ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN PHAN TÌNH

TÍNH CẬN TRÊN BỘ NHỚ LOG CỦA CHƯƠNG TRÌNH SỬ DỤNG GIAO DỊCH

LUẬN VĂN THẠC SỸ CÔNG NGHỆ THÔNG TIN

Hà Nội - 2016

ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

NGUYỄN PHAN TÌNH

TÍNH CẬN TRÊN BỘ NHỚ LOG CỦA CHƯƠNG TRÌNH SỬ DỤNG GIAO DỊCH

Ngành: Công Nghệ Thông Tin

Chuyên ngành: Kỹ thuật Phần Mềm

Mã số: 60480103

LUẬN VĂN THẠC SỸ CÔNG NGHỆ THÔNG TIN

NGƯỜI HƯỚNG DẪN KHOA HỌC: PGS.TS. Trương Anh Hoàng

Hà Nội - 2016

LỜI CAM ĐOAN

Tôi xin cam đoan luận văn này là do tôi thực hiện, được hoàn thành dưới sự hướng dẫn trực tiếp từ PGS.TS.Trương Anh Hoàng. Các trích dẫn có nguồn gốc rõ ràng, tuân thủ tôn trọng quyền tác giả. Luận văn này không sao chép nguyên bản từ bất kì một nguồn tài liệu nào khác.

Nếu có gì sai sót, tôi xin chịu mọi trách nhiệm.

Học viên

Nguyễn Phan Tình

LỜI CẢM ƠN

Để hoàn thành đề tài luận văn này, bên cạnh sự chủ động cố gắng của bản thân, tôi đã nhận được sự ủng hộ và giúp đỡ nhiệt tình từ các tập thể, cá nhân trong và ngoài trường.

Qua đây, cho phép tôi được bày tỏ lòng cảm ơn sâu sắc tới thầy PGS.TS.Trương Anh Hoàng, giảng viên trường Đại học công nghệ – Đại học Quốc gia Hà Nội, người đã trực tiếp động viên, định hướng và hướng dẫn tận tình trong quá trình học tập và hoàn thành đề tài luận văn này.

Đồng kính gửi lời cảm ơn đến tập thể các thầy, cô giáo trong trường Đại học Công Nghệ – Đại học Quốc gia Hà Nội đã trau dồi kiến thức cho tôi, điều đó là nền tảng quí báu góp phần to lớn trong quá trình vận dụng vào hoàn thiện luận văn.

Cuối cùng, tôi xin được gửi lòng biết ơn sâu sắc đến gia đình, bạn bè, đồng nghiệp đã tạo điều kiện về vật chất cũng như tinh thần, luôn sát cánh bên tôi, động viên giúp tôi yên tâm học tập và kết thúc khóa học.

Tôi xin chân thành cảm ơn!

MỤC LỤC

DANH MỤC CÁC KÝ HIỆU, THUẬT NGỮ, CHỮ VIẾT TẮT

STT	CHỮ VIẾT TẮT, THUẬT NGỮ, KÝ HIỆU	GIẢI NGHĨA
CHỮ VIẾT TẮT		
1	TM – Transactional Memory	Ngôn ngữ để viết chương trình giao dịch
2	STM - Software Transactional Memory	Bộ nhớ giao dịch phần mềm, một giải pháp viết các chương trình song song
THUẬT NGỮ		
1	Type System	Hệ thống kiểu
2	Transaction	Giao dịch
3	illegal memory reference	Tham chiếu bộ nhớ không hợp lệ
4	Data corruption	Hư hỏng dữ liệu
5	Thread	Luồng
6	Type checker	Bộ kiểm tra kiểu
7	Type checking	Trình kiểm tra kiểu
8	Well-behaved	Tính chất hành xử đúng của chương trình.
9	Well-formed	Tính chất thiết lập đúng của chương trình.
10	Ill-behaved	Tính chất hành xử yếu của chương trình.
11	Well-typed	Định kiểu tốt.
12	Ill-typed	Định kiểu yếu.
13	ADT-Abstract Data Type	Kiểu dữ liệu trừu tượng
14	Illegal instruction	Lệnh không hợp lệ
19	Atomicity	Tính nguyên tử
20	Consistency	Tính nhất quán
21	Isolation	Tính độc lập
22	Durability	Tính bền vững
23	Onacid	Trạng thái mở một giao dịch
24	Commit	Trạng thái kết thúc một giao dịch
25	Nested transactions	Các giao dịch lồng
26	Multi-threaded	Đa luồng
27	Spawn	Sinh luồng
28	Joint commits	Các commit của các luồng song song đồng thời thực hiện kết thúc một giao tác chung.
29	Local semantics	Ngữ nghĩa cục bộ
30	Global semantics	Ngữ nghĩa toàn cục
31	Local enviroments	Môi trường cục bộ
32	Global enviroments	Môi trường toàn cục

33	Syntax	Cú pháp
	KÝ HIỆU	
1	$^{+}$n	Mô tả thành phần + trong hệ thống kiểu dựa trên chuỗi số có dấu, mở giao dịch có kích thước là n đơn vị bộ nhớ
2	$^{-}$ n	Mô tả thành phần – trong hệ thống kiểu dựa trên chuỗi số có dấu, m thao tác *commit* liên tiếp.
3	$^{\#}$n	Mô tả thành phần # trong hệ thống kiểu,chỉ ra số đơn vị bộ nhớ lớn nhất được sử dụng bởi thành phần là n.
4	$^{\neg}$n	Mô tả thành phần ¬ thể hiện số luồng cùng đồng bộ tại một thời điểm joint commits

DANH MỤC CÁC BẢNG

DANH MỤC CÁC HÌNH VẼ

MỞ ĐẦU

Tính cấp thiết của đề tài

Cùng với sự phát triển như vũ bão của khoa học công nghệ, các vi xử lý hiện đại ngày càng thể hiện sức mạnh qua nhiều nhân (core) với tốc độ xử lý ngày càng cao. Có được như vậy là do bên trong các vi xử lý này được thiết kế các luồng (thread) có khả năng chạy và xử lý song song. Trước đây để lập trình đa luồng, người ta sử dụng cơ chế đồng bộ (synchronization) dựa trên khóa (lock) để áp đặt giới hạn về quyền truy cập tài nguyên trong một môi trường khi có nhiều luồng thực thi.Tuy nhiên, khi áp dụng phương pháp này thường nảy sinh các vấn đề như khóa chết (deadlock) hoặc các lỗi tiềm tàng…

Software Transactional Memory (STM- bộ nhớ giao dịch phần mềm) [8] là một giải pháp đơn giản hơn, nhưng vô cùng mạnh mẽ mà có thể giải quyết được hầu hết các vấn đề trên. Nó đã thay thế hoàn toàn giải pháp cũ trong việc truy cập bộ nhớ dùng chung. STM giao tiếp với bộ nhớ thông qua các giao dịch. Các giao dịch này cho phép tự do đọc, ghi để chia sẻ các biến và một vùng nhớ gọi là log sẽ được sử dụng để ghi lại các hoạt động này cho tới khi kết thúc giao dịch.

Một trong những mô hình giao dịch phức tạp sử dụng STM là mô hình giao dịch lồng và đa luồng (nested and multi-threaded transaction) [5]. Trong quá trình thực thi của các chương trình giao dịch lồng và đa luồng, khi các luồng mới được sinh ra hoặc một giao dịch được bắt đầu, các vùng bộ nhớ gọi là log sẽ được cấp phát. Các log này dùng để lưu lại bản sao của các biến dùng chung, nhờ vậy mà các luồng trên có thể sử dụng các biến này một cách độc lập.

Vấn đề đặt ra ở đây là tại thời điểm chương trình chạy liệu lượng bộ nhớ cần cấp phát cho các log có vượt quá tài nguyên bộ nhớ của máy, hay chương trình có thể chạy một cách trơn tru mà không gặp phải bất kỳ lỗi nào như hết bộ nhớ. Chính vì vậy, việc xác định cận trên của bộ nhớ ở thời điểm chạy chương trìnhcủa chương trình giao dịch là một vấn đề then chốt, có ý nghĩa hết sức quan trọng.

Chính vì lí do đó, trong luận văn thực hiện ở đây, một nghiên cứu sử dụng phương pháp phân tích tĩnh để giải quyết bài toán tính cận trên bộ nhớ log của chương trình có giao dịch sẽ được trình bày, dựa trên bài báo đã được các tác giả công bố trong [1].

Mục tiêu nghiên cứu

Luận văn được thực hiện dựa trên nghiên cứu trong bài báo [1]. Do vậy để có thể hiểu được giải pháp các tác giả đã đề xuất trong [1], trong luận văn này tập trung nghiên cứu các lý thuyết về hệ thống kiểu; Các khái niệm cơ bản cũng như tính chất của giao dịch; Nghiên cứu cú pháp và ngữ nghĩa của ngôn ngữ TM (Transactional Memory) – Một ngôn ngữ để viết các chương trình giao dịch. Từ việc nắm được giải pháp xây dựng hệ thống kiểu đề cập trong bài báo, một công cụ sẽ được cài đặt dựa trên ngôn ngữ C#.

Phương pháp nghiên cứu

Để thực hiện được mục tiêu đã đề ra trong luận văn, các phương pháp nghiên cứu sau đây đã được kết hợp:

- Phương pháp nghiên cứu lý thuyết: bao gồm phân tích, tổng hợp các tài liệu, bài báo có liên quan về lý thuyết hệ thống kiểu đặc biệt là hệ thống kiểu cho các chương trình TM, tài liệu về các thuật toán dựa trên hệ thống kiểu..

- Phương pháp thực nghiệm: Cài đặt thuật toán đã đề xuất, chạy thử để kiểm tra tính đúng đắn của chương trình.

Cấu trúc của luận văn

Luận văn được trình bày trong các phần, với nội dung chính của mỗi phần như sau:

Mở đầu: Nêu ra tính cấp thiết của đề tài, nêu ra các mục tiêu cần nghiên cứu, các phương pháp được sử dụng khi nghiên cứu và cấu trúc các phần của luận văn.

Chương 1: Giới thiệu bài toán

Trình bày nội dung cụ thể của bài toán tính cận trên bộ nhớ log của chương trình có sử dụng giao dịch, các vấn đề cần giải quyết trong bài toán này và hướng tiếp cận để giải quyết bài toán. Trong phần này, các điểm cải tiến của phương pháp giải quyết bài toán ở đây so với các phương pháp trước kia cũng được nêu ra. Ví dụ được trình bày trong mục 1.3 sẽ minh họa rõ ràng cho bài toán và hướng tiếp cận đã đưa ra.

Chương 2: Một số kiến thức cơ sở

Trình bày các lý thuyết cơ bản được sử dụng làm cơ sở để giải quyết bài toán, bao gồm: Lý thuyết về hệ thống kiểu như khái niệm, các thuộc tính hay ứng dụng của hệ thống kiểu trong thực tế; Lý thuyết về giao dịch, bộ nhớ giao dịch phần mềm gồm các khái niệm, tính chất, ứng dụng…

Chương 3: Ngôn ngữ giao dịch

Giới thiệu ngôn ngữ giao dịch TM (Transactional memory)- Một ngôn ngữ được dùng để viết các chương trình giao dịch. Trong chương này, cú pháp và ngữ nghĩa của ngôn ngữ TM sẽ được trình bày cụ thể.

Chương 4: Hệ thống kiểu cho chương trình giao dịch

Nghiên cứu hệ thống kiểu để giải quyết bài toán tính cận trên bộ nhớ log cho chương trình có giao dịch đã được trình bày trong bài báo [1]. Lý thuyết hệ thống kiểu được phát triển ở đây bao gồm các kiểu và các quy tắc kiểu.

Chương 5: Xây dựng công cụ và thực nghiệm

Cài đặt các thuật toán kiểu dựa trên hệ thống kiểu đã được trình bày ở chương 4. Từ các thuật toán đó, xây dựng công cụ để giải quyết bài toán tính cận trên bộ nhớ log và thực nghiệm để kiểm tra tính đúng đắn của công cụ.

Kết luận:

Đánh giá các kết quả đã đạt được, nêu ra tồn tại và hướng phát triển.

CHƯƠNG 1. GIỚI THIỆU BÀI TOÁN

1.1. Giới thiệu

Như chúng ta đã đề cập ở phần mở đầu, STM là giải pháp được sử dụng trong việc chia sẻ bộ nhớ dùng chung và một trong những mô hình giao dịch phức tạp sử dụng STM là mô hình giao dịch lồng và đa luồng (nested and multi-threaded transaction)

Ở đây, một giao dịch được gọi là lồng khi nó chứa một số giao dịch khác. Chúng ta gọi giao dịch cũ là giao dịch cha và gọi các giao dịch mà sinh ra trong giao dịch cha là giao dịch con. Các giao dịch con này phải được đóng trước giao dịch cha. Hơn nữa, giao dịch được gọi là đa luồng (multi-threaded) khi các luồng con sinh ra được chạy bên trong giao dịch đồng thời chạy song song với luồng cha đang thực thi giao dịch. Khi một giao dịch được bắt đầu một vùng bộ nhớ gọi là log được cấp phát dùng để lưu lại bản sao của các biến dùng chung. Một luồng mới hay luồng con, khi được sinh ra cũng sẽ tạo một bản sao các log giao dịch của luồng cha. Khi luồng cha thực hiện đóng (commit) giao dịch, tất cả các luồng con được tạo bên trong luồng cha phải cùng đóng với luồng cha. Chúng ta gọi kiểu đóng này là *join commit*, và thời điểm khi những commit này xảy ra được gọi là thời điểm joint commit. Ở thời điểm join commit bộ nhớ được cấp phát cho các log cũng được giải phóng. Join commit đóng vai trò như sự đồng bộ ngầm của các luồng song song. Chính vì hình thức đồng bộ này mà các luồng song song bên trong một giao dịch không hoàn toàn chạy độc lập.

Và vấn đề cần trả lời ở đây là liệu ở thời điểm chạy chương trình, liệu lượng bộ nhớ cần cấp phát cho các log có vượt quá tài nguyên bộ nhớ của máy, hay chương trình có thể chạy một cách trơn tru mà không gặp phải bất kỳ lỗi nào như hết bộ nhớ. Để trả lời cho câu hỏi này, chúng ta cần phải xác định được biên bộ nhớ của chương trình giao dịch hay chính là cận trên bộ nhớ được cấp phát cho các log ở thời điểm biên dịch.

Ở các nghiên cứu trước đây [2, 11], một hệ thống kiểu được phát triển để đếm số lượng log lớn nhất mà cùng tồn tại ở một thời điểm khi chương trình đang chạy. Con số này chỉ cho thông tin thô về bộ nhớ được sử dụng bởi các log giao dịch. Để quyết định thêm chính xác lượng bộ nhớ lớn nhất mà các log giao dịch có thể sử dụng, trong nghiên cứu [1] các tác giả đã đề xuất phương pháp giải quyết vấn đề với việc tính đến kích thước của mỗi log. Đây cũng là điểm cải tiến của hướng tiếp cận mới này so với các hướng tiếp cận trước đó.

Như vậy, bài toán cần giải quyết ở đây có thể phát biểu như sau: Tính toán lượng bộ nhớ yêu cầu lớn nhất cho toàn bộ chương trình giao dịch khi biết kích thước của các log.

1.2. Hướng tiếp cận

Để giải quyết bài toán đặt ra, trước hết chúng ta sẽ viết các chương trình giao dịch bằng một ngôn ngữ dành riêng cho nó, cụ thể là ngôn ngữ TM sẽ được trình bày trong chương 3.

Để thêm thông tin về kích thước của mỗi log, chúng ta sẽ mở rộng lệnh bắt đầu giao dịch trong các nghiên cứu trước để chứa thông tin này. Sau đó chúng ta phát triển một hệ thống kiểu để đánh giá tài nguyên bộ nhớ log mà các giao dịch có thể yêu cầu.

So với các nghiên cứu trước [2,11] thì ý tưởng về các cấu trúc kiểu trong nghiên cứu [1] không có gì thay đổi. Tuy nhiên, các ngữ nghĩa kiểu và các quy tắc kiểu là mới và khác so với các nghiên cứu trước đây.

