

## Chương 5

# Bộ nhớ đệm

Một hệ thống bộ nhớ bao gồm một tập hợp các bộ xử lý được kết nối với bộ nhớ bằng một giao diện truy cập nào đó mà chúng tôi gắn nhãn là memInt.



Trong phần này, chúng tôi chỉ định những gì bộ nhớ phải làm, sau đó chúng tôi chỉ định cách triển khai cụ thể của bộ nhớ bằng cách sử dụng bộ đệm. Chúng ta bắt đầu bằng cách chỉ định giao diện bộ nhớ, giao diện này chung cho cả hai thông số kỹ thuật.

### 5.1 Giao diện bộ nhớ

Giao diện không đồng bộ được mô tả trong Chương 3 sử dụng giao thức bắt tay. Việc nhận một giá trị dữ liệu phải được xác nhận trước khi có thể gửi giá trị dữ liệu tiếp theo. Trong giao diện bộ nhớ, chúng tôi trừu tượng hóa loại chi tiết này và thể hiện cả việc gửi và nhận giá trị dữ liệu dưới dạng một bước duy nhất. Chúng tôi gọi đó là bước Gửi nếu bộ xử lý đang gửi giá trị tới bộ nhớ; đó là bước Trả lời nếu bộ nhớ đang gửi đến bộ xử lý. Các bộ xử lý không gửi giá trị cho nhau và bộ nhớ chỉ gửi đến một bộ xử lý tại một thời điểm.

Chúng ta biểu thị trạng thái của giao diện bộ nhớ bằng giá trị của biến memInt. Bước Gửi thay đổi memInt theo một cách nào đó, nhưng chúng tôi không muốn chỉ định chính xác cách thực hiện. Cách để một cái gì đó không được chỉ định trong đặc tả là biến nó thành một tham số. Ví dụ: trong FIFO giới hạn của Phần 4.4, chúng tôi đã không xác định kích thước của bộ đệm bằng cách đặt nó làm tham số N. Thứ Tư

do đó muốn khai báo một tham số Gửi sao cho Send(p, d) mô tả cách memInt được thay đổi bởi một bước đại diện cho bộ xử lý p gửi giá trị dữ liệu d đến bộ nhớ. Tuy nhiên, TLA+ chỉ cung cấp các tham số không đổi và biến, không cung cấp các tham số hành động. Vì vậy, chúng ta khai báo Send là một toán tử không đổi và viết Send(p, d, memInt, memInt) thay vì Send(p, d).

Trong TLA+, chúng tôi khai báo Gửi là một toán tử hằng có bốn đối số bằng cách viết

$$\text{gửi liên tục}(p, d, \text{miOld}, \text{miNew})$$

Điều này có nghĩa là Send(p, d, miOld, miNew) là một biểu thức cho mọi biểu thức p, d, miOld và miNew, nhưng nó không nói gì về giá trị của biểu thức đó là gì. Chúng ta muốn nó là một giá trị Boolean đúng nếu một bước trong đó memInt bằng miOld ở trạng thái đầu tiên và miNew ở trạng thái thứ hai biểu thị việc gửi bởi p giá trị d tới bộ nhớ. Chúng ta có thể khẳng định rằng giá trị đó là a Boolean theo giả định

giả sử  $p, d, \text{miOld}, \text{miNew}$  :  
$$\text{Gửi}(p, d, \text{miOld}, \text{miNew}) \text{ boolean}$$

Điều này khẳng định rằng công thức

$$\text{Gửi}(p, d, \text{miOld}, \text{miNew}) \text{ boolean}$$

đúng với mọi giá trị của p, d, miOld và miNew. Ký hiệu boolean tích hợp biểu thị tập hợp {true, false}, có các phần tử là hai giá trị Boolean true và false.

Tuyên bố giả định này khẳng định một cách chính thức rằng giá trị của

$$\text{Gửi}(p, d, \text{miOld}, \text{miNew})$$

là một Boolean. Nhưng cách duy nhất để khẳng định một cách chính thức ý nghĩa của giá trị đó là nói giá trị thực sự của nó bằng gì—nghĩa là xác định Gửi thay vì biến nó thành một tham số. Chúng tôi không muốn làm điều đó, vì vậy chúng tôi chỉ nêu một cách không chính thức ý nghĩa của giá trị. Tuyên bố này là một phần của mô tả nội tại tại không chính thức về mối quan hệ giữa sự trừu tượng toán học của chúng ta và hệ thống bộ nhớ vật lý.

Để người đọc hiểu được đặc tả, chúng ta phải mô tả một cách không chính thức ý nghĩa của Gửi. Khi đó, câu lệnh giả định khẳng định rằng Send(p, d, miOld, miNew) là một Boolean sẽ không cần thiết để giải thích. Nhưng dù sao thì cũng nên đưa nó vào.

1 Ngay cả khi TLA+ cho phép chúng tôi khai báo một tham số hành động, chúng tôi cũng không có cách nào để chỉ định rằng hành động Gửi(p, d) chỉ ràng buộc memInt chứ không ràng buộc các biến khác.  
2 Chúng tôi hy vọng Send(p, d, miOld, miNew) chỉ có ý nghĩa này khi p là bộ xử lý và giá trị da mà p được phép gửi, nhưng chúng tôi đơn giản hóa đặc tả một chút bằng cách yêu cầu nó phải là Boolean cho tất cả các giá trị của p và d.

Đặc tả sử dụng giao diện bộ nhớ có thể sử dụng toán tử Gửi và Trả lời để chỉ định biến memInt thay đổi như thế nào. Thông số kỹ thuật cũng phải mô tả giá trị ban đầu của memInt. Do đó, chúng tôi khai báo một tham số không đổi InitMemInt là tập hợp các giá trị ban đầu có thể có của memInt.

Chúng tôi cũng giới thiệu ba tham số không đổi cần thiết để mô tả giao diện:

**Proc** Tập hợp các mã định danh bộ xử lý. (Chúng tôi thường rút ngắn mã định danh bộ xử lý thành bộ xử lý khi đề cập đến một phần tử của Proc.)

**Adr** Tập hợp các địa chỉ bộ nhớ.

**Val** Tập hợp các giá trị bộ nhớ có thể được gán cho một địa chỉ.

Cuối cùng, chúng tôi xác định các giá trị mà bộ xử lý và bộ nhớ gửi cho nhau qua giao diện. Bộ xử lý gửi yêu cầu tới bộ nhớ. Chúng tôi thể hiện một yêu cầu dưới dạng một bản ghi có trường op chỉ định loại yêu cầu và các trường bổ sung chỉ định các đối số của yêu cầu đó. Bộ nhớ đơn giản của chúng tôi chỉ cho phép đọc và ghi các yêu cầu. Yêu cầu đọc có trường op "Rd" và trường adr chỉ định địa chỉ cần đọc. Do đó, tập hợp tất cả các yêu cầu đọc là tập hợp

$[op : \{ "Rd" \}, adr : Adr]$

của tất cả các bản ghi có trường op bằng "Rd" (là một phần tử của tập hợp  $\{ "Rd" \}$  có phần tử duy nhất là chuỗi "Rd") và trường adr của nó là một phần tử của Adr. Yêu cầu ghi phải chỉ rõ địa chỉ cần ghi và giá trị cần ghi. Nó được biểu thị bằng một bản ghi có trường op bằng "Wr" và với các trường adr và val chỉ định địa chỉ và giá trị. Chúng ta định nghĩa MReq, tập hợp tất cả các yêu cầu, bằng sự kết hợp của hai tập hợp này. (Các thao tác thiết lập, bao gồm cả phép hợp, được mô tả trong Phần 1.2 trên trang 11.)

Bộ nhớ đáp ứng yêu cầu đọc với giá trị bộ nhớ mà nó đọc được. Chúng tôi cũng sẽ yêu cầu nó phản hồi yêu cầu ghi và có vẻ tốt nếu để phản hồi khác với phản hồi cho bất kỳ yêu cầu đọc nào. Do đó, chúng tôi yêu cầu bộ nhớ phản hồi yêu cầu ghi bằng cách trả về giá trị NoVal khác với bất kỳ giá trị bộ nhớ nào. Chúng ta có thể khai báo NoVal là một tham số không đổi và thêm NoVal /Val giả định. (Ký hiệu / được nhập vào ascii dưới dạng \notin .) Nhưng tốt nhất, khi có thể, nên tránh đưa ra các tham số.

