ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

LUẬN VĂN TỐT NGHIỆP ĐẠI HỌC

KỸ THUẬT DỊCH NGƯỢC

HỘI ĐỒNG: Khoa Học Máy Tính (Hội đồng 2)
GVHD: TS. Nguyễn Hứa Phùng
GVPB: Ths. Vương Bá Thịnh
---o0o---
SVTH 1: Nguyễn Đôn Bình - 51100296
SVTH 2: Nguyễn Tiến Thành - 51103220

TP. HỒ CHÍ MINH, THÁNG 01/2016

LỜI CAM ĐOAN

Chúng tôi xin cam đoan đây là công trình của chúng tôi. Các số liệu, kết quả nêu trong báo cáo luận văn tốt nghiệp là trung thực và chưa từng được ai công bố trong bất kỳ công trình nào khác.

Chúng tôi xin cam đoan rằng mọi sự giúp đỡ cho việc thực hiện báo cáo luận văn tốt nghiệp này đã được cảm ơn và các thông tin trích dẫn trong báo cáo đã được ghi rõ nguồn gốc.

Sinh viên thực hiện

Nguyễn Tiến Thành Nguyễn Đôn Bình

LỜI CẢM ƠN

Chúng tôi xin gửi lời cảm ơn đến Tiến Sĩ Nguyễn Hứa Phùng - Giảng viên hướng dẫn thực hiện đề tài Kỹ Thuật Dịch Ngược tại trường đại học Bách Khoa thành phố Hồ Chí Minh đã giúp đỡ chúng tôi trong suốt thời gian thực hiện đề tài này.

Cùng với đó, chúng tôi cũng chân thành gửi lời cảm ơn đến QuatumnG, Mike Van Emerik và các cộng sự, những người đã phát triển Framework Boomerang Decompiler. Chúng tôi cũng cảm ơn các tác giả của các tài liệu đã được trích dẫn.

Mục Lục

DANH MỤC BẢNG VÀ HÌNH

CHƯƠNG A
GIỚI THIỆU

Chương này sẽ trình bày sơ lược về các khái niệm liên quan đến Kỹ Thuật Dịch Ngược và cấu trúc của Luận Văn Tốt Nghiệp. Phần đầu của chương sẽ nêu rõ tầm quan trọng và ứng dụng của Kỹ Thuật Dịch Ngược đối với nhu cầu thực tế hiện tại cũng như các khó khăn khi tiếp cận với nó. Phần sau sẽ nêu ra một số bài toán cần giải quyết trong phạm vi giới hạn của đề tài Luận Văn Tốt Nghiệp.

I. Kỹ Thuật Dịch Ngược

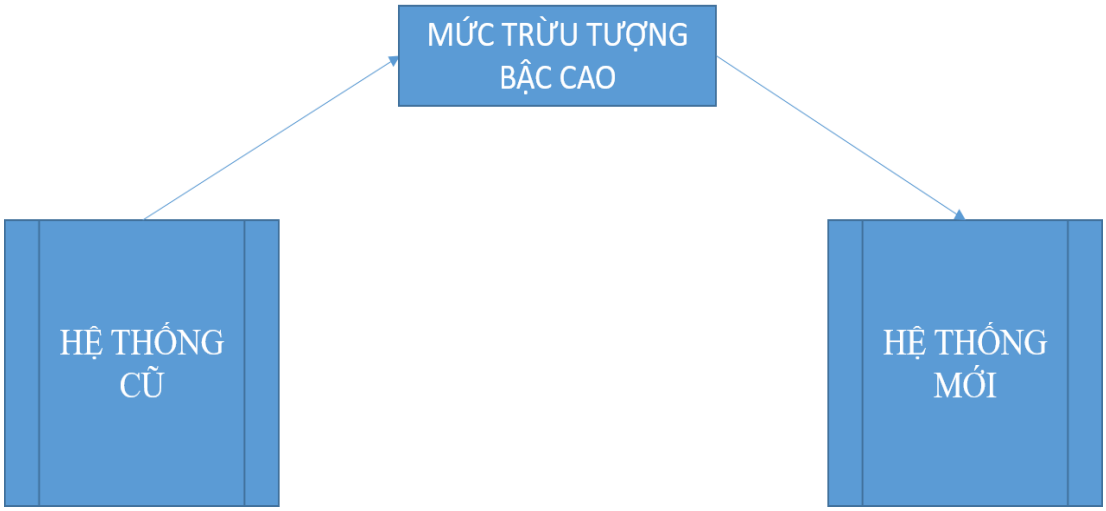
Việc tìm hiểu về cách thức hoạt động của một thành phần hay toàn bộ hệ thống phần mềm là một nhu cầu thiết yếu trong lĩnh vực Công Nghệ Phần Mềm. Nhu cầu đó phát sinh từ những bài toán tiêu biểu trong thực tế như tìm hiểu cơ chế hoạt động của các phần mềm độc hại (virus, malware), hay là kế thừa, nâng cấp, thay đổi một hệ thống phần mềm đã phát triển. Tuy nhiên, việc đáp ứng những nhu cầu này lại gặp nhiều khó khăn trong trường hợp thiếu sót tài liệu đặc tả. Nhất là đối với những phần mềm đã cũ hay các hệ thống lỗi thời (legacy), những tài liệu đặc tả liên quan thường không còn nữa, hoặc có còn thì cũng không còn đầy đủ, chính xác. Đối với các phần mềm độc hại như Virus, Malware thì tài liệu đặc tả thậm chí còn bị chủ ý che dấu. Ngoài ra, những khó khăn còn đến từ những phần mềm mới phát triển mà các tài liệu mô tả hệ thống không được xây dựng một cách chuyên nghiệp.

Ngay cả khi đã có các tài liệu mô tả hệ thống rõ ràng thì nhu cầu xây dựng lại mã nguồn hệ thống phần mềm vẫn rất quan trọng. Ví dụ như nhu cầu xây dựng lại những phần mềm được phát triển ở nền tảng cũ, lỗi thời thành những phần mềm mới có khả năng hoạt động trên những nền tảng hiện đại hơn sẽ cần đến những thông tin được rút trích từ mã nguồn. Việc khôi phục lại mã nguồn sẽ giúp cho quá trình tái xây dựng, bảo trì giảm thiểu được nhiều chi phí và bảo đảm tính chính xác hơn.

Một trong những phương pháp để giải quyết những bài toán nêu trên đó là xây dựng lại hệ thống dựa trên những tài liệu đặc tả đã có sẵn. Tuy nhiên phương pháp này gặp nhiều hạn chế. Đối với các hệ thống được viết trên ngôn ngữ lập trình đã cũ hoặc các hệ thống lỗi thời, đặc tả trở nên lỗi thời cho nên việc viết lại toàn bộ hệ thống rất khó khăn. Ngay cả đối với các hệ thống mới, việc xây dựng lại cũng mất nhiều chi phí, cùng với đó, tính ổn định cũng không dược đảm bảo khi so sánh với hệ thống đã hoạt động từ lâu.

Kỹ thuật dịch ngược

Phương pháp tiếp cận khác được đề xuất đó là đưa hệ thống có sẵn lên một mức trừu tượng cao hơn, rồi từ mức trừu tượng đó, có thể xây dựng lại tài liệu thiết kế cũng như nâng cấp, sửa đổi phần mềm theo nhu cầu. Với việc thực hiện công việc đó một cách tự động, chi phí cho việc xây dựng lại tài liệu mô tả hệ thống cũng như thừa kế, phát triển hệ thống hiện tại được cắt giảm đáng kể. Đồng thời, vì hệ thống mới được xây dựng lại từ nền tảng của hệ thống cũ, tính ổn định cũng sẽ được giữ nguyên như ban đầu.



Hình A.0. Phương pháp xây dựng lại hệ thống mới từ việc khôi phục lại hệ thống cũ

Trong Công Nghệ Phần Mềm, có một khái niệm liên quan đến phương pháp này gọi là **Kỹ Thuật Dịch Ngược (Reverse Engineering)**. Kỹ Thuật Dịch Ngược là một quá trình liên quan tới việc phân tích một hệ thống phần mềm để xây dựng nên những thông tin về hiện thực hay thiết kế ở mức trừu tượng bậc cao.. Đó có thể là việc chuyển đổi chương trình thực thi lên thành những đoạn mã nguồn hay chuyển đổi những đoạn mã ở ngôn ngữ lập trình bậc thấp lên thành những dạng mã ở ngôn ngữ lập trình bậc cao hơn.

Một trong những công cụ hỗ trợ Kỹ Thuật Dịch Ngược đó là **Trình Biên Dịch Ngược (Decompiler)**. Trình Biên Dịch Ngược là một chương trình thực hiện quá trình chuyển đổi ngôn ngữ cấp thấp lên thành chương trình ngôn ngữ cấp cao. Nói cách khác, một Trình Biên Dịch Ngược sẽ thực hiện quá trình đảo ngược của một Trình Biên Dịch (Compiler) điển hình. Hiện nay có rất nhiều công cụ Trình Biên Dịch Ngược ra đời với nhiều mục đích khác nhau như Hex-Rays, Boomerang, Retargetable Decompiler.

Mặc dù có khả năng lớn để đưa ra được lời giải cho các bài toán đã đề cập ở phía trên, nhưng việc xây dựng một hệ thống Dịch Ngược trong thực tế lại gặp nhiều khó khăn xuất phát từ việc thiếu thông tin. Khi đưa một chương trình từ ngôn ngữ bậc cao về ngôn ngữ bậc thấp (nhất là về mã máy), các thông tin về tên biến, kiểu biến thường bị mất mát. Các câu lệnh điều

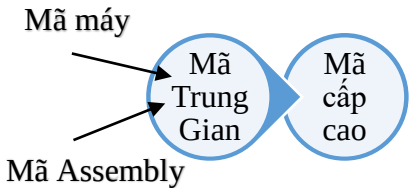
khiển rẽ nhánh hay vòng lặp sẽ không còn nữa, mà thay vào đó chỉ là cá quá trình xử lý dựa trên địa chỉ và thanh ghi. Ngoài ra, một yêu cầu quan trọng đối với hệ thống dịch ngược, mà người lập trình khó có thể định lượng được, là tính dễ đọc. Đây là một yêu cầu rất khó đáp ứng tuyệt đối, nhất là đối với các đoạn mã máy được viết một cách thủ công.

II. Đề tài luận văn tốt nghiệp

1. Bài toán dịch ngược mã Assembly

Bài toán dịch ngược mã Assembly trong thực tế có hai hướng giải quyết phổ biến. Hướng thứ nhất, đó là tiến hành biện dịch mã Assembly về mã máy tương ứng, rồi dùng các Trình Biên Dịch Ngược (vốn phần lớn được thiết kế cho việc dịch ngược mã máy) chuyển đổi lên các mã cấp cao. Tuy nhiên, phương pháp này không phải là tối ưu nhất vì có nhiều hạn chế. Lý do là một khi biên dịch mã Assembly về mã máy, một số thông tin quan trọng bị mất mát. Ví dụ như một số thư viện đặc trưng được liên kết trực tiếp vào tập tin thực thi, dẫn đến quá trình dịch ngược sau đó bị *nhiễu* khá nhiều.

Hướng giải quyết thứ hai được đề xuất, đó là phương pháp tận dụng nền tảng của các Trình Biên Dịch Ngược có sẵn. Đây là phương pháp được kỳ vọng tạo ra lời giải tương xứng do các công cụ Trình Biên Dịch Ngược được hiện thực rất tốt cho việc dịch ngược mã máy. Các kết quả từ việc biên dịch ngược cũng rất thành công và được minh chứng trong thực tế. Ngoài ra, vì phần lớn hệ thống biên dịch ngược đều trải qua giai đoạn sinh ra mã trung gian trước khi tạo ra mã đích, nên quá trình biên dịch ngược mã Assembly sẽ được thừa hưởng đáng kể phần cốt lõi nhất của hệ thống (dịch ngược mã trung gian gian sang mã cấp cao). Nói cách khác, việc dịch ngược mã máy và mã Assembly đều có thể chia sẽ chung một con đường quyết định như trong hình A.1. Qua đó góp phần giảm thiểu đáng kể chi phí nhờ tận dụng hệ thống kế thừa.



Hình A.1 Con đường quyết định của hệ thống dịch ngược

Với những lý do như vậy, nhằm giải quyết bài toán dịch ngược mã Assembly, phương pháp thứ hai sẽ được áp dụng dựa trên nền tảng của Framework Boomerang Decompiler – một công cụ Biên Dịch Ngược. Với những tính năng nổi bật như mã nguồn mở, dễ kế thừa, khả năng dịch ngược mã máy được đánh giá cao, Boomerang Decompile**r** sẽ là lựa chọn tối ưu phục vụ cho mục đích của đề tài.

2. Bài toán sinh hàm nguyên mẫu

Một trong những bài toán khác, cũng rất quan trọng đối các Trình biên dịch ngược là kỹ thuật sinh hàm nguyên mẫu (Prototype). Việc sinh hàm nguyên mẫu ra đúng là một phần giúp cho việc dịch ngược được thực hiện tốt, công việc gọi hàm, sử dụng các biến tốt hơn, nhờ đó mã sinh ra có tính chính xác, dễ đọc, dễ hiểu, đáp ứng yêu cầu để có thể là một trình biên dịch ngược tốt.

Tuy vậy, việc sinh hàm nguyên mẫu gặp rất nhiều khó khăn. Với ngôn ngữ cấp cao, các hàm được đặc tả hàm nguyên mẫu rất rõ ràng, bao gồm tên hàm, kiểu trả về, các đối số… Tuy nhiên, với ngôn ngữ máy, ở đây chỉ còn là những câu lệnh xử lý dựa trên thanh ghi, địa chỉ. Các thông tin liên quan đến việc đặc tả hàm, gọi hàm và truyền tham số đều không còn nữa. Việc mà một trình biên dịch ngược phải làm là dựa vào các lệnh nhị phân đó để phân tích và tái hiện lại các thông tin cần thiết về hàm nguyên mẫu.

Trong điều kiện Boomerang là một trong số ít các phần mềm dịch ngược mã nguồn mở, và cũng đã hiện thực được tương đối việc xây dựng lại hàm nguyên mẫu. Chức năng xác định hàm nguyên mẫu sẽ được cải tiến thêm nhằm hoàn thiện Framework này.