1.3. Ví dụ minh họa

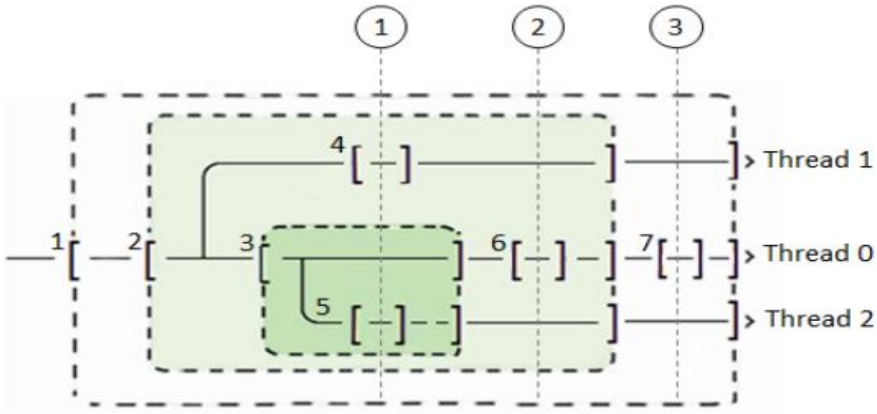
Để giải thích cho vấn đề và hướng tiếp cận đã trình bày trên, chúng ta sẽ xem xét một ví dụ, được mượn từ [1]. Trong ví dụ này chúng ta chỉ tập trung vào lõi của ngôn ngữ mà không quan tâm tới các cấu trúc khác ở một chương trình thật sự như các thủ tục, các phương thức gọi, truyền thông điệp, các biến và các tính toán cơ bản…

```
1  onacid(1); //thread 0
2    onacid(2);
3       spawn(onacid(4);commit;commit;commit);
4       onacid(3);
5         spawn(onacid(5); commit; commit; commit; commit);// thread 2
6       commit;
7       onacid(6); commit;
8    commit;
9    onacid(7);commit;
10 commit;
```

Hình 1.1 Đoạn mã cho mô hình giao dịch lồng và đa luồng

Trong đoạn mã ở trên, lệnh **onacid** và **commit** là các lệnh bắt đầu và đóng một giao dịch [8]. Lệnh **spawn** là lệnh tạo ra một luồng mới với mã được thể hiện bởi các tham số của lệnh. Lệnh **onacid** trong các nghiên cứu trước đây không có tham số, nhưng trong nghiên cứu này nó liên kết với một số để ký hiệu kích thước của bộ nhớ cần được cấp phát cho log của giao dịch ở thời điểm thực thi.

Các hành vi của chương trình này được miêu tả trong hình 1.2



Hình 2.2 Mô hình giao dịch lồng và đa luồng

Trong đó:

onacid : Lệnh bắt đầu giao dịch, ký hiêu bởi [

commit: Lệnh kết thúc giao dịch, bởi dấu].

Lệnh **spawn** tạo ra một luồng mới chạy song song với luồng cha của nó và được mô tả bằng đường kẻ ngang. Luồng mới sao chép các log của luồng cha cho việc lưu một bản sao giá trị các biến của luồng cha để nó có thể dùng các biến này một cách độc lập.

Trong hình vẽ trên các joint commit thể hiện thời điểm các luồng cha, con cùng đồng bộ được thể hiện bằng hình chữ nhật nét đứt, các điểm đồng bộ này được đánh dấu bằng cạnh bên phải của hình chữ nhật.

Sau đây chúng ta sẽ thực hiện tính tài nguyên bộ nhớ cho chương trình trong hình 1.2 tại 3 thời điểm khác nhau được chia ra theo các phân vùng độc lập 1, 2 và 3 như hình vẽ.

Ta thấy tại phân vùng 1, thì bộ nhớ log dành cho luồng 0 là1+2+3= 6 đơn vị bộ nhớ; Bộ nhớ log dành cho luồng 1 là (1+2)+4 = 7 đơn vị bộ nhớ, và Bộ nhớ log dành cho luồng 2 là (1+2)+3+5= 11 đơn vị bộ nhớ. Như vậy ở thời điểm này, tổng tài nguyên bộ nhớ được sử dụng là 6+7+11=24 đơn vị bộ nhớ.

Tại phân vùng 2, bộ nhớ log dành cho luồng 0 là 1+2+6=9 đơn vị bộ nhớ, bộ nhớ log dành cho luồng 1 là (1+2)= 3 đơn vị bộ nhớ; Và bộ nhớ log dành cho luồng 2 là (1+2) = 3 đơn vị bộ nhớ. Do đó tổng tài nguyên bộ nhớ ở thời điểm 2: 9+3+3= 15;

Tại phân vùng 3, ta tính được bộ nhớ log dành cho luồng 0 là 1+7= 8 đơn vị bộ nhớ, luồng 1 là 1 và luồng 2 là 1 đơn vị bộ nhớ. Do đó tổng tài nguyên bộ nhớ cho phân vùng 3 là 8+1+1=10 đơn vị bộ nhớ.

Như vậy tổng tài nguyên tổng hợp cho cả mô hình này phải là giá trị lớn nhất của tài nguyên tại các phân vùng, và giá trị thu được ở đây là 24.

Chúng ta thấy rằng trong các nghiên cứu [2,10,11], tài nguyên được ước tính bởi số giao dịch mở nhiều nhất hay số log lớn nhất mà cùng tồn tại ở thời điểm chương trình đang chạy.

Như chúng ta thấy trong ví dụ này, thì lượng bộ nhớ yêu cầu cho các log đạt tới có thể khác so với kết quả khi tính số lượng log lớn nhất đạt được trong các nghiên cứu như [2,11] (cũng ví dụ trên nhưng trong nghiên cứu [11] thì kết quả là 11).

CHƯƠNG 2. MỘT SỐ KIẾN THỨC CƠ SỞ

2.1. Hệ thống kiểu

2.1.1. Giới thiệu về hệ thống kiểu

Về định nghĩa hệ thống kiểu, có rất nhiều quan điểm được đưa ra. Trong các ngôn ngữ lập trình, hệ thống kiểu được định nghĩa là tập các quy tắc để gán thuộc tính được gọi là kiểu cho các cấu trúc của một chương trình máy tính bao gồm các biến, biểu thức, các hàm, hoặc module...Theo lý thuyết ngôn ngữ, một hệ thống kiểu là một tập các quy tắc quy định cấu trúc và lập luận cho ngôn ngữ. Trong lập trình, hệ thống kiểu được định nghĩa là một cơ chế cú pháp ràng buộc cấu trúc của chương trình bằng việc kết hợp các thông tin ngữ nghĩa với các thành phần trong chương trình và giới hạn phạm vi của các thành phần đó.

Mục đích cơ bản của hệ thống kiểu là ngăn chặn các sự cố do các lỗi thực thi trong quá trình chương trình chạy [3, 6]. Nó được thực hiện bằng cách định nghĩa các giao diện giữa các phần khác nhau của một chương trình máy tính, và sau đó kiểm tra xem các thành phần đã được ghép nối nhất quán hay chưa. Việc kiểm tra này có thể xảy ra tĩnh (tại thời gian biên dịch), hoặc động (tại thời gian chạy), hoặc kết hợp cả kiểm tra tĩnh và động. Ngoài ra hệ thống kiểu còn được sử dụng với nhiều mục đích khác, chẳng hạn như cho phép tối ưu hóa trình biên dịch nhất định, cung cấp một hình thức tài liệu…

Một hệ thống kiểu liên kết một kiểu với mỗi giá trị tính toán, bằng cách kiểm tra luồng (flow) của các giá trị, nỗ lực để đảm bảo hoặc chứng minh rằng không có lỗi kiểu có thể xảy ra.

Một trong những dấu hiệu của lỗi thực thi là lỗi phần mềm. Chẳng hạn lỗi có thể là lệnh sai (illegal instruction), tham chiếu bộ nhớ sai luật (illegal memory reference) hoặc hư hỏng dữ liệu (data corruption).

Một biến có thể có một miền giá trị trong suốt thời gian chạy chương trình. Kiểu của biến là cận trên của miền đó. Ví dụ, nếu một biến x kiểu Integer, giá trị của nó chỉ được phép là các giá trị nguyên trong bất kỳ lần thực thi nào của chương trình. Giả sử x và y có kiểu Interger, thì biểu thức x/y hợp lệ trong mọi lúc thực thi của chương trình. Ngược lại, nếu những biến này có kiểu String, thì x/y sẽ là nguyên nhân cho các lỗi phát sinh khác. Các ngôn ngữ đưa ra các kiểu không tầm thường (non- trivial) cho các biến được gọi là các ngôn ngữ định kiểu.

Các ngôn ngữ không định kiểu không giới hạn phạm vi giá trị của các biến. Tất cả các giá trị được chứa trong một kiểu phổ quát. Ví dụ về ngôn ngữ như vậy là assembly, ngôn ngữ này cho phép bất kỳ thao tác (phép tính) nào được thực hiện trên bất kỳ dữ liệu. Dữ liệu trong các ngôn ngữ đó được coi như khối các bit.

Một hệ thống kiểu trong một ngôn ngữ định kiểu theo dõi các kiểu của các biến và biểu thức trong một chương trình. Nó xác định liệu một tiến trình, một chương trình là hành xử đúng (well behaved) hay không. Các chương trình nguồn được kiểm chứng

bởi hệ thống kiểu để xác định rằng chúng cần được xem xét khi các chương trình hợp lệ hoặc cần bị loại bỏ khi các chương trình không an toàn. Một chương trình được cho là an toàn nếu nó không gây ra các lỗi mà không được chú ý trong một thời gian. Như các lỗi sẽ gây ra hành vi tùy ý (arbitrary behaviour). Ví dụ cho các lỗi đó là truy cập địa chi trái luật (illegal address accessing) (ví dụ : truy cập dữ liệu của bất kỳ mảng nào với chỉ số nằm ngoài các biên của mảng), nhảy tới địa chỉ sai (ví dụ bộ nhớ có hoặc không thể biểu diễn một luồng lệnh). Ngôn ngữ định kiểu tạo ra tính an toàn bằng cách sử dụng cả kiểm tra tĩnh hoặc cả kiểm tra động lẫn tĩnh cho tất cả chương trình. Bằng cách sử dụng kiểm tra tĩnh, ngôn ngữ định kiểu kiểm tra các chương trình trước khi chạy chúng (ví dụ ở thời điểm compile).Mặt khác, kiểm tra động được thực hiện khi chương trình đang chạy. Một ngôn ngữ có thể xác định như tập các lỗi chẳng hạn các lỗi cấm. Sau đó, ngôn ngữ kiểm chứng mỗi chương trình có là hành xử đúng (well behaved) hay không nếu chúng không gây ra bất kỳ lỗi nào mà không được phép xảy ra. Nói chung, một chương trình hành xử đúng phải là một chương trình an toàn . Điều đó có nghĩa là các lỗi không được phép phải bao gồm tất cả các lỗi không được lưu ý đã được mô tả trong phần trên. Trái ngược với hành xử tốt là hành xử yếu (ill behaved).

Các ngôn ngữ kiểu có thể thực hiện kiểm tra tĩnh để đảm bảo hành vi tốt và ngăn chặn tính không an toàn và các chương trình hành xử yếu được chạy. Quá trình kiểm tra được gọi là trình kiểm tra kiểu (typechecking), và các thuật toán được sử dụng được gọi là bộ kiểm tra kiểu (typechecker). Chương trình được cho là định kiểu tốt (well typed) nếu nó có thể vượt qua bộ kiểm tra kiểu; Ngược lại nếu không vượt qua, gọi là định kiểu yếu. Java hay Pascal là các ví dụ về ngôn ngữ sử dụng kiểm tra tĩnh .

Kiểm tra động là các kiểm tra trong thời gian thực thi để tìm ra tất cả các lỗi cấm. ngôn ngữ không định kiểu sử dụng kiểm tra động để thực thi hành vi tốt. Những ngôn ngữ này có thể kiểm tra các phép toán chia, giới hạn của mảng…khi lỗi xảy ra.

Để đạt được an toàn, các ngôn ngữ định kiểu có thể cần phải thực hiện các kiểm tra trong thời gian chạy. Ví dụ, giới hạn của mảng thường được kiểm tra động. Đó là trường hợp khi kiểm tra động sử dụng bởi một ngôn ngữ định kiểu. Vì vậy, một ngôn ngữ đã được kiểm tra tĩnh không có nghĩa là chương trình được thực hiện hoàn toàn mù quáng.

Theo định nghĩa, một chương trình hành xử tốt thì cũng an toàn. Mục tiêu cơ bản của hệ thống kiểu là để thực thi an toàn bằng cách loại trừ tất cả các lỗi không được chú ý trong tất cả các chương trình. Tuy nhiên, hầu hết các hệ thống loại này được thiết kế mạnh hơn. Chúng được sử dụng để đảm bảo thuộc tính hành xử tốt, và hoàn toàn an toàn. Hệ thống kiểu phân loại một chương trình như định kiểu yếu hoặc định kiểu tốt.

Một hệ thống kiểu đưa ra các quy tắc kiểu cho một ngôn ngữ lập trình. Trong hệ thống kiểu thuật toán của trình kiểm tra kiểu (typechecking) mà tương ứng với các

định nghĩa ngôn ngữ là độc lập với trình biên dịch. Với hệ thống kiểu cùng loại, trình biên dịch khác nhau có thể sử dụng các thuật toán kiểm tra kiểu khác nhau.

2.1.2. Các thuộc tính của hệ thống kiểu

Một hệ thống kiểu có một số thuộc tính sau:

Khả năng kiểm chứng: Hệ thống kiểu phải có thuật toán kiểm tra kiểu để phân loại các chương trình. Một hệ thống kiểu phải chủ động nắm bắt lỗi thực thi trước khi chúng xảy ra.

Tường minh: Các lập trình viên có thể dự đoán nếu một chương trình vượt qua bộ kiểm tra kiểu. Nếu nó lỗi khi kiểm tra, nên tìm được lí do một cách dễ dàng.

Khả năng thực thi: Các biến, biểu thức nên được kiểm tra tĩnh càng nhiều càng tốt. Mặt khác, chúng cũng cần được kiểm tra động. Sự nhất quán cần được kiểm chứng một cách thường xuyên.

2.1.3. Các ứng dụng của hệ thống kiểu

Hệ thống kiểu đóng vai trò quan trọng trong công nghệ phần mềm và trong lĩnh vực bảo mật mạng.

Đối với công nghệ phần mềm, nó được sử dụng trong trình biên dịch của các ngôn ngữ lập trình, tối ưu hóa, trong cơ sở dữ liệu và thậm chí là mô hình các ngôn ngữ tự nhiên… Trong ngôn ngữ lập trình, hệ thống kiểu có các chức năng chính sau :

a. Phát hiện lỗi

Khi chương trình chạy có thể xảy ra nhiều loại lỗi khác nhau. Có lỗi có thể tác động tức thì đến kết quả chương trình nhưng có những lỗi tiềm ẩn mà chỉ làm thay đổi dữ liệu nhưng không thấy ngay ở kết quả.

Ví dụ : Khi khai báo biến trong C#, nếu ta viết khai báo như sau:

```
bool x;
```

Trình biên dịch sẽ báo lỗi không hợp lệ vì không được phép biến khai báo mà không khởi tạo giá trị. Lỗi này sẽ dừng chương trình ngay lập tức. Để không bị báo lỗi, ta có thể sửa khai báo trên là : bool x= true;

Hệ thống kiểu có nhiệm vụ ngăn chặn các lỗi thực thi, lỗi mà có thể xảy ra khi chạy chương trình. Nhưng khi những lỗi này ở dạng tiềm tàng, hệ thống kiểu không thể nhận ra được. Vì vậy độ chính xác của hệ thống kiểu phụ thuộc vào nguyên nhân gây ra lỗi thực thi. Nó theo dõi kiểu của các đối số và có thể tìm ra các phần mã lệnh không hợp lệ. Hệ thống kiểu có thể theo dõi sự vắng mặt của lớp nào đó do lỗi lập trình nhờ vào khả năng phát hiện lỗi luồng dữ liệu logic.