Thay vào đó, chúng tôi định nghĩa NoVal bằng

$NoVal = \text{chọn } v : v / Val$

Biểu thức chọn  $x : F$  bằng một giá trị được chọn tùy ý  $x$  thỏa mãn công thức  $F$ . (Nếu không có  $x$  như vậy tồn tại, biểu thức có một giá trị hoàn toàn tùy ý.) Câu lệnh này định nghĩa NoVal là một giá trị nào đó không phải là một phần tử của





Toán học thông thường không có ký hiệu thuận tiện để viết một biểu thức có giá trị là một hàm số. TLA+ định nghĩa  $[x \ S \ e]$  là hàm  $f$  với miền  $S$  sao cho  $f[x] = e$  với mọi  $x \in S$ .<sup>3</sup> Ví dụ,

$$\text{thành công} = [n \in \text{Nat} \mid n + 1]$$

định nghĩa  $\text{succ}$  là hàm kế tiếp trên các số tự nhiên—hàm có miền  $\text{Nat}$  sao cho  $\text{succ}[n] = n + 1$  với mọi  $n \in \text{Nat}$ .

Bản ghi là một hàm có miền xác định là một tập hữu hạn các chuỗi. Ví dụ: một bản ghi có các trường  $\text{val}$ ,  $\text{ack}$  và  $\text{rdy}$  là một hàm có miền là tập hợp  $\{\text{"val"}, \text{"ack"}, \text{"rdy"}\}$  bao gồm ba chuỗi  $\text{"val"}, \text{"ack"}$  và  $\text{"rdy"}$ .

Biểu thức  $r.\text{ack}$ , trường  $\text{ack}$  của bản ghi  $r$  là viết tắt của  $r[\text{"ack"}]$ .

Ký lục

$$[\text{val} \ 42, \text{ack} \ 1, \text{rdy} \ 0]$$

có thể được viết

$$[i \in \{\text{"val"}, \text{"ack"}, \text{"rdy"}\} \mid \text{nếu } i = \text{"val"} \text{ thì } 42 \text{ else nếu } i = \text{"ack"} \text{ thì } 1 \text{ else } 0]$$

Cấu trúc ngoại trừ cho các bản ghi, được giải thích trong Phần 3.2, là trường hợp đặc biệt của cấu trúc ngoại trừ chung cho các hàm, trong đó  $!c$  là tên viết tắt của  $!\text{"c"}$ . giống như  $f$  ngoại trừ biểu thức  $[f \text{ ngoại trừ } ![c] = e]$  là hàm  $\hat{f}$  dành cho mọi hàm  $f$ , trừ với  $\hat{f}[c] = e$ . Hàm này cũng có thể được viết

$$[x \in \text{miền } f \mid \text{nếu } x = c \text{ thì } e \text{ khác } f[x]]$$

giả sử rằng ký hiệu  $x$  không xuất hiện trong bất kỳ biểu thức  $f$ ,  $c$  và  $e$  nào. Ví dụ:  $[\text{succ ngoại trừ } ![42] = 86]$  là hàm  $g$  giống như  $\text{succ}$  ngoại trừ  $g[42]$  bằng 86 thay vì 43.

Giống như trong cấu trúc ngoại trừ cho các bản ghi, biểu thức  $e$  trong

$$[f \text{ ngoại trừ } ![c] = e]$$

có thể chứa ký hiệu  $@$ , trong đó nó có nghĩa là  $f[c]$ . Ví dụ,

$$[\text{thành công ngoại trừ } ![42] = 2] @ = [\text{thành công ngoại trừ } ![42] = 2 \mid \text{thành công}[42]]$$

Nói chung,

$$[f \text{ ngoại trừ } ![c1] = e1, \dots, ![cn] = en]$$

<sup>3</sup> trong  $[x \ S \ e]$  chỉ là một phần của cú pháp; TLA+ sử dụng ký hiệu cụ thể đó để giúp bạn nhớ ý nghĩa của cấu trúc. Các nhà khoa học máy tính viết  $\lambda x : S$  để biểu thị một cái gì đó tương tự như  $[x \ S \ e]$ , ngoại trừ việc biểu thức  $\lambda$  của chúng không hoàn toàn giống với các hàm toán học thông thường được sử dụng trong TLA+.

là hàm  $f$  giống  $f$  ngoại trừ  $f[c_i] = e_i$  với mỗi  $i$ . Chính xác hơn, biểu thức này bằng

$$[ \dots [ [f \text{ ngoại trừ } ![c_1] = e_1] \text{ ngoại trừ } ![c_2] = e_2] \dots \text{ ngoại trừ } ![c_n] = e_n ]$$

Các hàm  $n$  ứng với mảng của ngôn ngữ lập trình. Miền của hàm  $n$  ứng với tập chỉ mục của một mảng. Hàm  $[f \text{ ngoại trừ } ![c] = e]$   $n$  ứng với mảng thu được từ  $f$  bằng cách gán  $e$  cho  $f[c]$ . Một hàm có phạm vi là một tập hợp các hàm  $n$  ứng với một mảng các mảng. TLA+ định nghĩa  $[f \text{ ngoại trừ } ![c][d] = e]$  là hàm  $n$  ứng với mảng thu được bằng cách gán  $e$  cho  $f[c][d]$ . Nó có thể được viết là

$$[f \text{ ngoại trừ } ![c] = [@ \text{ ngoại trừ } ![d] = e]]$$

Việc khái quát hóa thành  $[f \text{ ngoại trừ } ![c_1] \dots [c_n] = e]$  với mọi  $n$  đều hiển nhiên. Vì bản ghi là một hàm nên ký hiệu này cũng có thể được sử dụng cho các bản ghi. TLA+ duy trì thống nhất ký hiệu rằng  $\sigma.c$  là tên viết tắt của  $\sigma["c"]$ . Ví dụ, điều này ngụ ý

$$[f \text{ ngoại trừ } ![c].d = e] = [f \text{ ngoại trừ } ![c][\"d\"] = e]$$


$$= [f \text{ ngoại trừ } ![c] = [@ \text{ ngoại trừ } !.d = e]]$$

Định nghĩa TLA+ coi bản ghi là hàm giúp có thể thao tác chúng theo những cách không có đối tác trong ngôn ngữ lập trình. Ví dụ: chúng ta có thể định nghĩa một toán tử  $R$  sao cho  $R(r, s)$  là bản ghi thu được từ  $r$  bằng cách thay thế giá trị của mỗi trư ở  $c$  cũng là trư ở  $c$  của bản ghi  $s$  bằng  $sc$  nếu  $c$  là trư ở  $c$  của  $s$  thì  $R(r, s).c$   $r$  nếu không thì  $R(r, s).c = r.c$ . Định nghĩa  $R, = sc$ ; Nói cách khác, với mọi trư ở  $c$  của nghĩa là

$$R(r, s) = [c \text{ miền } r \text{ if } c \text{ miền } s \text{ then } s[c] \text{ else } r[c]]$$

Cho đến nay, chúng ta chỉ thấy các hàm của một đối số duy nhất,  $n$  tự về mặt toán học của mảng một chiều trong các ngôn ngữ lập trình.

Các nhà toán học cũng sử dụng các hàm có nhiều đối số,  $n$  tự như mảng nhiều chiều. Trong TLA+, giống như trong toán học thông thường, một hàm có nhiều đối số là một hàm có miền là một tập hợp các bộ dữ liệu. Ví dụ:  $f[5, 3, 1]$  là viết tắt của  $f[5, 3, 1]$ , giá trị của hàm  $f$  áp dụng cho bộ ba 5, 3, 1.

Cấu trúc hàm của TLA+ có phần mở rộng cho hàm nhiều đối số. Ví dụ:  $[g \text{ ngoại trừ } ![a, b] = e]$  là hàm  $g$  giống với  $g$  ngoại trừ  $g[a, b]$  bằng  $e$ . Cách diễn đạt

(5.1)  $[n \text{ Nat}, r \text{ Thực } n \text{ r}]$

bằng hàm  $f$  sao cho  $f[n, r]$  bằng  $n \text{ r}$  Cũng như, với mọi  $n \text{ Nat}$  và  $r \text{ Thực}$ .  
 $i \text{ S} : j \text{ S} : P$  có thể viết là  $i, j \text{ S} : P$ , ta có thể viết hàm  $[i \text{ S}, j \text{ S} : e]$  là  $[i, j \text{ S} : e]$ .

