trên đường chéo phụ là phần tử nhỏ nhất trên cùng cột đi qua phần tử đó.
2. Một xâu (string) w được gọi là *đối xứng* (palindrome) nếu $w = w^R$ hay đọc xuôi ngược đọc ngược đều như nhau (w^R là xâu đảo ngược của w). Ví dụ các xâu *omo*, *mannam*, ... đều là *đối xứng*. Viết đặc tả thể hiện các xâu *đối xứng*.
3. Đa thức cấp n được viết dưới dạng Toán học là :

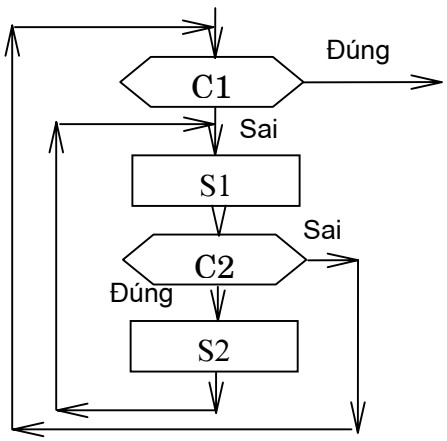
$$P_n(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_nx^n$$

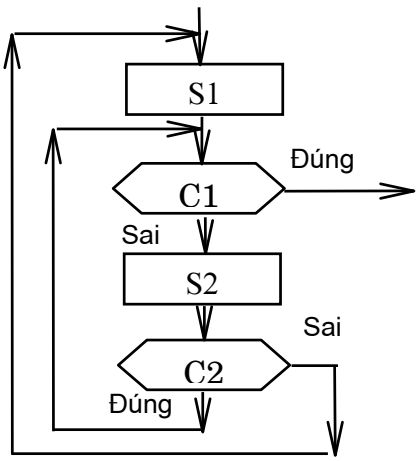
Viết đặc tả thể hiện phép cộng, so sánh hai đa thức $P_n(x)$ và $Q_m(x)$, nhân đa thức với một hằng số $a \times P_n(x)$ và nhân hai đa thức $P_n(x) \times Q_m(x)$.

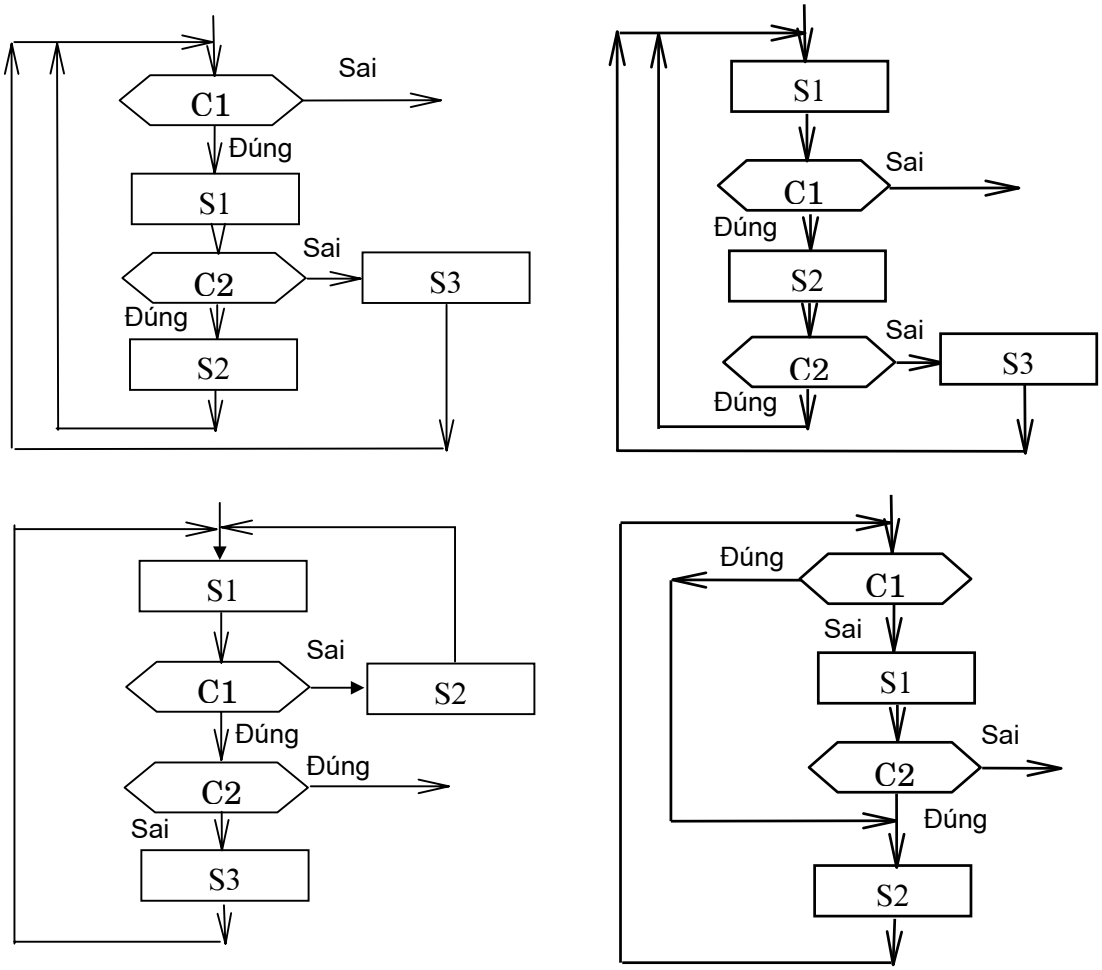
4. Các phân số (hay số hữu tỷ) được biểu diễn bởi danh sách (n, d), với n là tử số và d là mẫu số, là những số nguyên ($d \neq 0$). Viết đặc tả xây dựng các hàm xử lý phân số: rút gọn, trừ, chia và so sánh hai phân số, cộng, nhân hai phân số và chuyển đổi phân số thành số thực.

II. Lập trình cấu trúc (Structured programming)

1. Viết lệnh bằng giả ngữ (phỏng Pascal), chỉ sử dụng tối đa ba cấu trúc tuần tự, điều kiện if và lặp (while-repeat), theo các sơ đồ khối dưới đây :







2. Viết lệnh bằng giả ngữ (phỏng Pascal), chỉ sử dụng tối đa ba cấu trúc tuần tự, điều kiện if và lặp (while□repeat), theo sơ đồ khối dưới đây :

III. Thử nghiệm chương trình (Testing)

Giả sử các chương trình đã cho ở phần trên đây là các đơn thể gọi đến các đơn thể con S1, S2 và S3). Trình bày một phương pháp để thử nghiệm đơn thể gọi.