III. Cấu trúc của báo cáo luận văn tốt nghiệp

Báo cáo luận văn tốt nghiệp này gồm có 6 chương. Chương A, cũng là chương hiện tại sẽ giới thiệu tổng quát về kỹ thuật dịch ngược và các bài toán được đặt ra trong giai đoạn Luận Văn Tốt Nghiệp. Đó là các bài toán về dịch ngược từ mã assembly và cải tiến chức năng xác định hàm nguyên mẫu. Để giải quyết các bài toán đó, các nền tảng lý thuyết liên quan về Trình biên dịch, Trình biên dịch ngược, Framework Boomerang Decompiler và các kỹ thuật xác định hàm nguyên mẫu đã được tìm hiểu và trình bày trong chương B. Chương C sẽ đưa ra các mục tiêu cũng như giới hạn rõ ràng đối với các công việc sẽ được thực hiện. Chi tiết về việc xác định giải pháp, thiết kế hệ thống và hiện thực chức năng để giải quyết các bài toán dịch ngược từ mã assembly và cải tiến chức năng hàm nguyên mẫu lần lượt được trình bày trong chương D và chương E. Phần cuối cùng, chương F sẽ đưa ra kết luận về quá trình cũng như kết quả thực hiện đề tài, cũng như các hướng đi tiếp theo có thể thực hiện trong tương lai.

CHƯƠNG B
CƠ SỞ CỦA ĐỀ TÀI

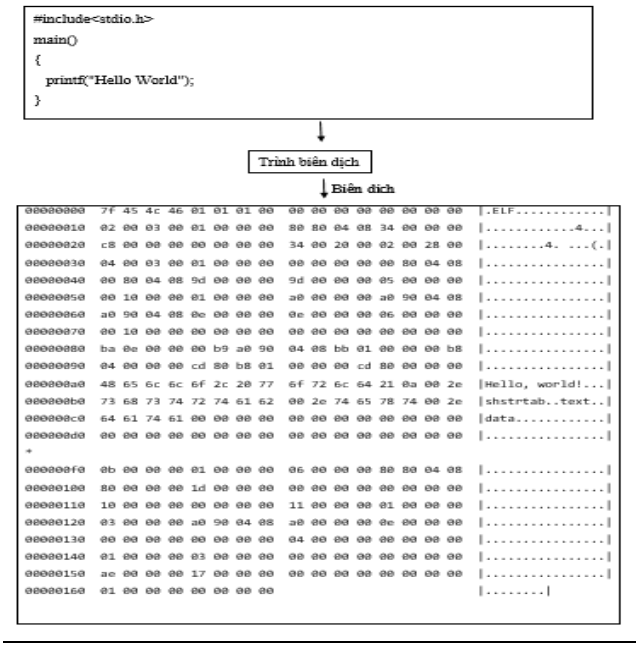
Chương này sẽ trình bày các cơ sở lý thuyết liên quan để làm nên tảng cho các mục tiêu của đề tài. Trong đó phần đầu tiên sẽ trình bày về thành phần cũng như quá trình hoạt động của một trình biên dịch. Phần tiếp theo sẽ nói về cấu trúc của một Trình biên dịch ngược và cơ chế hoạt động của nó. Chi tiết về Framework Boomerang Decompiler – một trình biên dịch ngược được sử dụng trong quá trình thực hiện đề tài cũng sẽ được nêu rõ trong phần này. Phần cuối cùng sẽ tập trung vào kỹ thuật Xác định hàm nguyên mẫu dựa vào kỹ thuật Phân tích dòng dữ liệu và dựa vào Quy ước gọi hàm.

I. Trình biên dịch (Compiler)

1. Giới thiệu

Trình biên dịch (Compiler) là một chương trình máy tính được dùng để chuyển đổi một chương trình viết ở ngôn ngữ bậc cao thành một chương trình viết ở ngôn ngữ bậc thấp[i]. Những chương trình bậc cao là những chương trình được viết trên ngôn ngữ gần gũi, dễ đọc và dễ hiểu với người lập trình. Còn những chương trình bậc thấp thì có thể được tạo nên bởi những đoạn mã gần với mã máy (như mã Assembly), hoặc là những đoạn mã máy mà máy tính có thể hiểu trực tiếp được (như mã Nhị Phân, Thập Lục Phân).

Ví dụ sau đây mô tả về quá trình biên dịch một chương trình viết trên ngôn ngữ lập trình C thành chương trình mã nhị phân:



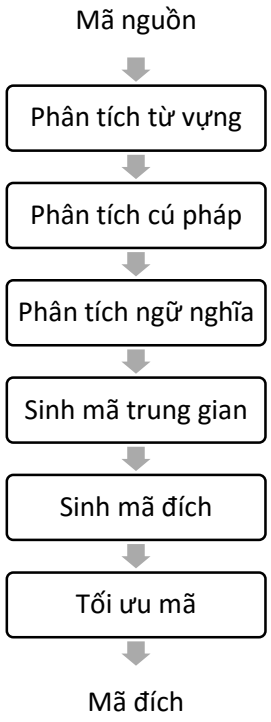
Hình B.1. Ví dụ về biên dịch mã C thành mã nhị phân.

Trong hình B.1, hàng đầu tiên là đoạn mã được viết bằng ngôn ngữ lập trình C, hàng thứ hai chứa đoạn mã nhị phân là kết quả của quá trình biện dịch đoạn mã C ban đầu. Có thể thấy, đoạn mã ở dòng thứ nhất được viết ở dạng ngôn ngữ rất dễ hiểu và dễ đọc với người lập trình so với đoạn mã nhị phân mà máy tính có thể hiểu trực tiếp được.

2. Các giai đoạn của một Trình biên dịch

Các giai đoạn chính của một trình biên dịch gồm có: Phân tích từ vựng (Lexical Analysis), Phân tích cú pháp (Syntax Analysis), Phân tích ngữ nghĩa (Semantic Analysis), Sinh mã trung gian (Immediate Code), Sinh mã đích (Code Generation) và Tối ưu mã (Optimization).[i]

Hình B.2 mô tả trình tự các giai đoạn của một Trình Biên Dịch:

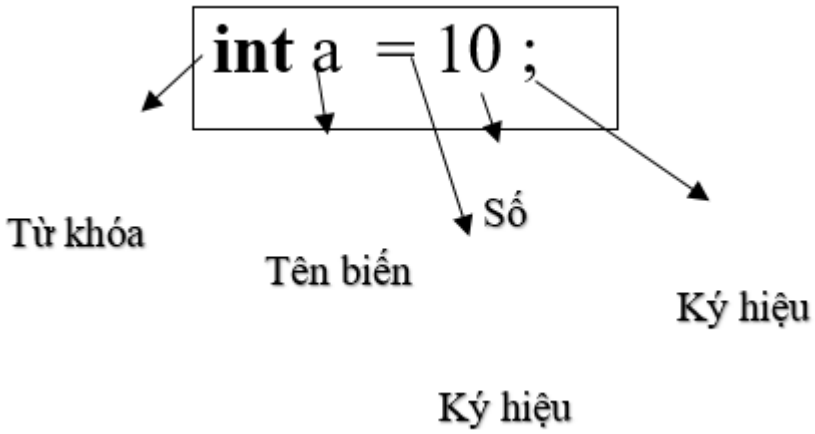


Hình B.2 Các giai đoạn của một Trình biên dịch.

a. Phân tích từ vựng

Giai đoạn này sẽ xem xét từng mẫu tự xuất hiện trong đoạn mã chương trình và phân nhóm chúng thành những đơn vị cú pháp gọi là từ tố (token). Từ tố trong ngôn ngữ lập trình, có thể bao gồm những đối tượng như tên biến, số, từ khóa, ký hiệu, ...

Ví dụ về các từ tố trong một câu lệnh viết bằng ngôn ngữ lập trình C:



Hình B.3. Ví dụ về các từ tố trong một câu lệnh lập trình.

Hình B.3 mô tả một câu lệnh về khai báo và gán giá trị cho một biến của ngôn ngữ lập trình C. Sau khi phân tích từ vựng, các ký tự sẽ được phân nhóm thành các từ tố riêng biệt. Với từ 'int', đây là một từ khóa về kiểu số nguyên được quy định trong ngôn ngữ lập trình C [(2)], nên được phân loại với nhóm Từ Khóa**.** Với chữ cái 'a', không thuộc về các từ khóa đã quy định trước, sẽ được phân vào nhóm Tên Biến. Chữ số '10' sẽ thuộc về nhóm "Số" và các Ký hiệu sẽ gồm dấu '=' và dấu ','. Ngoài ra, giai đoạn phân tích từ vựng còn nhận biết được các dấu khoảng trắng giữa các từ tố, dấu xuống dòng và các dấu khoảng cách.

Các nhóm các từ tố được xây dựng dựa trên đặc tả viết bằng Biểu Thức Chính Quy. Biểu thức chính quy là một chuỗi đặc biệt mô tả một tập hợp những chuỗi xác định khác [(1)]. Ví dụ như với tập hợp của chuỗi các số nguyên dương sẽ bao gồm các ký số từ số 0 đến số 9 kết hợp lại với nhau. Biểu Thức Chính Quy mô tả chuỗi này sẽ có dạng:

```
(0|1|2|3|4|5|6|7|8|9)+
```

Hình B.4. Biểu thức chính quy của chuỗi số nguyên dương.

Trong hình B.4, chuỗi ở giữa dấu '(' và dấu ')' là tập hợp các ký số từ 0 đến 9. Dấu '|' có ý nghĩa chọn lựa (hoặc là ký số 0, hoặc là ký số 1, …). Dấu '+' quy định biểu thức ở trước nó phải được lặp lại ít nhất là một lần. Tức là phải có ít nhất một ký số từ 0 đến 9 xuất hiện. Với biểu thức chính quy như vậy, các chuỗi sau đây là tương xứng với đặc tả: '1', '123', '1000'. Các chuỗi như 'a1', 'aa' là không tương xứng vì chứa ký tự chữ cái.

b. Phân tích cú pháp

Những từ tố được sinh ra ở giai đoạn Phân tích từ vựng sẽ được kết hợp lại với nhau để tạo thành dạng biểu diễn phản ánh cấu trúc của đoạn mã ban đầu. Dạng biểu diễn đó được gọi là Cây Cú Pháp hoặc là Cây Cú Pháp Trừu Tượng. Đúng như tên gọi của nó, Cây Cú Pháp là một cây trúc cây, gồm những nốt nội và các nốt lá kết hợp lại với nhau theo những quy luật nhất định[i].

Để biểu diễn các quy luật của đoạn mã ban đầu, dạng biểu diễn được sử dụng đó là Ngữ pháp phi ngữ cảnh. Ngữ pháp phi ngữ cảnh là tập hợp các luật đệ quy được sử dụng cho việc mô tả những chuỗi xác định[i]. Nó gồm có những thành phần như:

- Tập hợp *ký hiệu đầu cuối*, là tập hợp những mẫu tự xuất hiện trong đoạn mã, và là những nốt lá trong một cấu trúc cây.
- Tập hợp *ký hiệu không-đầu cuối*, là tập hợp các cái tên đại diện cho các *ký hiệu đầu cuối*, và là các nốt nội trong một cấu trúc cây,
- Tập hợp *phép sinh*, là tập hợp các quy luật về việc thay thế một *ký hiệu không đầu cuối* bằng một một *ký hiệu không đầu cuối* khác, hoặc bằng một *ký hiệu đầu cuối*.
- *Ký hiệu bắt đầu*, là một *ký hiệu không đầu cuối*. Đây là điểm bắt đầu của một Ngữ pháp phi ngữ cảnh. Ở trong một cấu trúc cây, *Ký hiệu bắt đầu* là nốt gốc.

Ví dụ về Ngữ pháp phi ngữ cảnh cho một đoạn văn bản viết bằng ngôn ngữ lập trình C sau:

```
int a = 10 ;
int b = 15 ;
```

Hình B.5. Đoạn mã C ví dụ

Đoạn mã C ở bảng B.5 bao gồm các dòng lệnh khai báo và gán giá trị cho một biến kiểu số nguyên dương. Mỗi dòng lệnh đều có cấu trúc như sau:

TỪ KHÓA TÊN BIẾN '=' SỐ NGUYÊN ';'

Hình B.6. Cấu trúc một dòng lệnh

Ngữ pháp phi ngữ cảnh cho đoạn mã ở hình B.5 có thể được biểu diễn ở dạng:

Program:
    Lines
;
Lines:
    Line
    | Lines Line
;
Line:
    KEYWORD ID '=' INTEGER ';'

A

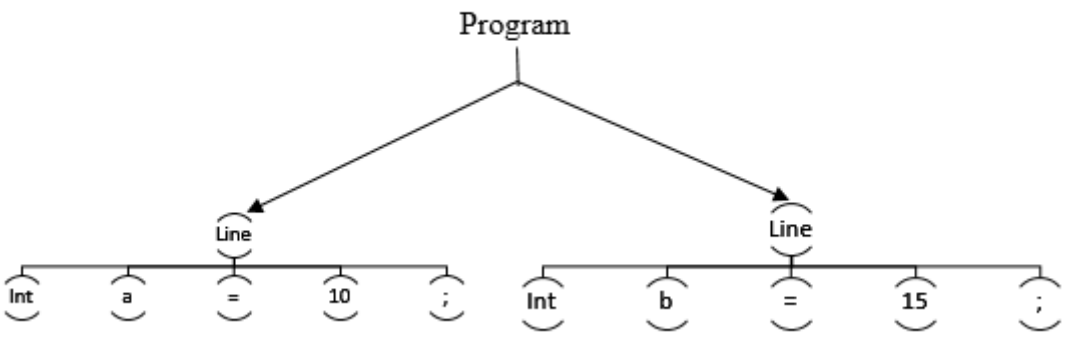
B

Hình B.7. Dạng biểu diễn ngữ pháp phi ngữ cảnh cho văn bản ở hình B.5

Trong hình B.7, 'Program' là *ký hiệu bắt đầu*, 'Lines' và 'Line' là các *Ký hiệu không kết thúc*. 'KEYWORD', 'ID', '=', 'INTEGER', '**;**' là các *Ký hiệu kết thúc*. Phần A đặc tả ý nghĩa đoạn văn bản sẽ gồm có các dòng lệnh ('Lines'). Phần B mô tả ý nghĩa các dòng lệnh sẽ gồm một hoặc nhiều lệnh ('Line'). Ở phần này, cấu trúc đệ quy được thể hiện ở dòng '| Lines Line'. Phần C đặc tả về cấu trúc của một dòng lệnh, gồm có các *Ký hiệu kết thúc*, vốn là các từ tố được sinh ra ở giai đoạn Phân Tích Từ Vựng. Các *Phép sinh* gồm có:

- Phần A, 'Program' có thể thay thế bởi 'Lines'
- Phần B, 'Lines' có thể thay thế bởi 'Line' hoặc là 'Lines Line'
- Phần C, 'Line' có thể thay thế bởi tập hợp theo thứ tự các từ tố 'KEYWORD', 'ID', '=', 'INTEGER', '**;**'

Kết thúc giai đoạn Phân Tích Cú Pháp, Trình biên dịch sẽ tạo ra Cây Cú Pháp. Với ví dụ có ở bảng B.8, Cây Cú Pháp sẽ có dạng:



Hình B.8. Cây Cú Pháp của được sinh ra của đoạn mã ở bảng B.5.

c. Phân tích ngữ nghĩa

Phân tích ngữ nghĩa là giai đoạn kiểm tra chương trình về mặt ngữ nghĩa bằng cách phân tích Cây Cú Pháp sinh ra ở giai đoạn Phân tích cú pháp [3].