Phần 16.1.7 trên trang 301 mô tả các phiên bản chung của cấu trúc hàm TLA+ cho các hàm có số lượng đối số bất kỳ. Tuy nhiên, chức năng của một đối số duy nhất là tất cả những gì bạn có thể cần. Hầu như bạn luôn có thể thay thế một hàm có nhiều đối số bằng một hàm có giá trị hàm—ví dụ: viết  $f[a][b]$  thay vì  $f[a, b]$ .

### 5.3 Bộ nhớ tuyến tính hóa

Bây giờ chúng ta chỉ định một hệ thống bộ nhớ rất đơn giản trong đó bộ xử lý p đưa ra yêu cầu bộ nhớ và sau đó chờ phản hồi trước khi đưa ra yêu cầu tiếp theo. Trong đặc tả của chúng tôi, yêu cầu được thực thi bằng cách truy cập (đọc hoặc sửa đổi) một biến mem, đại diện cho trạng thái hiện tại của bộ nhớ. Bởi vì bộ nhớ có thể nhận yêu cầu từ các bộ xử lý khác trước khi phản hồi bộ xử lý p, nên điều quan trọng là thời điểm truy cập mem. Chúng tôi cho phép quyền truy cập của mem xảy ra bất cứ lúc nào giữa yêu cầu và phản hồi. Điều này chỉ định cái được gọi là bộ nhớ tuyến tính hóa. Thông số kỹ thuật bộ nhớ ít hạn chế hơn, thiết thực hơn được mô tả trong Phần 11.2.

Ngoài mem, đặc tả còn có các biến nội bộ ctl và buf, trong đó  $ctl[p]$  mô tả trạng thái yêu cầu của bộ xử lý p và buf[p] chứa yêu cầu hoặc phản hồi. Hãy xem xét yêu cầu bằng

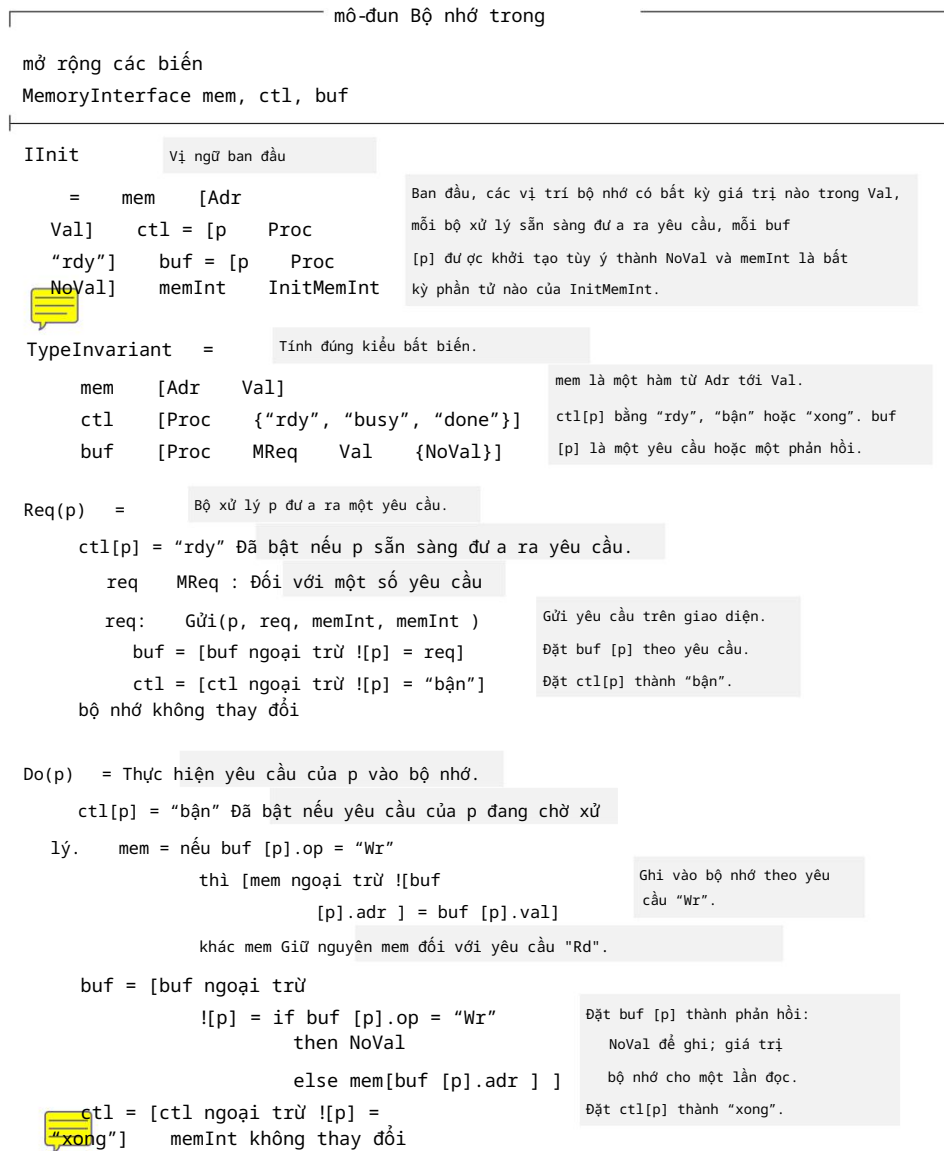
```
[op    "Wr", adr    a, val    v]
```

Đó là một yêu cầu ghi v vào địa chỉ bộ nhớ a và tạo ra phản hồi NoVal. Việc xử lý yêu cầu này được thể hiện bằng ba bước sau:

|                      |        |                  |
|----------------------|--------|------------------|
| ctl[p] = "rdy"       | yêu    | ctl[p] = "bận"   |
| buf[p] = . . .       | cầu(p) | buf[p] = yêu cầu |
| mem[a] = . . .       |        | mem[a] = . . .   |
|                      |        |                  |
| ctl[p] = "xong"      |        | ctl[p] = "rdy"   |
| Do(p) buf[p] = NoVal | Rsp(p) | buf[p] = NoVal   |
| mem[a] = v           |        | mem[a] = v       |

Bước Req(p) thể hiện việc đưa ra yêu cầu bởi bộ xử lý p. Nó được kích hoạt khi  $ctl[p] = \text{"rdy"}$ ; nó đặt  $ctl[p]$  thành "bận" và đặt buf[p] theo yêu cầu. Bước Do(p) thể hiện quyền truy cập bộ nhớ; nó được kích hoạt khi  $ctl[p] = \text{"busy"}$  và nó đặt  $ctl[p]$  thành "done" và buf[p] thành phản hồi. Bước Rsp(p) thể hiện phản hồi của bộ nhớ đối với p; nó được kích hoạt khi  $ctl[p] = \text{"done"}$  và nó đặt  $ctl[p]$  thành "rdy".

Viết đặc tả là một bài tập đơn giản trong việc thể hiện những thay đổi này đối với các biến trong ký hiệu TLA+. Thông số kỹ thuật bên trong, với mem, ctl và buf hiển thị (các biến miễn phí), xuất hiện trong mô-đun InternalMemory trên hai trang sau. Đặc tả bộ nhớ, ẩn ba biến bên trong, là mô-đun Bộ nhớ trong Hình 5.3 trên trang 53.



Hình 5.2a: Đặc tả bộ nhớ trong (bắt đầu).



Rsp(p) = Trả về phản hồi cho yêu cầu của p.

ctl[p] = "xong"

Trả lời(p, buf [p], memInt, memInt )

ctl = [ctl ngoại trừ ![p] = "rdy"]

không thay đổi mem, buf

Đã bật nếu có yêu cầu. đã xong như ng resp. chưa đ ược gửi.

Gửi phản hồi trên giao diện.

Đặt ctl[p] thành "rdy".

INext = p Proc : Req(p) Do(p) Rsp(p) Hành động ở trạng thái tiếp theo.

