HỆ THỐNG KIỂU ĐỂ TÍNH BỘ NHỚ LOG CỦA CHƯƠNG TRÌNH GIAO DỊCH TỪ BIẾN DÙNG CHUNG

Nguyễn Ngọc Khải[1], *Trương Anh Hoàng*[2]

Tóm tắt

Trong nghiên cứu trước đây, chúng tôi đã xây dựng một hệ thống kiểu để tính tài nguyên tối đa cần sử dụng của một chương trình giao dịch đa luồng. Tuy nhiên, tài nguyên này được tính toán dựa trên các tham số là tài nguyên mà mỗi giao dịch cần sử dụng. Những tham số này do người lập trình phải tự tính toán thủ công dựa trên mã nguồn của chương trình. Vì vậy, kết quả đó vẫn mang tính phương pháp và bán tự động, chưa thuận tiện cho người lập trình. Trong nghiên cứu này, chúng tôi cải tiến hệ thống kiểu để tính được tài nguyên tối đa cần sử dụng của chương trình giao dịch đa luồng một cách hoàn toàn tự động. Người lập trình không cần thực hiện bước tính toán thủ công các tham số như trong nghiên cứu trước. Tài nguyên trong nghiên cứu này được cụ thể hóa là bộ nhớ log của các giao dịch. Để thực hiện được công việc này, ngôn ngữ cũng đã được cải tiến, bổ sung và chúng cũng gần với ngôn ngữ thực tế hơn.

In previous works, we built a type system for caculating the maximum resource usage of a multi-thread transaction program. However, this resource was calculated based on the parameters. These parameters were the resources that transactions need to use and calculated manually based on analysis source code of program by the programmer. Therefore, the result was still methodical and semi-automatic leading to inconvenient to be used for programmers. In this work, we have improved the type system fully automated for calculating the maximum resources usage of multi-thread transaction programs. Based on this result, the programmers do not need to calculate manually parameters like previous work. The resources here are specified into the log memory of the transaction. In order to do this, the language is also improved, complementary, and it closer to the actual language.

Từ khóa

Đa luồng, Bộ nhớ giao dịch, Hệ thống kiểu, Ngôn ngữ lập trình, Tài nguyên bộ nhớ.

1. Giới thiệu

Mục đích của nghiên cứu này là xây dựng một hệ thống kiểu để giúp người lập trình ước lượng tĩnh bộ nhớ *log* tối đa cần sử dụng của các chương trình giao dịch đa luồng sử dụng cơ chế bộ nhớ giao dịch (gọi tắt là chương trình giao dịch). Từ đó, người lập trình có thể tối ưu chương trình của mình để sử dụng bộ nhớ hiệu quả hơn, đảm bảo không bị các lỗi tràn bộ nhớ. Nghiên cứu này được phát triển từ nghiên cứu [19], trong đó hệ thống kiểu đã được cải tiến để tính được bộ nhớ *log* tối đa cần sử dụng một cách

[1]Đại học Tài nguyên và môi trường Hà Nội, [2]Đại học Công nghệ - Đại học Quốc Gia Hà Nội.

tự động của các chương trình giao dịch. Ngôn ngữ trong nghiên cứu này cũng đã được cải tiến để thực hiện được điều này và chúng gần với ngôn ngữ thực tế hơn. Để thuận tiện cho việc mô tả vấn đề đặt ra và bạn đọc dễ theo dõi, phần tiếp theo chúng tôi sẽ trình bày tóm tắt những đặc điểm chính của cơ chế bộ nhớ giao dịch và ngôn ngữ giao dịch.

Đặc điểm chính của mô hình lập trình bộ nhớ giao dịch là cho phép tạo ra các giao dịch lồng nhau, tạo ra các luồng mới ngay trong các giao dịch đang mở. Khi một giao dịch lồng trong một giao dịch khác, ta gọi giao dịch được sinh ra trước là giao dịch cha, giao dịch sinh ra sau là giao dịch con. Một giao dịch con phải kết thúc trước giao dịch cha của chúng. Khi một giao dịch được bắt đầu, một vùng bộ nhớ gọi là *log* được cấp phát để lưu trữ các biến dùng chung. Một giao dịch được bắt đầu nhưng vẫn chưa kết thúc được gọi là một giao dịch đang mở. Bên trong một giao dịch đang mở, có thể sinh ra những luồng mới. Khi này, luồng mới sẽ tạo một bản sao các *log* của luồng cha của nó. Khi luồng cha kết thúc một giao dịch, tất cả các luồng con mà được tạo ra bên trong giao dịch đó phải cùng kết thúc với giao dịch cha của chúng. Loại kết thúc này được gọi là *đồng kết thúc*, thời điểm khi các kết thúc này diễn ra gọi là *điểm đồng kết thúc*. Những đồng kết thúc là sự đồng bộ ngầm định giữa các luồng. Nếu một giao dịch không có các luồng con, việc kết thúc là một kết thúc cục bộ thông thường. Cả hai loại kết thúc này đều giải phóng vùng nhớ đã được cấp phát cho các *log*. Việc sao chép các *log* từ luồng cha khi một luồng con được sinh ra sẽ làm tăng bộ nhớ *log* lên đáng kể. Vì vậy, việc ước lượng bộ nhớ *log* cần sử dụng của một chương trình giao dịch là hết sức cần thiết đối với người lập trình để đưa ra các giải pháp tối ưu chương trình, hạn chế các lỗi về bộ nhớ. Tuy nhiên, do đặc điểm của các chương trình giao dịch, các giao dịch có thể đan xen lồng nhau, các luồng hoạt động song song nhưng không độc lập mà có những điểm đồng bộ với nhau. Việc đồng bộ giữa các luồng là ngầm định của các ngôn ngữ lập trình mà không phải do người lập trình chủ động viết lệnh. Do đó chúng ta không thể phân tích ở mức mã nguồn chương trình mà phải phân tích ở mức ngữ nghĩa của ngôn ngữ lập trình. Vì những lý do này, việc ước lượng chính xác bộ nhớ *log* tối đa cần sử dụng của các chương trình này là bài toán thực sự phức tạp. Trong nghiên cứu [11], [18] chúng tôi đã xây dựng một hệ thống kiểu có thể tính được số *log* tối đa cùng tồn tại khi thực thi chương trình. Sau đó, nghiên cứu [19] phát triển tiếp từ các nghiên cứu trên và đã xây dựng được hệ thống kiểu để tính được tài nguyên tối đa chương trình cần sử dụng một cách bán tự động. Trong nghiên cứu này, chúng tôi tiếp tục cải tiến hệ thống kiểu của nghiên cứu [19] để tính được bộ nhớ *log* tối đa cần sử dụng của một chương trình giao dịch một cách tự động. Trong nghiên cứu này, nhiều ký hiệu được sử dụng lại từ những nghiên cứu trước, nhưng chúng có thể mang những ý nghĩa hoàn toàn khác trước.

Những đóng góp chính trong nghiên cứu này bao gồm: một ngôn ngữ giao dịch được cải tiến để gần với ngôn ngữ thực tế hơn; một hệ thống kiểu phù hợp với ngôn ngữ mới và tính được bộ nhớ *log* tối đa cần sử dụng của một chương trình giao dịch một cách hoàn toàn tự động.

Các nghiên cứu liên quan: Xác định tài nguyên cần sử dụng của chương trình đã được nhiều nhà khoa học quan tâm nghiên cứu dưới nhiều góc độ khác nhau. Trong

nghiên cứu [13], các tác giả Hofmann và Jost đã phân tích việc tính biên bộ nhớ heap cho các ngôn ngữ hàm tuần tự. Sau đó, họ sử dụng một hệ thống kiểu để tính biên bộ nhớ heap như một hàm của đầu vào cho một ngôn ngữ hướng đối tượng. Trong nghiên cứu [14], các tác giả Hughes và Pareto đã giới thiệu một ngôn ngữ hàm nghiêm ngặt, tuần tự với một hệ thống kiểu mà những chương trình có kiểu hợp lệ sẽ chạy với không gian được chỉ định bởi người lập trình. Trong nghiên cứu [10], tác giả Wei-Ngan Chin và các cộng sự đã nghiên cứu bộ nhớ sử dụng của các chương trình hướng đối tượng. Trong nghiên cứu [9], các tác giả đã đưa ra phương pháp tính toán tĩnh cận trên của tổng tài nguyên của một phương thức sử dụng một hàm tuyến tính của các tham số của phương thức. Trong nghiên cứu này, các biên tìm được chưa được chính xác, và phương pháp nghiên cứu không dựa trên hệ thống kiểu. Trong nghiên cứu [4], [8], tác giả Braberman và các cộng sự đã đưa ra phương pháp tính toán xấp sỉ hình thức của biên bộ nhớ cho các chương trình java. Trong nghiên cứu [5], các tác giả đã đề xuất hệ thống kiểu cho ngôn ngữ thành phần với các thành phần song song nhưng các luồng chạy độc lập, không có sự đồng kết thúc giữa các luồng như trong ngôn ngữ của của chúng tôi. Tác giả Albert và các cộng sự đã có nhiều nghiên cứu về ước lượng tài nguyên cần sử dụng cho chương trình. Trong nghiên cứu [3], các tác giả đã tính toán tổng bộ nhớ heap của một chương trình như một hàm của kích cỡ dữ liệu của nó. Trong [1], [2], các tác giả đã nghiên cứu vấn đề trong ngữ cảnh của các chương trình phân tán và đa luồng. Tuy nhiên, trong nghiên cứu này chi phí cho chương trình được tính bằng cách cộng dồn tất cả các chi phí cho các phương thức của chương trình ở tất cả các điểm mà chương trình đã thực thi. Trong nghiên cứu [17], tác giả Pham và các cộng sự đã đề xuất một thuật toán nhanh để tìm biên trên của bộ nhớ heap cho một lớp của các chương trình JavaCard. Trong nghiên cứu [12], tác giả Jan Hoffmann và Zhong Shao cũng sử dụng hệ thống kiểu để ước lượng tài nguyên sử dụng của các chương trình song song nhưng cho một ngôn ngữ lập trình hàm. Đối với chương trình tương tranh, trong [6], [7], các tác giả đã đo lường các chi phí về bộ nhớ đệm. Tuy nhiên, các nghiên cứu này không cung cấp cơ chế hỗ trợ cho việc tìm biên bộ nhớ tĩnh.