Một số lỗi khi lập trình là do sử dụng dữ liệu sai và ở các vị trí chưa đúng. Hệ thống kiểu, tùy theo chương trình mà phân vùng giá trị hợp lệ và kiểm tra ở thời điểm chạy chương trìnhxem phân vùng này đã thỏa mãn chưa. Nhờ vậy, lập trình viên có thể phát hiện lỗi khi chạy chương trình.

b. Trừu tượng hóa

Các các thành phần trong một chương trình được trừu tượng hóa bởi một số kiểu dữ liệu. Các kiểu dữ liệu trừu tượng (ADT- Abstract Data Type), là cơ sở để định

nghĩa các kiểu dữ liệu mới, giấu đi cấu trúc bên trong của nó với các thành phần khác trong chương trình. Nó có thể có các giao diện cơ bản, để ẩn đi thông tin và hạn chế sự phụ thuộc vào các module khác. Các ngôn ngữ hướng đối tượng thường sử dụng các kiểu dữ liệu trừu tượng.

c. Làm tài liệu

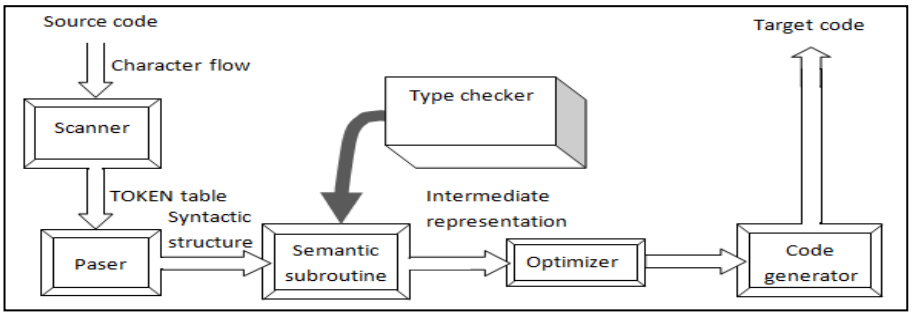
Trong nhiều ngôn ngữ kiểu tĩnh ví dụ C, một số thành phần của chương trình đòi hỏi người lập trình phải chú thích kiểu thông tin. Ví dụ,trong C biến phải được báo kiểu giá trị, mảng phải được khai báo kiểu mảng và kích thước tối đa, các định nghĩa hàm phải được khai báo kiểu trả về và kiểu của các đối số.

d. Tăng hiệu quả

Một chương trình được cho là định kiểu tốt (*well-typed*) sẽ cung cấp thông tin cho trình biên dịch, giúp trình biên dịch cải thiện quá trình dịch chương trình. Đối với các biến có kiểu tĩnh, vị trí thể hiện trong mã máy của các biến này có thể được quyết định bởi trình biên dịch. Việc này rất có giá trị khi một biến chỉ có thể có một kiểu giá trị cố định.

Trong lĩnh vực bảo mật mạng, hệ thống kiểu được sử dụng để xác minh các giao thức, cấu trúc thông tin trên web, đặc biệt trong nhân của một số mô hình bảo mật như Java hay JINI.

Ví dụ về hệ thống kiểu trong một trình biên dịch



Hình 2.1 Hệ thống kiểu trong trinh biên dịch

Một trình biên dịch thường có 3 trạng thái chính: scanning, syntatic analysis và type analysis tương ứng với 3 chức năng : scanner, parser và type checker. Trong đó, hệ thống kiểu sẽ nằm trong bộ Semantic subroutine, mà type checker là một thành phần trong đó. Với đầu vào là một chương trình được viết bằng một ngôn ngữ lập trình cụ thể, scanner sẽ sử dụng hữu hạn các trạng thái và ánh xạ chuỗi kí tự thu được từ đầu vào bảng TOKEN. Tiếp theo bảng TOKEN này sẽ được ánh xạ sang một cấu trúc trừu tượng nhờ có bộ parser. Bộ kiểm tra kiểu typechecker sẽ kiểm tra xem chương trình có là định kiểu tốt hay không.

2.2. Giao dịch và bộ nhớ giao dịch phần mềm (Software Transactional Memory-STM)

2.2.1. Giao dịch (Transaction)

Một giao dịch là một luồng điều khiển mà áp dụng một chuỗi hữu hạn các thao tác nguyên thủy (primitive) vào bộ nhớ [8]. Hay nói cách khác một giao dịch là một thực thi của một chương trình người dùng.

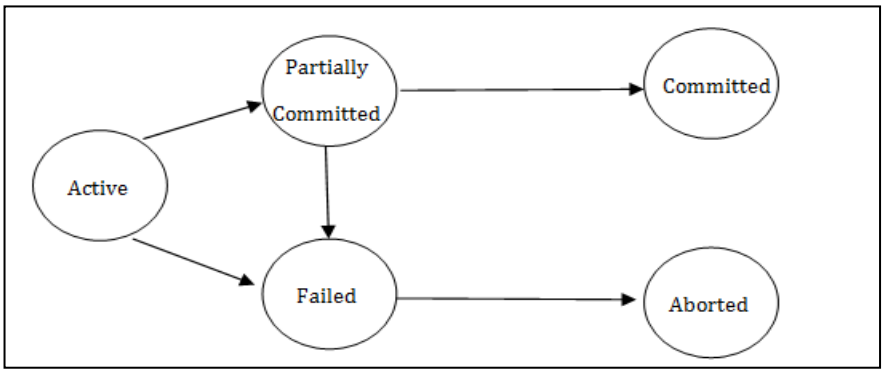
Các hệ quản trị cơ sở dữ liệu là một thể hiện điển hình cho các giao dịch trong các hệ thống phần mềm lớn.

Giao dịch có 4 tính chất, và được viết tắt ACID như sau :

a. Tính nguyên tử (Atomicity): Một giao dịch là một tập các thao tác, được thực hiện hoặc toàn bộ, hoặc không thực hiện gì cả.
b. Tính nhất quán (Consistency): Mỗi giao dịch được thực thi không được tranh chấp với các giao dịch khác.
c. Tính độc lập (Isolation): Người dùng có thể hiểu được một giao dịch mà không cần phải xem xét ảnh hưởng của các giao dịch tương tranh khác đang chạy.
d. Tính bền vững (Durability): Sau khi giao dịch đã hoàn toàn thành công, các trạng thái của nó được duy trì ngay cả khi hệ thống gặp sự cố.

Các trạng thái của một giao dịch bao gồm:

- Hoạt động (Active): Giao dịch giữ trạng thái này trong khi nó đang thực hiện.
- Đóng bộ phận (Partially Committed): Sau khi lệnh cuối cùng được thực hiện.
- Thất bại (Failed) : Khi giao dịch không thể tiếp tục thực hiện được
- Hủy bỏ (Aborted): Nếu giao dịch gặp trạng thái thất bại thì sau đó giao dịch cần phải khôi phục lại trạng thái của nó trước khi khởi động giao dịch. Hủy bỏ là kết quả cuối của quá trình đó.
- Committed: Sau khi giao dịch hoàn toàn thành công, nó sẽ đi vào trạng thái này.



Hình 2.2 Các trạng thái của giao dịch

2.2.2. Bộ nhớ giao dịch phần mềm (Software Transactional Memory- STM)

Từ năm 1986, ý tưởng cung cấp hỗ trợ phần cứng cho các giao dịch đã ra đời. Cho đến 1995 Nir Shavit và Dan Touitou đã mở rộng ý tưởng này cho bộ nhớ giao dịch phần mềm. Kể từ đó, nó đã trở thành trọng tâm của các các lý thuyết nghiên cứu chuyên sâu và các ứng dụng thực tế.

Trong khoa học máy tính, bộ nhớ phần mềm giao dịch (STM) là một cơ chế kiểm soát đồng thời tương tự như các giao dịch cơ sở dữ liệu cho việc kiểm soát quyền truy cập vào bộ nhớ dùng chung trong tính toán song song. Đây là một phương pháp thay thế cho cơ chế đồng bộ dựa trên khóa. STM là một chiến lược thực hiện trong phần mềm, chứ không phải là một thành phần phần cứng.

Một giao dịch trong bối cảnh này xảy ra khi một đoạn mã thực hiện một loạt các lần đọc và ghi vào bộ nhớ chia sẻ. Những lần đọc và ghi một cách logic xảy ra tại một thời điểm tức thì; Các trạng thái trung gian không thể nhìn thấy các giao dịch khác.

Ngoài các lợi ích về hiệu suất, STM làm đơn giản hóa sự hiểu biết về khái niệm của chương trình đa luồng và giúp cho các chương trình dễ bảo trì hơn bằng cách làm việc trong sự hòa hợp với các trừu tượng hóa mức cao đã có như các đối tượng và module. Lập trình dựa trên khóa có một số vấn đề mà thường xuyên phát sinh trong thực tế:

- Khóa đòi hỏi tư tưởng về các thao tác chồng chéo và thao tác bộ phận trong các phần tách biệt và dường như không liên quan của mã, một nhiệm vụ rất khó khăn và dễ bị lỗi.
- Khóa đòi hỏi lập trình viên phải áp dụng một chính sách khóa để ngăn chặn deadlock (khóa chết), livelock (khóa sống), và thất bại khác để kịp tiến độ. Các chính sách này thường được chính thức thi hành và có thể sai lầm, và khi những vấn đề phát sinh họ phải khó khăn để tái tạo và gỡ lỗi.
- Khóa có thể dẫn đến đảo ngược ưu tiên, một hiện tượng mà một luồng ưu tiên cao buộc phải chờ đợi cho một luồng ưu tiên thấp độc chiếm quyền truy cập vào tài nguyên mà nó cần.

Ngược lại, khái niệm về STM đơn giản hơn nhiều, bởi vì mỗi giao dịch có thể được xem trong sự cô lập như một tính toán đơn luồng. Deadlock và livelock được hoặc ngăn ngừa hoàn toàn hoặc bị xử lý bởi một trinh quản lý giao dịch bên ngoài; Các lập trình viên hầu như không cần phải lo lắng về nó. Đảo ngược ưu tiên vẫn có thể là một vấn đề, nhưng các giao dịch có mức ưu tiên cao có thể hủy bỏ xung đột với giao dịch ưu tiên thấp hơn mà vẫn chưa kết thúc.

Mặt khác, sự cần thiết phải hủy bỏ giao dịch thất bại cũng đặt những hạn chế về các hành vi giao dịch: Chúng không thể thực hiện bất kỳ thao tác nào mà không hoàn tất, bao gồm hầu hết các I/O. Những hạn chế như vậy thường được khắc phục trong thực tế bằng cách tạo bộ đệm mà hàng đợi hoạt động không thể đảo ngược và thực

hiện chúng ở một thời gian sau đó bên ngoài của bất kỳ giao dịch nào. Trong Haskell, hạn chế này được thi hành tại thời gian biên dịch bởi hệ thống kiểu.

CHƯƠNG 3. NGÔN NGỮ GIAO DỊCH

Trong chương này chúng ta sẽ nghiên cứu về cú pháp và ngữ nghĩa của một ngôn ngữ giao dịch được gọi là TM (Transactional Memory).

Một chương trình TM bắt đầu bằng lệnh onacid(n) (với n biểu diễn kích thước bộ nhớ được cấp phát cho log khi mở một giao dịch mới)và kết thúc bằng lệnh commit.

Dưới đây, chúng ta sẽ tìm hiểu cú pháp và ngữ nghĩa của TM. Trong đó, cú pháp nhằm mô tả các thành phần của một ngôn ngữ. Và các công thức thể hiện hoạt động của chương trình ở các mức cục bộ (bên trong một luồng), ở mức toàn cục (trong các luồng song song). Ngữ nghĩa thể hiện cách thức một chương trình được thực hiện như thế nào.

3.1. Cú pháp của TM [1]

Bảng 3.1 Bảng cú pháp của TM

$$
\begin{aligned}
P &::= 0 \quad | \quad P \parallel P \quad | \quad p(e) \\
e &::= \alpha \quad | \quad \mathbf{onacid}(n) \quad | \quad \mathbf{commit} \quad | \\
&\quad e_1; e_2 \quad | \quad e_1 + e_2 \quad | \quad \mathbf{spawn}(e)
\end{aligned}
$$

Trong dòng đầu tiên, một chương trình P có thể là rỗng kí hiệu 0 hoặc một luồng hoặc một số luồng song song. $p(e)$ là ký hiệu của một luồng với định danh là p và biểu thức thực thi e. Đây là cú pháp cho thực thi các luồng / tiến trình.

Với thành phần e, chúng ta giả sử ngôn ngữ có một tập các lệnh nguyên tử A, được giới hạn bởi **onacid(n)** và **commit** là lệnh bắt đầu và kết thúc một giao dịch. Tham số n trong onacid(n) biểu diễn số đơn vị bộ nhớ được cấp phát khi mở giao dịch mới. Chúng ta thấy trong thực tế n có thể được tổng hợp bởi trình biên dịch dựa trên kích thước của các biến dùng chung trong phạm vi của giao dịch. Điều đó có nghĩa là các lập trình viên không cần phải chú ý đến thông tin về kich thước này.

e_1; e_2 ký hiệu cho các lệnh tuần tự và e_1+ e_2 ký hiệu cho rẽ nhánh.

Câu lệnh cuối spawn(e) là lệnh tạo một luồng mới thực thi e.

3.2. Các ngữ nghĩa động

Ngữ nghĩa của TM được đưa ra bởi 2 mức tập hợp của các quy tắc hoạt động, tương ứng với các ngữ nghĩa cục bộ và toàn cục.

Môi trường thực thi (toàn cục) được cấu trúc như là một tập của các môi trường cục bộ. Mỗi môi trường cục bộ là một chuỗi các log cùng với kích thước của nó.

Môi trường cục bộ và môi trường toàn cục được định nghĩa như sau:

3.2.1. Ngữ nghĩa cục bộ

Các ngữ nghĩa cục bộ liên quan tới việc đánh giá một luồng đơn và các giao dịch cục bộ ở dạng $E, e \rightarrow E', e'$. E và E' ở đây là các môi trường cục bộ, trong khi e và e' là các biểu thức sẽ được thực thi bởi luồng, có nghĩa là một biểu thức e được đánh giá

trong môi trường cục bộ E thì nó sẽ được chuyển thành một biểu thức e', tương ứng với nó môi trường cục bộ E sẽ chuyển thành môi trường cục bộ E'.

Định nghĩa 1(Local environment – Môi trường cục bộ) *Một môi trường cục bộ E là một chuỗi tuần tự của các log và kích thước của nó:* $l_1{:}n_1;\ldots l_k{:}n_k$. *Môi trường không có phần tử nào được gọi là môi trường rỗng và ký hiêu bởi* ε *[1].*

Chúng ta ký hiệu |E| là kích thước của E, các cặp số l:log. |E| thể hiện mức sâu lồng nhau của các giao dịch. l_1 là giao dịch đầu tiên (ngoài nhất) và l_k là giao dịch trong nhất. Do vậy, commit sẽ được thực hiện từ phải sang trái.

Dãy *l: log* của giao tác được sử dụng để biểu diễn cấu trúc lồng.

$|E|$ trong phân tích của chúng ta là lượng bộ nhớ hiện tại xác định dành cho luồng.

log_i lưu những thay đổi từ bộ nhớ cục bộ của một luồng đối với giao dịch l_i và được coi như bản sao cục bộ. Nó sẽ được giải phóng ngay khi giao dịch có nhãn l_i thực thi xong

3.2.2. Ngữ nghĩa toàn cục

Ở mức toàn cục, ngữ nghĩa sẽ có dạng: $\Gamma, P \Rightarrow \Gamma', P'$ hoặc $\Gamma, P \Rightarrow error$ trong đó: Γ là môi trường toàn cục và P là tập các tiến trình có dạng (e) . Môi trường toàn cục là một tập các môi trường cục bộ mà không rỗng.

Định nghĩa 2 ***(Global environment – Môi trường toàn cục)***

Một môi trường toàn cục Γ *là một tập các luồng và môi trường cục bộ của nó, được viết là:* $\Gamma = \{ p_1 : E_1; \ldots; p_k : E_k\}$ *, với* p_i *là tên luồng và* E_i *là môi trường cục bộ của luồng [1].*

Chúng ta ký hiệu |Γ| cho kích thước của Γ, và $|\Gamma| = \sum_{i=0}^{k} |E_i|$.

Với 1 môi trường toàn cục Γ và tập các luồng P, chúng ta gọi cặp Γ và P là 1 trạng thái. Chúng ta có một trạng thái lỗi error cho các trạng thái mắc kẹt (stuck) trạng thái mà không có quy tắc giao dịch nào được áp dụng. Các ngữ nghĩa động được định nghĩa bởi các quy tắc giao dịch giữa các hình thức trạng thái $\Gamma, P \Rightarrow \Gamma', P'$ hoặc $\Gamma, P \Rightarrow error$ trong bảng dưới đây

Bảng 3.2. Bảng ngữ nghĩa động của TM

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

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

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

$$\frac{i = 1, 2}{\Gamma, P \parallel p(e_1 + e_2) \Rightarrow \Gamma, P \parallel p(e_i)} \text{ S-COND}$$

$$\frac{}{\Gamma, P \parallel p(\alpha; e) \Rightarrow \Gamma, P \parallel p(e)} \text{ S-SKIP}$$

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

Ý nghĩa các quy tắc trong bảng 3.2 :

Trong quy tắc S-SPAWN , hàm spawn(p, p', Γ) cho phép thêm vào Γ một phần tử mới với tên luồng p' và một môi trường cục bộ được sao chép từ môi trường cục bộ của p.