Một chương trình đúng về ngữ pháp chưa chắc là một chương trình đúng về ngữ nghĩa. Ví dụ như với đoạn mã:

```
Char a = 10;
```

Hình B.9. Đoạn mã ví dụ.

đúng về mặt ngữ pháp đặc tả ở hình B.7 nhưng lại sai về mặt ngữ nghĩa do từ khóa 'Char' trong ngôn ngữ lập trình C được dùng để khai báo biến kiểu ký tự một byte [2].

Giai đoạn Phân Tích Ngữ Nghĩa bao gồm việc *kiểm tra ngữ nghĩa tĩnh* và *kiểm tra ngữ nghĩa động.*

Kiểm tra ngữ nghĩa tĩnh gồm các công việc như:

- Kiểm tra kiểu
- Kiểm tra biến đã khai báo trước sử dụng
- Kiểm tra định danh được sử dụng trong ngữ cảnh phù hợp
- Kiểm tra tham số truyền vào một hàm
- Kiểm tra nhãn
- …

Kiểm tra ngữ nghĩa động gồm có:

- Kiểm tra lỗi tính toán (ví dụ như phép chia cho 0)
- Kiểm tra lỗi biến được sử dụng nhưng chưa được khởi tạo giá trị
- ….

d. Sinh mã trung gian

Sau khi kiểm tra tính đúng đắn của chương trình về mặt ngữ pháp và ngữ nghĩa. Trình biên dịch sẽ tạo ra những đoạn mã trung gian có ý nghĩa tương tự như đoạn mã ban đầu. Những đoạn mã trung gian thường này có cấu trúc đơn giản và gần với đoạn mã chương trình đích. Ví dụ như việc biên dịch mã C thành mã nhị phân thường sử dụng mã Assembly làm mã trung gian. Sau đó, đoạn mã Assembly sẽ được dịch về đoạn mã nhị phân cuối cùng.

Ưu điểm của giai đoạn sinh mã trung gian đó là nó có thể giảm thiểu chi phí của quá trình biên dịch [1]. Khi một trình biên dịch phải biên dịch cho nhiều kiến trúc máy khác nhau, thì vấn

đề cần làm đó là chỉ cần thay đổi việc biên dịch của mã trung gian (viết nhiều phiên bản) về đoạn mã máy cuối cùng. Không cần phải thay đổi những giai đoạn ở trước đó.

Ngoài ra, giai đoạn này còn giúp trình biên dịch giảm thiểu chi phí khi phải biên dịch nhiều ngôn ngữ lập trình khác nhau [i]. Vấn đề cần giải quyết đó là chuyển đổi các ngôn ngữ về một mã trung gian chung xác định, không cần thiết phải viết nhiều phiên bản mã trung gian.

e. Sinh mã đích

Đây là quá trình tạo ra mã đích từ mã trung gian sinh ra từ giai đoạn trước đó. Quá trình này gồm những nhiệm vụ chính như:

- Lựa chọn câu lệnh: quyết định câu lệnh trung gian nào sẽ được hiện thực
- Cấp phát thanh ghi và các phép gán: quyết định giá trị nào được giữ ở trong thanh ghi.
- Trật tự câu lệnh: quyết định trật tự thực thi của các câu lệnh

Kết thúc giai đoạn này, Trình Biên Dịch sẽ sinh ra đoạn mã đích cuối cùng.

f. Tối ưu mã

Tối ưu mã tập hợp những kỹ thuật nhằm cải thiện đoạn mã đích cuối cùng bằng cách làm cho nó ít tốn tài nguyên (bộ nhớ, CPU) hơn và nâng cao tốc độ thực thi.

Việc tối ưu mã có thể không nằm ở giai đoạn cuối cùng của quá trình biên dịch. Nó có thể nằm ở giai đoạn mã ban đầu, ở chương trình gốc, khi người lập trình hiện thực bằng những giải thuật tối ưu. Nó cũng có thể nằm ở giai đoạn sinh ra mã trung gian, nơi trình biên dịch có thể chỉnh sửa mã bằng cách tính toán địa chỉ và cải thiện những vòng lặp [(4)].

Những công việc chủ yếu của giai đoạn tối ưu mã đó là:

- Loại bỏ *mã chết:* loại bỏ những đoạn mã không bao giờ được thực thi hoặc có thực thi nhưng kết quả không bao giờ được sử dụng.
- Tối ưu vòng lặp
- Loại bỏ dư thừa

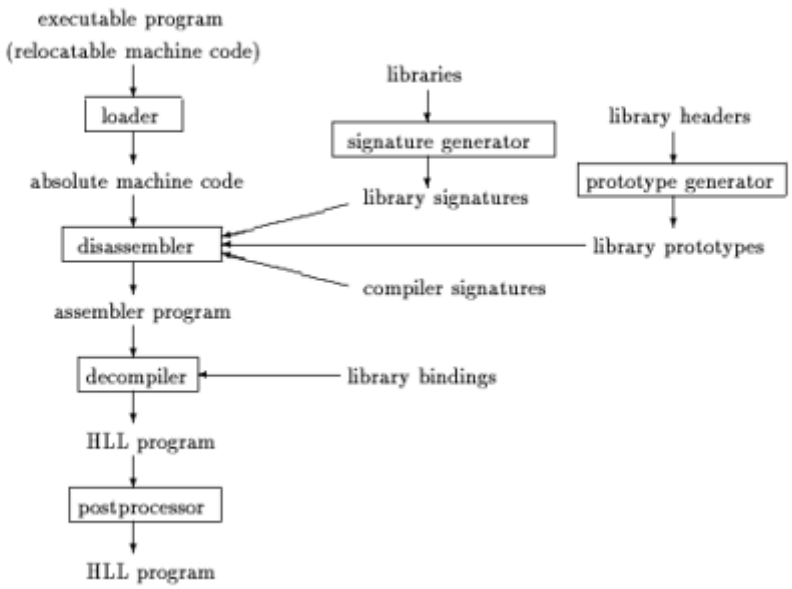
II. Trình biên dịch ngược và Framework Boomerang Decompiler

1. Hệ thống Trình Biên Dịch Ngược (Decompiler)

Trình Biên Dịch Ngược là một phần mềm đặc biệt thực hiện chực năng đảo ngược của một Trình Biên Dịch điển hình. Đó là chuyển đổi một đoạn mã ở ngôn ngữ bậc thấp lên thành đoạn mã ở ngôn ngữ bậc cao. Hệ thống của một Trình Biên Dịch ngược gồm có 4 thành phần chính:

Kỹ thuật dịch ngược

Loader, Disassembler, Decompiler, và Post Processor và 2 thành phần hỗ trợ cho phần Disassembler: Signature generator, Prototype generator[ii]. Hình B.10 mô tả hệ thống đó.



Hình B10. Mô hình một hệ thống biên dịch ngược

Loader: Loader là phần sẽ nạp những thông tin đoạn mã máy (thường là mã nhị phân) vào hệ thống. Đó là các thông tin về cấu trúc tập tin nhị phân, số lượng bộ nhớ để chạy trong chương trình hay địa chỉ của các câu lệnh, giá trị thanh ghi khởi tạo. Tùy theo loại máy và hệ điều hành mà cấu trúc một tập tin nhị phân có thể khác nhau.

Dù vậy, một chương trình nhị phân bất kỳ cũng đều có 3 thành phần chính: phần đầu (Header), bảng tái định địa chỉ (Relocation Table), ảnh nhị phân (Binary Image). Header chứa thông tin dùng để xác định kiểu của chương trình nhị phân (như ELF, DOSW64). Binary Image bao gồm các ảnh liên kết của chương trình được nạp vào trong bộ nhớ. Relocation Table: chứa độ dời địa chỉ của câu lệnh trong mã nhị phân và câu lệnh được nạp vào trong bộ nhớ. Kết thúc giai đoạn này, hệ thống Decompiler sẽ có được những đoạn mã máy thuần túy (Absolute Machince Code).

Signature Generator: Đây là một chương trình sinh ra các Signature từ đoạn mã máy. Một signature là một mẫu mã nhị phân được dùng để nhận dạng virus, trình biên dịch, thư viện hay chương trình con. Phần lớn mục đích của giai đoạn này đó là nhận dạng các thư viện hay chương trình con được gọi trong thời gian thực thi (ví dụ như gọi hàm printf để xuất ra màn hình). Hệ thống biên dịch ngược sẽ gom các thông tin về các thư viện liên quan (như library signatures, library prototypes, compiler signatures) để sử dụng cho giai đoạn tiếp theo.

Kỹ thuật dịch ngược

Library Prototype Generator: Là một chương trình tự động tạo ra thông tin nguyên mẫu (Prototype) của một hàm thư viện, giúp Trình Biên Dịch Ngược kiểm tra được đúng kiểu và tham số của hàm thư viện đó. Ví dụ, đoạn mã Assembly dưới đây dùng hàm thư viện 'PRINTF' để xuất ra màn hình giá trị 42:

```
MOV AX, 42
PUSH AX
CALL PRINTF
```

Hình B.11: Đoạn mã ví dụ.

Sau quá trình phân tích, trình biên dịch nhân biết được một hàm thư viên tên PRINTF và có một tham số truyền vào, giá trị tham số là 42.

Disassembler: Disassembler là một chương trình chuyển đổi chương trình viết bằng ngôn ngữ máy sang chương trình đích ở dạng mã Assembly. Thành phần có chức năng tương tự giai đoạn Sinh Mã Trung Gian của Trình Biên Dịch, đó là tạo ra những mã trung gian như Assembly, dùng cho giai đoạn sinh mã đích.

Library Bindings: Library Bindings là thành phần ràng buộc những hàm thư viện của một ngôn ngữ đối với một ngữ khác. Trong quá trình biên dịch lại chương trình đích của quá trình biên dịch ngược (chương trình kết quả sau khi dịch ngược), Library Bindings sẽ hộ trợ trong việc thực thì các lệnh gọi hàm thư viện.

Decompiler: Decompiler là thành phần sinh ra ngôn ngữ bậc cao (HLL- High Level Language) tương ứng về ngữ nghĩa đối với đoạn mã trung gian tạo ra bởi Disassembler.

Postprocessor: Postprocessor tạo ra đoạn mã cuối cùng của hệ thống Trình Biên Dịch Ngược bằng việc thực hiện các quá trình tối ưu mã. Ví dụ như đoạn mã sinh ra ở giai đoạn Decompiler ở hình B.12 sẽ chuyển thành đoạn mã kết quả ở hình B.13.

```
loc1 = 10;
while (loc1 < 20) {
        loc1 = loc1 +1;
        //code;
}
```

Hình B.12: Đoạn mã ví dụ.

```
for (loc1 = 10; loc1<20; loc1 ++)
{
    //code;
}
```

Hình B.13. Đoạn mã kết quả sau quá trình chuyển đoạn mã ở hình B.12

Có thể thấy ở giai đoạn này đoạn mã đích đã được tối ưu hóa bằng cách thay các cấu trúc điều khiển bằng cách cấu trúc đặc tả ngôn ngữ.

2. Framework Boomerang Decompiler

Dự án Boomerang Decompiler là một dự án mã nguồn mở do QuatumnG và Mike Van Emerik cùng các cộng sự phát triển. Hiện tại dự án không còn được tiếp tục phát triển, lần cập cập nhật cuối cùng được công bố là vào tháng 11 năm 2006[iii].

Chức năng chính của Boomerang cũng tương tự như chức năng của một Trình Biên Dịch Ngược điển hình. Đó là nhận đầu vào là một chương trình thực thi, sau đó xử lý và chuyển chương trình đó thành một chương trình ở dạng ngôn ngữ bậc cao. Ở đây, Boomerang dịch ngược một tập tin nhị phân lên thành một tập tin ở ngôn ngữ lập trình C.

Mục tiêu mà Boomerang muốn nhắm tới là tạo ra một Trình Biên Dịch Ngược có thể dễ dàng sữa đổi cho phù hợp với đa nền tảng. Tức là người sử dụng có thể biên dịch ngược được nhiều loại mã máy khác nhau như X86-windows, sparc-solaris, … mà không tốn quá nhiều công sức chuyển đổi và thử nghiệm.

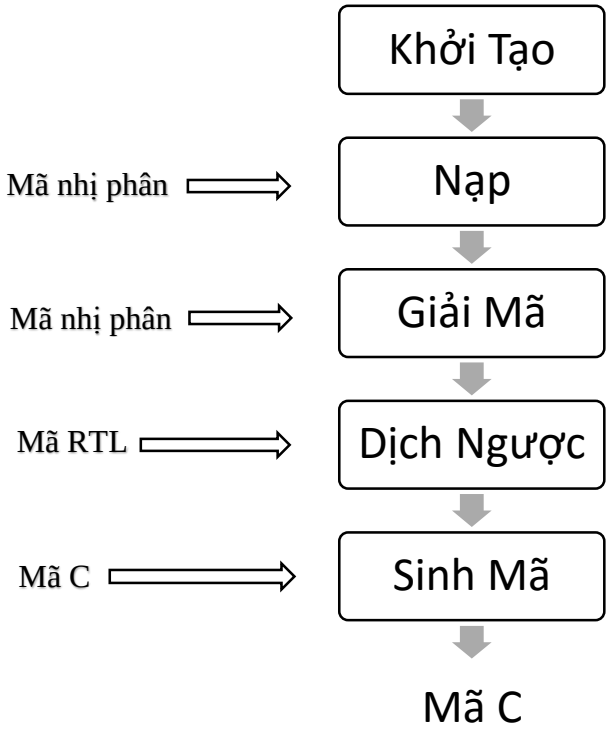
Boomerang độc lập với những hành vi của một Trình Biên Dịch điển hình nhờ việc chuyển đổi ngữ nghĩa của những câu lệnh và sử dụng những kỹ thuật mạnh mẽ như kỹ thuật phân tích "Static Single Assignment DataFlow". Việc tối ưu hóa của Trình Biên Dịch cũng sẽ không ảnh hưởng đến kết quả cuối cùng.

Ngôn ngữ phát triển: Boomerang được xây dựng trên ngôn ngữ lập trình hướng đối tượng C++.

Các dòng máy được hỗ trợ: Boomerang hiện có thể hỗ trợ việc biên dịch ngược chương trình trên các máy: X86, Sparc, Power PC.

3. Quá trình hoạt động của Framework Boomerang

Hình B.14 mô tả quá trình hoạt động của của hệ thống Boomerang. Gồm có 5 giai đoạn chính: Khởi Tạo (Initialize), Nạp (Load), Giải Mã (Decode), Dịch Ngược (Decompile) và Sinh Mã (Generate Code)[iii].