Nghiên cứu của chúng tôi tập trung vào ngôn ngữ lập trình đa luồng sử dụng cơ chế bộ nhớ giao dịch. Trong đó, chúng tôi tập trung giải quyết những vấn đề về giao dịch lồng nhau, quá trình đồng bộ giữa các luồng, do đó biên bộ nhớ tìm được sẽ sắc hơn những phương pháp trước. Phương pháp đưa ra ở đây là phương pháp phân tích chương trình tĩnh, dựa trên hệ thống kiểu và đã được chứng minh và kiểm thử chặt chẽ.

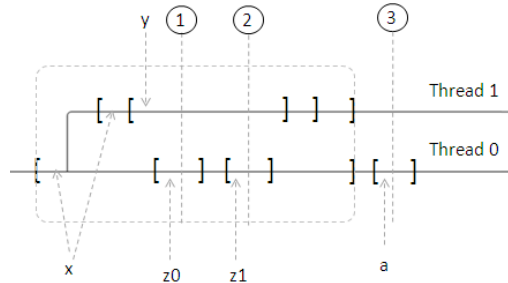
2. Ví dụ minh họa

Chúng ta xem xét một ví dụ về chương trình giao dịch được mô tả trong Hình *1.* . Ví dụ này được phát triển dựa trên ví dụ trong [16] của chúng tôi. Trong đoạn mã trong Hình *1.* .a, các lệnh `onacid` và `commit` dùng để bắt đầu và kết thúc một giao dịch. Lệnh `spawn` dùng để tạo một luồng mới. Lệnh `global int x = 0;` để khai báo và khởi tạo các biến dùng chung, đây chính là lệnh yêu cầu cấp phát bộ nhớ cho các giao dịch của chương trình. Lệnh `spawn` tạo ra một luồng mới chạy song song với luồng cha của nó. Luồng mới sẽ nhân bản các biến dùng chung của luồng cha để lưu trữ thành một bản

sao dành riêng cho chúng, vì vậy nó có thể làm việc với các biến này một cách độc lập. Hành vi của đoạn chương trình này được mô tả trong Hình *1.* .b.

```
1  onacid  //thread 0
2   global int x=0;
3   spawn{ //thread 1
4    onacid
5     //do something
6     onacid
7      global int y=0;
8      //do something
9     commit;
10   commit; commit;};
11   int i=0; //local variable
12   while(i<2){
13    onacid
14     if (i=0) then {global int z0=0;}
15      else {global int z1=0;};
16       //do something
17    commit;
18    i++;};
19 commit;
20 onacid
21  global int a=0;
22  //do something
23 commit;
```

(a)



Mô tả hành vi hoạt động của đoạn chương trình bên:
- Ký hiệu [và] mô tả việc bắt đầu và kết thúc một giao dịch,
- Các đường kẻ liền nằm ngang mô tả các luồng chạy song song,
- Các đồng kết thúc được mô tả bằng hình chữ nhật đứt nét, cạnh phải đánh dấu việc đồng bộ giữa các luồng, cạnh trái là các giao dịch được mở tương ứng.

(b)

Hình 1. Ví dụ một đoạn chương trình giao dịch

Tại dòng 1, luồng chính (luồng 0) mở một giao dịch, tiếp theo dòng 2 khai báo và khởi tạo biến số nguyên dùng chung x, dòng 3 tạo một luồng mới (luồng 1) chạy song song với luồng 0 (luồng cha của nó). Luồng 1 sẽ sao chép biến x để dùng riêng cho nó. Như vậy bộ nhớ đã bị nhân lên gấp đôi. Từ dòng 4 đến dòng 6, luồng 1 mở hai giao dịch lồng nhau. Dòng 7, khai báo và khởi tạo một biến dùng chung y mới. Từ dòng 9 đến dòng 10, có 3 lệnh `commit` trong đó 2 lệnh `commit` đầu là để đóng hai giao dịch vừa mở trước nó của chính luồng 1, lệnh `commit` thứ 3 sẽ kết hợp với lệnh `commit` tại dòng 19 của luồng 0 để cùng đóng giao dịch đã mở bằng lệnh `onacid` của luồng 0 (dòng 1). Loại kết thúc này gọi là đồng kết thúc.

Dòng 11, khởi tạo một biến i. Tuy nhiên, đây là biến cục bộ chỉ dùng trong luồng 0 để điều khiển vòng lặp `while` và lệnh `if` vì vậy chúng sẽ không bị các luồng khác sao chép, nên không làm ảnh hưởng đến việc tăng bộ nhớ do cơ chế bộ nhớ giao dịch, vì thế ta bỏ qua việc tính bộ nhớ cho các biến loại này. Giả sử mỗi biến số nguyên sẽ tiêu thụ 4 bytes bộ nhớ, bây giờ ta sẽ tính bộ nhớ tối đa được sử dụng bởi các *log* của đoạn chương trình này như sau:

- Tại thời điểm ①: Tổng bộ nhớ *log* được sử dụng cho các biến được ký hiệu là m_1, là tổng của:
  - bộ nhớ cho luồng 0: bộ nhớ cho biến x của giao dịch đầu tiên, bộ nhớ cho biến z0 của giao dịch thứ 2 (4+4=8 bytes).

- bộ nhớ cho luồng 1: bộ nhớ cho biến x mà luồng 1 đã sao chép từ luồng 0 và bộ nhớ cho biến y của giao dịch thứ 2 của luồng 1 (4+4=8 bytes).

Vì vậy m_1=8+8=16 bytes.

- Tại thời điểm ②: Bộ nhớ *log* cần sử dụng ký hiệu là m_2 bằng với bộ nhớ *log* cần sử dụng tại thời điểm ①.
- Tại thời điểm ③: Bộ nhớ *log* cần sử dụng ký hiệu m_3, là bộ nhớ cần sử dụng cho biến a của giao dịch cuối cùng của luồng 0 (4 bytes).

Như vậy, ta thấy trong trường hợp xấu nhất, bộ nhớ lớn nhất cần sử dụng cho các biến là $\max(m_1, m_2, m_3) = 16$ bytes.

3. Ngôn ngữ giao dịch

Ngôn ngữ của chúng tôi được xây dựng bắt nguồn từ ngôn ngữ trong [15], sau đó chúng tôi đã phát triển chúng trong [19]. Ngôn ngữ trong nghiên cứu này và ngôn ngữ trong [16] cùng được phát triển từ [19] với việc bổ sung thêm những câu lệnh về khai báo và khởi tạo các biến, các phép toán số học, logic, cấu trúc vòng lặp. Cùng với đó, chúng tôi đã bổ sung ngữ nghĩa của ngôn ngữ và các quy tắc kiểu để phù hợp với ngôn ngữ mới này. Việc bổ sung các lệnh khai báo, khởi tạo các biến là một nội dung quan trọng vì cùng với quy tắc kiểu sau cùng trong Định nghĩa 4 ta thấy rõ vai trò của lệnh `commit` mà các nghiên cứu trước chưa thể hiện được. So với ngôn ngữ trong [16], nghiên cứu này đã cải tiến để phân biệt biến cục bộ và biến dùng chung.