Trong quy tắc S-TRANS, hàm start(l:n,p, Γ) tạo ra thêm một log với nhãn l và kích thước n đơn vị bộ nhớ ở cuối môi trường cục bộ,

Quy tắc S-SPAWN chỉ ra một luồng mới được tạo ra bằng lệnh spawn. Câu lệnh **spawn(e_1)** tạo một luồng p' thực thi e_1 song song với luồng cha p và thay đổi môi trường từ Γ sang Γ'.

Quy tắc S-TRANS dành cho những trường hợp mà luồng p tạo một giao dịch mới với lệnh **onacid.** Một giao dịch mới với nhãn l được tạo ra, và thay đổi môi trường từ Γ sang Γ'.

Quy tắc S- COMM dành cho việc đóng giao dịch. Trong quy tắc này $\coprod_1^k p_i\,(E_i)$ được hiểu là $p_1(e_1) \parallel \cdots \parallel p_k(e_k)$. Nếu giao dịch của luồng p là l thì tất cả các luồng trong giao dịch la phải join commit khi giao dịch l đóng.

Quy tắc S-COND là để chọn lựa một trong những nhánh e_1 hoặc e_2 để tiếp tục.

Quy tắc S-SKIP là để cho việc tính toán các lệnh khác của ngôn ngữ, các lệnh mà ta giả sử là nó không can thiệp vào các ngữ nghĩa lồng và đa luồng, chúng ta có thể bỏ qua chúng.

Quy tắc S-ERROR-C và S-ERROR-O được sử dụng trong các trường hợp có nhiều điểm không khớp trong việc bắt đầu và kết thúc giao dịch.

CHƯƠNG 4. HỆ THỐNG KIỂU CHO CHƯƠNG TRÌNH GIAO DỊCH

Mục đích chính của hệ thống kiểu là để xác định lượng bộ nhớ lớn nhất mà một chương trình TFJ có thể yêu cầu.

Kiểu của một thành phần (term) trong hệ thống được tính toán từ chuỗi các số có dấu, là một biểu diễn trừu tượng của thành phần hành vi giao dịch liên quan tới bộ nhớ log.

4.1. Các kiểu

Theo [1], các kiểu của chúng ta là các chuỗi giới hạn trên tập được gọi là chuỗi số có dấu. Một số có dấu là một cặp của các dấu và các số tự nhiên không âm N^+. Chúng ta sử dụng 4 dấu {+, −, ¬, #} cho việc ký hiệu tương ứng mở, đóng, joint commit, và bộ nhớ tích lũy lớn nhất mà các log sử dụng.

Tập tất cả các chuỗi số có dấu được ký hiệu là ${}^{T}N$.

Do đó ${}^{T}N$= {${}^{+}n, {}^{-}n, {}^{\neg}n, {}^{\#}n$}

Ý nghĩa của những số có dấu này được mô tả như sau :

- Số có dấu ${}^{+}n$ chỉ ra rằng mở giao dịch có kích thước của log là n đơn vị bộ nhớ. Lưu ý là ngữ nghĩa này khác so với các nghiên cứu [5,6], ở đó nó ký hiệu cho n lệnh mở giao dịch onacid liên tiếp.
- Số có dấu ${}^{-}n$ có nghĩa là có n lệnh commit liên tiếp.
- Số có dấu ${}^{\neg}n$ có nghĩa là n luồng yêu cầu sự đồng bộ ở thời điểm Join commit.
- Số có dấu ${}^{\#}n$ chỉ ra số đơn vị bộ nhớ lớn nhất hiện tại mà thành phần sử dụng là n.

Ví dụ 4.1 :

onacid (5) có kiểu ${}^{+}5$;

commit có kiểu ${}^{-}1$;

Chuỗi onacid(5); commit có kiểu là ${}^{+}5\ {}^{-}1$;

Chuỗi onacid(1); onacid(3); commit; commit có kiểu là ${}^{+}1{}^{+}3\ {}^{-}1{}^{-}1$, nó tương đương với ${}^{+}1\ {}^{\#}3{}^{-}1$ hoặc ${}^{\#}4$; ${}^{\#}4{}^{\#}5$ và ${}^{\#}5{}^{\#}2$ đều chuyển được về kiểu tương đương là ${}^{\#}5$ vì chúng phản ánh số đơn vị bộ nhớ lớn nhất được sử dụng là 5.

Tiếp theo, cho s trên tập ${}^{T}N$, ${}^{T}\overline{N}$ là tập tất cả các chuỗi số có dấu, S trên tập ${}^{T}\overline{N}$, và cho m, n , l trên N.

Một chuỗi rỗng được ký hiệu ∅.

Cho một chuỗi S, chúng ta ký hiệu |S| là độ dài của S, và S(i) là phần tử thứ i của S.

Với một số có dấu s, chúng ta ký hiệu tag(s) là dấu của s, và |s| là số tự nhiên của s (hay nói cách khác s=$^{tag(s)}|s|$).

Với một chuỗi S ∈ ${}^{T}\overline{N}$, chúng ta viết tag(S) cho chuỗi các dấu của các phần tử của S và $\{S\}$ cho tập các dấu xuất hiện trong S.

Lưu ý : tag (s_1...s_k) = tag(s_1)…tag(s_k)

Để cho đơn giản, chúng ta cũng viêt tag(s) ∈ S thay vì viết tag(s) ∈ $\{S\}$

Tập ${}^{T}\overline{N}$ có thể chia thành các lớp tương đương mà các phần tử trong cùng một lớp biểu diễn cùng hành vi giao dịch, và cho mỗi lớp chúng ta sử dụng chuỗi gọn nhất là đại diện cho lớp và gọi chúng là phần tử chuẩn tắc.

Định nghĩa 5 (Chuỗi chuẩn tắc): *Một chuỗi gọi là chuẩn tắc nếu tag(S) không chứa '−−', '##', '+ −', '+ ¬', '+ #¬' hoặc '+# −' và $|S(i)|>0$ với mọi i [1].*

Theo trực quan ta thấy chúng ta có thể rút gọn một chuỗi S mà không cần thay đổi sự biểu diễn của nó. Hàm seq dưới đây rút gọn một chuỗi trong ${}^{T}\overline{N}$ thành một chuỗi chuẩn tắc. Ta thấy là *'+ −'* không xuất hiện ở bên trái nhưng chúng ta có thể chèn ${}^{\#}0$ để áp dụng hàm. Hai mẫu cuối *'+ ¬' và '+ #¬'* sẽ được giải quyết bởi hàm j_c trong định nghĩa 10.

Ví dụ 4.2: ${}^{+}5\ {}^{-}1$ là chuỗi không chuẩn tắc nhưng ${}^{+}5\ {}^{+}1$ lại là chuỗi chuẩn tắc.

Định nghĩa 6 (Rút gọn):

Hàm rút gọn seq được định nghĩa đệ quy như sau:

seq(S) = S khi S là chuẩn tắc

seq($S\ {}^{\#}m\ {}^{\#}n\ S'$) = seq($S\ {}^{\#}\ max(m,n)S'$)

seq($S\ {}^{-}m\ {}^{-}n\ S'$) = seq($S\ {}^{-}\ (m+n)S'$)

seq($S\ {}^{+}k\ {}^{\#}l\ {}^{-}n\ S'$) = seq($S\ {}^{\#}(l+k)\ {}^{-}\ (n-1)\ S'$) [1]

Trong định nghĩa này, dòng 2 và 3 dành cho trình bày rút gọn. Dòng cuối là cho các commit cục bộ, các commit mà không đồng bộ với các luồng khác.

Ví dụ 4.3: Chuỗi *${}^{\#}3\ {}^{\#}2$ rút gọn sẽ được chuỗi ${}^{\#}5$; ${}^{+}3\ {}^{\#}2\ {}^{-}2$ rút gọn được chuỗi ${}^{\#}5\ {}^{-}1$*

Như ví dụ trong hình 1.2, các luồng được đồng bộ bởi các join commit (các hình chữ nhật nét đứt). Vì vậy những joint commit này chia một luồng thành các phân đoạn (segment) và chỉ một số phân đoạn có thể chạy song song. Ví dụ, trong khi chạy chương trình trong ví dụ ở hình 1.1, onacid(5) trên dòng 5 không thể chạy song song với onacid(6) trên dòng 7.

Với kiểu được đưa ra cho thành phần e, các phân đoạn có thể được định kiểu bằng cách kiểm tra kiểu của e trong spawn (e) để mở rộng – hoặc ¬. Ví dụ trong spawn (e_1);e_2; Nếu chuỗi chuẩn tắc của e_1 có – hoặc¬, thì luồng e_1 phải được đồng bộ với luồng cha của nó là luồng e_2 . Hàm gộp (merge) trong định nghĩa 8 được sử dụng trong tình huống này, nhưng để xác định nó cần một số hàm bổ trợ :

Với S∈ ${}^{T}\overline{N}$ và với một dấu sig ∈{+, −, ¬, #}, chúng ta đưa ra hàm first(S,sig) trả về giá trị nhỏ nhất chỉ số i mà tag(S(i))= sig. Nếu không có phần tử như vậy tồn tại thì hàm trả về 0. Một commit có thể là commit cục bộ hoặc hoàn toàn là join commit. Đầu tiên, chúng ta giả sử tất cả các commit là các commit cục bộ. Khi chúng ta tìm ra không có lệnh bắt đầu giao dịch (onacid) cục bộ nào để ghép với một commit cục bộ, thì commit phải là một join commit.

Hàm sau thực thi công việc đó và chuyển thành một chuỗi chuẩn tắc mà không có phần tử + nào, được gọi là chuỗi cộng.

Định nghĩa 7 (Cộng):

Cho S= $s_1...s_k$ là một chuỗi chuẩn tắc mà + không nằm trong {S } và giả sử i=first(S, −) . Thì hàm cộng join(S) định nghĩa đệ quy thay thế − trong S bởi ¬ như sau:

join (S)= S *nếu i=0;*

join(S) = $s_1...s_{i-1}$ $\neg_1$ join ($\neg$ ($|s_i| - 1$) s_{i+1} ...s_k) *ngược lại [1]*

Ví dụ 4.4: Chuỗi cộng của $^{\#}5^{\neg}$ 1 là $^{\#}5^{\neg}$ 1; Chuỗi cộng của $^{\#}2^{\neg}$ 1$^{\#}3^{\neg}$ 1 là $^{\#}2^{\neg}$ 1$^{\#}3^{\neg}$ 1

Ta thấy trong định nghĩa 5 chuỗi chuẩn tắc chỉ chứa thành phần # xen kẽ với – hoặc ¬. Sau khi áp dụng hàm join, chúng ta nhận chuỗi cộng. Chuỗi cộng này chỉ chứa thành phần # xen kẽ với ¬

Một chuỗi cộng được sử dụng để định kiểu một thành phần bên trong một spawn hoặc một thành phần trong luồng chính.

Chuỗi cộng được gộp cùng nhau trong định nghĩa sau đây :

Định nghĩa 8 (Gộp):

Cho S_1 và S_2 là các chuỗi cộng mà số các thành phần ¬ trong S_1 và S_2 là như nhau (có thể là 0). Hàm merge được định nghĩa đệ quy như sau :

Merge ($^{\#}$ m_1, $^{\#}$ m_2)= $^{\#}$ (m_1 + m_2)

Merge ($^{\#}$ m_1 $\neg$ n_1 S_1' , $^{\#}$ m_2 $\neg$ n_2 S_2')= $^{\#}$ (m_1 + m_2) $\neg$(n_1 + n_2)merge(S_1', S_2') [1]

Định nghĩa này là thiết lập đúng (well- formed) vì S_1, S_2 là các chuỗi cộng vì vậy chúng chỉ có thành phần # và ¬. Thêm vào đó, theo giả sử trong định nghĩa số các thành phần # là như nhau. Do đó chúng ta có thể chèn thêm $^{\#}$ 0 để làm cho 2 chuỗi phù hợp với các mẫu đã xác định.

Chúng ta thấy rằng hàm gộp merge được sử dụng để định kiểu cho các thành phần spawn (e_1); e_2, khi ta tính kiểu cho e_1, sau đó áp dụng hàm join để thu được một chuỗi cộng – kiểu của spawn (e_1). Sau đó, ta cần tính toán kết hợp một chuỗi cộng từ e_2 để gộp với chuỗi cộng của kiểu spawn (e_1).

Để định kiểu các thành phần có dạng e_1+e_2, chúng ta có thêm một số hàm. Với những thành phần này, yêu cầu các hành vi giao dịch bên ngoài của e_1 và e_2 là như nhau.Chẳng hạn, khi loại bỏ tất cả các phần tử mà có dấu # từ chúng, các chuỗi còn lại là giống hệt nhau. Cho S_1 và S_2 là 2 chuỗi như vậy. Thì chúng có thể luôn được viết là

S_i= $^{\#}$$m_i$ *nS_i' , i= 1,2, * = {+, −, ¬} trong đó S_1' và S_2' lần lượt có cùng các hành vi giao dịch.

Với điều kiện trên cho S_1 và S_2, chúng ta xác định toán tử chọn (Choice) như sau:

Định nghĩa 9 (Chọn) :

Cho S_1 và S_2 là 2 chuỗi mà nếu chúng ta loại bỏ thành phần # từ chúng, thì hai chuỗi còn lại là giống hệt nhau. Hàm alt được định nghĩa đệ quy như sau :

alt ($^{\#}$$m_1$, $^{\#}$$m_2$)= $^{\#}$max(m_1,m_2)

alt ($^{\#}$ m_1 * n_1 S_1' , $^{\#}$ m_2 * n_2 S_2')= $^{\#}$ max(m_1, m_2) *n alt (S_1', S_2') [1]

4.2. Các quy tắc kiểu

Ngôn ngữ của các kiểu *T* được xác định bởi cú pháp sau

$$T = S \mid S^{\rho} \mid \quad [1]$$

Loại thứ hai của kiểu S^{ρ} được sử dụng cho thành phần **spawn(e)** khi nó cần đồng bộ với luồng cha nếu có bất kỳ join commit nào. Việc xử lý 2 trường hợp là toàn khác nhau, do vậy chúng ta ký hiệu *kind(T)* là kiểu của *T*, có thể là rỗng (thông thường) hoặc ρ phụ thuộc vào trường hợp của *T*.

Kiểu của môi trường mã hóa ngữ cảnh giao dịch cho các thành phần đang được định kiểu, các đánh giá kiểu có dạng :

$$n \vdash e : T$$

khi n∈ *N* là kiểu của môi trường. Khi n âm, điều đó có nghĩa là e sử dụng n đơn vị bộ nhớ cho các log của nó khi thực thi e. Khi n âm, có nghĩa là e có thể giải phóng n đơn vị bộ nhớ của một số log.

Các quy tắc kiểu được biểu diễn trong bảng 4.1 dưới đây:

Bảng 4.1 Các quy tắc kiểu

$$\frac{}{E \vdash onacid(n) : {}^{+}n} \quad T-ONACID$$

$$\frac{n \in N^{+}}{n \vdash commit : {}^{-}1} \quad T-COMMIT$$

$$\frac{n \vdash e : S}{n \vdash spawn(e) : join(S)^{p}} \quad T-SPAWN$$

$$\frac{n \vdash e : S}{n \vdash e : join(S)^{p}} \quad T-PREP$$

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

$$\frac{n_1 \vdash e_1 : S_1 \quad n_2 \vdash e_2 : S_2^{\rho} \;\; S = jc(S_1\;, S_2\;)}{n_1 + n_2 \vdash e_1 ; e_2 \;\; : S} \quad T-JC$$

$$\frac{n \vdash e_i : S_i^{\rho} \;\; i = 1,2 \;\; S = merge(S_1\;, S_2\;)}{n \vdash e_1 ; e_2 \;\; : S^{\rho}} \quad T-MERGE$$

$$\frac{n \vdash e_i : S_i^{\rho} \;\; i = 1,2 \;\; kind(T_1) = kind(T_2)\; T_i \;=\; S_i^{kind(T_i)}}{n \vdash e_1 + e_2 \;\; : alt(S_1, S_2)^{kind(S_1)}} \quad T-COND$$

Trong bảng trên, quy tắc T-ONACID cho phép chuyển onacid(n) thành +n.