ISpec = IInit [INext]memInt, mem, ctl, buf Thông số kỹ thuật.

định lý ISpec TypeInvariant

Hình 5.2b: Thông số bộ nhớ trong (cuối).

## 5.4 Bộ dữ liệu là hàm

Trước khi viết đặc tả bộ nhớ đệm, chúng ta hãy xem xét kỹ hơn các bộ dữ liệu. Hãy nhớ rằng a, b, c là bộ 3 có các thành phần a, b và c. Trong TLA+, bộ 3 này thực sự là hàm có miền {1, 2, 3} ánh xạ 1 đến a, 2 đến b và 3 đến c. Do đó, a, b, c [2] bằng b.

TLA+ cung cấp toán tử tích Descartes  $\times$  của toán học thông thường, trong đó  $A \times B \times C$  là tập hợp tất cả 3 bộ a, b, c sao cho a ∈ A, b ∈ B và c ∈ C. Lưu ý rằng  $A \times B \times C$  khác với  $A \times (B \times C)$ , là tập hợp các cặp a, p với a thuộc A và p thuộc tập hợp các cặp B  $\times$  C.

Mô-đun Sequences xác định các chuỗi hữu hạn là các bộ dữ liệu. Do đó, một dãy có độ dài n là một hàm có miền xác định 1..N. Trong thực tế, s là một dãy nếu nó bằng [i 1..Len(s) s[i]]. Dưới đây là một số định nghĩa toán tử từ mô-đun Trình tự. (Ý nghĩa của các toán tử đ ược mô tả trong Phần 4.1.)

(Các) đầu = s[1]

(Các) đuôi = [i 1..(Len(s) - 1) s[i +

s t 1]] = [i 1..(Len(s) +

Len(t)) if i ≤ Len(s) then s[i] else t[i - Len(s)] ]

bộ nhớ mô-đun

mở rộng MemoryInterface Inner

(mem, ctl, buf ) = instance InternalMemory Spec =

mem, ctl, buf : Inner (mem, ctl, buf )!ISpec



Hình 5.3: Thông số bộ nhớ.

## 5.5 Định nghĩa hàm đệ quy

Chúng ta cần thêm một công cụ để viết đặc tả bộ nhớ đệm: định nghĩa hàm đệ quy. Các hàm được định nghĩa đệ quy rất quen thuộc với các lập trình viên.

Ví dụ kinh điển là hàm giai thừa, mà tôi sẽ gọi là hàm số thực. Nó thường được xác định bằng cách viết

$$\text{Fact}[n] = \text{nếu } n = 0 \text{ thì } 1 \text{ khác } n \quad \text{Fact}[n - 1]$$

với mọi  $n \in \text{Nat}$ . Ký hiệu TLA+ cho hàm viết gợi ý việc cố gắng xác định sự kiện bằng cách

$$\text{thực tế} = [n \in \text{Nat} \mid \text{nếu } n = 0 \text{ thì } 1 \text{ khác } n \mid \text{thực tế}[n - 1]]$$

Định nghĩa này là bất hợp pháp vì sự xuất hiện của sự kiện ở bên phải  $\text{thực tế}[n - 1]$  là không xác định—sự kiện chỉ được xác định sau định nghĩa của nó.

TLA+ cho phép tính tuần hoàn rõ ràng của các định nghĩa hàm đệ quy.

Chúng ta có thể định nghĩa hàm giai thừa thực tế bằng cách

$$\text{thực tế}[n \in \text{Nat}] = \text{nếu } n = 0 \text{ thì } 1 \text{ khác } n \mid \text{Fact}[n - 1]$$

Nói chung, định nghĩa có dạng  $f[x \in S] = e$  có thể được sử dụng để định nghĩa đệ quy một hàm  $f$  với miền  $S$ .

Ký hiệu định nghĩa hàm có sự khái quát hóa đơn giản để định nghĩa sự khởi đầu của hàm của nhiều đối số. Ví dụ,

$$\begin{aligned} \text{Acker}[m, n \in \text{Nat}] = & \\ & \text{nếu } m = 0 \text{ thì } n + 1 \\ & \text{ngược lại nếu } n = 0 \text{ thì } \text{Acker}[m - 1, 0] \\ & \text{khác } \text{Acker}[m - 1, \text{Acker}[m, n - 1]] \end{aligned}$$

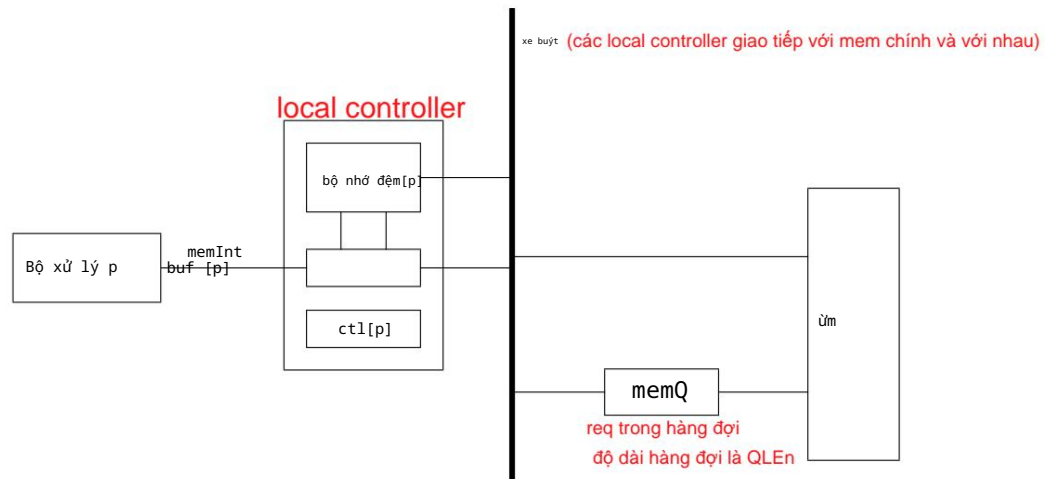
định nghĩa Acker  $[m, n]$  cho mọi số tự nhiên  $m$  và  $n$ .

Phần 6.3 giải thích chính xác ý nghĩa của các định nghĩa đệ quy. Hiện tại, chúng tôi sẽ chỉ viết các định nghĩa đệ quy mà không cần lo lắng về ý nghĩa của chúng.

## 5.6 Bộ nhớ đệm ghi qua

Bây giờ chúng tôi chỉ định một bộ nhớ đệm ghi qua đơn giản để thực hiện đặc tả bộ nhớ. Hệ thống được mô tả bằng hình ảnh Hình 5.4 ở trang tiếp theo.

Mỗi bộ xử lý p giao tiếp với bộ điều khiển cục bộ, bộ điều khiển này duy trì ba thành phần trạng thái:  $\text{buf}[p]$ ,  $\text{ctl}[p]$  và  $\text{cache}[p]$ . Giá trị của  $\text{cache}[p]$  đại diện cho bộ đệm của bộ xử lý;  $\text{buf}[p]$  và  $\text{ctl}[p]$  đóng vai trò tương tự như trong đặc tả bộ nhớ trong (mô-đun `InternalMemory`). (Tuy nhiên, như chúng ta sẽ thấy bên dưới,  $\text{ctl}[p]$  có thể giả định một giá trị bổ sung là “đang chờ”.) Các bộ điều khiển cục bộ này



Hình 5.4: Bộ đệm ghi qua.

giao tiếp với bộ nhớ chính wmem và với nhau qua xe buýt.

Các yêu cầu từ bộ xử lý tới bộ nhớ chính nằm trong hàng đợi memQ có độ dài tối đa Qlen.

Yêu cầu ghi của bộ xử lý p được thực hiện bởi hành động DoWr (p).

Đây là bộ đệm ghi qua, nghĩa là mọi yêu cầu ghi sẽ cập nhật bộ nhớ chính.


Vì vậy, hành động DoWr (p) ghi giá trị vào cache[p] và thêm yêu cầu ghi vào đuôi memQ. Khi yêu cầu đến phần đầu của memQ, hành động MemQWr sẽ lưu giá trị trong wmem. Hành động DoWr (p) cũng cập nhật cache[q] cho mọi bộ xử lý q khác có bản sao địa chỉ trong bộ đệm của nó.