3.1. Cú pháp

Cú pháp của ngôn ngữ được thể hiện trong Hình 2. . Dòng đầu tiên mô tả một chương trình P, chúng gồm một tập các luồng. Chúng cũng có thể là trống, ký hiệu là ϕ, hoặc các luồng/tiến trình song song ký hiệu là $P \parallel P$. Ký hiệu $p(e)$ mô tả cho một luồng có định danh p, thực thi thành phần e. Dòng thứ hai mô tả các kiểu dữ liệu, chúng có thể là số nguyên, hoặc kiểu boolean. Dòng thứ ba mô tả việc khai báo và khởi tạo các biến, trong đó có hai loại biến là biến dùng chung và biến cục bộ. Lệnh khởi tạo biến dùng chung cũng là các câu lệnh yêu cầu việc cấp phát bộ nhớ trong các giao dịch. Ta gọi các biến dùng chung trong một giao dịch là một *log* và có chung một định danh *log*. Ta giả sử rằng các biến được khai báo tập trung ở đầu mỗi giao dịch, trước các lệnh `spawn`. Như vậy, khi một luồng mới được tạo ra chúng sẽ sao chép toàn bộ các biến dùng chung của giao dịch đang chứa nó của luồng cha của nó. Dòng thứ tư mô tả các phép toán, trong đó • thể hiện các phép toán số học, ■ thể hiện các phép tính so sánh, ◆ thể hiện các phép toán lôgic và, hoặc, ▲ thể hiện phép toán phủ định. Dòng tiếp theo mô tả các giá trị, chúng có thể là một số nguyên n, một giá trị lôgic $true$ hoặc $false$. Phần còn lại mô tả các biểu thức, e_i mô tả biểu thức số nguyên, e_b mô tả các biểu thức lôgic, e mô tả cho một biểu thức chung. Biểu thức e có thể là một giá trị v, phép gán một giá trị của một biểu thức vào biến x. $e; e$ ký hiệu chuỗi các lệnh tuần tự. Câu lệnh `if`$\{e_b\}$ `then` $\{e\}$ `else` $\{e\}$ là các lệnh rẽ nhánh. Lệnh `while`$(e_b)\{e\}$ thực hiện vòng lặp, trong đó biểu thức e trong thân vòng lặp không chứa các lệnh `spawn` để sinh ra luồng mới. Điều này nhằm đảm bảo cho chương trình có kiểu hợp lệ. Trong

dòng cuối cùng, lệnh `spawn`$\{e\}$ để tạo ra một luồng mới thực thi e, lệnh `onacid` và `commit` là các lệnh bắt đầu và kết thúc một giao dịch.

P	$::=$	$\phi \mid P \parallel P \mid p(e)$	luồng/tiến trình
$\mathcal{T}$	$::=$	$int \mid bool$	kiểu
D	$::=$	$global\ \mathcal{T}\ \vec{x} := v \mid \mathcal{T}\ \vec{x} := v$	khai báo và khởi tạo biến
O	$::=$	$\bullet \mid \blacksquare \mid \blacklozenge \mid \blacktriangle$	phép toán
v	$::=$	$n \mid$ **true** $\mid$ **false**	giá trị
e_i	$::=$	$e_i \bullet e_i \mid n$	biểu thức số nguyên
e_b	$::=$	$e_i \blacksquare e_i \mid e_b \blacklozenge e_b \mid \blacktriangle\ e_b \mid$ **true** $\mid$ **false**	biểu thức lôgic
e	$::=$	$v \mid x := e \mid e; e \mid$ **if**(e_b) **then** $\{e\}$ **else** $\{e\}$	biểu thức
		$\mid$ **while**$(e_b)\{e\}$	vòng lặp
		$\mid$ **spawn**$\{e\}$ $\mid$ **onacid** e $\mid$ **commit**	tạo luồng, mở/đóng giao dịch

Hình 2. Cú pháp của ngôn ngữ giao dịch

Lưu ý rằng, trong ngôn ngữ này, các lệnh `commit` trong các luồng con để đồng kết thúc với luồng cha (ví dụ như lệnh `commit` thứ 2 trong dòng 10 của đoạn mã trong ví dụ *1.*) sẽ làm cho ngôn ngữ khó dùng hơn nhưng điều này giúp người lập trình có thể chủ động kiểm soát được các điểm kết thúc giao dịch. Bởi vì giữa những lệnh `commit` có thể có các tính toán khác. Thực tế, các ngôn ngữ có thể được đơn giản bằng cách trình biên dịch sẽ tự động chèn thêm các `commit` vào cuối của mỗi luồng, nhưng như vậy người lập trình sẽ hạn chế hơn trong việc điều khiển các hành vi của các giao dịch này. Thậm chí với một trình biên dịch thông minh có thể chèn các `commit` còn thiếu cho các luồng con ngay khi các biến dùng chung không còn được sử dụng bởi các luồng đó. Trong cả hai tình huống trên, chương trình đều có thể chuyển về dạng ngôn ngữ của chúng tôi để tính toán.

3.2. Ngữ nghĩa động

Ta gọi một môi trường toàn cục của chương trình là một tập hợp các môi trường cục bộ của các luồng, mỗi môi trường cục bộ là một chuỗi các *log* và kích cỡ của chúng. Các biến dùng chung trong cùng một giao dịch ta gọi chung là một *log* và có cùng một định danh *log*. Kích cỡ của một *log* của một giao dịch là tổng kích cỡ các biến dùng chung trong giao dịch đó. Kế thừa từ nghiên cứu trước, để thuận tiện cho người đọc theo dõi, chúng tôi trình bày định nghĩa về môi trường cục bộ và môi trường toàn cục như sau:

Định nghĩa 1 (Môi trường cục bộ). *Một* môi trường cục bộ E *là một chuỗi hữu hạn của các* định danh các *log và kích cỡ của chúng:* $l_1{:}n_1; \ldots; l_k{:}n_k$. *Môi trường không có phần tử nào thì gọi là* môi trường trống*, ký hiệu là* ϵ.

Với một môi trường $E = l_1{:}n_1; \ldots; l_k{:}n_k$, bộ nhớ log cần sử dụng trong E được xác định là $[\![E]\!] = \sum_{i=1}^{k} n_i$, và $|E| = k$ là số phần tử trong E.

Định nghĩa 2 (Môi trường toàn cục). *Một* môi trường toàn cục Γ *là một tập của* định danh các luồng *và môi trường cục bộ của chúng,* $\Gamma = \{p_1{:}E_1, \ldots, p_k{:}E_k\}$.

Bộ nhớ *log* sử dụng bởi Γ, ký hiệu là $[\![\Gamma]\!]$, được xác định là: $[\![\Gamma]\!]= \sum_{i=1}^{k}[\![E_i]\!]$.

Với một môi trường toàn cục Γ và một tập các luồng P, ta gọi cặp Γ, P là một *trạng thái* của chương trình. Một trạng thái đặc biệt $error$ là trạng thái lỗi - trạng thái mà không có quy tắc giao dịch nào có thể áp dụng. Ngữ nghĩa động được định nghĩa bởi *các quy tắc giao dịch* giữa các trạng thái có dạng $\Gamma, P{\Rightarrow}\Gamma', P'$ hoặc $\Gamma, P{\Rightarrow} error$ trong Hình *3.* .

$$\frac{p'\,fresh \quad spawn(p,p',\Gamma)=\Gamma'}{\Gamma, P \parallel p(\textbf{spawn}(e_1);e_2) \Rightarrow \Gamma', P \parallel p(e_2) \parallel p'(e_1)}\ \text{S-SPAWN}$$

$$\frac{l\,fresh \quad start(l{:}n,p,\Gamma)=\Gamma'}{\Gamma, P \parallel p(\textbf{onacid}(e_1);e_2) \Rightarrow \Gamma', P \parallel p(e_2)}\ \text{S-TRANS}$$

$$\frac{intranse(\Gamma, l:n)=\mathbf{p}=\{p_1,..,p_k\} \quad commit(\mathbf{p},\Gamma)=\Gamma'}{\Gamma, P \parallel \coprod_1^k p_i(\textbf{commit};e_i) \Rightarrow \Gamma', P \parallel \coprod_1^k p_i(e_i)}\ \text{S-COMM}$$

$$\frac{x, v : \mathcal{T}}{\Gamma, P \parallel p(global\ \mathcal{T}\ x := v; e) \Rightarrow \Gamma', P \parallel p(e)}\ \text{S-INIT}$$

$$\frac{e_b \downarrow \textbf{true}}{\Gamma, P \parallel p(\textbf{if}(e_b)\textbf{then}\{e_1\}\textbf{else}\{e_2\}) \Rightarrow \Gamma, P \parallel p(e_1)}\ \text{S-COND}_1$$

$$\frac{e_b \downarrow \textbf{false}}{\Gamma, P \parallel p(\textbf{if}(e_b)\textbf{then}\{e_1\}\textbf{else}\{e_2\}) \Rightarrow \Gamma, P \parallel p(e_2)}\ \text{S-COND}_2$$

$$\frac{e_b \downarrow true \quad e \text{ không chứa } spawn}{\Gamma, P \parallel p(\textbf{while}(e_b)\{e\}) \Rightarrow \Gamma, P \parallel p(e;\textbf{while}(e_b)\{e\})}\ \text{S-WHILE}$$

$$\frac{e_b \downarrow false \quad e_1 \text{ không chứa } spawn}{\Gamma, P \parallel p(\textbf{while}(e_b)\{e_1\};e_2) \Rightarrow \Gamma, P \parallel p(e_2)}\ \text{S-NO WHILE}$$

$$\frac{x, e_1, e_2 : \mathcal{T}}{\Gamma, P \parallel p(x := e_1; e_2) \Rightarrow \Gamma, P \parallel p(e_2)}\ \text{S-ASSIGN} \qquad \frac{}{\Gamma, P \parallel p(\alpha; e) \Rightarrow \Gamma, P \parallel p(e)}\ \text{S-SKIP}$$

$$\frac{\Gamma = \Gamma' \cup \{p:E\} \quad |E|=0}{\Gamma, P \parallel p(\textbf{commit};e) \Rightarrow error}\ \text{S-ERROR-C} \qquad \frac{\Gamma = \Gamma' \cup \{p:E\} \quad |E|>0}{\Gamma, P \parallel p() \Rightarrow error}\ \text{S-ERROR-O}$$

Hình 3. *Ngữ nghĩa động của ngôn ngữ giao dịch*

Hình *3.* sử dụng các hàm phụ trợ được mô tả như dưới đây, trong đó tên các hàm được lấy từ [15] và một số quy tắc được áp dụng cho các tiến trình bao gồm: $P \parallel P' \equiv P' \parallel P$, $P \parallel (P' \parallel P'') \equiv (P \parallel P') \parallel P''$ và $P \parallel 0 \equiv P$.