Quy tắc T- COMMIT cho phép chuyển commit thành -1;

Quy tắc T- SPAWN chuyển S từ chuỗi cộng và đánh dấu các kiểu mới bởi ρ do đó chúng ta có thể gộp nó với chuỗi của luồng cha trong hàm T- MERGE.

Quy tắc T- PREP cho phép chúng ta tìm kiểu phù hợp cho e trong T- MERGE.

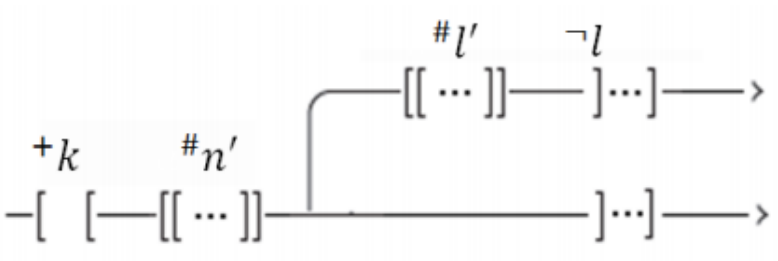
Quy tắc T-JC giải quyết join commit giữa các luồng chạy song song và sử dụng tời công thức jc trình bày ở định nghĩa 10. Trong đó, phần tử + cuối cùng trong S_1, gọi là ^{+}n sẽ được kết hợp với phần tử ¬ đầu tiên trong S_2, gọi là $^{\neg}l$ (Hình 4.1). Nhưng sau ^{+}n, có thể có thành phần #, gọi là $^{\#}n'$, do vậy cận của các đơn vị bộ nhớ cục bộ được sử dụng bởi thành phần có kiểu $^{+}n^{\#}n'$ là n+ n'. Trước $^{\neg}l$ có thể có $^{\#}l'$, vì vậy khi thực hiện join commit các thành phần có kiểu $^{\neg}l$ với giao dịch bắt đầu của nó có kiểu ^{+}n, kiểu của các phân đoạn sẽ là $l'+l*n$. Sau khi kết hợp ^{+}n từ S_1 với $^{\neg}l$ từ S_2 chúng ta có thể rút gọn các chuỗi mới và lặp lại các join commit của jc. Do đó hàm jc được xác định như sau :

***Định nghĩa 10 (Join commit)**:*

Hàm jc được xác định đệ quy như sau :

$jc(S_1' +n\ ^{\#}n', ^{\#}l'^{\neg}l\ S_2') = jc(seq(\ S_1'\ ^{\#}(n+\ n'), seq(^{\#}(n+n'), seq(^{\#}(\ l' + l*n)\ S_2'\ ))$ *nếu l>0*

$jc(^{\#}n, ^{\#}l'\ S_2') = seq(^{\#}\ max(n', l')\ S_2')$ *ngược lại [1]*



Hình 4.1 Các luồng song song Joincommit

Định nghĩa 11 (Chương trình well- typed): *Một thành phần e được gọi là well- typed nếu tồn tại một dẫn xuất cho e mà 0 ⊢ e :# n với một số n* [1].

Do các kiểu phản ánh hành vi của một thành phần, vì vậy kiểu của một chương trình định kiểu tốt (well- typed) chứa chỉ một chuỗi $^{\#}n$ ở đó n là số đơn vị bộ nhớ lớn nhất được sử dụng khi thực hiện chương trình.

Dưới đây chúng ta sẽ sử dụng các lý thuyết ở trên để giải bài toán tính cận trên cho chương trình trong hình 1.3.

Để thuận lợi cho việc tính toán, trước hết ta chia nhỏ chương trình thành các biểu thức

e_1 = onacid (1); onacid (2);

e_2 = spawn (onacid (4); commit; commit; commit; commit)

e_3 =onacid (3);

e_4 = spawn (onacid (5); commit; commit; commit; commit)

e_5 = commit;onacid (6);commit;commit;onacid (7);commit;commit;

Trước hết ta tính kiểu cho e_4;

Sử dụng các quy tắc T-ONACID, T-COMMIT và T-SEQ ta có kiểu của

Onacid (5);commit;commit;commit;commit : $^{\#}5^{\neg}1^{\neg}1^{\neg}1$

Áp dụ ng T-SPAWN, ta được

$6\vdash e_4 :(^{\#}5^{\neg}1^{\neg}1^{\neg}1)^p$

Tiếp theo ta tính e_5 ; cũng áp dụng các hàm T-ONACID, T-COMMIT và T-SEQ ta có

$6\vdash e_5 : {}^{\neg}1^{\#}6^{\neg}1^{\#}7^{\neg}1$

Sử dụng T-PREP, chúng ta có kiểu phù hợp với S_4 , tiếp tục áp dụng hàm T- MERGE ta được

$6 \vdash e_{45} : (^{\#}5^{\neg}2^{\#}6^{\neg}2^{\#}7^{\neg}2)^p$

Với $-3 \vdash e_3 :^{+}3$, chúng ta áp dụng T-JC ta được

$3\vdash e_{345}: {}^{\#}11^{\neg}2^{\#}7^{\neg}2$

vì $jc(^{+}3, {}^{\#}5^{\neg}2^{\#}6^{\neg}2^{\#}7^{\neg}2) = jc(seq(^{\#}3), seq(^{\#} (5+3*2) {}^{\#}6^{\neg}2^{\#}7^{\neg}2))$

$=jc(^{\#}3, {}^{\#}11^{\neg}2^{\#}7^{\neg}2)= {}^{\#}11^{\neg}2^{\#}7^{\neg}2;$

Tương tự ta tính được kiểu của e_2 : $3\vdash e_3 : (^{\#}4^{\neg}1^{\neg}1)^p$. Kiểu của e_3 phù hợp với e_{345} , do vậy ta lại sử dụng hàm T-MERGE và được kiểu của e_{2345}

$3\vdash e_{2345} : {}^{\#}15^{\neg}3^{\#}7^{\neg}3$

Áp dụng T-JC cho e_1 và e_2 ta đươc

$0 \vdash e_{12345} : {}^{\#}24$

vì $jc(^{+}1^{+}2, {}^{\#}15^{\neg}3^{\#}7^{\neg}3)= jc(seq(^{+}1^{\#}2),seq(^{\#}21, {}^{\#}7^{\neg}3)= jc(^{+}1^{\#}2, {}^{\#}21^{\neg}3)= jc(seq(^{\#}3), seq(^{\#}24)) = {}^{\#}24$

Vậy chương trình này là well-typed và lượng bộ nhớ lớn nhất mà chương trình cần là 24 đơn vị bộ nhớ.

CHƯƠNG 5. XÂY DỰNG CÔNG CỤ VÀ THỰC NGHIỆM

5.1. Giới thiệu ngôn ngữ lập trình/ nền tảng

Ngôn ngữ lập trình được sử dụng để xây dựng công cụ trong luận văn này là ngôn ngữ C#, trên nền .NET. Đây một ngôn ngữ lập trình ứng dụng, ngôn ngữ biên dịch, ngôn ngữ đa năng được phát triển bởi Microsoft. Ngôn ngữ C# là một ngôn ngữ được phát triển từ C, C++ và Java, nhưng nó được tạo từ nền tảng phát triển hơn. C# được thêm vào những đặc tính mới để giúp cho nó uyển chuyển và dễ sử dụng hơn. Nhiều đặc tính trong ngôn ngữ C# khá giống với ngôn ngữ Java. Cụ thể, C# có những đặc tính cơ bản sau :

- Đơn giản, dễ học : Chỉ có khoảng hơn 80 từ khóa và mười mấy kiểu dữ liệu được dựng sẵn
- Gần gũi với các ngôn ngữ lập trình thông dụng như C, C++, Java
- Xây dựng dựa trên nền tảng của những ngôn ngữ lập trình mạnh nên thừa hưởng được các ưu điểm của các ngôn ngữ đó.
- Hướng đối tượng
- Mạnh mẽ và mềm dẻo
- Cung cấp những đặc tính hướng thành phần như property, event…
- C# có bộ Garbage Collector sẽ tự động thu gom vùng nhớ khi không sử dụng nữa.
- Hỗ trợ khái niệm giao diện

Song hành với ngôn ngữ C# là nền tảng .NET Framework. Đây là một nền tảng lập trình và cũng là một nền tảng thực thi ứng dụng chủ yếu trên hệ điều Windows. Nó cũng được phát triển bởi Microsoft và bao gồm 2 thành phần chính:

- CLR: Các chương trình được viết trên nền.NET sẽ được triển khai trong môi trường được gọi là CLR (Common Language Runtime). Môi trường phần mềm này đóng vai trò là một máy ảo cung cấp các dịch vụ như bảo mật, quản lý ngoại lệ hay bộ nhớ.
- Thư viện các lớp: .NET framework gồm nhiều lớp thư viện, những thư viện này sẽ hỗ trợ các lập trình viên trong việc xây dựng giao diện, kết nối cơ sở dữ liệu, giao tiếp mạng…

5.2. Xây dựng công cụ và thực nghiệm

Trong phần này của luận văn, dựa vào các quy tắc kiểu đã được đề cập ở chương 4, tôi sẽ xây dựng thuật toán tính kiểu đề giải quyết bài toán đã nêu trong chương 2. Bước tiếp theo, dựa trên thuật toán có được viết một chương trình bằng ngôn ngữ C# để tính toán cận trên tài nguyên cho một chương trình giao dịch

Trước tiên để xây dựng thuật toán ta quy ước một số ký hiệu được sử dụng để biểu diễn một chuỗi số có dấu khi lập trình như sau:

{+n, −n, #n,! n } (trong đó *n* là số tự nhiên) tương ứng với các thành phần dạng

{ ^{+}n, ^{-}n, $^{\#}n$, $^{\neg}n$ } đã đề cập trong phần 4.1

Chúng ta sẽ chuyển đổi mã của chương trình giao dịch thành một chuỗi gồm các dấu, các số và các dấu đóng mở ngoặc tương ứng với các lệnh sinh một luồng mới và đóng luồng. Cụ thể:

onacid (n): Tương ứng với chuỗi "+n";

commit: Tương ứng với "-1";

spawn: Tương ứng với "( )" – Khởi tạo một luồng

Qua bước chuyển đổi các thành phần trên, ta sẽ kết xuất được một xâu là đầu vào để thực hiện tính toán.

Chương trình được xây dựng bao gồm các phương thức cơ bản sau:

- CalculResourses (string Input): Phương thức thực hiện việc tính cận trên bộ nhớ log của chương trình sử dụng giao dịch. Phương thức này sẽ gọi tới các phương thức Seq, Joint, Merge, JointCommit được trình bày ở bên dưới.
- Seq (string InputString): Phương thức rút gọn một chuỗi gồm dấu và số.
- Joint (string InputString): Phương thức chuyển các dấu "─ " trong chuỗi đã được rút gọn (sau khi thực hiện hàm Seq ở trên) về dấu "⌐".
- Merge (string s1, string s2): Phương thức gộp 2 chuỗi chuẩn tắc.
- JoinCommit(string s1, string s2): Phương thức jointcommit 2 chuỗi chuẩn tắc

Sau đây chúng ta sẽ đi xây dựng thuật toán cụ thể cho các phương thức đã nêu ở trên:

5.2.1. Thuật toán rút gọn (chính tắc hóa) một chuỗi

5.2.1.1. Mô tả thuật toán:

Thuật toán dựa trên các quy tắc được trinh bày trong định nghĩa 6, mục 4.1.

Đầu vào: Một chuỗi số có dấu chưa được chính tắc, cho trước các quy tắc rút gọn như sau:

seq(S) = S khi S là một chuỗi chính tắc;

seq($S^{\#}m^{\#}nS'$) = seq($S^{\#}max(m,n)S'$)

seq($S^{-}m^{-}nS'$) = seq($S^{-}(m^{+}n)S'$)

seq($S^{+}m^{\#}l^{-}nS'$) = seq($S^{\#}$ (m+ 1) $^{-}$ (n - 1)S')

Đầu ra: Chuỗi S đã được rút gọn hay chính tắc

Thuật toán :

Bước 1: Duyệt lần lượt các phần tử theo chiều từ trái qua phải, sau đó kiểm tra xem 2 dấu liền nhau trong chuỗi có thuộc danh sách --, ##, +-, +#-…(như định nghĩa 5 hay không). Nếu chuỗi không chứa các dấu liền nhau như trên thì chuỗi đã chính tắc,hay đã được rút gọn.

Bước 2: Gọi R là mẫu thỏa mãn điều kiện về các cặp dấu như trên, khi tìm thấy R ta sẽ thay thế R bằng mẫu S theo các quy tắc ở trên

5.2.2.2. Cài đặt và kiểm chứng

```
1.  public static string Seq(string InputString)
2.  {
3.    string Spattern = @"-\d+-\d+|#\d+#\d+|\+\d+#\d+-\d+|\+\d+-\d+";
4.    while (Regex.IsMatch(InputString, Spattern))
5.    {
6.      string pattern = @"-\d+-\d+";
7.      while (Regex.IsMatch(InputString, pattern))
8.      {
9.        foreach (Match m in Regex.Matches(InputString, pattern))
10.       {
11.        string[] operation_temp = Regex.Split(m.Value, @"\D");
12. operation_temp = operation_temp.Where(s => !String.IsNullOrEmpty(s)).ToArray();
13.        int sum = 0;
14.        foreach (string temp in operation_temp)
15.            sum = sum + Convert.ToInt32(temp);
16.       InputString = Regex.Replace(InputString, m.Value, "-" + sum.ToString());
17.        }
18.      }
19.         //   find and replace ##
20.      string pattern1 = @"#\d+#\d+";
21.      while (Regex.IsMatch(InputString, pattern1))
22.      {
23.        foreach (Match m1 in Regex.Matches(InputString, pattern1))
24.        {
25.          string[] operation_temp = Regex.Split(m1.Value, @"\D");
26. operation_temp = operation_temp.Where(s => !String.IsNullOrEmpty(s)).ToArray();
27.          int max = Convert.ToInt32(operation_temp[0].ToString());
28.          foreach (string temp in operation_temp)
29.            if (max < Convert.ToInt32(temp))
30.              max = Convert.ToInt32(temp);
31.     InputString = Regex.Replace(InputString, m1.Value, "#" + max.ToString());
32.            }
33.          }
34.        //-find and replace +#-
35.      string pattern2 = @"\+\d+#\d+-\d+";
36.      while (Regex.IsMatch(InputString, pattern2))
37.      {
38.        foreach (Match m2 in Regex.Matches(InputString, pattern2))
39.        {
40.          string[] operation_temp = Regex.Split(m2.Value, @"\D");
41. operation_temp = operation_temp.Where(s => !String.IsNullOrEmpty(s)).ToArray();
42.          int heso1 = (Convert.ToInt32(operation_temp[0]) + Convert.ToInt32(operation_temp[1]));
43.          int heso2 = (Convert.ToInt32(operation_temp[8]   ) - 1);
44.          if (heso2 != 0)
45. InputString = InputString.Replace(m2.Value.ToString(),"#" + heso1.ToString()+ "-" + heso2.ToString());
46.          else InputString = InputString.Replace(m2.Value.ToString(), "#" + heso1.ToString());
47.          }
48.        }
```

```
49.
50.          //find and replace +-
51.
52.        string pattern3 = @"\+\d+-\d+";
53.        while (Regex.IsMatch(InputString, pattern3))
54.        {
55.          foreach (Match m3 in Regex.Matches(InputString, pattern3))
56.          {
57.            string[] operation_temp = Regex.Split(m3.Value, @"\D");
58.  operation_temp = operation_temp.Where(s => !String.IsNullOrEmpty(s)).ToArray();
59.            int heso1 = Convert.ToInt32(operation_temp[0]);
60.            int heso2 = (Convert.ToInt32(operation_temp[1]) - 1);
61.            if (heso2 != 0)
62.  InputString = InputString.Replace(m3.Value.ToString(), "#" + heso1.ToString() + "-" + heso2.ToString());
63.            else InputString = InputString.Replace(m3.Value.ToString(), "#" + heso1.ToString());
64.          }
65.        }
66.            }
67.        return InputString;
68.
69.         }
```

Để kiểm tra tính đúng đắn của thuật toán, ta kiểm tra với các đầu vào khác nhau và được kết quả cho ở bảng sau:

Bảng 5.1 Bảng kết quả kiểm thử hàm rút gọn

Lượt	Dữ liệu đầu vào	Kết quả
1	#1#2-1	#3-1
2	-1-1	-2
3	+3#5-1	#8
4	+1#2-1#4-2	#4-2
5	+1-1-1-1	#1-2

5.2.2. Thuật toán Cộng (Joint)

5.2.2.1. Mô tả thuật toán

Thuật toán dựa trên các quy tắc được trinh bày trong định nghĩa 7, mục 4.1

Đầu vào: Chuỗi có dấu chính tắc S không chứa dấu +, và i là vị trí chứa dấu – đầu tiên trong chuỗi S, i # 0. Hàm join(S) được định nghĩa đệ quy như sau:

$join(S) = S$ nếu i=0;

$join(S) = s_1 \ldots s_{i-1}\ \neg 1\ join\ (-(|s_i| - 1)\ s_{i+1} \ldots s_k)$ nếu i≠0;

Đầu ra: Chuỗi S chỉ chứa dấu # và ⌐ (trong lập trình dùng ký hiệu "!" thay cho ⌐)

Thuật toán:

Bước 1: Duyệt chuỗi từ trái qua phải và kiểm tra tất cả các mẫu bắt đầu bằng dấu ─

Bước 2:

- Nếu tìm được , ta sẽ thay thế tất cả các thành phần có dạng –m thành ⌐1─ (m-1)
- Lặp lại quá trình cho tới khi toàn bộ dấu – của chuỗi được khử hết

5.2.2.2. Cài đặt và kiểm chứng

Thuật toán Join được cài đặt như sau:

```
1.  public static string Join(string InputString)
2.  {
3.    string pattern = @"-\d+";
4.    while (Regex.IsMatch(InputString, pattern))
5.    {
6.      foreach (Match m in Regex.Matches(InputString, pattern))
7.      {
8.        string[] operation_temp = Regex.Split(m.Value, @"\D");
9.   operation_temp = operation_temp.Where(s => !String.IsNullOrEmpty(s)).ToArray();
10.       int m1 = Convert.ToInt32(operation_temp[0].ToString()) - 1;
11.       if (m1 > 0)
12.   InputString = Regex.Replace(InputString, m.Value, "!1-" + m1.ToString());
13.       else
14.         InputString = Regex.Replace(InputString, m.Value, "!1");
15.     }
16.   }
17.     return InputString;
18.  }
```

Để kiểm tra tính đúng đắn của thuật toán, ta kiểm tra với các đầu vào khác nhau và được kết quả cho ở bảng sau:

Bảng 5.2 Bảng kết quả kiểm thử hàm cộng

Lượt	Dữ liệu vào	Kết quả
1	-1	!1
2	-2#1	!1!1#1
3	#3 -2	#3!1!1
4	#21!3	#21!1!1!1
5	#4-3#1-1	#4!1!1!1#1!1

5.2.3. Thuật toán gộp (Merge)

5.2.3.1. Mô tả thuật toán

Thuật toán dựa trên các quy tắc được trinh bày trong định nghĩa 8, mục 4.1

Đầu vào: S_1, S_2 là 2 chuỗi có dấu đã chính tắc, với số phần tử chứa dấu *"¬"* bằng nhau (có thể bằng 0). Hàm *merge* định nghĩa đệ quy như sau:

Merge $(S_1, S_2) = {}^{\#}(m_1+m_2)$ khi $S_i = {}^{\#}m_i$, $i = 1, 2$

$$Merge(^{\#} m_1 \neg n_1 S_1', ^{\#} m_2 \neg n_2 S_2') = ^{\#} (m_1 + m_2) \neg (n_1 + n_2) Merge(S_1', S_2')$$

Đầu ra: Chuỗi mới được gộp từ 2 chuỗi có dấu chính tắc ban đầu (chỉ chứa # và ¬).

Thuật toán:

Duyệt 2 chuỗi S_1, S_2 theo chiều từ trái qua phải và kiểm tra:

- Nếu 2 phần tử lấy ra có dạng là $^{\#}m_i$
  - Loại 2 phần tử đầu của S_1, S_2 ra khỏi chuỗi.
  - Thêm phần tử $^{\#}(m_1 + m_2)$ vào đầu chuỗi kết quả.
  - Lặp lại bước 1 với các phần tử còn lại của 2 chuỗi.
- Nếu 2 phần tử lấy ra có dạng là $^{!}m_i$
  - Loại 2 phần tử đầu của S_1, S_2 ra khỏi chuỗi
  - Thêm phần tử $\neg(m_1 + m_2)$ vào đầu chuỗi kết quả
  - Lặp lại bước 1 với các phần tử còn lại của 2 chuỗi
- Nếu 2 phần tử có dạng là $\neg m_1, ^{\#}m_2$
  - Loại phần tử đầu của S_2 ra khỏi chuỗi.
  - Lặp lại bước 1 với S_1và các phần tử còn lại của S_2.
- Nếu 2 phần tử có dạng là $^{\#}m_1, \neg m_2$
  - Loại phần tử đầu của S_1 ra khỏi chuỗi
  - Quay lại bước 1 với các phần tử còn lại của S_1, S_2.

5.2.3.2. Cài đặt và kiểm chứng

```
1.  public static string Merge(string s1, string s2)
2.  {
3.      string[] S1 = ConvertToArray(s1);
4.      List<string> L1 = new List<string>(S1);
5.
6.      string[] S2 = ConvertToArray(s2);
7.      List<string> L2 = new List<string>(S2);
8.
9.
10.       StringBuilder result = new StringBuilder();
11.
12.       //int m1 = 0, m2 = 0;
13.       while ((L1.Count > 0) && (L2.Count > 0))
14.       {
15.       if ((L1[0] == "#") & (L2[0] == "#"))
16.       {
17.
18.          int m = Int32.Parse(L1[1]) + Int32.Parse(L2[1]);
19.      result.Append("#" + m.ToString());
20.      L1.RemoveRange(0, 2);
21.      L2.RemoveRange(0, 2);
22.       }
23.       else if ((L1[0] == "!") & (L2[0] == "!"))
24.       {
25.       int m = Int32.Parse(L1[1]) + Int32.Parse(L2[1]);
26.      result.Append("!" + m.ToString());
27.
28.      L1.RemoveRange(0, 2);
29.      L2.RemoveRange(0, 2);
30.
31.       }
32.       else if ((L1[0] == "#") & (L2[0] == "!"))
33.       {
34.      int m = Int32.Parse(L1[1]);
35.      result.Append("#" + m.ToString());
36.
37.      L1.RemoveRange(0, 2);
38.       }
39.       else if ((L1[0] == "!") & (L2[0] == "#"))
40.       {
41.      int m = Int32.Parse(L2[1]);
42.      result.Append("#" + m.ToString());
43.      L2.RemoveRange(0, 2);
44.       }
45.
46.       else break;
47.       }
48.
49.       if ((L1.Count == 0) & (L2.Count > 0))
50.       {
51.
52.       string s = string.Join("", L2.ToArray());
53.       result.Append(s);
54.       }
55.       else if ((L2.Count == 0) & (L1.Count > 0))
56.       {
57.       string t = string.Join("", L1.ToArray());
```

Để kiểm tra tính đúng đắn của thuật toán, ta kiểm tra với các đầu vào khác nhau và được kết quả cho ở bảng sau:

Bảng 5.3 Bảng kết quả kiểm thử hàm gộp

Lượt	Chuỗi 1	Chuỗi 2	Kết quả
1	!1	!1	!2
2	!1	#1 !1	#1!2
3	#2!2	!1	#2!3
4	#3	#3	#6
5	#3!1#2!2	#1!1!1	#4!2#2#2!3

5.2.4. Thuật toán JoinCommit

5.2.4.1.Mô tả thuật toán:

Thuật toán dựa trên các quy tắc được trinh bày trong định nghĩa 10, mục 4.1

Đầu vào: S_1, S_2 là 2 chuỗi chính tắc.

Hàm *joint commit* được định nghĩa như sau:

Jc ($^{\#}n_1$, $^{\#}l_1$) = $^{\#}$max (n_1, l_1)

Jc ($S'_1{}^{+}n_1{}^{\#}n_2$, $^{\#}l_1{}^{\neg}l_2\ S'_2$) = jc (Sequence ($S'_1{}^{\#}(n_1+n_2)$), Sequence($^{\#}(l_1+1_2*n)\ S'_2$)

Đầu ra: Chuỗi chính tắc mới có dạng #m

Bước 1: Duyệt chuỗi S_1, S_2 :

- Nếu S_1, S_2 có dạng #m, #n → jc (S_1, S_2)= #max(m,n).
- Nếu các phần tử cuối của S_1 có dạng "$^{+}n_1{}^{\#}n_2$" thì kiểm tra phần đầu của S_2:
  - Nếu phần đầu S_2 có dạng $^{\neg}l_2$ thì:
    - Loại bỏ phần đầu của S_2
    - Gọi hàm Seq với đầu vào là chuỗi S_2 sau khi đã thêm phần tử $^{\#}(l_2 *n_1)$ vào đầu chuỗi thu được ở bước trên.
    - Gọi hàm Jc với $S_1 = {}^{\#}(n_1+n_2)$ và chuỗi kết quả thu được ở trên
  - Nếu phần đầu S_2 có dạng $^{\#}l_1{}^{\neg}l_2$ thì:
    - Loại bỏ 2 phần tử đầu của S_2.
    - Gọi hàm Seq($^{\#}(l_1 + l_2 *n_1)\ S_2$) với S_2 thu được ở bước trên
    - Gọi đệ quy jc với chuỗi $^{\#}$ (n_1+n_2) và chuỗi kết quả thu được ở bước trên
- Nếu phần cuối của S_1 có dạng "+n1" thì kiểm tra phần đầu của S_2:
  - Nếu phần đầu lst2 có dạng $^{\neg}$l thì:
    - Loại bỏ phần tử đầu của S_2
    - Gọi hàm Seq ($^{\#}(l_2 *n_1)\ S_2$) với S_2 thu được ở bước trên
    - Gọi đệ quy joint commit với chuỗi S1 = $^{\#}n_1$ và chuỗi kết quả thu được ở bước trên
  - Nếu các phần đầu S_2 có dạng $^{\#}l_1{}^{\neg}l_2$ thì:
    - Loại bỏ phần đầu của S_2
    - Seq($^{\#}(l_1 + l_2 *n_1)\ S_2$) với S_2 thu được ở bước trên

- Gọi đệ quy joint commit với chuỗi S1= $^{\#}n_1$ và S_2 thu được ở trên

5.2.4.2. Cài đặt và kiểm chứng

```
1.  public static string JoinCommit(string s1, string s2)
2.      {
3.          string[] S1 = ConvertToArray(s1);
4.          List<string> M1 = new List<string>(S1);
5.          string[] S2 = ConvertToArray(s2);
6.          List<string> M2 = new List<string>(S2);
7.          string result = "";
8.          int l1, l2, n1, n2;
9.          if ((M1[0] == "#") & (M2[0] == "#") & (M1.Count == 2))
10.         {
11.         n2 = Int32.Parse(M1[1]);
12.         l1 = Int32.Parse(M2[1]);
13.         int max = n2 > l1 ? n2 : l1;
14.         M2.RemoveRange(0, 2);
15.         string t = string.Join("", M2.ToArray());
16.         result = Seq("#" + max.ToString() + t);
17.         return result;
18.
19.         }
20.         else
21.         {
22.
23.  if ((M1.Count >= 4) && (M1[M1.Count - 4] == "+") && (M1[M1.Count - 2] == "#"))
24.         {
25.       if (M2[0] == "!")
26.       {
27.       l2 = Int32.Parse(M2[1]);
28.       n1 = Int32.Parse(M1[M1.Count - 3]);
29.       n2 = Int32.Parse(M1[M1.Count - 1]);
30.       int x1 = l2 * n1;
31.       int n = n1 + n2;
32.       M2.RemoveRange(0, 2);
33.       M1.RemoveRange(M1.Count - 4, 4);
34.
35.       string t11 = string.Join("", M1.ToArray());
36.       string t12 = string.Join("", M2.ToArray());
37.       string st1 = Seq(t11 + "#" + n.ToString());
38.       string st2 = Seq("#" + x1.ToString() + t12);
39.
40.       return JoinCommit(st1, st2);
41.       }
42.       else if ((M2[0] == "#") & (M2[8]    == "!"))
43.       {
44.       l1 = Int32.Parse(M2[1]);
45.       l2 = Int32.Parse(M2[5]);
46.       n1 = Int32.Parse(M1[M1.Count - 3]);
47.       n2 = Int32.Parse(M1[M1.Count - 1]);
48.       int x2 = l1 + l2 * n1;
49.       int m = n1 + n2;
```

```
50.      M2.RemoveRange(0, 4);
51.      M1.RemoveRange(M1.Count - 4, 4);
52.      string t21 = string.Join("", M1.ToArray());
53.      string t22 = string.Join("", M2.ToArray());
54.      string st3 = Seq(t21 + "#" + m.ToString());
55.      string st4 = Seq("#" + x2.ToString() + t22);
56.      return JoinCommit(st3, st4);
57.      }
58.        }
59.
60.           else if (M1[M1.Count - 2] == "+")
61.           {
62.         if (M2[0] == "!")
63.         {
64.         l2 = Int32.Parse(M2[1]);
65.         n1 = Int32.Parse(M1[M1.Count - 1]);
66.         int k1 = l2 * n1;
67.         M2.RemoveRange(0, 2);
68.         M1.RemoveRange(M1.Count - 2, 2);
69.         string t32 = string.Join("", M2.ToArray());
70.         string t31 = string.Join("", M1.ToArray());
71.
72.         string st5 = Seq(t31 + "#" + n1.ToString());
73.         string st6 = Seq("#" + k1.ToString() + t32);
74.
75.         return JoinCommit(st5, st6);
76.         }
77.         else if ((M2[0] == "#") & (M2[8]    == "!"))
78.         {
79.         l1 = Int32.Parse(M2[1]);
80.         l2 = Int32.Parse(M2[5]);
81.         n1 = Int32.Parse(M1[M1.Count - 1]);
82.         int k2 = l1 + l2 * n1;
83.         M2.RemoveRange(0, 4);
84.         M1.RemoveRange(M1.Count - 2, 2);
85.         string t42 = string.Join("", M2.ToArray());
86.         string t41 = string.Join("", M1.ToArray());
87.         string st7 = Seq(t41 + "#" + n1.ToString());
88.
89.         string st8 = Seq("#" + k2.ToString() + t42);
90.         return JoinCommit(st7, st8);
91.         }
92.         else Console.WriteLine("Chuong trinh khong hop le");
93.
94.           }
95.           }
96.
97.           return null;
98.
99.                 }
```

Để kiểm tra tính đúng đắn của thuật toán, ta kiểm tra với các đầu vào khác nhau và thu được kết quả cho ở bảng sau:

Bảng 5.4 Bảng kiểm thử hàm JoinCommit

Lượt	Chuỗi 1	Chuỗi 2	Kết quả
1	+1	!1	#1
2	#1	#1!1	#2
3	+1#2	#1 !1	#3
4	+2#3	#2!1	#5

5.2.5. Thuật toán tính cận trên tài nguyên của chương trình giao dịch

5.2.5.1. Mô tả thuật toán

Đầu vào: Chuỗi kết xuất được từ chương trình Featherweight Java có sử dụng giao dịch

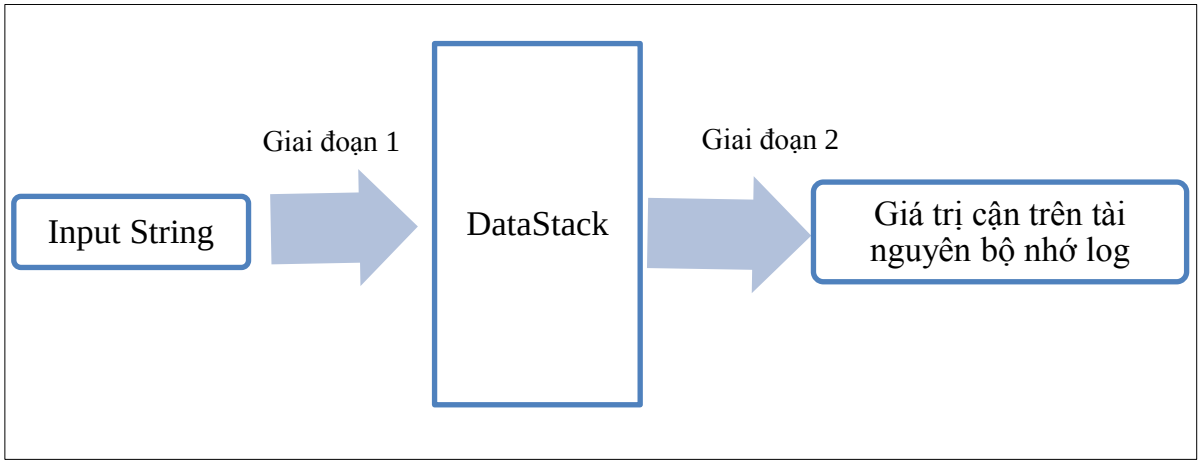
Đầu ra: Giá trị cận trên tài nguyên bộ nhớ log của chương trình hoặc thông báo chương trình thành lập không hợp lệ

Thuật toán:

Thuật toán này có thể chia nhỏ thành các giai đoạn sau:

Giai đoạn 1: Chuyển chuỗi kết xuất được từ mã chương trình TFJ vào ngăn xếp, trong đó mỗi phần tử ở dạng "(" hoặc ")" hoặc X, với X là 1 chuỗi.