Yêu cầu đọc của bộ xử lý p được thực hiện bởi hành động DoRd(p), lấy giá trị từ bộ đệm. Nếu giá trị không có trong bộ đệm, hành động RdMiss(p) sẽ thêm yêu cầu vào đuôi memQ và đặt ctl[p] thành "đang chờ".

Khi yêu cầu được xếp hàng đợi đến phần đầu của memQ, hành động MemQRd sẽ đọc giá trị và đặt nó vào cache[p], kích hoạt hành động DoRd(p).

Chúng tôi có thể mong đợi hành động MemQRd đọc giá trị từ wmem. Tuy nhiên, điều này có thể gây ra lỗi nếu có lệnh ghi vào địa chỉ đó được đưa vào hàng đợi trong memQ đằng sau yêu cầu đọc. Trong trường hợp đó, việc đọc giá trị từ bộ nhớ có thể dẫn đến việc hai bộ xử lý có các giá trị khác nhau cho địa chỉ trong bộ đệm của chúng: một bộ xử lý đưa ra yêu cầu đọc và một bộ xử lý đưa ra yêu cầu ghi theo sau việc đọc trong memQ. Vì vậy, hành động MemQRd phải đọc giá trị từ lần ghi cuối cùng vào địa chỉ đó trong memQ, nếu có lần ghi như vậy; nếu không, nó sẽ đọc giá trị từ wmem.

4Chúng tôi sử dụng tên wmem để phân biệt biến này với biến mem của mô-đun InternalMemory. Chúng ta không cần phải làm vậy, vì mem không phải là biến miễn phí (hiển thị) của đặc tả bộ nhớ thực tế trong mô-đun Bộ nhớ, nhưng nó giúp chúng ta tránh bị nhầm lẫn.

 Việc loại bỏ một địa chỉ khỏi bộ đệm của bộ xử lý p được thể hiện bằng một hành động `Evict(p)` riêng biệt. Vì tất cả các giá trị được lưu trong bộ đệm đã được ghi vào bộ nhớ nên việc trục xuất không làm gì khác ngoài việc xóa địa chỉ khỏi bộ đệm. Không có lý do gì để loại bỏ một địa chỉ cho đến khi cần dung lượng, vì vậy trong quá trình triển khai, hành động này sẽ chỉ được thực thi khi nhận được yêu cầu về một địa chỉ không được lưu trong bộ nhớ từ p và bộ đệm của p đã đầy. Nhưng đó là sự tối ưu hóa hiệu suất; nó không ảnh hưởng đến tính đúng đắn của thuật toán nên không xuất hiện trong đặc tả. Chúng tôi cho phép xóa một địa chỉ đã lưu trong bộ nhớ khỏi bộ nhớ của p bất kỳ lúc nào—trừ khi địa chỉ đó vừa được đặt ở đó bằng hành động `MemQrd` cho yêu cầu đọc mà hành động `DoRd(p)` chưa được thực hiện. Đây là trường hợp khi `ctl[p]` bằng “đang chờ” và `buf[p].adr` bằng địa chỉ được lưu trong bộ nhớ cache.

Các hành động `Req(p)` và `Rsp(p)`, đại diện cho bộ xử lý p đưa ra yêu cầu và bộ nhớ đưa ra phản hồi cho p, giống như các hành động tương ứng của đặc tả bộ nhớ, ngoại trừ việc chúng cũng để lại các biến mới cache và memQ không thay đổi và chúng để lại vmem không thay đổi thay vì mem.

Để chỉ định tất cả các hành động này, chúng ta phải quyết định cách bộ xử lý lưu vào bộ nhớ đệm và hàng đợi yêu cầu tới bộ nhớ được biểu thị bằng các biến memQ và cache. Chúng ta đặt memQ là một chuỗi các cặp có dạng p, req, trong đó req là một yêu cầu và p là bộ xử lý đã đưa ra yêu cầu đó. Đối với bất kỳ địa chỉ bộ nhớ a nào, chúng ta đặt `cache[p][a]` là giá trị trong bộ đệm của p cho địa chỉ a (“bản sao” của a trong bộ đệm của p). Nếu bộ nhớ đệm của p không có bản sao của a, chúng ta để `cache[p][a]` bằng `NoVal`.

Thông số kỹ thuật xuất hiện trong mô-đun `WriteThroughCache` trên trang 57-59. Bây giờ tôi sẽ xem xét đặc tả này, giải thích một số điểm tốt hơn và một số ký hiệu mà chúng ta chưa từng gặp trước đây.

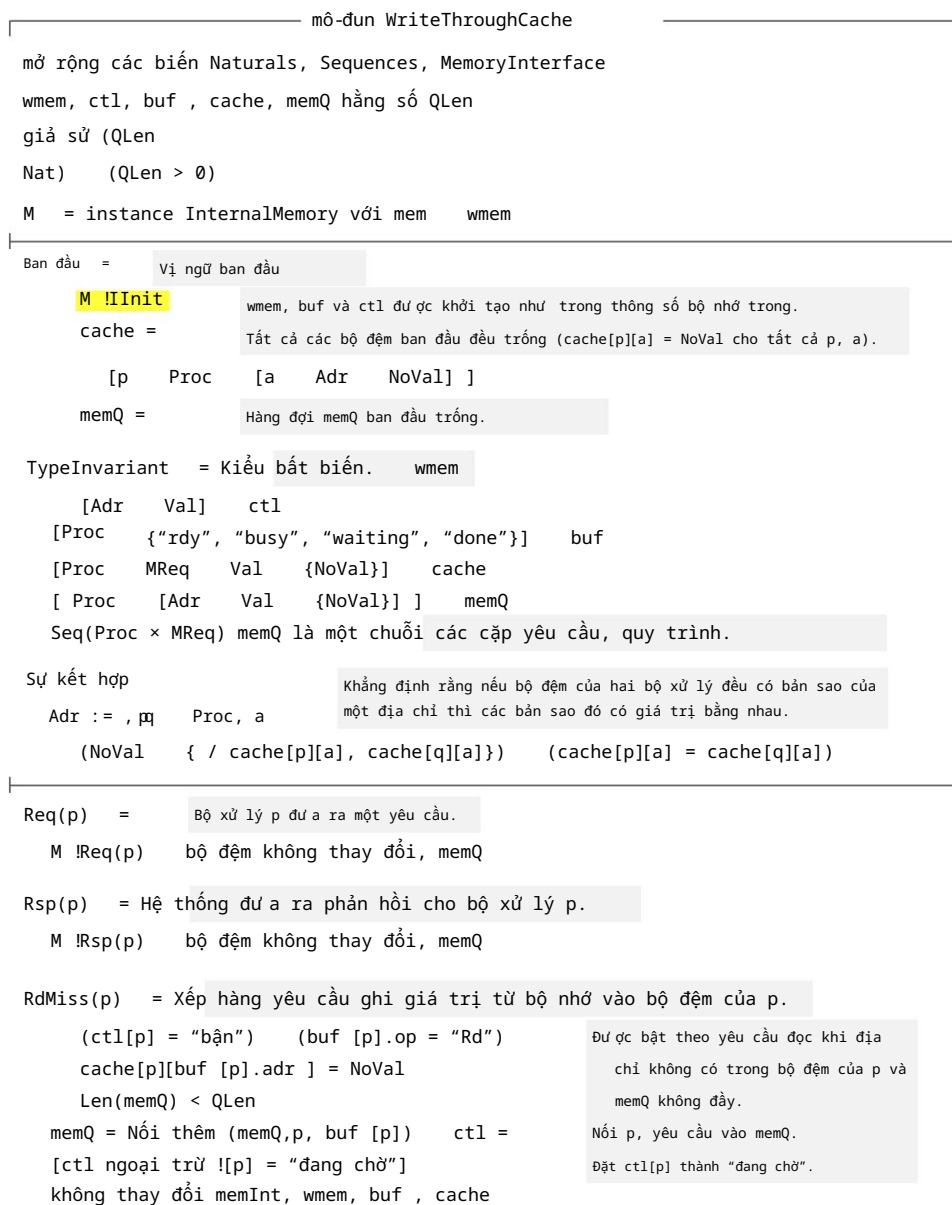
Các câu lệnh mở rộng, khai báo và giả sử đều quen thuộc. Chúng ta có thể sử dụng lại một số định nghĩa từ mô-đun `InternalMemory`, do đó, một câu lệnh **phiên bản** sẽ tạo bản sao của mô-đun đó bằng wmem được thay thế cho mem. (Các tham số khác của mô-đun `InternalMemory` được khởi tạo bằng các tham số cùng tên trong mô-đun `WriteThroughCache`.)