- Trong quy tắc S-SPAWN, hàm $spawn(p,p',\Gamma)$ thêm vào Γ một luồng mới với định danh là p' và một môi trường cục bộ được nhân bản từ môi trường cục bộ của p. Một cách hình thức, giả sử $\Gamma = \{p : E\} \cup \Gamma''$ và $spawn(p,p',\Gamma) = \Gamma'$, thì $\Gamma' = \Gamma \cup \{p' : E'\}$ trong đó $E' = E$.
- Trong quy tắc S-TRANS, hàm $start(l : n, p, \Gamma)$ tạo thêm một *log* với nhãn l và kích cỡ n ở cuối của môi trường cục bộ của p_i. Nếu $start(l{:}n, p_i, \Gamma) = \Gamma'$ trong đó $\Gamma = \{p_1 : E_1, \ldots, p_i : E_i, \ldots, p_k : E_k\}$ và l là một nhãn được làm mới, thì $\Gamma' = \{p_1 : E_1, \ldots, p_i : E_i', \ldots, p_k : E_k\}$, trong đó $E_i' = E_i; l : n$.

  Lưu ý rằng, trong quy tắc này, khi mở một giao dịch, tài nguyên bộ nhớ vẫn chưa

được cấp phát, khi khởi tạo các biến trong giao dịch tài nguyên mới được cấp phát. Đây là một điểm khác biệt của nghiên cứu này so với nghiên cứu [19].

- Trong quy tắc S-COMM, hàm $intranse(\Gamma, l : n)$ trả lại một tập của tất cả các luồng, được ký hiệu là $\mathbf{p}$, trong Γ mà môi trường cục bộ bao gồm định danh *log* l và định danh *log* này là phần tử sau cùng của môi trường cục bộ. Điều đó có nghĩa là nếu $intranse(\Gamma, l{:}n) = \mathbf{p} = \{p_1, .., p_k\}$ thì:
  - với mọi $i \in \{1..k\}$, p_i có dạng $E'_i; l : n$.
  - với mọi $p' : E' \in \Gamma$ giả sử $p' \notin \{p_1, .., p_k\}$, ta có E' không chứa *log* có định danh l.
- Cũng trong quy tắc S-SPAWN, hàm $commit(\mathbf{p}, \Gamma)$ hủy bỏ *log* sau cùng trong môi trường cục bộ của tất cả các luồng trong $\mathbf{p}$. Nghĩa là, giả sử $intranse(\Gamma, l{:}n) = \mathbf{p}$ và $commit(\mathbf{p}, \Gamma) = \Gamma'$, thì với mọi $p' : E' \in \Gamma'$, nếu $p' \in \mathbf{p}$, thì $p' : (E'; l : n) \in \Gamma$. Các trường hợp khác, $p' : E' \in \Gamma$.

Chú ý rằng hàm $spawn$ sao chép các nhãn của môi trường của luồng cha của nó sang môi trường cục bộ của luồng mới và hàm $intranse$ tìm các nhãn của chúng để xác định các luồng cần đồng bộ trong một phép đồng kết thúc.

Các quy tắc trong Hình *3.* có ý nghĩa như sau:

- Quy tắc S-SPAWN được sử dụng để tạo một luồng mới với lệnh `spawn`. Lệnh `spawn` $\{e_1\}$ tạo một luồng mới p' thực thi e_1 song song với luồng cha của nó p, và thay đổi môi trường từ Γ sang Γ'.
- Quy tắc S-TRANS dùng trong trường hợp luồng p tạo một giao dịch mới với câu lệnh **onacid**. Một giao dịch mới với nhãn l được tạo, và thay đổi môi trường từ Γ sang môi trường Γ'.
- Quy tắc S-COMM để kết thúc một giao dịch. Trong quy tắc này $\coprod_1^k p_i(E_i)$ viết tắc cho $p_1(e_1) \parallel .. \parallel ..p_k(e_k)$. Nếu giao dịch hiện tại của luồng p là l, thì tất cả các luồng trong giao dịch l phải đồng kết thúc khi giao dịch l kết thúc.
- Quy tắc S-COND$_1$ và S-COND$_2$ áp dụng cho các lệnh điều kiện.
- Quy tắc S-WHILE và S-NO WHILE áp dụng cho các lệnh vòng lặp.
- Quy tắc S-INT áp dụng cho việc khai báo và khởi tạo một biến x kiểu $\mathcal{T}$. Khi khởi tạo một biến dùng chung, tài nguyên bộ nhớ mà chương trình sử dụng sẽ tăng lên tương ứng với bộ nhớ cấp phát cho biến đó, vì vậy môi trường sẽ thay đổi từ Γ sang Γ'. Giả sử $\Gamma = \{p_1{:}E_1, \ldots, p_i{:}E_i \ldots, p_k{:}E_k\}$, và $E_i = \{l_1{:}n_1; \ldots; l_i{:}n_i; \ldots; l_k{:}n_k\}$ thì $\Gamma' = \{p_1{:}E_1, \ldots, p_i{:}E'_i \ldots, p_k{:}E_k\}$, và $E'_i = \{l_1{:}n_1; \ldots; l_i{:}n'_i; \ldots; l_k{:}n_k\}$, trong đó $n'_i = n_i + m$ với m là bộ nhớ cấp phát cho biến mới được khởi tạo.
- Quy tắc S-ASSIGN áp dụng cho phép gán giá trị của biểu thức e_1 vào biến x.
- Quy tắc S-SKIP dùng cho các lệnh tính toán khác của ngôn ngữ mà chúng không ảnh hưởng gì tới ngữ nghĩa giao dịch và đa luồng, vì vậy ta có thể bỏ qua chúng.
- Quy tắc S-ERROR-C và S-ERROR-O được sử dụng trong trường hợp chương trình có các lỗi bắt đầu và kết thúc các giao dịch. Ví dụ, `onacid;spawn{commit;commit };commit;` có một lỗi trong commit thứ hai trong `spawn{commit;commit}`. $p()$ trong S-ERROR-O nghĩa là có một lỗi kết thúc trong chương trình.

4. Hệ thống kiểu

Hệ thống kiểu trong nghiên cứu này được xây dựng với mục đích xác định bộ nhớ *log* tối đa một chương trình giao dịch cần sử dụng một cách hoàn toàn tự động. Kiểu của một thành phần của chương trình được thể hiện bằng một chuỗi *số được gán dấu*, chúng mô tả trừu tượng các hành vi giao dịch của chương trình. Hệ thống kiểu này được phát triển từ Hệ thống kiểu trong nghiên cứu [19] với những bổ sung cải tiến để tính được bộ nhớ *log* tối đa của chương trình giao dịch một cách tự động. Một số ký hiệu trong hệ thống kiểu này được dùng lại từ các nghiên cứu trước nhưng ý nghĩa của chúng có thể khác.

4.1. Kiểu

Kiểu của một thành phần là một chuỗi hữu hạn các *số có dấu*. Một số có dấu là một cặp của một số tự nhiên không âm thuộc tập $\mathbb{N}^+$ và một *dấu*. Ta sử dụng tập các dấu (ký hiệu) $\{\star, +, -, \neg, \sharp\}$ để lần lượt ký hiệu cho việc khởi tạo một biến dùng chung, bắt đầu, kết thúc, đồng kết thúc một giao dịch và bộ nhớ tối đa cấp phát cho các *log*. Tập các số có dấu được ký hiệu là ${}^T\mathbb{N}$, nghĩa là ${}^T\mathbb{N} = \{ {}^{\star}n, {}^{+}n, {}^{-}n, {}^{\sharp}n, {}^{\neg}n \mid n \in \mathbb{N}^+\}$. Các số được gán dấu này có ý nghĩa như sau:

- ${}^{\star}n$: khởi tạo một biến dùng chung, và bộ nhớ cần cấp phát cho biến đó là n.
- ${}^{+}n$: mở một giao dịch với bộ nhớ *log* cần cấp phát cho giao dịch đó là n. Khi $n = 0$, nghĩa là khởi tạo một giao dịch nhưng chưa khởi tạo biến nào trong giao dịch đó.
- ${}^{-}n$: có n lệnh commit liên tiếp để kết thúc các giao dịch trước đó.
- ${}^{\neg}n$: có n luồng cần đồng bộ tại một thời điểm.
- ${}^{\sharp}n$: bộ nhớ *log* tối đa cần sử dụng cho một thành phần của chương trình là n.