Giai đoạn 2: Tính giá trị cận trên tài nguyên cho bộ nhớ log



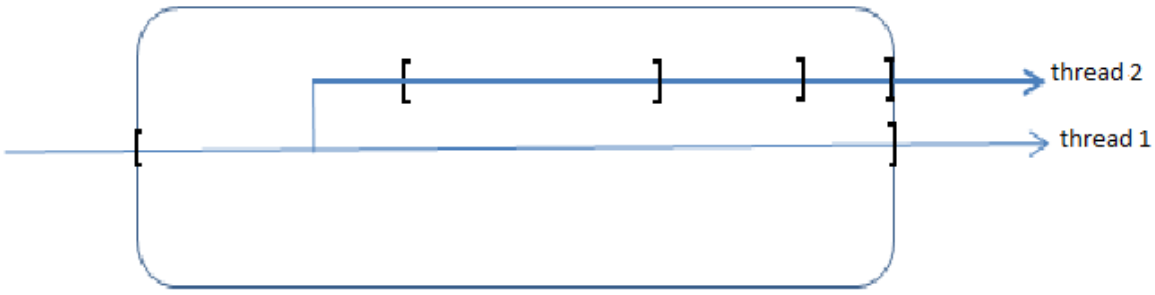
Hình 5.1 Hình mô tả các giai đoạn tính cận trên tài nguyên bộ nhớ log

Ví dụ 5.1

Tính giá trị cận trên tài nguyên bộ nhớ log của chương trình có sử dụng giao dịch với đoạn mã như sau:

```
onacid(1);
spawn(onacid(1),commit,commit,commit);
commit;
```

Khi đó, đoạn mã trên có thể mô hình hóa ở dạng các luồng lồng nhau như hình vẽ dưới đây:

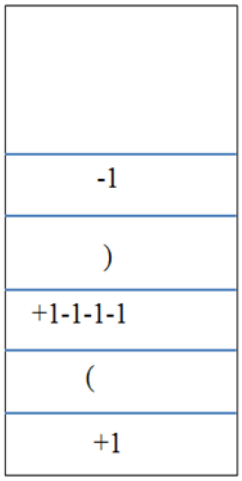


Hình 5.2 Mô hình giao dịch lồng và đa luồng cho ví dụ 5.1

Với đoạn mã như trên, ta kết xuất được xâu đầu vào tương ứng:

Input= +1(+1-1-1-1)-1

Giai đoạn 1: Chuyển xâu đầu vào Input thành một ngăn xếp



Hình 5.3 Mô tả giai đoạn 1 của thuật toán tính tài nguyên

Giai đoạn 2: Tính giá trị cận trên tài nguyên với xâu đầu vào đã được chuyển thành ngăn xếp như hình 5.3.

Sau đây là thuật toán cụ thể cho từng giai đoạn:

Giai đoạn 1: Chuyển chuỗi kết xuất được từ mã chương trình TFJ vào ngăn xếp, trong đó mỗi phần tử ở dạng "(" hoặc ")" hoặc X, với X là 1 chuỗi.

Thuật toán:

Bước 1: Nhận giá trị đầu vào xâu Input;

Khởi tạo DataStack= null để lưu trữ xâu trả về;

i=0, biến đếm lưu chỉ số cho từng ký tự;

Xâu S lưu lại giá trị mỗi phần tử trong ngăn xếp DataStack.

Bước 2: Kiểm tra nếu i>= Input.Length thì trả về ngăn xếp DataStack và kết thúc.

Bước 3: Nếu Input[i]!= ‘(’ và Input[i] != ‘)’ thì

S= + Input[i], i=i+1 và quay về Bước 2.

Bước 4: Nếu Input[i] = ‘(’ thì

Bước 4.1: Kiểm tra nếu S.Length>0 thì đẩy S vào ngăn xếp.

Bước 4.2: Đẩy “(” vào ngăn xếp, quay về Bước 2.

Bước 5: Nếu Input[i] = ‘)’ thì

Bước 5.1: Kiểm tra nếu S.Length >0 thì đẩy S vào ngăn xếp.

Bước 5.2: Đẩy “)” vào ngăn xếp, quay về Bước 2.

Giai đoạn 2: Tính giá trị cận trên tài nguyên cho bộ nhớ log

Thuật toán:

Bước 1: Đầu vào là chuỗi Input đã được chuyển về ngăn xếp DStack qua giai đoạn 1; Khởi tạo result= “ ”, result là xâu lưu lại kết quả cận trên bộ nhớ log.

Bước 2: Thực hiện khi ngăn xếp không rỗng. Lấy ra phần tử đầu tiên X của ngăn xếp,

Bước 3: Nếu X = “)” thì thực hiện

Bước 3.1: Lấy phần tử trên cùng của ngăn xếp là Y

Bước 3.2: Gọi hàm Join(Seq(Y)), kết quả trả về của hàm Join(Seq(Y)) là chuỗi S12

Bước 3.3: Gán result = Merge(S12,result); Và quay trở lại bước 2.

Bước 4: Nếu X = “(” thì thực hiện

Bước 4.1: Lấy phần tử trên cùng của ngăn xếp là Y2

Bước 4.2: Gọi hàm Join(Seq(Y2)), kết quả Join(Seq(Y2)) là chuỗi S22

Bước 4.3: Gán result= JoinCommit(S22,result); Và quay trở lại bước 2

Bước 5: Nếu X là xâu

Bước 5.1: Gọi hàm Join(Seq(X)), kết quả thu được chuỗi S32

Bước 5.2: Nếu result= “ ” thì gán result= S32; Và quay trở lại bước 2

Bước 5.3: Nếu result != “ ” thì gán result= JoinCommit(S32,result); Và quay trở lại bước 2.

5.2.5.2. Cài đặt và kiểm chứng

Cài đặt cho giai đoạn 1:

```
1.  public static Stack<string> ConvertStringToStack(string Input)
2.      {
3.
4.      Stack<string> DataStack = new Stack<string>();
5.      String S = "";
6.      for (int i = 0; i < Input.Length; i++)
7.      {
8.        if (Input[i] == '(')
9.        {
10.       if (S.Length > 0) DataStack.Push(S);
11.       DataStack.Push("(");
12.       S = "";
13.       }
14.       else if (Input[i] == ')')
15.       {
16.       if (S.Length > 0) DataStack.Push(S);
17.       DataStack.Push(")");
18.       S = "";
19.       }
20.       else S += Input[i];
21.     }
22.     if (S != "") DataStack.Push(S);
23.
24.     return DataStack;
25.     }
```

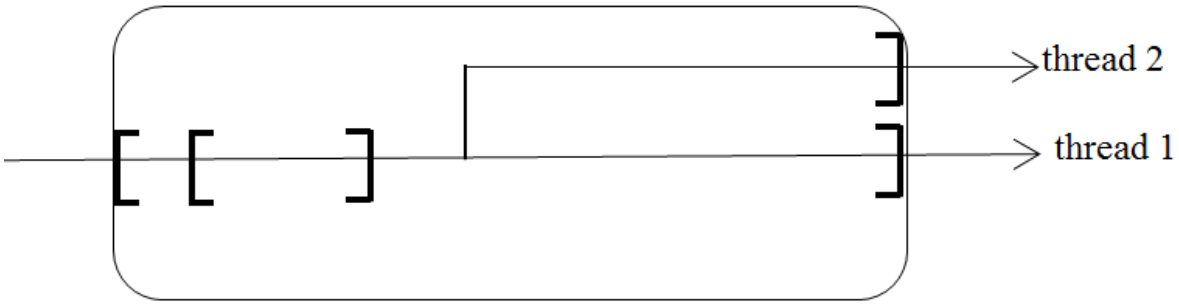
Cài đặt cho giai đoạn 2:

```
1.  public static string CalculResourses(string Input)
2.      {
3.
4.      Stack<string> DStack = ConvertStringToStack(Input);
5.      string result = "";
6.
7.      while (DStack.Count > 0)
8.      {
9.          string X = DStack.Pop();
10.         if (X == ")")
11.         {
12.         string Y = DStack.Pop();
13.         string S12 = Join(Seq(Y));
14.         result = Merge(S12, result);
15.         }
16.         else if (X == "(")
17.         {
18.         string Y2 = DStack.Pop();
19.
20.         string S22 = Join(Seq(Y2));
21.         result = JoinCommit(S22, result);
22.
23.         }
24.         else
25.         {
26.
27.         string S32 = Join(Seq(X));
28.         if (result == "") result = S32;
29.         else result = JoinCommit(S32, result);
30.
31.         }
32.
33.
34.     }
35.     return result;
36.     }
```

Để kiểm tra tính đúng đắn của thuật toán, ta kiểm tra với các đầu vào khác nhau qua các thực nghiệm sau:

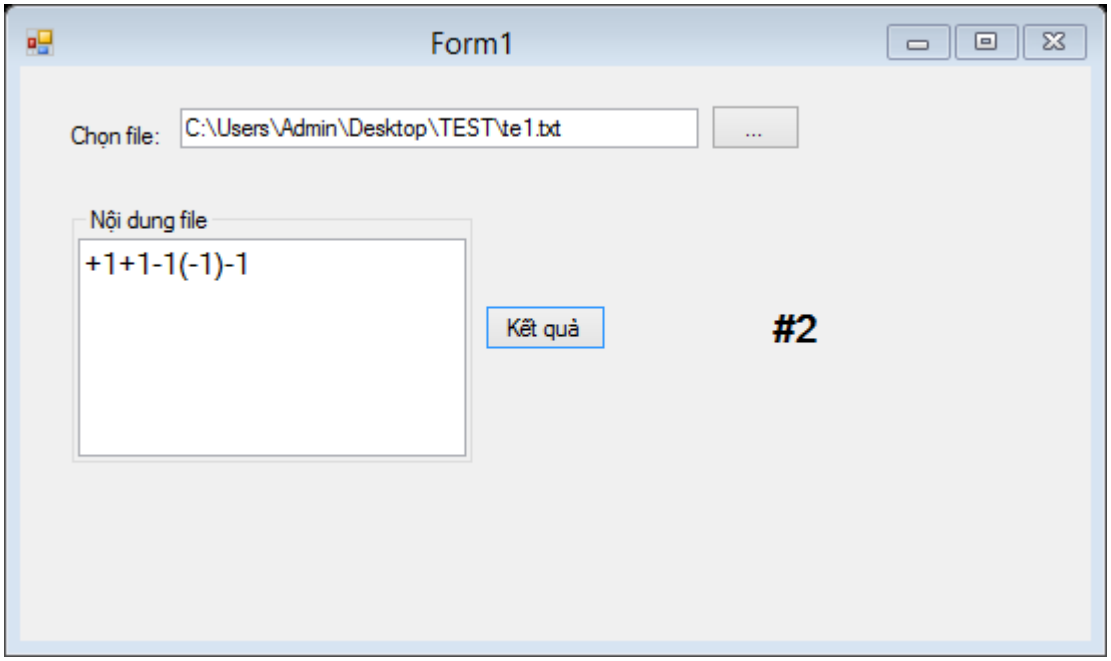
Thực nghiệm 1: Tính cận trên tài nguyên bộ nhớ log của chương trình có giao dich, với chuỗi đầu vào: S=+1+1-1(-1)-1

Mô hình giao dịch lồng và đa luồng tương ứng với chuỗi trên:



Hình 5.4 Mô hình giao dịch lồng và đa luồng cho thực nghiệm 1

Chạy thực nghiệm chương trình ta được kết quả: #2



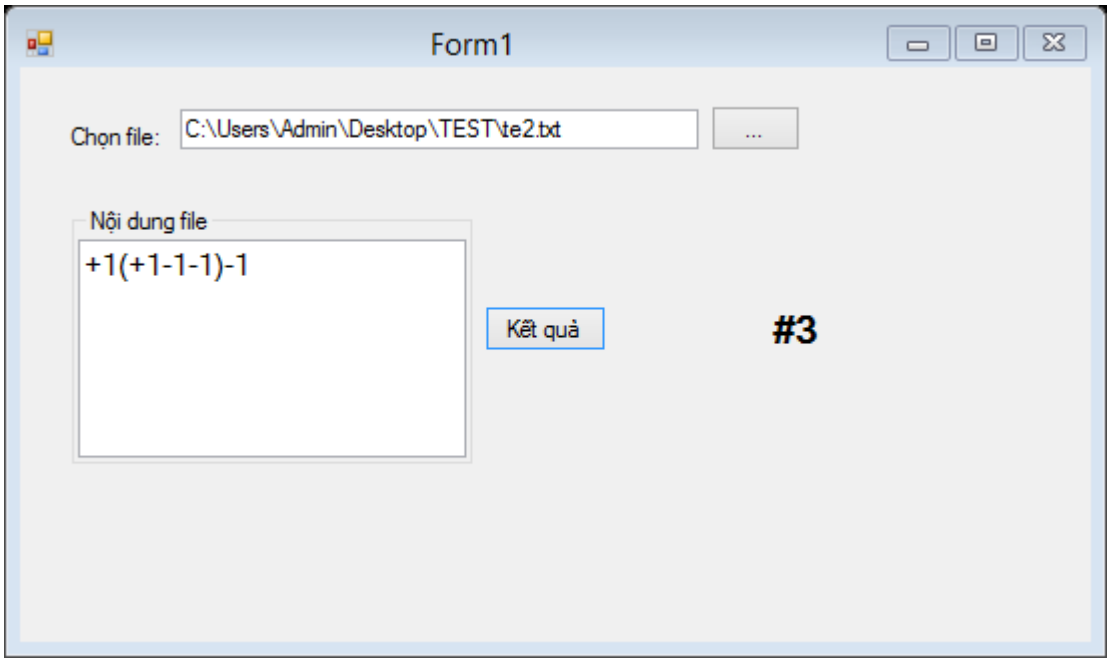
Hình 5.5 Màn hình kết quả thực nghiệm 1

Thực nghiệm 2: Tính cận trên tài nguyên bộ nhớ log của chương trình có giao dich, với chuỗi đầu vào: S=+1(+1-1-1)-1

Mô hình giao dịch lồng và đa luồng tương ứng cho thực nghiệm 2 như sau:

Hình 5.6 Mô hình giao dịch lồng và đa luồng cho thực nghiệm 2

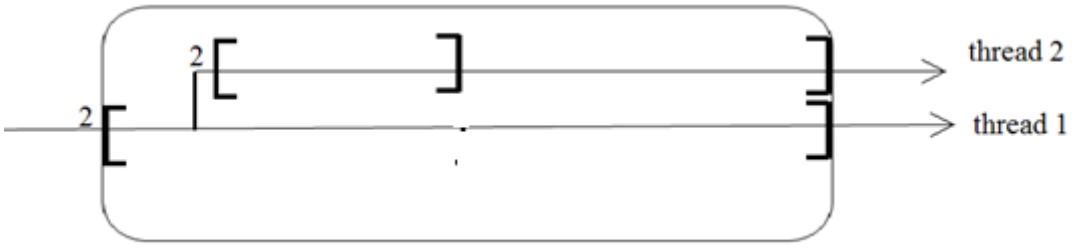
Chạy thực nghiệm chương trình, ta nhận được giá trị cận trên bộ nhớ log là #3