Vì từ ban đầu `Init` chứa liên kết `M !Init`, khẳng định rằng `ctl` và `buf` có cùng giá trị ban đầu như trong đặc tả bộ nhớ trong và wmem có cùng giá trị ban đầu như mem trong đặc tả đó.

Bộ nhớ đệm ghi cho phép `ctl[p]` có giá trị “đang chờ” mà nó không có trong đặc tả bộ nhớ trong, vì vậy chúng ta không thể sử dụng lại kiểu bất biến `M !TypeInvariant` của bộ nhớ trong. Do đó, Công thức `TypeInvariant` mô tả rõ ràng các loại wmem, `ctl` và `buf`. Loại memQ là tập hợp các chuỗi bộ xử lý, cặp yêu cầu.

Tiếp theo, mô-đun này xác định Sự kết hợp vị ngữ, xác nhận thuộc tính kết hợp bộ đệm cơ bản của bộ đệm ghi: đối với mọi bộ xử lý p và q và bất kỳ địa chỉ a nào, nếu cả p và q đều có bản sao của địa chỉ a trong bộ đệm của chúng, thì những bản sao đó đều bình đẳng. Lưu ý thủ thuật viết  $x \in \{y, z\}$  thay vì công thức tương đương như ng dài hơn  $(x = y) \vee (x = z)$ .





Hình 5.5a: Đặc tả bộ đệm ghi qua (bắt đầu).

DoRd(p) = Thực hiện đọc p một giá trị trong bộ đệm

```

của nó.   ctl[p]   {"bận",
"đang chờ"}   buf
[p].op = "Rd"   cache[p][buf
[p].adr ] = NoVal   buf = [buf ngoại trừ ![p ] =
cache[p][buf [p].adr ] ]   ctl = [ctl
ngoại trừ ![p] = "xong"]   không thay đổi memInt, wmem, cache, memQ

```

Được kích hoạt nếu đọc  
yêu cầu đang chờ xử lý và  
địa chỉ nằm trong bộ đệm.  
Nhận kết quả từ bộ đệm.  
Đặt ctl[p] thành "xong".

DoWr (p) = Ghi vào bộ nhớ đệm của p, cập nhật các bộ nhớ đệm khác và xếp hàng cập nhật bộ

```

hãy để   nhớ.   = buf [p] Yêu cầu của bộ xử lý p.

trong   (ctl[p] = "busy")   (r .op = "Wr")   Được bật nếu yêu cầu ghi đang chờ
xử lý   Len(memQ) < QLen và memQ chứa đầy.   cache = Cập nhật
bộ đệm của p và bất kỳ bộ đệm nào khác có bản sao.

```

```

[q Proc   if (p = q)   (cache[q][r .adr ] = NoVal)
thì [cache[q] ngoại trừ ![r .adr ] = r .val]
else cache[q]]

```

```

memQ = Nối(memQ, p, r )   buf =
[buf ngoại trừ ![p] = NoVal]   ctl =
[ctl ngoại trừ ![p] = "xong"]   memInt,
wmem không thay đổi

```

Enqueue viết ở đuôi memQ.  
Tạo phản hồi.  
Đặt ctl để cho biết yêu cầu đã được thực hiện.

wmem = Giá trị wmem sẽ có sau khi tất cả việc ghi vào memQ được thực hiện.

```

đặt f [i 0 . . Len(memQ)] =
thực hiện. nếu i = 0 thì

```

```

wmem else if memQ[i][2].op = "Rd"
then f [i
1] else [f [i 1] ngoại trừ ![memQ[i][2].adr ]
= memQ[i][2].val]

```

trong f [Len(memQ)]

MemQWr = Thực hiện ghi ở đầu memQ vào bộ nhớ.

```

hãy để   = Head(memQ)[2] Yêu cầu ở đầu memQ. trong   (memQ = )

```

```

(r .op = "Wr")
wmem =

```

Được bật nếu Head(memQ) ghi.  
Thực hiện ghi vào bộ nhớ.

```

[wmem ngoại trừ ![r .adr ] = r .val]

```

```

memQ = Tail(memQ)

```

Xóa ghi khỏi memQ.

```

không thay đổi memInt, buf , ctl, cache

```

Hình 5.5b: Đặc tả bộ đệm ghi qua (giữa).

MemQRd = Thực hiện đọc vào bộ nhớ theo hàng đợi.

hãy để p = Head(memQ)[1] Bộ xử lý yêu cầu. =

r Head(memQ)[2] Yêu cầu ở đầu memQ.

trong (memQ = ) (r.op = "Rd") Đư ợc bật nếu Head(memQ) là một lần  
đọc. memQ = Đuôi(memQ) Loại bỏ phần đầu của memQ.

cache = Đư a giá trị từ bộ nhớ hoặc memQ vào cache của p.

[bộ nhớ đệm ngoại trừ ![p][r.adr] = vmem[r.adr]]

không thay đổi memInt, wmem, buf, ctl

Evict(p, a) = Xóa địa chỉ a khỏi bộ đệm của

p. (ctl[p] = "đang chờ") (buf [p].adr

Không thể xóa nếu nó vừa đư ợc đọc  
vào bộ nhớ đệm từ bộ nhớ.

= a) cache = [cache ngoại trừ ![p][a] =

NoVal] không thay đổi memInt, wmem, buf, ctl, memQ

Tiếp theo = p Proc : Req(p)

Rsp(p) RdMiss(p) DoRd(p) DoWr

(p) a Adr : Evict(p, a)

MemQWr MemQRd

Thông số = Ban đầu [Tiếp theo]memInt, wmem, buf, ctl, cache, memQ

định lý Spec (Loại Bất biến Kết hợp)

LM = instance Định lý bộ nhớ Thông số bộ nhớ. với các biến nội bộ bị ẩn.

Spec LM !Spec Công thức Spec triển khai thông số bộ nhớ.

Hình 5.5c: Đặc tả bộ đệm ghi qua (cuối).

Các hành động Req(p) và Rsp(p), đại diện cho bộ xử lý gửi yêu cầu lại và nhận phản hồi, về cơ bản giống với các hành động tương ứng trong mô-đun InternalMemory. Tuy nhiên, họ cũng phải chỉ định rằng các biến cache và memQ, không có trong mô-đun InternalMemory, sẽ đư ợc giữ nguyên.

Trong định nghĩa của RdMiss, biểu thức Append(memQ, p, buf [p]) là chuỗi thu đư ợc bằng cách thêm phần tử p, buf [p] vào cuối memQ.

Hành động DoRd(p) thể hiện việc thực hiện việc đọc từ bộ đệm của p. Nếu ctl[p] = "busy", thì địa chỉ ban đầu nằm trong bộ đệm. Nếu ctl[p] = "waiting", thì địa chỉ vừa đư ợc đọc vào bộ đệm từ bộ nhớ.

Hành động DoWr (p) ghi giá trị vào bộ đệm của p và cập nhật giá trị trong bất kỳ bộ đệm nào khác có bản sao. Nó cũng xếp hàng yêu cầu ghi vào memQ.

Trong quá trình triển khai, yêu cầu đư ợc đư a lên bus, truyền nó đến các bộ nhớ đệm khác và tới hàng đợi memQ. Trong chế độ xem hệ thống cấp cao của chúng tôi, chúng tôi trình bày tất cả những điều này dư ới dạng một bư ớc duy nhất.

Định nghĩa của  $\text{DoWr}$  giới thiệu cấu trúc  $\text{TLA}^+$   $\text{let/in}$ . Mệnh đề  $\text{let}$  bao gồm một chuỗi các định nghĩa có phạm vi mở rộng cho đến hết mệnh đề  $\text{in}$ . Trong định nghĩa của  $\text{DoWr}$ , mệnh đề  $\text{let}$  định nghĩa  $r$  bằng  $\text{buf}[p]$  trong mệnh đề  $\text{in}$ . Quan sát rằng định nghĩa của  $r$  chứa tham số  $p$  của định nghĩa  $\text{DoWr}$ . Do đó, chúng tôi không thể di chuyển định nghĩa của  $r$  ra ngoài định nghĩa của  $\text{DoWr}$ .