Để hiểu rõ hơn về hệ thống kiểu này, ta xem xét một số ví dụ sau: giả sử một biến số nguyên cần 4 bytes bộ nhớ thì câu lệnh `global int x:=0;` có kiểu là ${}^{\star}4$; `onacid` có kiểu là ${}^{+}0$; `commit` có kiểu là ${}^{-}1$; hay `onacid global int x:=0; commit;` có kiểu là ${}^{+}0\ {}^{\star}4\ {}^{-}1$, hoặc có thể rút gọn thành ${}^{\sharp}4$. Với `spawn(onacid global int x:=0; commit;commit;commit)`, kiểu của nó là $({}^{+}0\ {}^{\star}4\ {}^{-}1\ {}^{-}1\ {}^{-}1)^{\rho}$ và có thể rút gọn thành $({}^{\sharp}4\ {}^{\neg}1\ {}^{\neg}1)^{\rho}$ bằng cách kết hợp các phần tử $+ \star -$ và xác định các phần tử đồng kết thúc.

Phần tiếp theo chúng tôi sẽ trình bày các quy tắc kiểu để kết hợp, rút gọn các chuỗi số có dấu cho một thành phần của chương trình. Trong quá trình tính toán, một dấu gắn với số không (ví dụ ${}^{\sharp}0$, ${}^{-}0$, ...) có thể được tạo ra nhưng nó không ảnh hưởng tới ngữ nghĩa của chuỗi vì vậy chúng ta sẽ tự động loại bỏ khi nó xuất hiện. Để thuận tiện cho quá trình tính toán, chúng ta cũng có thể chèn thêm phần tử ${}^{\sharp}0$ khi nào cần thiết.

Ta gọi s là phần tử thuộc tập ${}^T\mathbb{N}$, S thuộc tập ${}^T\bar{\mathbb{N}}$ với ${}^T\bar{\mathbb{N}}$ là tập các chuỗi số có dấu, và $m, n, l, ..$ thuộc tập $\mathbb{N}$, chuỗi trống được ký hiệu là ϵ. Với chuỗi S ta ký hiệu $|S|$ cho độ dài của S, và $S(i)$ cho phần tử thứ i của S. Với số có dấu s, ta ký hiệu $\mathrm{tag}(s)$ là dấu của s, và $|s|$ là số tự nhiên của s (nghĩa là $s = {}^{\mathrm{tag}(s)}|s|$). Với chuỗi $S \in {}^T\bar{\mathbb{N}}$, ta viết $\mathrm{tag}(S)$ cho chuỗi các dấu của các phần tử của S và $\{S\}$ cho tập các dấu xuất

hiện trong S. Chú ý rằng $\mathrm{tag}(s_1 \dots s_k) = \mathrm{tag}(s_1) \dots \mathrm{tag}(s_k)$. Để đơn giản ta cũng viết $\mathrm{tag}(s) \in S$ là rút gọn cho $\mathrm{tag}(s) \in \{S\}$.

Tập ${}^{T}\bar{\mathbb{N}}$ có thể được phân thành các lớp tương đương, tất cả các phần tử trong lớp tương đương mô tả các hành vi giao dịch tương tự, và mỗi lớp ta sử dụng chuỗi ngắn gọn nhất để mô tả cho lớp và ta gọi nó là chuỗi *chính tắc*.

Định nghĩa 3 (Chuỗi chính tắc)**.** *Một chuỗi S là* chính tắc *nếu* $\mathrm{tag}(S)$ *không chứa các thành phần* '$\star\star$', '$+\star$', '$--$', '$\sharp\sharp$', '$+-$', '$+\sharp-$', '$+\sharp\star-$', '$+\neg$', '$+\sharp\neg$, '$+\sharp\star\neg$' *và* $|S(i)| > 0$ *với mọi* i.

Ta luôn có thể rút gọn một chuỗi S mà không làm thay đổi ý nghĩa của chúng. Hàm seq dưới đây sẽ rút gọn một chuỗi ${}^{T}\bar{\mathbb{N}}$ thành chuỗi chính tắc. Chú ý mẫu '$+-$' không xuất hiện bên trái, nhưng ta có thể chèn thêm ${}^{\sharp}0$ để áp dụng hàm. Hai mẫu '$+\neg$' và '$+\sharp\neg$', sẽ được sử lý bởi hàm jc trong Định nghĩa 8.

Định nghĩa 4 (Rút gọn chuỗi)**.** *Hàm rút gọn một chuỗi được định nghĩa đệ quy như sau:*

$$
\begin{aligned}
\mathrm{seq}(S) &= S \textit{ khi } S \textit{ là chuỗi chính tắc} \\
\mathrm{seq}(S\,{}^{\star}m\ {}^{\star}n\,S') &= \mathrm{seq}(S\,{}^{\star}(m+n)S') \\
\mathrm{seq}(S\,{}^{\sharp}m\ {}^{\sharp}n\,S') &= \mathrm{seq}(S\,{}^{\sharp}\max(m,n)\,S') \\
\mathrm{seq}(S\,{}^{-}m\ {}^{-}n\,S') &= \mathrm{seq}(S\,{}^{-}(m+n)\,S') \\
\mathrm{seq}(S\,{}^{+}m\ {}^{\star}n\,S') &= \mathrm{seq}(S\,{}^{+}(m+n)\,S') \\
\mathrm{seq}(S\,{}^{+}k\ {}^{\sharp}l\ {}^{-}n\,S') &= \mathrm{seq}(S\,{}^{\sharp}(l+k)\ {}^{-}(n-1)\,S') \\
\mathrm{seq}(S\,{}^{+}k\ {}^{\sharp}l\ {}^{\star}m\ {}^{-}n\,S') &= \mathrm{seq}(S\,{}^{\sharp}(l+k)\ {}^{\sharp}(m+k)\,{}^{-}(n-1)\,S')
\end{aligned}
$$

Trong định nghĩa này, từ dòng thứ 2 đến dòng thứ 4 dùng để rút gọn chuỗi. Dòng thứ 5 mô tả việc cộng thêm kích cỡ của *log* của giao dịch khi khởi tạo thêm biến trong giao dịch. Dòng số 6 dùng cho các commit cục bộ – các commit này không đồng bộ với các luồng khác. Dòng sau cùng cũng dùng cho việc commit cục bộ như dòng số 6, nhưng sau lệnh `commit` sau cùng tạo nên ${}^{\sharp}l$ có thêm các lệnh khởi tạo biến (tạo ra kiểu ${}^{*}m$). Quy tắc này thể hiện rõ vai trò của lệnh `commit` mà trong các nghiên cứu trước chưa thể hiện được. Như minh họa trong Hình *1.* , các luồng được đồng bộ bởi các đồng kết thúc. Các đồng kết thúc này chia một luồng thành các *phân đoạn* và chỉ một số phân đoạn có thể chạy song song. Ví dụ, trong Hình *1.* , các `onacid` trên dòng 4, dòng 6 không thể chạy song song với `onacid` trên dòng 20.

Với kiểu cho một thành phần e, phân đoạn có thể được xác định bằng cách kiểm tra các thành phần $-$ hoặc $\neg$ trong kiểu của e trong `spawn`$\{e\}$. Ví dụ, trong `spawn`$\{e_1\}; e_2$, nếu chuỗi chính tắc của e_1 có $-$ hoặc $\neg$ thì luồng thực thi e_1 phải được đồng bộ với cha của nó đó là luồng thực thi e_2. Hàm merge trong Định nghĩa 6 được sử dụng trong tình huống này, nhưng để định nghĩa nó ta cần một số hàm phụ trợ sau đây:

Với $S \in {}^{T}\bar{\mathbb{N}}$ và một dấu $sig \in \{\star, +, -, \neg, \sharp\}$, hàm *first*$(S, sig)$ trả về chỉ số nhỏ nhất i mà $\mathrm{tag}(S(i)) = sig$. Nếu không có phần tử nào, hàm sẽ trả về 0. Một lệnh `commit`

có thể là một kết thúc cục bộ, hoặc một đồng kết thúc. Trước tiên, ta giả sử tất cả các `commit` là cục bộ, khi ta duyệt qua nếu không có lệnh bắt đầu giao dịch cục bộ nào (lệnh `onacid`) để khớp với một kết thúc cục bộ thì những kết thúc cần được chuyển thành đồng kết thúc. Hàm sau thực hiện việc đó và chuyển một chuỗi chính tắc không có phần tử $+$ thành *chuỗi đồng bộ*.