Hình B.14. Mô hình hoạt động của Framework Boomerang.

a. Khởi Tạo

Đây là giai đoạn khởi tạo của hệ thống. Boomerang sẽ nhận thiết lập những thông số đầu vào của người dùng như bật tắt chế độ Debug, sinh cây AST, hiển thị mã RTL, … để chuẩn bị bước vào giai đoạn Nạp.

b. Nạp

Đây là giai đoạn mà hệ thống sẽ nhận tập tin nhị phân đầu vào. Nhận diện kiểu tập tin nhị phân (ví dụ như là ELF hay DOS…), sau đó nhận diện kiến trúc máy tương ứng (ví dụ như kiến trúc Sparc, Pentium…). Sau khi có được các thông tin cần thiết, hệ thống sẽ nạp tập lệnh của tập tin nhị phân cũng như các tập tin thư viện liên quan. Tiếp đến, Boomerang sẽ phân tích tập tin đặc tả kiến trúc máy (Semantic Specification Language) để có được nhưng thông tin cần thiết cho phần quá trình Giải Mã.

c. Giải Mã:

Giải Mã có nhiệm vụ chuyển đoạn mã nhị phân lên thành một dạng mã trung gian. Ở giai đoạn này, hệ thống sẽ sử dụng công cụ NJMC Toolkit để chuyển tập lệnh nhị phân đã nạp lên trước đó sang một dạng mã trung gian cấp cao gọi là RTL (Register Transfer Language). Đây là dạng mã mà giai đoạn Dịch Ngược sử dụng để chuyển lên thành mã C.

d. Dịch Ngược:

Ở giai đoạn này, hệ thống Boomerang sẽ phân tích và xử lý luồng hoạt động của đoạn mã RTL sinh ra ở phần trước để tạo ra những đoạn mã C tương ứng. Bước đầu tiên là phân tích luồng dữ liệu (dataflow analysis) và tối ưu hóa. Quá trình này sẽ loại bỏ các câu lệnh không cần thiết (không được gọi hay không được sử dụng), và với các thuật toán phân tích, quá trình này sẽ tái hiện lại các thông tin về dữ liệu như các biểu thức, tham số và kết quả trả về, kiểu biến. Sau khi tối ưu, hệ thống sẽ tiến hành phân tích luồng điều khiển (control flow analysis) để có thông tin về cấu trúc điều khiển ngôn ngữ cấp cao (như if else, while, go to…), và cuối cùng sẽ tạo ra các luồng điều khiển tương ứng.

e. Sinh Mã:

Sau khi kết thúc quá trình phân tích các luồng dữ liệu, các lệnh trung gian trong Khối cơ bản (Basic Block) sẽ được chuyển thành các câu lệnh cấp cao. Những câu lệnh rẽ nhánh, điều khiển cấu trúc phụ thuộc vào cấu trúc đồ thị điều khiển (Control Flow Graph).

Giai đoạn Sinh Mã cũng là giai đoạn kết thúc luồng hoạt động của hệ thống Boomerang. Chương trình sẽ tạo ra kết quả cuối cùng là các đoạn mã C có ngữ nghĩa giống với mã nguồn nhưng có sự khác biệt về tên (như tên biến, tên hàm…) và cấu trúc chương trình (do ảnh hưởng của việc tối ưu hóa).

III. Kỹ thuật xác định hàm nguyên mẫu

1. Hàm nguyên mẫu (function prototype)

Hàm nguyên mẫu là các thông tin của một hàm bao gồm tên của hàm, kiểu dữ liệu trả về, số lượng các đối số, kiểu dữ liệu và các đối số đó[iv]. Hàm nguyên mẫu thường được khai báo ở ngôn ngữ cấp cao dưới dạng:

```
data_type function_name(type_1 argument_1, type_2 argument_2, … , type_n argument_n);
```

Hình B.15. Cú pháp khai báo hàm nguyên mẫu ở ngôn ngữ cấp cao.

Ví dụ:
```
int add(int a, int b);
```

Hiện nay, các công cụ dịch ngược có các cơ chế để xác định hàm nguyên mẫu khác nhau. Trong đó, hai cơ chế xác định hàm nguyên mẫu nổi bật là dựa trên phân tích luồng dữ liệu và quy ước gọi hàm. Hai cơ chế xác định đó sẽ được phân tích ở phần bên dưới.

2. Xác định hàm nguyên mẫu dựa trên kỹ thuật phân tích dòng dữ liệu [v]

a. Phân tích dòng dữ liệu (Data Flow Analysis)

Phân tích dòng dữ liệu là quá trình phân tích các mã tĩnh để có thể trích xuất được các thông tin về hoạt động của chương trình[vi].

Như đã nói ở trên, giai đoạn phân tích dòng dữ liệu sẽ quan tâm đến việc định nghĩa và sử dụng các địa chỉ, trong đó có việc phân tích và tìm ra tham số cũng như kết quả trả về, nghĩa là xác định hàm nguyên mẫu.

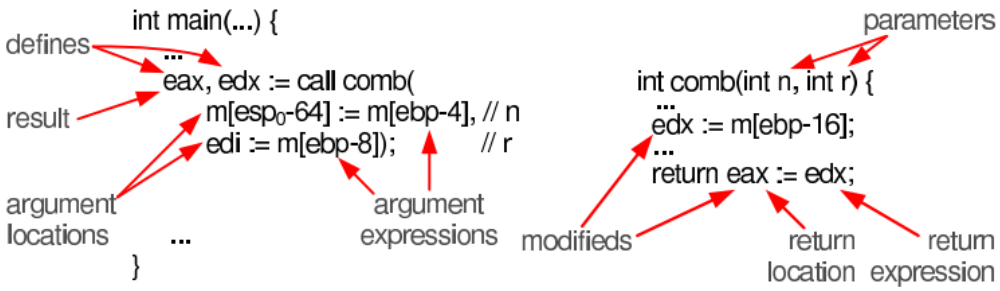
b. Các định nghĩa cần biết

Vị trí dữ liệu (location): là một địa chỉ có thể gán giá trị vào. Nó có thể là một thanh ghi, hoặc là địa chỉ trên bộ nhớ. Các vị trí dữ liệu này tương ứng với các biến ở ngôn ngữ cấp cao. VD: %esp (thanh ghi), m[sp - 4](ô nhớ tại vị trí có offset = -4 so với con trỏ-chồng)

Tính sống (live) của vị trí dữ liệu: một vị trí dữ liệu được gọi là sống tại vị trí x khi giá trị của nó tại x ảnh hưởng đến chương trình. Ví dụ: a = 3(*); b = a; print b; thì tại *, a là sống. Còn trong trường hợp a = 3; a=b; print b; thì tại *, a không sống;

Biến cục bộ (local variable): các vị trí dữ liệu nằm trên bộ nhớ chồng (stack). Các biến này chỉ có giá trị trong tầm vực nhất định.

Biến toàn cục (global variable): Biến toàn cục có thể là thanh ghi hoặc vị trí dữ liệu trên bộ nhớ. Các biến này không thuộc hàm hay thủ tục nào cả.



Hình B.16: Ví dụ về vị trí thay đổi, tham số, vị trí trả về…

Vị trí thay đổi (modifieds) của một hàm là các vị trí dữ liệu có giá trị thay đổi bởi hàm đó. Ví dụ như "edx", "eax" ở ví dụ trong hình B.14 trên.

Định nghĩa (defines) của một hàm là các vị trí dữ liệu mà hàm đó sẽ thay đổi. Ví dụ như "eax", "edx" ở ví dụ trong hình B.14 trên.

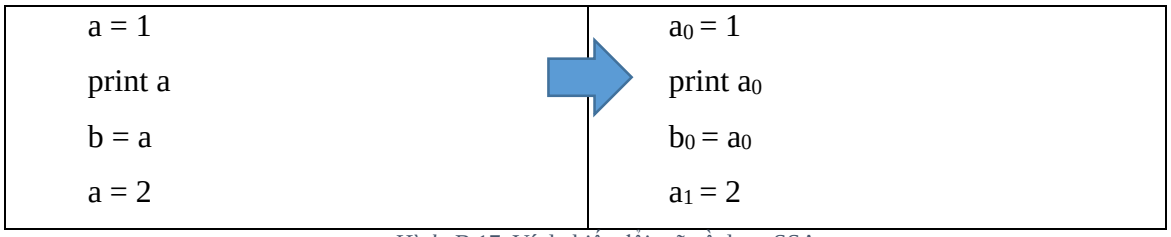
Giữ gìn (preserved): một vị trí dữ liệu được giữ gìn bởi một hàm a khi giá trị của vị trí dữ liệu đó tại thời điểm bắt đầu và kết thúc hàm đó không thay đổi. Như trong trường hợp gọi hàm,

có các thanh ghi được lưu giá trị vào bộ nhớ(như chồng), sau khi kết thúc hàm, thanh ghi đó được trả lại giá trị cũ. Không phụ thuộc vào việc thanh ghi đó được sử dụng trong hàm như thế nào.

Hàm gọi(caller) và hàm được gọi (callee): khi hàm A gọi hàm B, ta nói hàm A là hàm gọi, còn hàm B là hàm được gọi.

c. Các giai đoạn phân tích, biến đổi trước khi xử lý

Biến đổi về dạng SSA (Single Static Assignment): Dạng SSA là định dạng mà ở đó mỗi biến, hay vị trí dữ liệu chỉ được định nghĩa một lần trong suốt chương trình. Như vậy, các biến cần phải đổi tên lại theo như ví dụ trong hình B.17 sau:



Hình B.17. Ví dụ biến đổi mã về dạng SSA.

Như vậy, sau khi đổi tên, từ a thành a_0 và a_1, các biến a_0 và a_1 chỉ được định nghĩa 1 lần trong suốt chương trình.

Lan truyền biểu thức (Expression Propagation - EP): lan truyền giá trị của một biến (thanh ghi hoặc trên bộ nhớ), đến các lệnh tiếp theo, nhằm mục đích tối thiểu việc biến thiên của nhiều biến, phục vụ cho giai đoạn loại bỏ mã chết.

Loại bỏ mã chết (Deadcode Elimination - DE): Giai đoạn loại bỏ đi các đoạn mã mà không được dùng đến nữa sau giai đoạn lan truyền biểu thức.

Ngoài ra còn một số các giai đoạn phân tích khác áp dụng cho một số trường hợp riêng.

Một ví dụ dễ hiểu được nêu ra sau đây:

```
         0 esp0 := esp          ; Save esp; see text
80483b0  1 esp := esp - 4
         2 m[esp] := ebp         ; push ebp
80483b1  3 ebp := esp
80483b3  4 esp := esp - 4
         5 m[esp] := esi         ; push esi
80483b4  6 esp := esp - 4
         7 m[esp] := ebx         ; push ebx
80483b5  8 esp := esp - 4
         9 m[esp] := ecx         ; push ecx
80483b6 10 tmp1 := esp
        11 esp := esp - 8        ; sub esp, 8
80483b9 13 edx := m[ebp+8]       ; Load n to edx
```

Hình B.18: Chương trình trước khi thực hiện EP và DE

Hình B.18 thể hiện chương trình trước giai đoan lan truyền biểu thúc và loại bỏ mã chết. Trong đó thanh ghi esp có giá trị biến đổi và liên tục được sử dụng.

```
         0 esp0 := esp
80483b0  1 esp := esp0 - 4
         2 m[esp0-4] := ebp
80483b1  3 ebp := esp0-4
80483b3  4 esp := esp0 - 8
         5 m[esp0-8] := esi
80483b4  6 esp := esp0 - 12
         7 m[esp0-12] := ebx
80483b5  8 esp := esp0 - 16
         9 m[esp0-16] := ecx
80483b6 10 tmp1 := esp0 - 16
        11 esp := esp0 - 24
80483b9 13 edx := m[esp0+4]      ; Load n to edx
```

Hình B.19: Chương trình sau giai đoạn EP và trước giai đoạn DE.

Hình B.19 thể hiện chương trình sau giai đoạn lan truyền biểu thức. Giá trị của esp sẽ được lan truyền xuống các lệnh tiếp theo. Sau giai đoạn EP, việc lấy giá trị trong chồng chỉ dựa vào một thanh ghi có giá trị không đổi là esp0.

```
80483b0  2 m[esp0-4] := ebp
80483b3  5 m[esp0-8] := esi
80483b4  7 m[esp0-12] := ebx
80483b5  9 m[esp0-16] := ecx
80483b9 13 edx := m[esp0+4]      ; Load n to edx
```

Hình B.20: Chương trình sau giai đoạn DE.

Hình B.20 thể hiện chương trình sau giai đoạn loại bỏ mã chết. Trong giai đoạn này các phép gán cho esp không cần thiết nữa nên được bỏ đi.

d. Cơ chế nhận dạng tham số đầu vào

Dưới đây là các biểu thức có thể áp dụng để tìm tham số đầu vào:

$$\text{param-filter} = \{\text{non-memory expressions} + \text{local-variables}\} \quad (1)$$

$$\text{init-params}(p) = \text{live-on-entry}(p) \cap \text{param-filter} \quad (2)$$

$$\text{final-param}(p) = \text{init-params}(p) - \text{preseved}(p) \quad (3)$$

Giải thích:

- (1) **param-filter** là các điều kiện chung để vị trí dữ liệu có thể là một tham số. Đó là vị trí dữ liệu đó không phải các giá trị nằm trên bộ nhớ (**non-memory expression**), ngoại trừ là các **local-variable** (nằm trên chồng). Để dễ hiểu, đó có thể là các thanh ghi (edx…) hoặc vị trí dữ liệu trên chồng (m[esp0+4])
- (2) **init-params(p)** là danh sách các vị trí dữ liệu *có thể* là tham số của hàm p. là các vị trí dữ liệu sống tại điểm nhập của p (**live-on-entry(p)**), và thỏa mãn điều kiện 1 (**param-filter**).
- (3) **final-params(p)** là danh sách vị trí dữ liệu là tham số của hàm p. Đó là danh sách vị trí dữ liệu *có thể* là tham số của p, ngoại trừ các vị trí dữ liệu được giữ gìn bởi p (**preserved(p)**). Ở đây có thể hiểu là có một số vị trí dữ liệu có thể làm tham số, tuy nhiên điều đó là không cần thiết.

e. Cơ chế nhận dạng tham số trả về

Dưới đây là các biểu thức có thể áp dụng để tìm tham số trả về:

$$\text{ret-filter} = \{\text{non-memory expressions}\} \quad (1)$$

$$\text{modifieds}(p) = \text{reach-exit}(p) - \text{preserveds}(p) \quad (2)$$

$$\text{defines (caller)} = \text{modifieds (callee)} \quad (3)$$

$$\text{return-location}\ (p) = \text{modifieds}(p) \cap \text{ret-filter} \cap \text{live(caller)} \quad (4)$$

Giải thích:

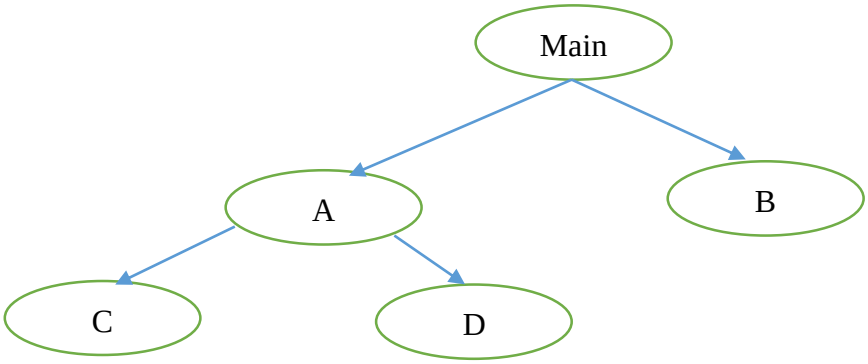
- (1) **ret-filter** là các điều kiện chung để vị trí dữ liệu có thể là một vị trí trả về. Đó là vị trí dữ liệu không phải các giá trị nằm trên bộ nhớ (**non-memory expression**).(Giá trị nằm ở thanh ghi, ví dụ như eax)
- (2) **modifieds(p)** là danh sách vị trí thay đổi của một hàm p. danh sách đó được xác định là vị trí dữ liệu có giá trị đến thời điểm kết thúc hàm p (**reach-exit(p)**). Ngoại trừ các vị trí dữ liệu được giữ gìn trong hàm p (**preserved(p)**).