Một định nghĩa trong  $\text{let}$  cũng giống như một định nghĩa thông thường trong một mô-đun; đặc biệt, nó có thể có các tham số. Các định nghĩa cục bộ này có thể được sử dụng để rút ngắn biểu thức bằng cách thay thế các biểu thức con thông thường bằng một toán tử. Trong định nghĩa của  $\text{DoWr}$ , tôi đã thay thế năm trừ ở trong hợp  $\text{buf}[p]$  bằng ký hiệu duy nhất  $r$ . Đây là một việc làm ngắn gọn vì nó hầu như không tạo ra sự khác biệt nào về độ dài của định nghĩa và nó đòi hỏi người đọc phải nhớ định nghĩa của ký hiệu mới  $r$ . Nhưng việc sử dụng  $\text{let}$  để loại bỏ các biểu thức con phổ biến thường có thể rút ngắn và đơn giản hóa một biểu thức rất nhiều.

$\text{let}$  cũng có thể được sử dụng để làm cho biểu thức dễ đọc hơn, ngay cả khi các toán tử mà nó định nghĩa chỉ xuất hiện một lần trong biểu thức  $\text{in}$ . Chúng ta viết một đặc tả với một chuỗi các định nghĩa, thay vì chỉ xác định một công thức nguyên khối duy nhất, bởi vì một công thức sẽ dễ hiểu hơn khi được trình bày thành các phần nhỏ hơn. Cấu trúc  $\text{let}$  cho phép quá trình chia công thức thành các phần nhỏ hơn được thực hiện theo thứ bậc.  $\text{let}$  có thể xuất hiện dưới dạng biểu thức con của biểu thức  $\text{in}$ . Lets lồng nhau phổ biến trong các thông số kỹ thuật lớn, phức tạp.

Tiếp theo là định nghĩa hàm trạng thái  $\text{vmem}$ , được sử dụng trong hành động xác định  $\text{MemQRd}$  bên dưới. Nó bằng giá trị mà bộ nhớ chính  $\text{wmem}$  sẽ có sau khi tất cả các thao tác ghi hiện tại trong  $\text{memQ}$  đã được thực hiện.

Hãy nhớ lại rằng giá trị được  $\text{MemQRd}$  đọc phải là giá trị gần đây nhất được ghi vào địa chỉ đó—một giá trị có thể vẫn còn trong  $\text{memQ}$ . Giá trị đó là giá trị trong  $\text{vmem}$ . Hàm  $\text{vmem}$  được định nghĩa theo hàm  $f$  được xác định đệ quy, trong đó  $f[i]$  là giá trị mà  $\text{wmem}$  sẽ có sau khi thao tác  $i$  đầu tiên trong  $\text{memQ}$  được thực hiện. Lưu ý rằng  $\text{memQ}[i][2]$  là thành phần thứ hai (yêu cầu) của  $\text{memQ}[i]$ ,  $i$  phần tử trong dãy  $\text{memQ}$ .

Hai hành động tiếp theo,  $\text{MemQWr}$  và  $\text{MemQRd}$ , thể hiện việc xử lý yêu cầu ở đầu hàng đợi  $\text{memQ}$ — $\text{MemQWr}$  cho yêu cầu ghi và  $\text{MemQRd}$  cho yêu cầu đọc. Những hành động này cũng sử dụng  $\text{let}$  để tạo định nghĩa cục bộ. Ở đây, định nghĩa của  $p$  và  $r$  có thể được chuyển trước định nghĩa của  $\text{MemQWr}$ . Trên thực tế, chúng ta có thể tiết kiệm không gian bằng cách thay thế hai định nghĩa cục bộ của  $r$  bằng một định nghĩa toàn cục (trong mô-đun). Tuy nhiên, việc đưa ra định nghĩa toàn cục về  $r$  theo cách này sẽ hơi gây mất tập trung, vì  $r$  chỉ được sử dụng trong các định nghĩa của  $\text{MemQWr}$  và  $\text{MemQRd}$ . Thay vào đó, có thể tốt hơn nếu kết hợp hai hành động này thành một. Việc bạn đặt định nghĩa vào  $\text{let}$  hay làm cho nó mang tính tổng quát hơn sẽ phụ thuộc vào điều gì làm cho đặc tả dễ đọc hơn.

Hành động  $\text{Evict}(p, a)$  thể hiện thao tác xóa địa chỉ  $a$  khỏi bộ đệm của bộ xử lý  $p$ . Như đã giải thích ở trên, chúng tôi cho phép xóa một địa chỉ bất cứ lúc nào—trừ khi địa chỉ đó chỉ được viết để đáp ứng yêu cầu đọc đang chờ xử lý,



đó là trạng hợp iff  $ctl[p] = \text{"waiting"}$  và  $buf[p].adr = a$ . Lưu ý việc sử dụng “chỉ số dư đôi kép” trong biểu thức ngoại trừ liên kết thứ hai của hành động.  
Liên từ này “gán NoVal cho  $cache[p][a]$ ”. Nếu địa chỉ  $a$  không có trong bộ đệm của  $p$  thì  $cache[p][a]$  đã bằng NoVal và bư ớc  $Evict(p, a)$  là một bư ớc giặt hình.

Các định nghĩa về hành động trạng thái tiếp theo Tiếp theo và về Thông số kỹ thuật hoàn chỉnh rất đơn giản. Mô-đun kết thúc với hai định lý sẽ đư ợc thảo luận tiếp theo.

## 5.7 Bất biến

Mô-đun WriteThroughCache chứa định lý

định lý Spec (Loại Bất biến Kết hợp)  
khẳng định rằng TypeInvariant Coherence là một bất biến của Spec. Vị từ trạng thái  $P \rightarrow Q$  luôn đúng nếu cả  $P$  và  $Q$  đều đúng, vì vậy  $(P \rightarrow Q)$  tương đương đư ợc với  $P \rightarrow Q$ . Điều này suy ra định lý trên tương đương với hai định lý

định lý Spec TypeĐịnh lý bất biến Spec Coherence

Định lý đầu tiên là khẳng định bất biến kiểu thông thường. Điều thứ hai khẳng định rằng Coherence là bất biến của Spec, thể hiện một thuộc tính quan trọng của thuật toán.  
Mặc dù TypeInvariant và Coherence đều là bất biến của công thức thời gian Spec, nhưng chúng khác nhau về cơ bản. Nếu  $s$  là bất kỳ trạng thái nào thỏa mãn TypeInvariant thì bất kỳ trạng thái  $t$  nào  $s \rightarrow t$  là Bư ớc tiếp theo cũng thỏa mãn TypeInvariant. Tính chất này đư ợc thể hiện bằng

định lý TypeInvariant Next TypeInvariant (Hãy  
nhớ rằng TypeInvariant là công thức thu đư ợc bằng cách mồi tất cả các biến trong công thức TypeInvariant.) Nói chung, khi  $P \rightarrow N \rightarrow P$  đúng, chúng ta nói rằng vị từ  $P$  là một bất biến của hành động  $N$ . Vị từ TypeInvariant là một bất biến Một bất biến của đặc tả Spec vì biến của Next và nó đư ợc ngụ ý bởi vị từ ban đầu  $S$  cũng là Init. một bất biến của nó là bất biến của Next và nó đư ợc ngụ ý bởi vị từ ban đầu  $S$  cũng là Init. Sự kết hợp vị từ không phải là một bất biến của hành động trạng thái tiếp theo Tiếp theo. Đối với hành động ở trạng thái tiếp theo của nó đôi khi đư ợc gọi là bất biến quy nạp của  $S$ .  
Ví dụ: giả sử  $s$  là trạng thái trong đó

- $cache[p1][a] = 1$
- $cache[q][b] = \text{NoVal}$ , với mọi  $q, b$  khác  $p1, a$
- $wmem[a] = 2$
- $memQ$  chứa phần tử đơn  $p2$ ,  $[op \text{ " Rd", } adr \text{ } a]$

cho hai bộ xử lý khác nhau p1 và p2 và một số địa chỉ a. Trạng thái s như vậy (gán giá trị cho các biến) tồn tại, giả sử rằng có ít nhất hai bộ xử lý và ít nhất một địa chỉ. Khi đó Coherence đúng ở trạng thái s. Gọi t là trạng thái thu được từ s bằng cách thực hiện bước MemQRd. Ở trạng thái t, chúng ta có  $cache[p2][a] = 2$  và  $cache[p1][a] = 1$ , do đó Coherence là sai. Do đó, tính mạch lạc không phải là một bất biến của hành động ở trạng thái tiếp theo.