Định nghĩa 5 (Chuyển dạng chuỗi)**.** *Cho $S = s_1 \dots s_k$ là một chuỗi chính tắc mà $+ \notin \{S\}$ và giả sử $i = \mathit{first}(S, -)$, hàm $\mathrm{join}(S)$ thay thế đệ quy $-$ trong S bằng $\neg$ như sau:*

$$\begin{aligned} \mathrm{join}(S) &= S && \textit{nếu } i = 0 \\ \mathrm{join}(S) &= s_1..s_{i-1}\ {}^{\neg}1\ \ \mathrm{join}(\,{}^{\neg}(\,|s_i| - 1)s_{i+1}..s_k) && \textit{trường hợp khác} \end{aligned}$$

Chú ý rằng trong Định nghĩa 5 chuỗi chính tắc S chỉ bao gồm các phần tử $\sharp$ xen kẽ với các phần tử $-$ hoặc $\neg$. Sau khi áp dụng hàm join, ta nhận được chuỗi đồng bộ. Các chuỗi đồng bộ này chỉ bao gồm các phần tử $\sharp$ xen kẽ với các phần tử $\neg$. Một chuỗi đồng bộ được sử dụng để định kiểu một thành phần bên trong một `spawn` hoặc một thành phần bên trong luồng chính. Các chuỗi đồng bộ được hợp với nhau theo định nghĩa sau:

Định nghĩa 6 (Hợp)**.** *Giả sử S_1 và S_2 là những chuỗi đồng bộ, khi đó số các phần tử $\neg$ trong S_1 và S_2 là tương tự như nhau (có thể bằng không). Hàm hợp được định nghĩa đệ quy như sau:*

$$\begin{aligned} \mathrm{merge}(\,{}^{\sharp}m_1\,,\,{}^{\sharp}m_2\,) &= {}^{\sharp}(m_1 + m_2) \\ \mathrm{merge}(\,{}^{\sharp}m_1\ {}^{\neg}n_1\ \ S_1',\,{}^{\sharp}m_2\ {}^{\neg}n_2\ \ S_2') &= {}^{\sharp}(m_1 + m_2)\ {}^{\neg}(n_1 + n_2)\ \mathrm{merge}(S_1', S_2') \end{aligned}$$

Định nghĩa là hợp lệ, bởi vì S_1, S_2 là các chuỗi đồng bộ vì vậy chúng chỉ có các phần tử $\sharp$ và $\neg$. Thêm nữa, số các phần tử ${}^{\sharp}$s là tương tự như nhau. Vì vậy ta có thể chèn thêm ${}^{\sharp}0$ để tạo ra hai chuỗi phù hợp với mẫu được định nghĩa. Chú ý rằng hàm hợp được sử dụng cho những thành phần có dạng `spawn`$\{e_1\}; e_2$, trong đó ta định kiểu cho e_1 trước, sau đó áp dụng hàm chuyển dạng ở trên để có được một chuỗi đồng bộ – kiểu của `spawn`$\{e_1\}$. Sau đó, ta cần định kiểu của e_2 để hợp với chuỗi đồng bộ của `spawn`$\{e_1\}$.

Ta cần thêm một hàm để định kiểu cho thành phần của dạng `if`$\{e_b\}$ `then` $\{e_1\}$ `else` $\{e_2\}$. Đối với những thành phần này, ta yêu cầu các hành vi giao dịch với bên ngoài của e_1 và e_2 phải tương tự nhau, nghĩa là khi gỡ bỏ tất cả các phần tử với dấu $\sharp$ từ chúng, chuỗi còn lại phải giống nhau. Giả sử S_1, S_2 là hai chuỗi như vậy thì chúng luôn có thể được viết lại thành $S_i = {}^{\sharp}m_i\ {}^{*}n\ S_i'$, $i = 1, 2$, $* = \{+, -, \neg\}$, trong đó S_1' và S_2' lần lượt có các hành vi giao dịch tương tự. Với điều kiện này, ta định nghĩa phép toán chọn như sau:

Định nghĩa 7 (Chọn)**.** *Chuỗi S_1 và S_2 là hai chuỗi mà nếu chúng ta loại bỏ tất cả các phần tử $\sharp$ từ chúng, thì phần còn lại hai chuỗi là giống nhau. Hàm* alt *được định*

nghĩa đệ quy như sau:

$$\text{alt}({}^{\sharp}m_1\,,\,{}^{\sharp}m_2\,) = {}^{\sharp}\text{max}(m_1, m_2)$$
$$\text{alt}({}^{\sharp}m_1\ {}^{*}n\ S_1', {}^{\sharp}m_2\ {}^{*}n\ S_2') = {}^{\sharp}\text{max}(m_1, m_2)\,{}^{*}n\ \ \text{alt}(S_1', S_2')$$

4.2. Quy tắc kiểu

Cú pháp của ngôn ngữ kiểu T có dạng: $T = S \mid S^{\rho}$. Dạng kiểu S^{ρ} được sử dụng cho `spawn`(e), chúng cần đồng bộ với luồng cha của nó nếu có đồng kết thúc. Các tính toán trên hai trường hợp là khác nhau, vì vậy chúng tôi ký hiệu $kind(T)$ cho loại của T, chúng có thể trống (thông thường) hoặc ρ nếu T có dạng S^{ρ}.

Môi trường kiểu cung cấp thông tin ngữ cảnh cho biểu thức đang được định kiểu. Các phát biểu kiểu có dạng: $n \vdash e : T$, trong đó $n \in \mathbb{N}$ là môi trường kiểu. Khi n âm, có nghĩa là khi thực thi e sẽ sử dụng n bytes bộ nhớ cho các *log* của nó. Khi n dương, nó có nghĩa là e cần giải phóng n bytes bộ nhớ của các *log* do các giao dịch đã mở trước đó.

$$\frac{x, e : int}{-4 \vdash global\ \mathcal{T}\ x := e :\ {}^{\star}4}\ \text{T-INT} \qquad \frac{x, e : bool}{-1 \vdash \mathcal{T}\ x := e :\ {}^{\star}1}\ \text{T-BOOL} \qquad \frac{}{0 \vdash \textbf{onacid} :\ {}^{+}0}\ \text{T-ONACID}$$

$$\frac{n \in \mathbb{N}^{+}}{n \vdash \textbf{commit} :\ {}^{-}1}\ \text{T-COMMIT} \qquad \frac{n \vdash e : S}{n \vdash \textbf{spawn}(e) : \text{join}(S)^{\rho}}\ \text{T-SPAWN} \qquad \frac{n \vdash e : S}{n \vdash e : \text{join}(S)^{\rho}}\ \text{T-PREP}$$

$$\frac{n_i \vdash e_i : S_i \quad i = 1,2 \quad S = \text{seq}(S_1 S_2)}{n_1 + n_2 \vdash e_1; e_2 : S}\ \text{T-SEQ} \qquad \frac{n_1 \vdash e_1 : S_1 \quad n_2 \vdash e_2 : S_2^{\rho} \quad S = \text{jc}(S_1, S_2)}{n_1 + n_2 \vdash e_1; e_2 : S}\ \text{T-JC}$$

$$\frac{n \vdash e_i : S_i^{\rho} \quad i = 1,2 \quad S = \text{merge}(S_1, S_2)}{n \vdash e_1; e_2 : S^{\rho}}\ \text{T-MERGE} \qquad \frac{n_1 \vdash e_b : bool \quad n_1 \vdash e :\ {}^{\sharp}n_2}{n_1 \vdash \textbf{while}(e_b)\{e\} :\ {}^{\sharp}n_2}\ \text{T-WHILE}$$

$$\frac{n \vdash e_i : T_i \quad i = 1,2 \quad n \vdash e_b : bool \quad kind(T_1) = kind(T_2) \quad T_i = S_i^{kind(T_i)}}{n \vdash \textbf{if}(e_b)\textbf{then}\{e_1\}\textbf{else}\{e_2\} : \text{alt}(S_1, S_2)^{kind(S_1)}}\ \text{T-COND}$$

Hình 4. *Quy tắc kiểu*

Các quy tắc định kiểu được thể hiện trong Hình *4.* . Ta không có quy tắc cho việc định kiểu α vì chúng không ảnh hưởng đến ngữ nghĩa giao dịch và đa luồng, vì vậy ta có thể loại bỏ chúng khi định kiểu chương trình. Ta giả sử rằng trong các quy tắc này hàm $\text{seq}, \text{jc}, \text{merge}, \text{alt}$ có thể áp dụng, nghĩa là các tham số của chúng thỏa mãn điều kiện của các hàm.