- (3) **defines(caller)** là danh sách định nghĩa của hàm gọi. Đó là các vị trí thay đổi của hàm được gọi **modifieds (callee)** (Các vị trí dữ liệu eax và edx ở ví dụ trên)
- (4) **return-location (p)** là các vị trí trả về của hàm p. Đó là các giá trị vị trí thay đổi của hàm đó p (**modifieds(p)**) thỏa mãn các điều kiện 1 (**ret-filter**) và sống tại thời điểm gọi hàm **live(caller).**

f. Các xử lý khác đối với hàm đệ quy.

i. *Quy ước về các tham số hàm đệ quy.*

Khi thực hiện việc phân tích xác định hàm nguyên mẫu đối với các hàm có chứa các lệnh gọi hàm, thì chúng ta cần có các thông tin của các hàm được gọi (các tham số đầu vào và tham số trả về). Vấn đề này được giải quyết bằng thứ tự phân tích theo giải thuật Depth First Search (DFS) như ví dụ sau:



Hình B.21: Ví dụ về thứ tự phân tích các hàm

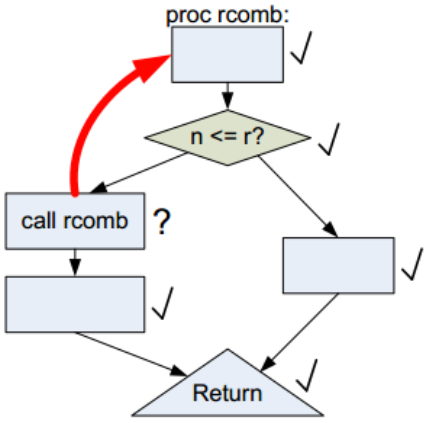
Hình B.21 thể hiện thứ tự gọi hàm của một chương trình, trong đó từ hàm Main sẽ có gọi đến hàm A và hàm B, ham A sẽ có gọi đến hàm C và hàm D. Chúng ta sẽ lần lượt phân tích theo thứ tự DFS như sau: C -> D -> A -> B -> Main. Với thứ tự đó, khi phân tích hàm A, chúng ta đã có được các thông tin cần thiết từ các hàm con (C, D), và khi phân tích hàm Main, chúng ta cũng đã có được các thông tin của các hàm A và B.

Tuy nhiên, trong trường hợp đối với hàm đệ quy, thì việc xác định các thông tin này lại không dễ dàng. Bởi vì hàm được gọi cũng là hàm mà ta đang phân tích xác định hàm nguyên mẫu. Trong trường hợp này, chúng ta sẽ giả sử các định nghĩa đến được lệnh gọi hàm sẽ là tham số, còn các vị trí dữ liệu đang sống sẽ là các vị trí trả về.

ii. *Quy ước về vị trí được giữ gìn ở hàm đệ quy.*

Việc xác định các vị trí được giữ gìn ở hàm đệ quy cũng gặp các khó khăn tương tự khi các thông tin của hàm được gọi chưa được xác định rõ ràng. Một vị trí dữ liệu được gọi là được giữ

gìn khi mà nó được giữ gìn trên tất cả các nhánh rẽ trong chương trình. Tuy nhiên, khi gọi đệ quy, thì sẽ tồn tại nhánh rẽ không có thông tin rõ ràng. Cụ thể trong ví dụ sau:



Hình B.22: Ví dụ về xác định vị trí được giữ gìn ở hàm đệ quy

Hình B.22 thể hiện luồng xử lý của hàm rcomb, trong đó các ô vuông có dấu tick (✔) là các khối lệnh mà vị trí dữ liệu đang được xem xét đã được giữ gìn. Tuy nhiên, tính được giữ gìn của vị trí dữ liệu ở lệnh gọi hàm call rcomb là vẫn chưa xác định được. Nếu vị trí dữ liệu được giữ gìn ở lệnh gọi hàm này thì sẽ được giữ gìn ở hàm rcomb, còn nếu không thì vị trí dữ liệu sẽ không giữ gìn ở cả hàm rcomp. Trong trường hợp này, chúng ta sẽ giả sử vị trí dữ liệu được giữ gìn ở lệnh gọi hàm call rcomb, và đi đến xác định xem nó có được giữ gìn ở hàm rcomb hay không.

iii. ***Loại bỏ các tham số dư thừa.***

Đối với trường hợp đệ quy, biểu thức xác định tham số (init-params(p) = live-on-entry(p) ∩ param-filter) không hoàn toàn đúng. Ở đây nói không hoàn toàn đúng bởi vì có các tham số trong cả chỉ được đưa vào để có thể làm tham số của lệnh gọi đệ quy. Việc loại bỏ các tham số dư thừa đó sẽ giúp chương trình đúng đắn và dễ đọc hơn. Ví dụ như chương trình dưới đây:

```
int, int a(p, q) {
   ...                  /* No use of p, q, r or s*/
   r, s := b(p, q);     /* b is always called by a */
   ...                  /* No use of p, q, r or s */
   return r, s;
}
int,int b(p, q) {
   if (q > 1)          /* q is used other than in a call to a */
      r, s := a(p, q-1);
   else
      ...              /* No use of p, q, r, or s */
      print(r);        /* Use r */
   return r, s;
}
```

Hình B.23: Ví dụ về hàm đệ quy có các tham số dư thừa

Ở hình B.23 trên, có bao gồm hàm a và hàm b gọi đệ quy lẫn nhau, cả 2 hàm sử dụng tham số p và q. Tuy nhiên, nếu để ý, chúng ta sẽ thấy trong 2 tham số đó, chỉ có q là thật sự được sử dụng (lệnh if (q > 1)), còn tham số p chỉ được truyền vào để có tham số truyền vào lệnh gọi đệ quy. Nếu chỉ xem xét riêng mỗi hàm a, chúng ta sẽ không thể nhận biết được p và q có phải làm các tham số dư thừa hay không. Mà chúng ta phải xem xét tất cả các hàm gọi đệ quy có liên quan (ở đây là hàm a và b), đối với mỗi tham số, nếu nó chỉ được sử dụng để truyền tham số vào lệnh gọi hàm, thì đó là tham số dư thừa.

3. Xác định hàm nguyên mẫu dựa trên quy ước gọi hàm

a. Quy ước gọi hàm (ABI Calling convention) là gì?

ABI (Application Binary Interface) là tập hợp các quy ước kết nối giữa các module ở mức mã máy, hay nói cách khác là xác định cách mà các module giao tiếp với nhau để thực hiện một mục đích chung[vii]. Trong đó có các quy ước về quy ước gọi hàm

Quy ước gọi hàm (calling convention) là phương thức chuẩn để hiện thực và gọi hàm. Trong đó có quy định các thông tin về cách truyền tham số vào hàm, cách đưa kết quả trả về ra, cách quản lý chồng…[viii] Với các quy ước này, lập trình viên không cần phải tìm hiểu mã nguồn để tìm ra cách truyền tham số vào hàm, cũng như là việc sử dụng các hàm thư viện được dễ dàng và thuận tiện hơn.

Hiện tại có rất nhiều loại quy ước gọi hàm, phụ thuộc vào từng kiến trúc máy và trình biên dịch được sử dụng … Các kiến trúc máy khác nhau (86k, x86, sparc, mips…) có các tiêu chuẩn khác nhau về việc này. Ngoài ra, đối với mỗi kiến trúc còn phụ thuộc vào trình biên dịch đang sử dụng (GCC, …).

b. Quy ước gọi hàm của một số kiến trúc máy

i. *Máy SPARC*[ix]

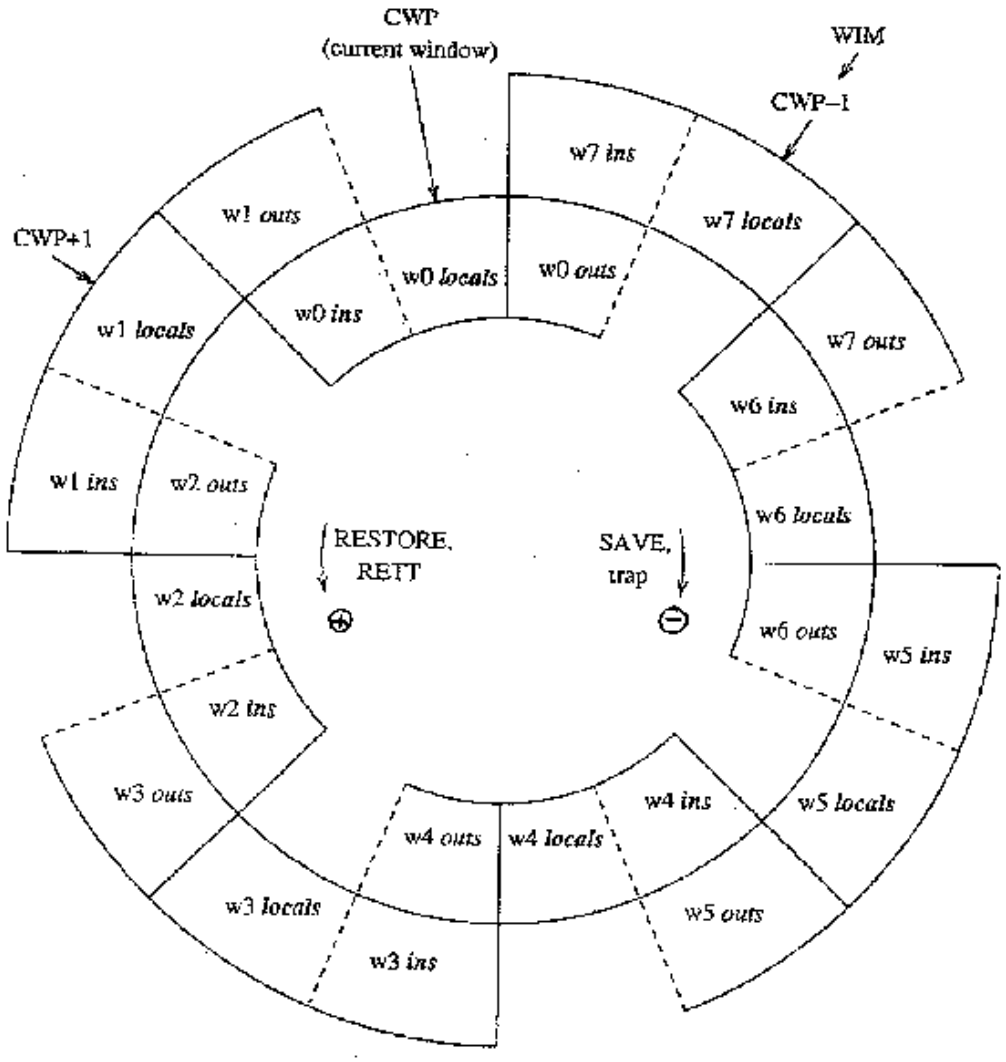
Đối với kiến trúc máy Sparc, tại mỗi thời điểm sẽ có 32 thanh ghi có thể được sử dụng, trong đó có 8 thanh ghi toàn cục(global) và 24 thanh ghi nằm trong "cửa sổ thanh ghi" (register window) như bảng B.1 sau:

Bảng B.1: Các thanh ghi kiến trúc máy Sparc

Nhóm thanh ghi	Tên gợi nhớ	Vị trí thanh ghi
Global	%g0 - %g7	r[0] – r[7]
Out	%o0 - %o7	r[8] – r[15]
Local	%l0 - %l7	r[16] – r[23]
In	%i0 - %i7	r[24] – r[31]

Theo như bảng trên, thì sẽ có 4 nhóm thanh ghi là global, out, local và in, trong đó các thanh ghi thuộc nhóm out, local và in sẽ nằm trong cửa sổ thanh ghi. Các thanh ghi được sử dụng sẽ có tên %g0-g7, %o0 - %o7, %l0 - %l7, %i0 - %i7 và sẽ nằm ở các vị trí thanh ghi số 0 đến 31. Các thanh ghi đó sẽ được sử dụng theo một số quy ước sau đây:

- Các thanh ghi %go - %g7 sẽ là các thanh ghi dùng chung trong chương trình.
- Các thanh ghi %o0 - %o5 sẽ được dùng để truyền tham số vào hàm được gọi.
- Các thanh ghi %i0 - %i5 sẽ được dùng để nhận các tham số của hàm gọi.
- Trong trường hợp có nhiều hơn 6 tham số, các tham số còn lại sẽ được truyền qua stack.
- Các tham số sẽ được truyền theo thứ tự từ phải qua trái. Ví dụ như đối với hàm func(param1, param2), thì giá trị của param2 sẽ được đưa vào trước, sau đó sẽ là param1.
- Thanh ghi %o6 sẽ là thanh ghi con trỏ chồng, và cũng được gọi là %sp.
- Thanh ghi %i6 là thanh ghi con trỏ khung(frame pointer), và cũng được gọi là %fp.
- Thanh ghi %i7 và %o7 được sử dụng để truyền địa chỉ của hàm gọi.