Sự kết hợp là một bất biến của công thức Spec vì các trạng thái như s không thể xảy ra trong một hành vi thỏa mãn Spec. Việc chứng minh tính bất biến của nó không dễ dàng như vậy. Chúng ta phải tìm một vị từ Inv là bất biến của Next sao cho Inv ngụ ý Sự mạch lạc và được ngụ ý bởi vị từ ban đầu Init.

Các thuộc tính quan trọng của đặc tả thư ởng có thể được biểu diễn dưới dạng bất biến. Chứng minh rằng vị từ trạng thái P là bất biến của một đặc tả có nghĩa là chứng minh một công thức có dạng

$$\text{Ban đầu} \quad [\text{Tiếp theo}]v \quad P$$

Điều này được thực hiện bằng cách tìm một vị từ trạng thái thích hợp Inv và chứng minh

$$\text{Ban đầu} \quad \text{Đầu vào}, \quad \text{Inv} \quad [\text{Tiếp theo}]v \quad \text{Inv}, \quad \text{Mới} \quad P$$

Vì chủ đề của chúng ta là đặc tả chứ không phải bằng chứng nên tôi sẽ không thảo luận về cách tìm Inv.

## 5.8 Chứng minh việc thực hiện



Mô-đun WriteThroughCache kết thúc bằng định lý

$$\text{định lý Spec} \quad \text{LM !Spec}$$

trong đó LM !Spec là công thức Spec của mô-đun Bộ nhớ. Định lý này khẳng định rằng mọi hành vi đáp ứng thông số kỹ thuật Spec của bộ đệm ghi cũng đáp ứng LM !Spec, thông số kỹ thuật của bộ nhớ tuyến tính hóa. Nói cách khác, nó khẳng định rằng bộ nhớ đệm ghi thực hiện một bộ nhớ có thể tuyến tính hóa. Trong TLA, việc thực hiện là hàm ý. Một hệ thống được mô tả bằng công thức Sys thực hiện một đặc tả Spec iff Sys ngụ ý Spec—tức là, iff  $\text{Sys} \Rightarrow \text{Spec}$  là một định lý.

TLA không phân biệt giữa mô tả và thông số kỹ thuật của hệ thống; cả hai đều chỉ là công thức.

Theo định nghĩa công thức Spec của module Memory (trang 53), ta có thể phát biểu lại định lý như sau:

$$\text{định lý Spec} \quad \text{mem, ctl, buf} : \text{LM !Inner (mem, ctl, buf) !ISpec}$$

trong đó LM !Inner (mem, ctl, buf) !ISpec là công thức ISpec của mô-đun InternalMemory. Các quy tắc logic cho chúng ta biết rằng để chứng minh một định lý như vậy, chúng ta phải tìm “nhân chứng” cho các biến lưu trữ mem, ctl và buf. Những nhân chứng này là

các hàm trạng thái (các biểu thức thông thường không có số nguyên tố), mà tôi sẽ gọi là  $omem$ ,  $octl$ , và  $obuf$ , điều đó thỏa mãn

(5.2)  $Spec \quad LM \ !Inner (omem, octl, obuf) !ISpec$

Công thức  $LM \ !Inner (omem, octl, obuf) !ISpec$  là công thức  $ISpec$  với các thay thế

$mem \quad omem, \quad ctl \quad octl, \quad buf \quad obuf$

Bộ dữ liệu  $omem$ ,  $octl$ ,  $obuf$  của các hàm chứng kiến được gọi là ánh xạ sàng lọc, và chúng tôi mô tả (5.2) như là sự khẳng định rằng  $Spec$  thực hiện công thức  $ISpec$  theo ánh xạ sàng lọc này. Theo trực giác, điều này có nghĩa là  $Spec$  ngụ ý rằng giá trị của bộ dữ liệu  $memInt$ ,  $omem$ ,  $octl$ ,  $obuf$  của các hàm trạng thái thay đổi cách  $ISpec$  khẳng định rằng bộ  $memInt$ ,  $mem$ ,  $ctl$ ,  $buf$  của các biến sẽ thay đổi.

Bây giờ tôi sẽ mô tả ngắn gọn cách chúng tôi chứng minh (5.2); để biết chi tiết, xem kỹ thuật các bài viết về TLA, có sẵn trên trang Web TLA. Hãy để tôi giới thiệu đầu tiên một chút ký hiệu không phải TLA+. Đối với bất kỳ công thức  $F$  nào của mô-đun `InternalMemory`, hãy  $F$  bằng  $LM \ !Inner (omem, octl, obuf) !F$ , là công thức  $F$  với  $omem$ ,  $octl$ ,  $obuf$  thay thế cho  $mem$ ,  $ctl$  và  $buf$ . Cụ thể là  $mem$ ,  $ctl$  và  $buf$  tương ứng bằng  $omem$ ,  $octl$  và  $obuf$ .

Với ký hiệu này, chúng ta có thể viết (5.2) dư dãi dạng  $Spec \quad ISpec$ . Thay thế thông số kỹ thuật và  $ISpec$  theo định nghĩa của họ, công thức này trở thành

(5.3) Ban đầu  $[Tiếp] memInt, wmem, buf, \quad ctl, cache, memQ$   
 $IInit \quad [INext] \quad memInt, mem, ctl, buf$

Công thức (5.3) sau đó được chứng minh bằng cách tìm  $Inv$  bất biến của  $Spec$  sao cho

Ban đầu  $IInit$   
 $Inv \quad Tiếp \quad theo \quad INext$   
 $không \quad thay \quad đổi \quad memInt, mem, ctl, buf$

Liên từ thứ hai được gọi là mô phỏng bước. Nó khẳng định rằng Bước tiếp theo bắt đầu ở trạng thái thỏa mãn  $Inv$  bất biến hoặc là bước  $INext$ —một bước mà thay đổi 4 bộ  $memInt$ ,  $omem$ ,  $octl$ ,  $obuf$  theo cách bước  $INext$  thay đổi  $memInt$ ,  $mem$ ,  $ctl$ ,  $buf$ —hoặc nếu không thì giữ nguyên bộ 4 đó. Cho chúng ta thông số kỹ thuật bộ nhớ, các hàm trạng thái  $omem$ ,  $octl$  và  $obuf$  được xác định bởi

$ôi \quad = \quad vmem$   
 $bất \quad phân \quad = \quad [p \quad Proc \quad if \quad ctl[p] = "đang \quad chờ" \quad thì \quad "bạn" \quad else \quad ctl[p]]$   
 $obuf \quad = \quad buf$

Toán học của một bằng chứng triển khai rất đơn giản, do đó, bằng chứng là đơn giản - về mặt lý thuyết. Đối với các thông số kỹ thuật của hệ thống thực, những bằng chứng như vậy có thể khá khó khăn. Đi từ lý thuyết đến thực hành đòi hỏi phải biến đổi toán học

$memInt$  bằng  $memInt$ , vì  $memInt$  là một biến khác biệt với  $mem$ ,  $ctl$ , và  $buf$ .

của việc chứng minh vào một chuyên ngành kỹ thuật. Đây là một chủ đề xứng đáng được viết thành một cuốn sách và tôi sẽ không cố gắng thảo luận về nó ở đây.

Bạn có thể sẽ không bao giờ chứng minh được rằng một đặc tả này thực hiện một đặc tả khác. Tuy nhiên, bạn nên hiểu ánh xạ sàng lọc và mô phỏng ngược. Sau đó, bạn sẽ có thể sử dụng TLC để kiểm tra xem một đặc tả có triển khai một đặc tả khác hay không; Chương 14 giải thích cách thực hiện.