Quy tắc T-SPAWN chuyển đổi S thành chuỗi đồng bộ và đánh dấu kiểu mới bởi ρ vì vậy có thể hợp với cha của nó bằng T-MERGE. Quy tắc T-PREP cho phép ta tạo ra một kiểu cho e để sử dụng trong T-MERGE. Quy tắc T-WHILE sử dụng để định kiểu cho biểu thức vòng lặp `while`. Thực tế, việc phân tích, định kiểu cho biểu thức vòng lặp phức tạp hơn. Tuy nhiên để đảm bảo cho chương trình có kiểu hợp lệ, và việc định kiểu được thuận lợi trong nghiên cứu này chúng tôi chỉ định kiểu cho những vòng lặp mà thân của nó không chứa các lệnh `spawn` và kiểu của thân vòng lặp có dạng ${}^{\sharp}n$. Đây cũng là hạn chế mà chúng tôi sẽ tiếp tục cần phải hoàn thiện trong các nghiên cứu tiếp theo. Các quy tắc còn lại là đơn giản ngoại trừ quy tắc T-JC trong đó chúng ta cần thêm một hàm mới jc (trong Định nghĩa 8). Quy tắc T-JC để xử lý đồng kết thúc giữa các luồng đang chạy song song. Phần tử $+$ cuối cùng trong S_1, gọi là ${}^{+}n$, sẽ khớp với phần tử $\neg$

đầu tiên trong S_2, gọi là ${}^{\neg}l$. Nhưng sau ${}^{+}n$, có thể có phần tử $\sharp$, gọi là ${}^{\sharp}n'$, vì vậy bộ nhớ lớn nhất sử dụng bởi thành phần có kiểu ${}^{+}n\ {}^{\sharp}n'$ sẽ là $n + n'$. Trước ${}^{\neg}l$ có thể có ${}^{\sharp}l'$, vì vậy khi thực hiện đồng kết thúc của phần tử có kiểu ${}^{\neg}l$ với phần tử có kiểu ${}^{+}n$ trước đó, ta được kiểu của chúng là ${}^{\sharp}(l' + l * n)$. Sau khi kết hợp ${}^{+}n$ từ S_1 và ${}^{\neg}l$ từ S_2 ta có thể đơn giản hóa chuỗi mới và lặp lại việc đồng kết thúc với jc. Vì vậy, hàm jc được định nghĩa như sau:

Định nghĩa 8 (Đồng kết thúc)**.** *Hàm đồng kết thúc được định nghĩa đệ quy như sau:*

$$\mathrm{jc}(S_1'\ {}^{+}n\ {}^{\sharp}n',\ {}^{\sharp}l'\ {}^{\neg}l\ S_2') = \mathrm{jc}(\mathrm{seq}(S_1'\ {}^{\sharp}(n+n')), \mathrm{seq}({}^{\sharp}(l'+l*n)\ S_2'))\ \textit{nếu}\ l > 0$$
$$\mathrm{jc}({}^{\sharp}n',\ {}^{\sharp}l'\ S_2') = \mathrm{seq}({}^{\sharp}\mathrm{max}(n', l')\ S_2')\ \textit{cho các trường hợp khác}$$

Trong định nghĩa này mẫu phù hợp với dòng đầu tiên có mức ưu tiên hơn dòng thứ hai.

Vì kiểu trong nghiên cứu này phản ánh hành vi của một thành phần của một chương trình, vì vậy kiểu của một chương trình có kiểu hợp lệ là một chuỗi chỉ gồm một phần tử ${}^{\sharp}n$ với n là bộ nhớ *log* tối đa cần sử dụng khi thực thi chương trình.

Định nghĩa 9 (Kiểu hợp lệ)**.** *Một thành phần e có* kiểu hợp lệ *nếu có một phép suy diễn kiểu cho e mà $0 \vdash e : {}^{\sharp}n$ với một số n nào đó.*

Một phát biểu kiểu có một tính chất quan trọng đó là nếu kết hợp môi trường kiểu của một thành phần với kiểu của thành phần đó thì sẽ tạo ra một cấu trúc 'kiểu hợp lệ'.

Định lý 1 (Tính chất kiểu)**.** *Nếu $n \vdash e : T$ và $n \geqslant 0$, thì $\mathrm{sim}({}^{+}n, T) = {}^{\sharp}m$ với một số m nào đó (nghĩa là $\mathrm{sim}({}^{+}n, T)$ chỉ là một phần tử đơn $\sharp$) và $m \geqslant n$ trong đó $\mathrm{sim}(T_1, T_2) = \mathrm{seq}(\mathrm{jc}(S_1, S_2))$ với S_1, S_2 là T_1, T_2 không có ρ.*

Chứng minh (rút gọn): Việc chứng minh được thực hiện bằng cách xem xét các quy tắc kiểu trong Hình *4.* . ■

4.3. Định kiểu cho chương trình ví dụ

Bây giờ chúng ta sẽ xác định kiểu cho chương trình ví dụ trong phần 2. Ta ký hiệu e_m^l cho đoạn chương trình từ dòng l tới dòng m. Đầu tiên, sử dụng T-SEQ, T-ONACID, T-INT, T-COMMIT ta có $4 \vdash e_{10}^4 : {}^{+}0\ {}^{+}0\ {}^{\star}4\ {}^{-}1\ {}^{-}1\ {}^{-}1$ hay $4 \vdash e_{10}^4 : {}^{\sharp}4\ {}^{-}1$. Tiếp theo, áp dụng quy tắc T-SPAWN, ta có $4 \vdash e_{10}^3 : ({}^{\sharp}4\ {}^{\neg}1)^{\rho}$. Bây giờ, ta muốn áp dụng T-MERGE, ta cần tìm một đoạn chương trình mà kiểu của nó thể khớp với kiểu của e_{10}^3. Ta tìm được e_{19}^{11} thỏa mãn điều kiện này, bởi vì áp dụng T-WHILE, T-COND, T-ONACID, T-INT, T-COMMIT, T-COND ta có $4 \vdash e_{19}^{11} : {}^{\sharp}4\ {}^{\sharp}4\ {}^{-}1$ hay $4 \vdash e_{19}^{11} : {}^{\sharp}4\ {}^{-}1$. Bằng việc áp dụng T-PREP, ta có một kiểu phù hợp với kiểu của e_{10}^3. Vì vậy ta có thể áp dụng T-MERGE để nhận được kiểu cho e_{19}^3 như sau: $4 \vdash e_{19}^3 : ({}^{\sharp}8\ {}^{\neg}2)^{\rho}$. Tại dòng 12, ta có khởi tạo biến `i`, tuy nhiên đây chỉ là biến cục bộ của luồng, không phải biến dùng chung, vì vậy nó không ảnh hưởng tới bộ nhớ *log* theo cơ chế bộ nhớ giao dịch. Vì vậy ta bỏ qua biến này. Tiếp theo, áp dụng quy tắc T-ONACID, T-INT, ta có: $-4 \vdash e_2^1 : {}^{+}0\ {}^{\star}4$ hay $-4 \vdash e_2^1 : {}^{+}4$. Áp dụng T-JC ta nhận được kiểu cho e_{19}^1 là: $0 \vdash e_{19}^1 : {}^{\sharp}16$ bởi vì $\mathrm{jc}({}^{+}4, {}^{\sharp}8\ {}^{\neg}2) = \mathrm{jc}(\mathrm{seq}({}^{\sharp}4), \mathrm{seq}({}^{\sharp}(8 + 4 * 2))) = \mathrm{jc}({}^{\sharp}4, {}^{\sharp}16) = {}^{\sharp}16$.

Áp dụng quy tắc T-ONACID, T-INT, T-COMMIT ta có: $-4 \vdash e_{23}^{20} : {}^{+}0\ {}^{*}4\ {}^{-}1$ hay $-4 \vdash e_{23}^{20} : {}^{\sharp}4$. Áp dụng T-SEQ kết hợp e_{19}^{1} với e_{23}^{20} ta có: $0 \vdash e_{23}^{1} : {}^{\sharp}16$ Như vậy chương trình có kiểu hợp lệ và bộ nhớ *log* tối đa cần sử dụng trong trường hợp này là 16 bytes.

Phần tiếp theo, chúng ta sẽ chứng minh tính đúng đắn của hệ thống kiểu đã đề xuất.

5. Tính đúng của hệ thống kiểu

Tính đúng đắn của hệ thống kiểu trong nghiên cứu này được hiểu là với một chương trình có kiểu hợp lệ sẽ không sử dụng nhiều bộ nhớ *log* hơn mô tả trong kiểu của nó.

Giả sử một chương trình có kiểu hợp lệ e và có kiểu là ${}^{\sharp}n$, ta cần chứng minh khi thực thi e theo ngữ nghĩa trong Phần 3, tổng số bộ nhớ *log* của e trong môi trường toàn cục luôn nhỏ hơn hoặc bằng n.

Một trạng thái của chương trình là một cặp Γ, P trong đó $\Gamma = \{p_1{:}E_1, \ldots, p_k{:}E_k\}$ và $P = \coprod_1^k p_i(e_i)$. Ta nói Γ thỏa mãn P, ký hiệu $\Gamma \models P$, nếu tồn tại $S_1, \ldots, S_k$ mà $[\![E_i]\!] \vdash e_i : S_i$ với mọi $i = 1, \ldots, k$. Với một thành phần i, E_i mô tả bộ nhớ *log* đã được cấp phát cho các biến mới được tạo ra hay các biến được sao chép trong luồng p_i, và S_i mô tả bộ nhớ *log* sẽ được cấp phát khi thực thi e_i.