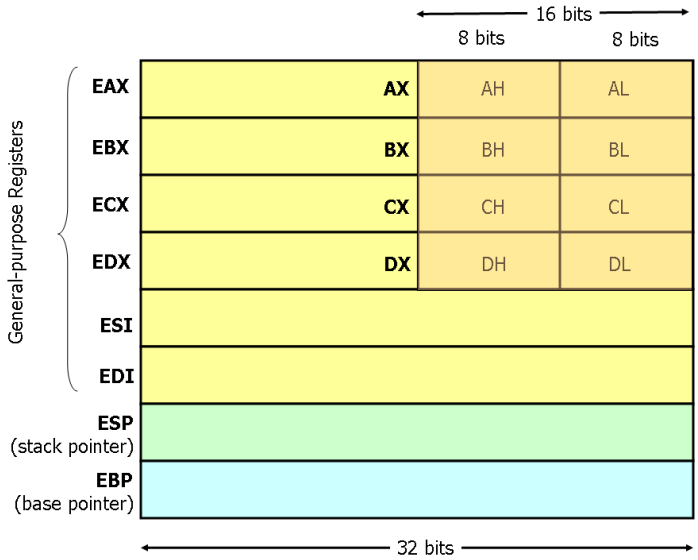


Hình B.24: Chuyển đổi con trỏ cửa sổ hiện tại ở máy Sparc

Hình B.24 thể hiện sự thay đổi của cửa sổ thanh ghi khi sử dụng lệnh SAVE và RESTORE trong việc gọi hàm. Giả sử hiện tại, CWP (Current windows pointer - con trỏ cửa sổ hiện tại) đang nằm ở vị trí w0 như hình. Sau khi gọi hàm con, chúng ta sẽ sử dụng lệnh SAVE, khi đó CWP sẽ nằm ở vị trí w1, giá trị các thanh ghi ở vị trí w0 ins sẽ truyền vào các thanh ghi w1 outs. Sau khi thực hiện các chức năng của hàm và gọi lệnh RESTORE, giá trị của các thanh ghi ở vị trí w1 outs cũng sẽ được trả về các thanh ghi w0 ins. Đó cũng là cách để truyền các tham số vào và nhận giá trị trả về.

ii. ***Máy PENTIUM^x***

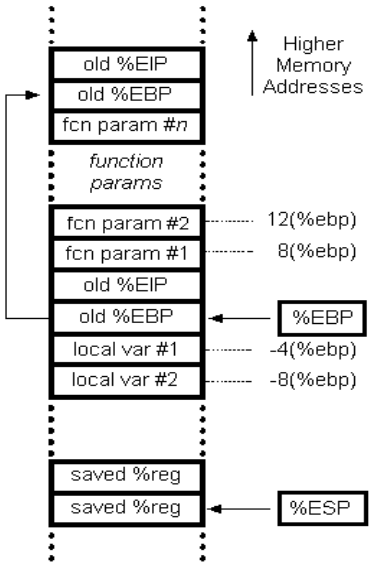
Đối với kiến trúc máy Pentium, sẽ có một số các thanh ghi thông dụng được thể hiện trong hình B.23 sau:



Hình B25: Các thanh ghi của kiến trúc máy x86(Pentium)

Theo như hình B.25 trên, các thanh ghi thường được sử dụng là các thanh ghi ESP(Stack Pointer), EBP (Base Pointer), các thanh ghi đa chức năng (General-purpose register) như là EAX, EBX, ECX, EDX, ESI, EDI. Đối với các thanh ghi này, chúng ta có thể truy xuất xuống các vùng nhớ nhỏ hơn. Ví dụ như chỉ sử dụng 16 bit đầu của thanh ghi EAX qua AX, 8 bit tiếp theo qua thanh ghi AH, và 8 bit cuối cùng qua thanh ghi AL.

Theo như các quy ước trên, thì sẽ không có thanh ghi nào được sử dụng cho việc truyền các tham số vào hàm được gọi cả, mà hàm gọi sẽ truyền tham số vào qua stack frame. Hình B.26 sau đây sẽ thể hiện việc truyền các các tham số vào hàm được gọi qua stack frame:



Hình B.26: Bộ nhớ stack trước và sau khi gọi hàm ở máy x86(Pentium)

Thực chất của việc đưa tham số vào stack frame là sử dụng lệnh push, lệnh này sẽ đưa giá trị tham số của lệnh vào stack, đồng thời giảm giá trị của thanh ghi ESP. Theo như hình trên, với việc đưa 2 biến param#1 và param#2 vào stack, giá trị của các tham số đó sẽ được truy xuất bằng cách thực hiện load giá trị tại các ô nhớ ESP + 8 và ESP + 12.

Tuy nhiên, sau khi gọi hàm, trách nhiệm của hàm gọi là lưu giá trị của thanh ghi EBP lại, đồng thời gán giá trị thanh ghi ESP của hàm gọi vào thanh ghi EBP. Do đó, thực tế việc truy xuất giá trị các tham số này được thực hiện bằng việc load giá trị tại các ô nhớ EBP + 8 và EBP + 12.

Sau khi hàm được gọi thực hiện các tác vụ cần thiết xong, giá trị trả về sẽ được trả về qua thanh ghi EAX.

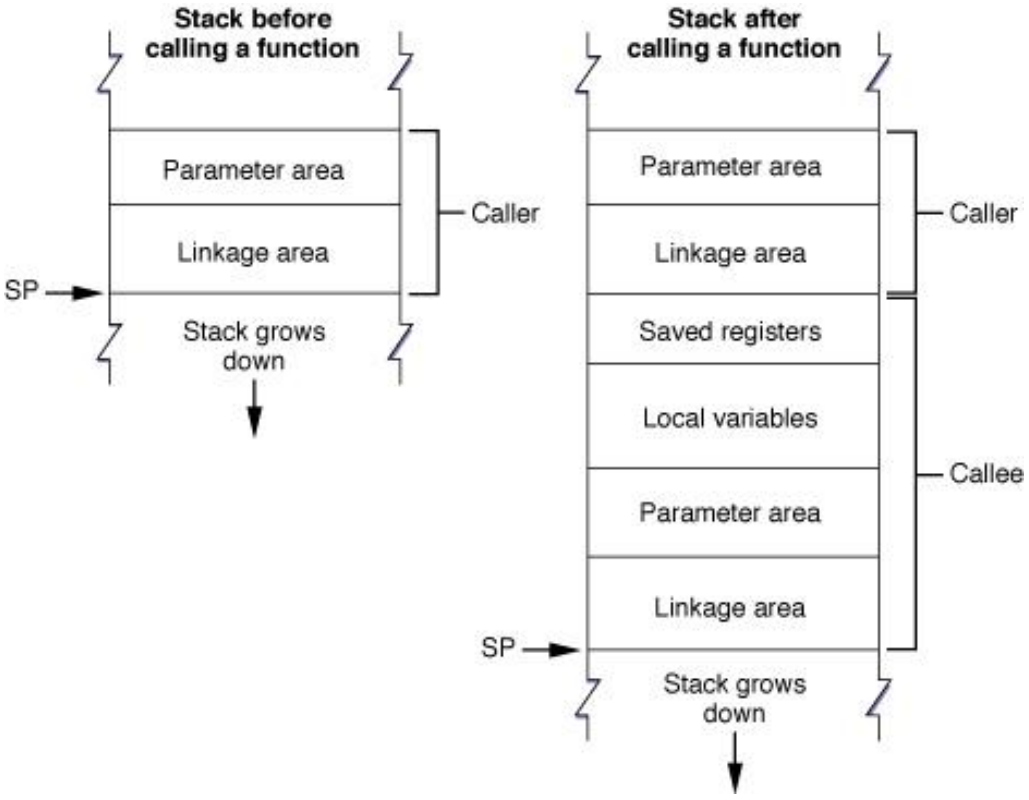
iii. ***Máy Power PC***[xi]

Cũng như các kiến trúc máy khác, máy Power PC(PPC) cũng có các thanh ghi thông dụng và được sử dụng theo một số quy ước sẵn như bảng B.2 sau:

Bảng B.2: Các thanh ghi của kiến trúc máy PowerPC

Thanh ghi	Mục đích sử dụng
GPR0	Để lưu link register cũ
GPR1	Con trỏ chồng
GPR2	Sử dụng đa mục đích
GPR3	Để truyền tham số vào, cũng như là nhận kết quả trả về
GPR4 - GPR10	Để truyền tham số vào
GPR11 – GPR12	Dùng để xử lý trong việc gọi hàm
GPR13 – GPR31	Các thanh ghi global

Hình B.27 thể hiện chồng của máy PPC trước và sau khi gọi hàm:



Hình B.27: Bộ nhớ chồng trước và sau khi gọi hàm ở máy PowerPC.

Theo như hình B.27 trên, mỗi khi hàm mới được gọi, chương trình sẽ chiếm thêm các vùng nhớ mới trong chồng để lưu trữ các thông tin liên kết, các biến cục bộ, lưu giá trị của các thanh ghi và để truyền tham số vào. Trong đó, vùng tham số(parameter area) sẽ có vai trò trong việc truyền tham số từ hàm gọi vào. Chúng ta sẽ có 1 thanh ghi để nhận giá trị trả về là GPR3 và 8 thanh ghi để truyền tham số vào là các thanh ghi từ GPR3 đến GPR10(general-purpose register). 8 thanh ghi này sẽ tương ứng với 32 byte đầu tiên của vùng tham số như trong bảng B.3 sau:

Bảng B.3: các thanh ghi và vị trí trong chồng đối với máy PowerPC

Vị trí trong stack frame	Thanh ghi
%SP + 24	GPR3
%SP + 28	GPR4
%SP + 32	GPR5
%SP + 36	GPR6
%SP + 40	GPR7
%SP + 44	GPR8
%SP + 48	GPR9
%SP + 52	GPR10

Với sự tương ứng giữa các thanh ghi vị trí của nó trong stack frame, chúng ta có thể lưu giá trị vào thanh ghi và load từ stack lên hoặc ngược lại. Ví dụ như trước khi gọi hàm, chúng ta có

thể đưa giá trị của tham số vào địa chỉ %SP + 24, và sau khi gọi hàm rồi thì sử dụng giá trị lấy được từ thanh ghi GPR3.

Trong trường hợp có nhiều hơn 8 tham số, hoặc kích thước của các tham số vượt quá 32 bytes, chúng ta có thể tiếp tục sử dụng các vùng nhớ ở stack frame có địa chỉ cao hơn như %SP + 56, %SP + 60 …

c. Ví dụ áp dụng

Lấy ví dụ dối với hàm tính tổng của bảy số được thực hiện trên máy Sparc:

```
int calculateSum(int a, int b, int c, int d, int e, int f, int g) {
    int sum = a + b + c + d + e + f + g ;
    return sum;
}
```

Hình B.28: Đoạn mã ví dụ

Để dễ hiểu thì chúng ta sẽ biên dịch mã nguồn thành mã assembly trên máy sparc như sau:

```
Main:
        save    %sp, -136, %sp
        mov     1, %g1
        st      %g1, [%fp-4]
        mov     2, %g1
        st      %g1, [%fp-8]
        mov     3, %g1
        st      %g1, [%fp-12]
        mov     4, %g1
        st      %g1, [%fp-16]
        mov     5, %g1
        st      %g1, [%fp-20]
        mov     6, %g1
        st      %g1, [%fp-24]
        mov     7, %g1
        st      %g1, [%fp-28]
        ld      [%fp-28], %g1
        st      %g1, [%sp+92]
        ld      [%fp-24], %o5
```

```
calculateSum:
        save    %sp, -104, %sp
        st      %i0, [%fp+68]
        st      %i1, [%fp+72]
        st      %i2, [%fp+76]
        st      %i3, [%fp+80]
        st      %i4, [%fp+84]
        st      %i5, [%fp+88]
        ld      [%fp+68], %g2
        ld      [%fp+72], %g1
        add     %g2, %g1, %g2
        ld      [%fp+76], %g1
        add     %g2, %g1, %g2
        ld      [%fp+80], %g1
        add     %g2, %g1, %g2
        ld      [%fp+84], %g1
        add     %g2, %g1, %g2
        ld      [%fp+88], %g1
        add     %g2, %g1, %g2
```

```
        ld      [%fp-20], %o4
        ld      [%fp-16], %o3
        ld      [%fp-12], %o2
        ld      [%fp-8], %o1
        ld      [%fp-4], %o0
        call    calculateSum, 0
         nop
        st      %o0, [%fp-32]
        ld      [%fp-32], %o1
        sethi   %hi(.LC0), %g1
        or      %g1, %lo(.LC0), %o0
        call    printf, 0
         nop
        mov     0, %g1
        mov     %g1, %i0
        restore
        jmp     %o7+8
         nop
```

```
        ld      [%fp+92], %g1
        add     %g2, %g1, %g1
        st      %g1, [%fp-4]
        ld      [%fp-4], %g1
        mov     %g1, %i0
        restore
        jmp     %o7+8
         nop
```

Hình B.29: Ví dụ áp dụng quy ước gọi hàm trong việc gọi hàm

Hình B.29 trên thể hiện lệnh assembly tuân theo quy ước gọi hàm. Phía bên trái là các lệnh của hàm main, phía bên phải là các lệnh của hàm caculateSum. Ở hàm main, tính từ lệnh gọi hàm đi lên (call calculateSum, 0), các tham số lần lượt được truyền vào các thanh ghi %o0 đến %o5 (ld [%fp-4], %o0, load giá trị từ địa chỉ [%fp-4] vào thanh ghi %o0), sau đó đến tham số thứ 7, giá trị sẽ được truyền vào chồng (st %g1, [%sp+92], lưu giá trị của g1 vào địa chỉ [%sp+92]). Nếu tính từ trên xuống, thì thứ tự các tham số được truyền vào từ phải qua trái (tham số cuối cùng trước, rồi từ từ đến tham số đầu tiên).

Tại hàm caculateSum, giá trị tính được sẽ được đưa vào thanh ghi %i0 (mov %g1, %i0, đưa giá trị từ thanh ghi %g1 vào thanh ghi %i0), và sau đó trả về thanh ghi %o0 ở hàm Main.

CHƯƠNG C
MỤC TIÊU VÀ GIỚI HẠN

Chương này sẽ trình bày khái quát các mục tiêu của đề tài Luận văn tốt nghiệp. Mục tiêu chi tiết đối với bài toán Dịch ngược mã Assembly cũng như bài toán Sinh hàm nguyên mẫu được trình bày ở phần ở phần đầu tiên. Phần kế tiếp sẽ trình bày những giới hạn chi tiết của 2 bài toán trong phạm vi Luận văn tốt nghiệp.

I. Mục Tiêu

Mục tiệu đầu tiên trong phạm vi Luận Văn Tốt Nghiệp đó là cải tiến để Boomerang chấp nhận được mã Assembly là một trong hai loại mã đầu vào cùng với mã nhị phân. Phần hiện thực của Boomerang cũng được chỉnh sửa để có thêm chức năng dịch ngược mã Assembly lên thành mã C tương ứng. Một bộ công cụ giả lập bộ công cụ New Jersey Machine Code (NJMC) được phát triển, dành riêng cho việc giải mã câu lệnh Assembly (thay vì câu lệnh nhị phân của NJMC trước đây). Ngoài ra, Boomerang còn được tích hợp thêm bộ phân tích mã Assembly (Assembly Parser) để rút trích thông tin từ tập tin Assembly đầu vào. Bộ đặc tả tập lệnh kiến trúc máy Intel-MCS-51 (8051) cũng được bổ sung vào để hỗ trợ phần dịch ngược mã Assembly 8051.