Hình 5.7 Màn hình kết quả thực nghiệm 2

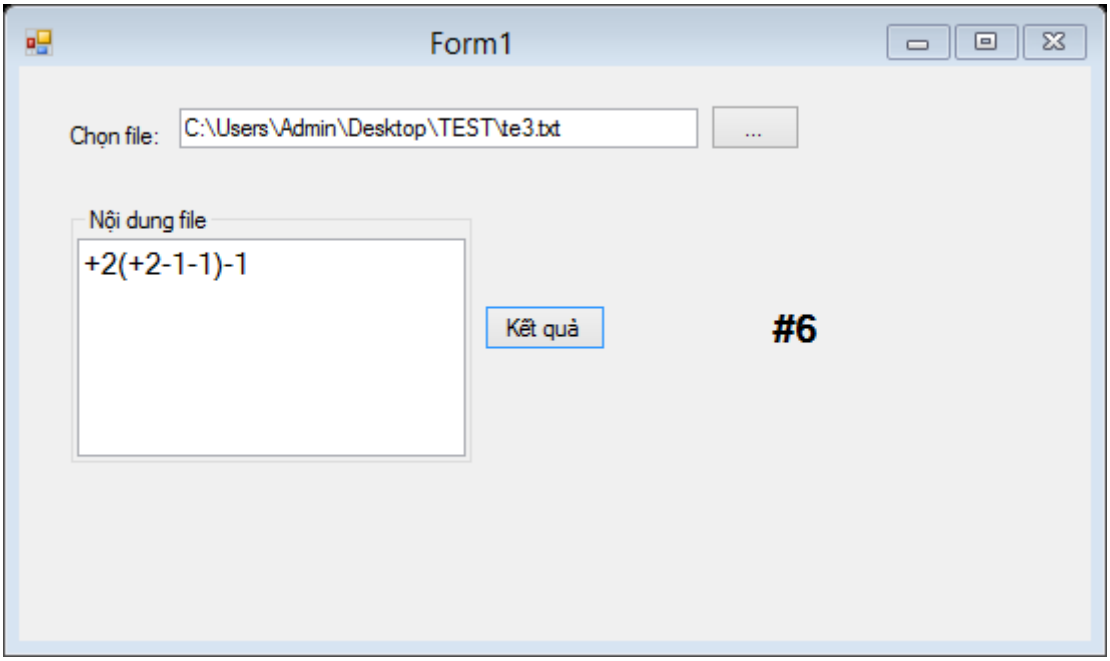
Thực nghiệm 3: Tính cận trên tài nguyên bộ nhớ log của chương trình có giao dich, với chuỗi đầu vào: S= +2(+2-1-1)-1

Chuỗi trên có thể được mô hình hóa như sau:



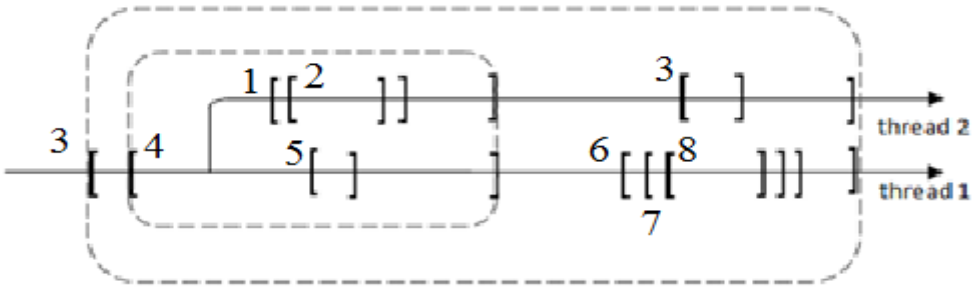
Hình 5.8 Mô hình giao dịch lồng và đa luồng cho thực nghiệm 3

Chạy thực nghiệm chương trình, ta nhận được giá trị cận trên bộ nhớ log là #6.



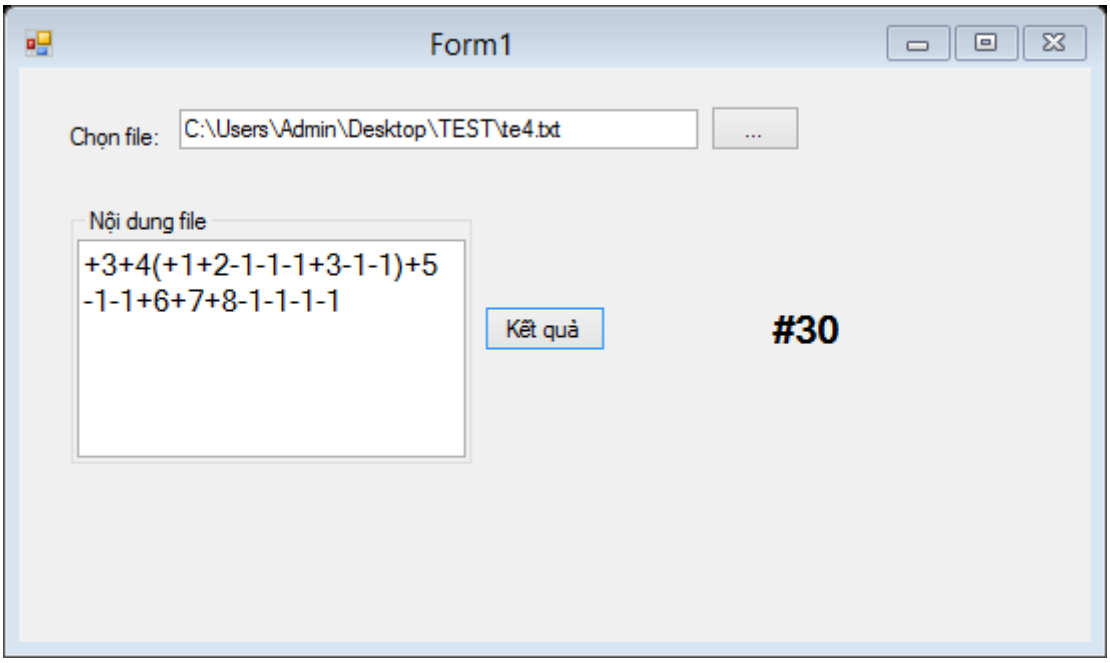
Hình 5.9 Màn hình kết quả chạy thực nghiệm 3

Thực nghiệm 4: ***Tính cận trên tài nguyên bộ nhớ log của chương trình có giao dich, với chuỗi đầu vào: S= +3+4(+1+2-1-1-1+3-1-1)+5-1-1+6+7+8-1-1-1-1***
Chuỗi trên có thể được mô hình hóa như sau:



Hình 5.10 Mô hình giao dịch cho thực nghiệm 4

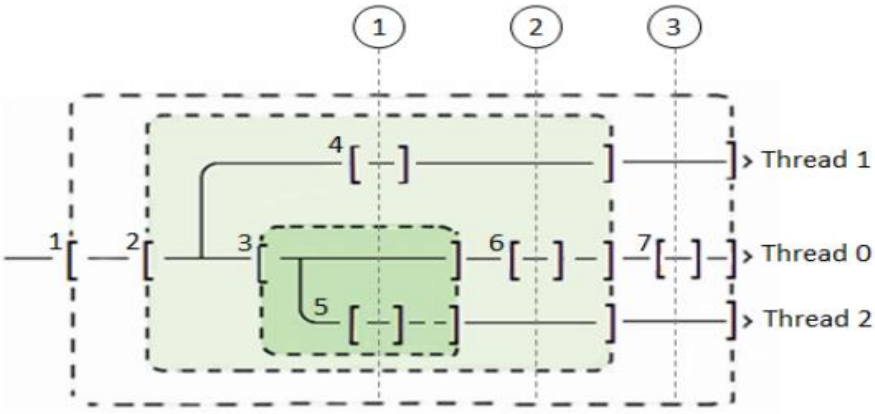
Chạy thực nghiệm chương trình, ta nhận được giá trị cận trên bộ nhớ log là #30.



Hình 5.11 Màn hình kết quả thực nghiệm 4

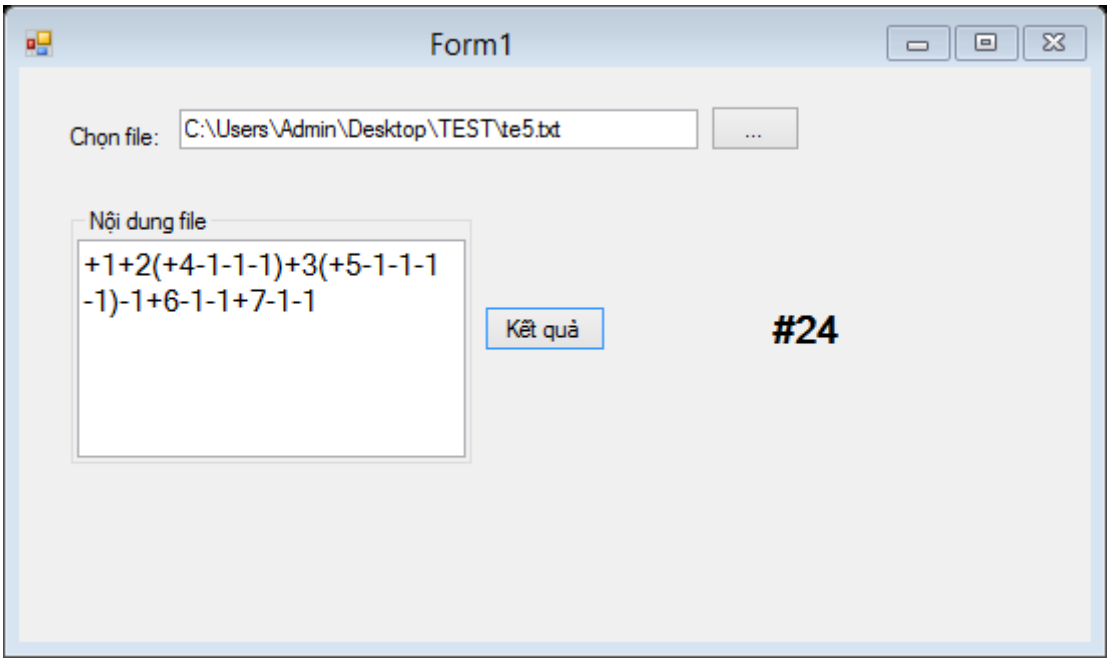
Thực nghiệm 5: ***Tính cận trên tài nguyên bộ nhớ log của chương trình có giao dịch, với chuỗi đầu vào: S= +1+2(+4-1-1-1)+3(+5-1-1-1-1)-1+6-1-1+7-1-1***

Mô hình giao dịch lồng và đa luồng tương ứng với chuỗi đầu vào như trên là:



Hình 5.12 Mô hình giao dịch lồng và đa luồng cho thực nghiệm 5

Chạy thực nghiệm chương trình, ta nhận được giá trị cận trên bộ nhớ log là #24.



Hình 5.13 Màn hình kết quả thực nghiệm 5

Đánh giá kết quả thực nghiệm:

Qua chạy thực nghiệm công cụ với một số trường hợp, các kết quả nhận được có giá trị bằng các kết quả mong đợi.

KẾT LUẬN

Qua thời gian nghiên cứu và tìm hiểu đề tài, luận văn đã được hoàn thành và đạt được những nội dung đề ra với mục tiêu chính là giải quyết bài toán tính cận trên bộ nhớ log cho các chương trình sử dụng giao dịch.

Về lý thuyết, luận văn đã trình bày được các kiến thức cơ sở về hệ thống kiểu nói chung bao gồm định nghĩa hệ thống kiểu, các thuộc tính cơ bản của hệ thống kiểu và ứng dụng của hệ thống kiểu trong thực tế. Ngoài ra, luận văn còn trình bày các khái niệm cơ bản về giao dịch và bộ nhớ giao dịch phần mềm. Tiếp theo, cú pháp và ngữ nghĩa của ngôn ngữ giao dịch TM cũng được giới thiệu trong luận văn. Từ cú pháp và ngữ nghĩa của ngôn ngữ TM, luận văn đã trình bày phương pháp xây dựng hệ thống kiểu để xác định cận trên bộ nhớ log của chương trình sử dụng giao dịch, dựa trên nghiên cứu được các tác giả thực hiện trong bài báo [1]. Một chương trình có giao dịch được cấu thành từ các thành phần cơ bản, mỗi thành phần thể hiện hành vi giao dịch và được định kiểu thông qua một dạng chuỗi số đặc biệt, chuỗi số có dấu. Hệ thống kiểu được trình bày ở đây bao gồm các kiểu, các quy tắc kiểu trong đó chứa định nghĩa các phép toán được sử dụng để định kiểu cho từng thành phần trong chương trình sử dụng giao dịch.

Về thực nghiệm, một công cụ được viết bằng ngôn ngữ C# đã được cài đặt để tính cận trên bộ nhớ log của chương trình sử dụng giao dịch. Chương trình bao gồm các phương thức được xây dựng để thực hiện các phép toán như rút gọn một chuỗi số có dấu, gộp 2 chuỗi số có dấu, Joincommit…Và đặc biệt là phương thức để tính cận trên bộ nhớ log. Chương trình đã được thực nghiệm với nhiều chuỗi được kết xuất từ các chương trình giao dịch khác nhau và cho kết quả tương đối chính xác.

Tuy nhiên, do thời gian có hạn và tài liệu nghiên cứu liên quan chưa nhiều. Hơn nữa, đây là một đề tài khó, đòi hỏi sự đầu tư nhiều về thời gian và công sức nên trong luận văn này không tránh khỏi những hạn chế. Trong quá trình nghiên cứu về đề tài, chúng tôi cũng nhận thấy các kết quả nghiên cứu mới chỉ dừng ở mức độ thực hiện và kiểm chứng về mặt lý thuyết mà chưa hề được kiểm chứng ở thực tế. Do vậy trong tương lai, hi vọng đề tài có thể được nghiên cứu và kiểm chứng ở thực tế. Nếu thành công, các kết quả đạt được này sẽ đóng góp đáng kể vào việc tối ưu các chương trình phần mềm và làm tăng hiệu quả sử dụng tài nguyên bộ nhớ.

TÀI LIỆU THAM KHẢO

Tiếng Anh

[1] Anh-Hoang Truong, Ngoc-Khai Nguyen, Dang Van Hung, and Dang Duc Hanh (2016), "Calculate statically maximum log memory used by multi-threaded transactional programs", *Theoretical Aspects of Computing – ICTAC 2016,* pp. 82-99

[2] 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*, pp. 157-168

[3] Hoang Truong (2006), *Type Systems for Guaranteeing Resource Bounds of Component Software*, Dissertation for the degree Philosophiae Doctor (PhD) University of Bergen, Norway

[4] Igarashi, Atsushi, Pierce Benjamin C, Wadler, Philip (2001), "Featherweight Java: a minimal core calculus for Java and GJ", *Journal ACM Transactions on Programming Languages and Systems,* Volume 23, pp. 396- 450

[5] Jiang Hui,Lin Dong, Zhang Xingyuan, Xie Xiren(2001), "Type System in Programming Languages", *Journal of Computer Science and Technology*, Volume 16, pp. 286-292

[6] Luca Cardelli (1996), "Type system", *ACM Computing Surveys (CSUR),*Volume 28 Issuse1, pp. 263-264

[7] Martin Steffen and Thi Mai Thuong Tran (2009), "Safe commits for Transactional Featherweight Java", *Integrated Formal Methods*, pp. 290-304

[8] N.Shavit, and D.Touitou (1995), "Software Transational Memory", Proceeding PODC *'95 Proceedings of the fourteenth annual ACM symposium on Principles distributed computing*, pp 204-213

[9] Thi Mai Thuong Tran, O.Owe, and Martin Steffen (2010), "Safe typing for transactional vs. lock-based concurrency in multi-threaded Java", *KSE '10 Proceedings of the 2010 Second International Conference on Knowledge and Systems Engineering*, pp.188-193

[10] Thi Mai Thuong Tran, Martin Steffen, and Hoang Truong (2011), "Estimating Resource Bounds for Software Transactions", *SEFM 2013 Proceedings of the 11th International Conference on Software Engineering and Formal Methods*, Volume 8137, pp. 212-228

[11] Xuan-Tung Vu, Thi Mai Thuong Tran, Anh-Hoang Truong, and Martin Steffen. "A type system for finding upper resource bounds of multi-threaded programs with nested transactions", In *Symposium on Information and Communication Technologys 2012, SoICT '12, Halong City, Quang Ninh, Viet Nam, August 23-24, 2012*, pp. 21-30