Vì vậy, tổng bộ nhớ *log* được sử dụng bởi luồng p_i được mô tả bởi $\mathrm{sim}({}^{+}[\![E_i]\!], S_i)$, trong đó hàm sim được định nghĩa trong Định lý 1. Ta sẽ chứng minh rằng $\mathrm{sim}({}^{+}[\![E_i]\!], S_i) = {}^{\sharp}n$. Ta ký hiệu giá trị n này là $[\![E_i, S_i]\!]$. Tổng bộ nhớ *log* của một trạng thái chương trình, bao gồm tại Γ và các *log* sẽ được tạo ra khi thực thi phần còn lại của chương trình, ký hiệu là $[\![\Gamma, P]\!]$, và được định nghĩa bởi công thức: $[\![\Gamma, P]\!] = \sum_{i=1}^{k} [\![E_i, S_i]\!]$.

Vì $[\![\Gamma, P]\!]$ mô tả bộ nhớ *log* tối đa *từ* trạng thái hiện tại và $[\![\Gamma]\!]$ là bộ nhớ *log* tối đa *ở* thời điểm hiện tại, ta có định lý sau:

Bổ đề 1. *Nếu* $\Gamma \models P$, *thì* $[\![\Gamma, P]\!] \geqslant [\![\Gamma]\!]$.

Chứng minh (rút gọn): Theo Định nghĩa của $[\![\Gamma, P]\!]$ và $[\![\Gamma]\!]$, ta cần chứng minh rằng $[\![E_i, S_i]\!] \geqslant [\![E_i]\!]$ với mọi i. Điều này được suy ra từ Định lý 1. ■

Bổ đề 2 (Tính bất biến)**.** *Nếu* $\Gamma \models P$ *và* $\Gamma, P \Rightarrow \Gamma', P'$, *thì* $\Gamma' \models P'$ *và* $[\![\Gamma, P]\!] \geqslant [\![\Gamma', P']\!]$.

Chứng minh (rút gọn): Chứng minh được thực hiện bằng việc kiểm tra từng quy tắc ngữ nghĩa trong Hình *3.* . Với mỗi quy tắc, ta cần chứng minh hai phần: (i) $\Gamma' \models P'$ và (ii) $[\![\Gamma, P]\!] \geqslant [\![\Gamma', P']\!]$. ■

Định lý 2 (Tính đúng)**.** *Giả sử* $0 \vdash e : {}^{\sharp}n$ *và* $p_1 : \epsilon, p_1(e) \Rightarrow^* \Gamma, P$, *thì* $[\![\Gamma]\!] \leqslant n$.

Chứng minh (rút gọn): Với môi trường ban đầu ta có: $[\![p_1 : \epsilon, p_1(e)]\!] = \mathrm{sim}(0, {}^{\sharp}n) = {}^{\sharp}n$. Vì vậy từ Định lý 2 và Định lý 1, định lý được chứng minh theo quy nạp trên độ dài của giao dịch.

Định lý cuối cùng này cho ta thấy rằng tổng bộ nhớ *log* mà chương trình cần sử dụng khi thực thi không bao giờ vượt quá mô tả trong kiểu của chúng, nghĩa là hệ thống kiểu của chúng tôi đề xuất cho kết quả chính xác. ■

6. Kết luận

Chúng tôi đã trình bày một ngôn ngữ với các đặc điểm chính về đa luồng và bộ nhớ giao dịch. Trong nghiên cứu này, chúng tôi đã mở rộng ngôn ngữ với một số cấu trúc lệnh để gần với thực tế hơn. Kết quả chính của nghiên cứu này là một hệ thống kiểu để xác định tài nguyên bộ nhớ *log* tối đa mà chương trình giao dịch đa luồng cần sử dụng. Khác với những nghiên cứu trước đây, tài nguyên mà chương trình cần sử dụng trong nghiên cứu này là bộ nhớ *log* được tính toán cụ thể từ bộ nhớ cấp phát cho các biến dùng chung của các giao dịch. Các kết quả trên đã được chứng minh, thử nghiệm đầy đủ và cho các kết quả đúng với các tính toán lý thuyết. Hệ thống kiểu của nghiên cứu này là nền tảng để chúng ta xây dựng một hệ thống kiểu cho ngôn ngữ lập trình thực tế. Áp dụng những quy tắc của hệ thống kiểu này, người lập trình có thể tính được bộ nhớ *log* tối đa mà chương trình cần sử dụng từ đó có thể tối ưu chương trình của mình để tiết kiệm bộ nhớ, hạn chế các lỗi tràn bộ nhớ. Trong các nghiên cứu tiếp theo chúng tôi sẽ tiếp tục cải tiến hệ thống kiểu này để sử dụng thuận tiện hơn trong thực tế.

Tài liệu tham khảo

[1] Elvira Albert, Puri Arenas, Jesús Correas Fernández, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, and Guillermo Román-Díez. Object-sensitive cost analysis for concurrent objects. *Softw. Test., Verif. Reliab.*, 25(3):218–271, 2015.

[2] Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. Peak cost analysis of distributed systems. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014*, volume 8723 of *LNCS*, pages 18–33. Springer, 2014.

[3] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 105–116, New York, NY, USA, 2007. ACM.

[4] David Aspinall, Robert Atkey, Kenneth MacKenzie, and Donald Sannella. Symbolic and analytic techniques for resource analysis of Java bytecode. In *Proceedings of the 5th International Conference on Trustworthly Global Computing*, TGC'10, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Marc Bezem, Dag Hovland, and Hoang Truong. A type system for counting instances of software components. *Theor. Comput. Sci.*, 458:29–48, 2012.

[6] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96), Philadelphia, Pennsylvania, May 24-26, 1996.*, pages 213–225, 1996.

[7] Guy E. Blelloch and Robert Harper. Cache and I/O efficent functional algorithms. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 39–50, 2013.

[8] Victor Braberman, Diego Garbervetsky, Samuel Hym, and Sergio Yovine. Summary-based inference of quantitative bounds of live heap objects. *Science of Computer Programming*, 92, Part A:56 – 84, 2014. Special issue on Bytecode 2012.

[9] Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[10] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. Memory usage verification for OO programs. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 70–86, Berlin, Heidelberg, 2005. Springer-Verlag.

[11] Truong Anh Hoang and Nguyen Ngoc Khai. A type system for counting logs of a minimal language with multithreaded and nested transactions. *Journal of science of hnue*, 60:80–93, 2015.

[12] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *LNCS*, pages 132–157. Springer, 2015.

[13] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. volume 38, pages 185–197, New York, NY, USA, January 2003. ACM.

[14] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ml programming. *SIGPLAN Not.*, 34(9):70–81, September 1999.

[15] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, August 2005.

[16] Ngoc-Khai Nguyen and Anh-Hoang Truong. A compositional type systems for finding log memory bounds of transactional programs. In *Proceedings of the Eighth International Symposium on Information and Communication Technology*, SoICT 2017, pages 409–416, New York, NY, USA, 2017. ACM.

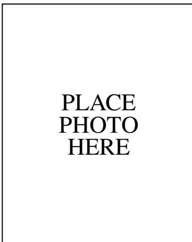
[17] Tuan-Hung Pham, Anh-Hoang Truong, Ninh-Thuan Truong, and Wei-Ngan Chin. A fast algorithm to compute heap memory bounds of Java Card applets. In Antonio Cerone and Stefan Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 259–267. IEEE Computer Society, 2008.

[18] Anh-Hoang Truong, Dang Van Hung, Duc-Hanh Dang, and Xuan-Tung Vu. A type system for counting logs of multi-threaded nested transactional programs. In Nikolaj Bjørner, Sanjiva Prasad, and Laxmi Parida, editors, *Distributed Computing and Internet Technology - 12th International Conference, ICDCIT 2016, Proceedings*, volume 9581 of *LNCS*, pages 157–168. Springer, 2016.

[19] Anh-Hoang Truong, Ngoc-Khai Nguyen, Dang Van Hung, and Duc-Hanh Dang. Calculating statically maximum log memory used by multi-threaded transactional programs. In *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, 2016, Proceedings*, Lecture Notes in Computer Science, pages 3–27 (to appear). Springer, 2015.

Ngày nhận bài 09-11-2017; Ngày chấp nhận đăng 21-03-2018.

■



Nguyễn Ngọc Khải tốt nghiệp Đại học Quốc gia Hà Nội năm 2003, tốt nghiệp thạc sĩ tại trường Đại học Công nghệ - Đại học Quốc gia Hà Nội năm 2010. Hiện nay anh là giảng viên trường Đại học Tài nguyên và Môi trường Hà Nội. Hướng nghiên cứu chính của anh là lý thuyết kiểu, phương pháp hình thức và công nghệ tác tử.

Trương Anh Hoàng nhận học vị tiến sĩ năm 2006 tại Đại học Bergen, Na Uy, được phong hàm phó giáo sư năm 2015 và hiện nay là giảng viên trường Đại học Công nghệ - Đại học Quốc gia Hà Nội. Hướng nghiên cứu chính của ông là các lý thuyết kiểu, phương pháp hình thức và kiểm thử phần mềm.