Mục tiêu thứ hai đó là nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm của các kiến trúc máy. Việc nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm có những ưu và khuyết điểm riêng đối với cơ chế nhận dạng hiện tại của Boomerang (dựa trên giải thuật). Việc áp dụng cả hai cơ chế này sẽ giúp hoàn thiện hơn việc xác định hàm nguyên mẫu, nhằm sinh ra mã nguồn đúng và dễ hiểu. Để thực hiện được mục tiêu này thì một công cụ tự động sinh mã nhận dạng hàm nguyên mẫu dựa trên đặc tả quy ước gọi hàm được xây dựng. Việc xây dựng công cụ tự động sinh mã này giúp việc hiện thực trên các kiến trúc máy một cách nhanh chóng, chính xác. Đồng thời tạo điều kiện cho việc phát triển sau này được dễ dàng hơn.

II. Giới hạn

Quá trình dịch ngược mã Assembly được giới hạn về số lượng kiến trúc máy. Có hai kiến trúc máy được thử nghiệm đó là Sparc và máy 8051. Đối với từng kiến trúc máy Sparc, một số tập lệnh và trường hợp bị giới hạn.

Đối với bài toán cải thiện kỹ thuật sinh hàm nguyên mẫu cũng có những giới hạn về số lượng kiến trúc máy. Hiện tại Boomerang chỉ hỗ trợ dịch ngược tập tin thực thi của các kiến

trúc máy Sparc, Pentium và PowerPC trên nền tảng 32bit. Vì vậy, đề tài cũng sẽ giới hạn việc hiện thực xác định tham số dựa trên quy ước gọi hàm trên các kiến trúc máy này.

CHƯƠNG D

DỊCH NGƯỢC MÃ ASSEMBLY

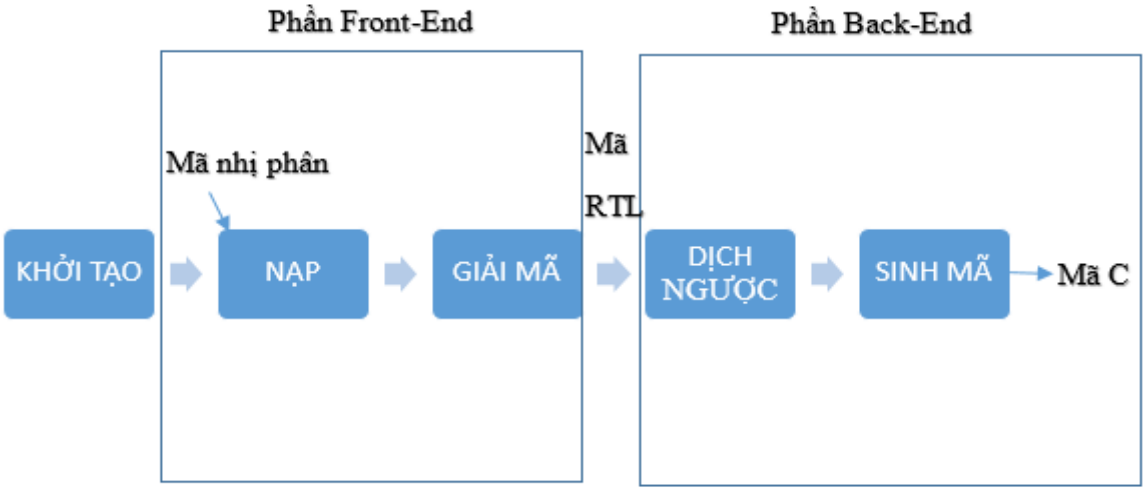
Nội dung chương này trình bày chi tiết về quá trình hiện thực chức năng dịch ngược mã Assembly dựa vào Framework Boomerang. Phần đầu tiên sẽ xác định những nhiệm vụ cần thực hiện cũng như khái quát những giải pháp khi dịch ngược mã Assemby. Phần tiếp theo nói về chi tiết hiện thực để xây dựng nên cấu trúc lưu trữ nội dung mã Assembly để sử dụng cho những giai đoạn tiếp theo. Phần cuối sẽ sẽ trình bày về việc xây dựng nên công cụ Boomerang Toolkit và quá trình chỉnh sửa Front-End để phục vụ quá trình chuyển đổi mã Assembly về mã trung trung RTL.

I. Xác định giải pháp

Trong chương I này, phần đầu tiên sẽ mô tả lại cơ chế hoạt động của hệ thống Boomerang khi dịch ngược mã nhị phân để xác định những nhiệm vụ cần phải thực hiện khi dịch ngược mã Assembly. Phần cuối sẽ trình bày về cơ chế hoạt động mới được xây dựng nhằm khái quát những giải pháp cho những nhiệm vụ đó.

1. Xác định những nhiệm vụ

Quá trình dịch ngược mã Assembly được thực hiện bằng phương pháp dựa trên nền tảng của công cụ Boomerang Decompiler. Cho nên giải pháp chi tiết cũng như nhiệm vụ cần thực hiện phụ thuộc vào cơ chế hoạt động của Boomerang. Hình D.1 khái quát lại cơ chế đó:

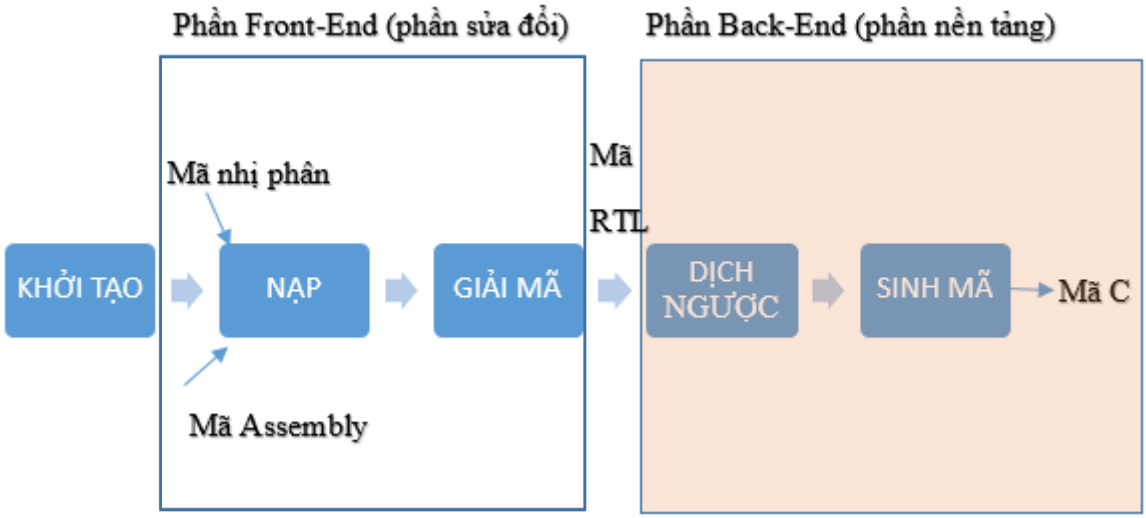


Hình D.1. Mô hình khái quát cơ chế hoạt động Boomerang.

Kỹ thuật dịch ngược

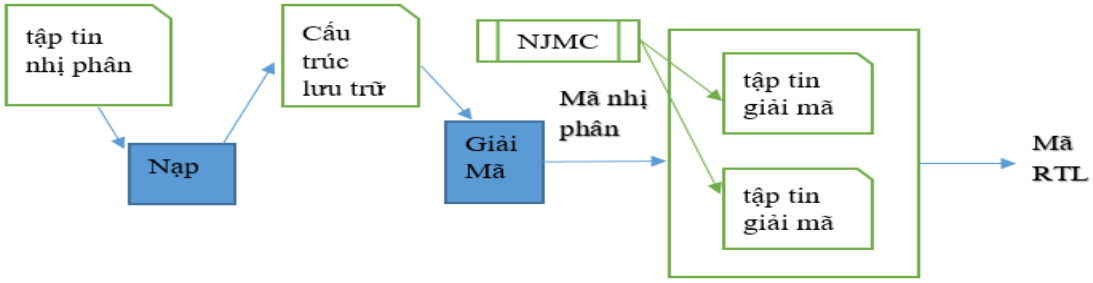
Trong mô hình D.1, giai đoạn Khởi Tạo (Initialize) chỉ dùng để hỗ trợ cho chương trình chính lúc thực thi nên không được xét đến. Giai đoạn Nạp (Load) có nhiệm vụ phân tích tập tin nhị phân và rút trích thông tin tập lệnh trong tập tin đó. Giai đoạn Giải Mã (Decode) sẽ chuyển đổi tập lệnh mã nhị phân sang mã trung gian RTL. Hai giai đoạn Nạp và Giải Mã được gộp chung vào một phần gọi là Front-End**,** hai giai đoạn Dịch Ngược (Decompile) và Sinh Mã (Generate Code) được gộp chung vào một phần gọi là Back-End**.** Phần Front-End sẽ tạo mã RTL, vốn là đoạn mã mà phần Back-End sử dụng để tạo nên mã C cuối cùng**.**

Như vậy, có thể thấy mã trung gian RTL chính là điểm mấu chốt của vấn đề. Phần Back-End không cần quan tâm đến loại mã được dịch ngược là gì (nhị phân hay Assembly) mà chỉ cần biết mã RTL là mã đầu vào của nó. Do đó, trên tinh thần của phương pháp đã đề ra, vấn đề dịch ngược mã Assembly sẽ tập trung vào phần Front-End và coi Back-End như là nền tảng chung được thừa kế. Hình D.2 khái quát lại cơ chế hoạt động mới của Boomerang khi bổ sung chức năng dịch ngược mã Assembly:



Hình D.2. Cơ chế hoạt động mới của Boomerang tập trung vào sửa đổi Front-End

Để xử lý Front-End, cả 2 giai đoạn Nạp và Giải Mã đều cần phải xử lý. Hình D.3 mô tả chi tiết cơ chế hoạt động của 2 giai đoạn này (khi dịch ngược mã nhị phân):



Hình D.3: Chi tiết cơ chế hoạt động của 2 giai đoạn Nạp và Giải Mã.

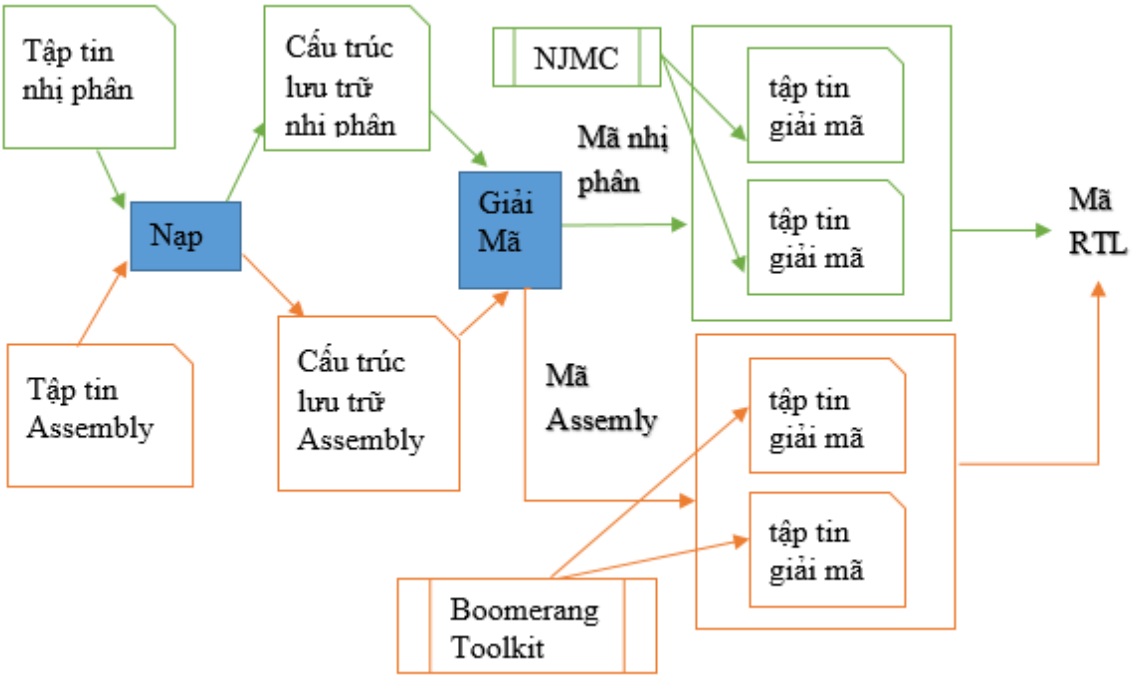
Kỹ thuật dịch ngược

Đối với giai đoạn Nạp, tập tin nhị phân sẽ được phân tích và lưu trữ trong một cấu trúc lưu trữ xác định. Cấu trúc này chứa thông tin về chương trình chính, các hàm trong chương trình, và các câu câu lệnh mã nhị phân trong từng hàm. Giai đoạn Giải mã sẽ sử dụng cấu trúc thông tin này để giải mã tuần tự từng câu lệnh nhị phân và sinh ra mã trung gian. Quá trình biến đổi được thực hiện bởi những Tập Tin Giải Mã (decoder). Đây là những tập tin có chức năng nhận dạng câu lệnh nhị phân và biến đổi chúng sang dạng mã RTL tương ứng. Tập Tin Giải Mã được sinh ra bởi bộ công cụ NJMC (New Jersey Machine Code).

Như vậy, để giải quyết bài toán dịch ngược mã Assembly bằng Boomerang Decompiler, có hai nhiệm vụ cần phải giải quyết. Nhiệm vụ đầu tiên đó là cần phải hiện thực một cấu trúc để lưu trữ thông tin của mã Assembly. Nhiệm vụ thứ hai, nằm ở giai đoạn Giải Mã**,** đó là cần phải xử lý được những thông tin có được sao cho có thể biến đổi mã Assembly thành mã RTL tương ứng.

2. Khái quát hóa những giải pháp

Hình D.4 minh họa giải pháp cho cơ chế hoạt động mới của 2 giai đoạn Nạp và Giải Mã:



Hình D.4: Chi tiết cơ chế hoạt động mới của 2 giai đoạn Nạp và Giải Mã.

Đối với nhiệm vụ đầu tiên, có hai giải pháp được đề ra: một là chỉnh sửa cấu trúc lại cấu trúc lưu trữ thông tin nhị phân cho phù hợp với mã Assembly, hai là hiện thực một cấu trúc lưu trữ mới.

Kỹ thuật dịch ngược

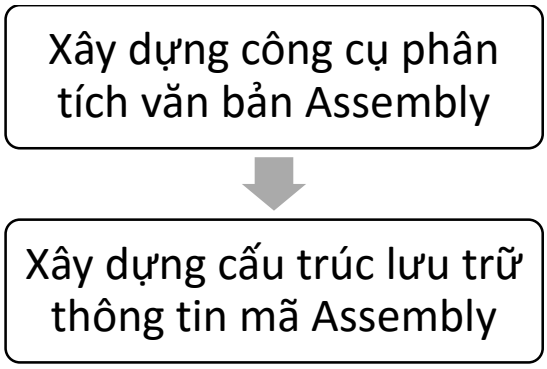
Với phương pháp đầu tiên, chi phí hiện thực là không hiệu quả vì cấu trúc này đã được thiết kế đặc trưng cho mã nhị phân. Hơn nữa, việc đi chỉnh sửa sẽ tiềm ẩn rủi ro hư hại cho chức năng ban đầu của Boomerang. Phương pháp thứ hai có nhiều ưu điểm hơn. Vì việc đi xây dựng một cấu trúc lưu trữ mới sẽ tạo ra hai con đường song song cho việc xử lý hai loại mã. Chức năng dịch ngược mã nhị phân không bị tổn hại và chi phí chỉnh sửa ở các giai đoạn sau được giảm thiểu đi nhiều. Như trong hình D.4, một cấu trúc mới được hiện thực song song với cấu trúc cũ.

Đối với Nhiệm vụ thứ hai**,** yêu cầu chủ yếu được đặt ra, đó là phải có những tập tin giải mã dành riêng cho việc chuyển đổi mã Assembly thành mã trung RTL.Giải pháp được lựa chọn cũng là xây dựng nên con đường độc lập với con đường ban đầu. Như trong hình D.4, giai đoạn Giải Mã sẽ sử dụng những Tập Tin Giải Mã mới độc lập với những tập tin giải mã dành cho mã nhị phân. Một công cụ mới cũng được phát triển gọi là Boomerang Toolkit dùng để sinh ra những Tập Tin Giải Mã đó.

Như vậy, để giải quyết tốt bài toán dịch ngược mã Assembly bằng Boomerang Decompiler thì nhất thiết phải giải quyết thật tốt hai nhiệm vụ đã đề ra.

II. Hiện thực cấu trúc lưu trữ thông tin mã Assembly

Các nhiệm vụ cần thực hiện của chương II được mô tả ở hình D.5:



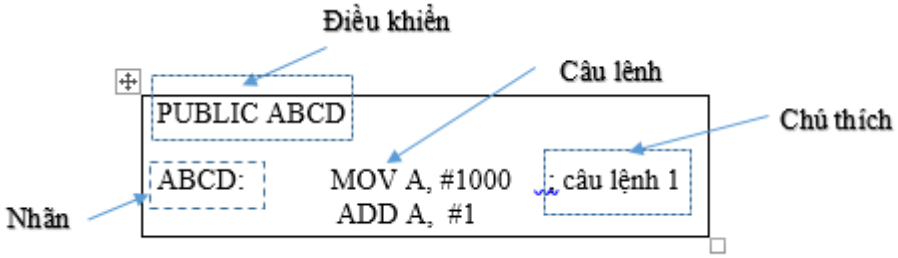
Hình D.5. Các nhiệm cần thực hiện ở chương II.

Vì đoạn mã Assembly được biểu diễn ở dưới dạng văn bản cho nên vấn đề đầu tiên cần phải giải quyết đó là xây dựng nên một công cụ tự động có khả năng phân tích văn bản để rút trích những thông tin cần thiết chứa trong văn bản đó. Công cụ tự động này giúp giảm thiểu chi phí của người dùng khi sử dụng Framework Boomerang. Người dùng cũng có thể tự định nghĩa thủ công như thông tin cần dùng đến nhưng việc làm đó sẽ tốn nhiều công sức ban đầu cũng như không đảm bảo được tính đúng đắn của thông tin. Phần 1 của chương này sẽ trình bày về công cụ đó.

Nhiệm vụ thứ hai là sau khi có được những thông tin cần thiết đó là hiện thực cấu trúc lưu trữ mã Assembly để phục vụ cho quá trình giải mã. Quá trình hiện thực này được trình bày ở phần 2.

1. Hiện Thực Công Cụ Phân Tích Văn Bản Mã Assembly

Assembly cũng là một ngôn ngữ lập trình dành cho máy tính. Chương trình viết bằng ngôn ngữ Assembly cũng có những quy tắc về cú pháp và ngữ nghĩa nhất định. Tùy vào từng loại kiến trúc máy mà có một loại mã Assembly khác nhau về tập lệnh, cú pháp… Nhưng về tổng thể, một chương trình mã Assembly đều có các thành phần cơ bản như sau: *Câu lệnh*, *Nhãn*, *Điều Khiển*, *Chú thích*. Ví dụ mẫu về một chương trình Assembly được mô tả trong hình D.6.



Hình D.6. Đoạn mã Assembly của máy 8051

Câu lệnh là thành phần cơ bản nhất của mọi chương trình mà máy tính sẽ nạp vào để thực thi. *Nhãn* giống như một kiểu dấu hiệu đặc biệt đại diện cho một đoạn chương trình, ví dụ như trong hình D.6 *nhãn* 'ABCD' đại diện cho 2 câu lệnh bên dưới. *Điều khiển* không phải là thành phần được nạp trực tiếp vào máy tính, nó giống như sự hướng dẫn dành cho chương trình Assembly. Mỗi loại ngôn ngữ Assembly khác nhau có các kiểu *Điều khiển* khác nhau, như trong hình D.6 *Điều khiển* 'PUBLIC ABCD' quy định đoạn chương trình sẽ được thực thi nằm ở label ABCD. Thành phần *chú thích* chỉ hỗ trợ cho việc đọc hiểu chương trình của người lập trình.

Dựa vào những thành phần cơ bản như vậy, công cụ phân tích văn bản mã Assembly (gọi là Assembly Parser**)** sẽ được xây dựng tương tự với 2 giai đoạn đầu tiên của một Trình Biên Dịch, đó là: Phân Tích Từ Vựng và Phân tích cú pháp. Các giai đoạn khác không cần thiết được áp dụng vì mục đích chính của nhiệm vụ này đó là rút trích thông tin, không phải biên dịch lại đoạn mã nguồn.

Kỹ thuật dịch ngược

a. Ngôn ngữ và công cụ được sử dụng

Công cụ được xây dựng trên bộ thư viện Flex và Bison. Đây là bộ thư viện phân tích ckhả tương thích với ngôn ngữ lập trình C++, do đó có thể tích hợp trực tiếp vào hệ thống Boomerang.

b. Phân tích từ vựng

Giai đoạn phân tích từ vựng có khả năng tạo nên các từ tố thông dụng. Một số từ tố được phân tích bởi Assembly Parser:

- Định danh: dành cho tên của nhãn, tên câu lệnh, toán hạng …

  Biểu thức chính quy:

  [A-Za-z_\$\@\#] [A-Za-z_$0-9\-]*

  Ví dụ như trong hình D.6: ABCD, MOV, A là các định danh.

- Số thực: dành cho toán hạng số thực

  Biểu thức chính quy:

  [0-9]+\.[0-9]+

  Ví dụ: 1.25

- Số nguyên: dành cho toán hạng số nguyên

  Biểu thức chính quy:

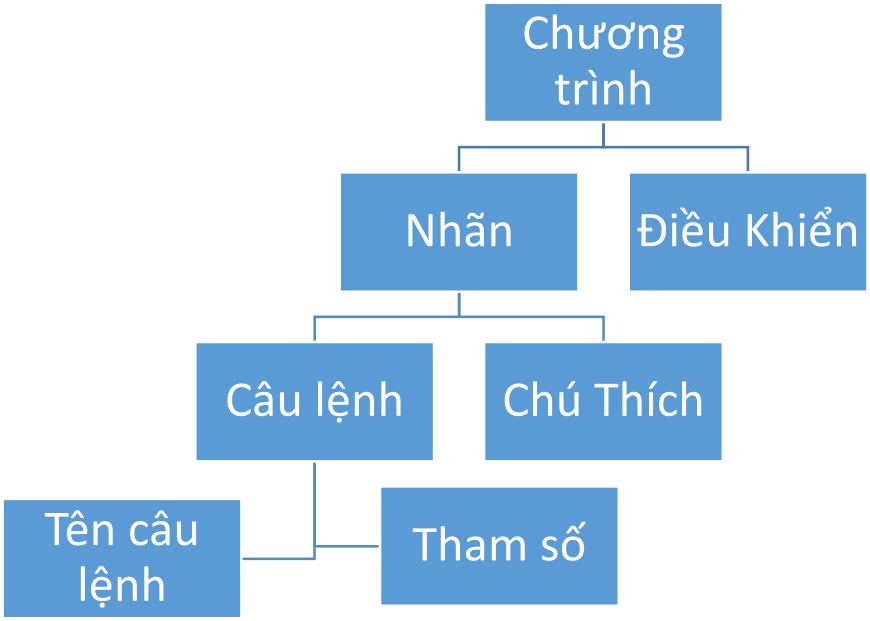
  [0-9]+

  Ví dụ: 10

Ngoài ra, với mỗi loại mã Assembly khác nhau thì còn có những nhóm từ tố khác nhau.

c. Phân tích cú pháp

Cấu trúc một chương trình Assembly đơn giản gồm có các *Điều Khiển* và các *Nhãn. Ở trong mỗi nhãn* gồm có nhiều *câu lệnh*. Mỗi *câu lệnh* có tên câu lệnh và các tham số. Hình D.7 minh họa cấu trúc như vậy:



Hình D.7. Minh họa cấu trúc đơn giản của chương trình Assembly

Dựa vào cấu trúc này, *Ngữ Pháp Phi Ngữ Nghĩa* sẽ được xây dựng. Cũng tùy thuộc vào mỗi loại mã Assembly mà *Ngữ Pháp Phi Ngữ Nghĩa* có chút khác nhau, nhưng vẫn có những thành phần chung cơ bản. Bảng D.1 dưới đây minh họa *Ngữ Pháp Phi Ngữ Nghĩa* của mã Assembly 8051 (gồm những phần chung):

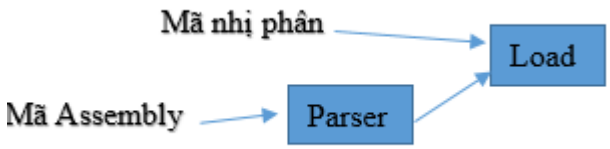
Bảng D.1. Minh họa về Ngữ Pháp Phi Ngữ Nghĩa của mã Assembly 8051.

NGỮ PHÁP PHI NGỮ NGHĨA	GIẢI THÍCH
Program: directives labels ;	Chương trình (Program) gồm có các điều khiển (directives) và các nhãn (labels).
labels: labels label | label ;	Các nhãn có thể gồm 1 hay nhiều nhãn (label).
label: ID ':' instruction ;	Nhãn thì có tên nhãn (ID), dấu hai chấm (':') và các câu lệnh (instruction).
instruction: ID | ID arguments	Câu lệnh có thể chỉ gồm tên (ID), hoặc gồm cả tên và các tham số (arguments).

;	
arguments: argument | arguments ‘,’ argument ;	Các tham số có thể gồm một tham số (argument), hoặc là nhiều tham số cách nhau bởi dấu phẩy (‘,’)
argument: ID |INTEGER | FLOAT	Tham số có thể gồm có hoặc là định danh, hoặc là kiểu số nguyên, hoặc là kiểu số thực

d. Kết quả

Boomerang Decompiler đã được bổ sung thêm chức năng tự động phân tích cho văn bản ngôn ngữ Asssembly đầu vào. Một khi Boomerang thực thi lựa chọn là dịch ngược mã Assembly, tập tin đầu vào sẽ được tự động nhận diện và phân tích bởi Assembly Parser (như trong hình D.7):



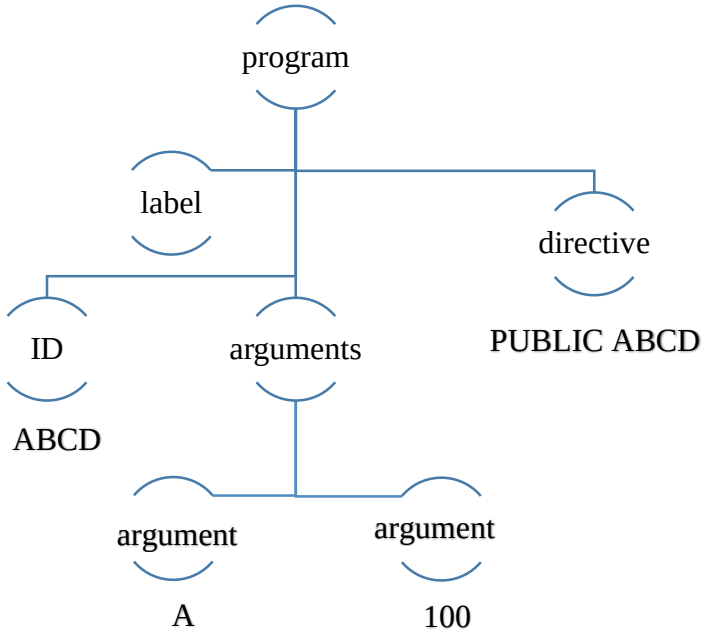
Hình D.7. Công cụ Assembly Parser được tích hợp vào Boomerang.

Ví dụ, với tập tin mã Assembly 8051 dưới đây:

```
PUBLIC ABCD
ABCD:
        MOV A, 100
```

Hình D.8. Đoạn văn bản ví dụ chương trình Assembly 8051

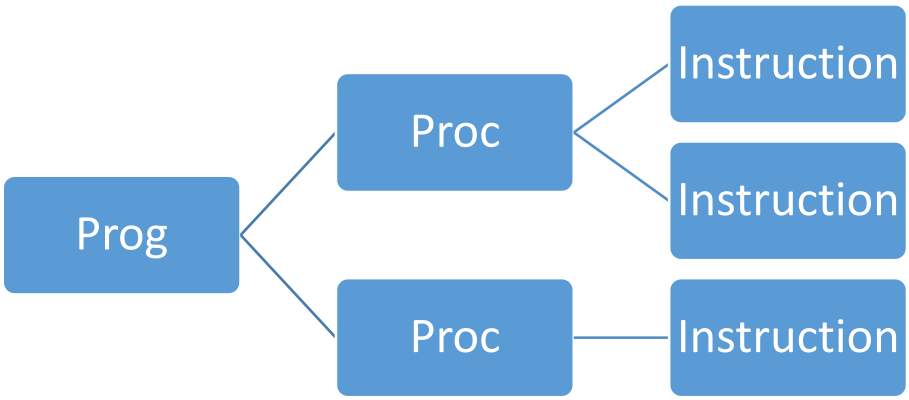
Cùng với ngữ pháp phi ngữ nghĩa đặc tả ở bảng D.1, thông tin thu được sau khi được phân tích bởi Assembly Parser:



Hình D.9. Thông tin thu được sau khi phân tích văn bản ở bảng D.1

2. Hiện thực cấu trúc lưu trữ thông tin chương trình Assembly

Để cho quá trình giải mã ở giai đoạn Giải Mã được thuận lợi, cấu trúc lưu trữ phải được hiện thực phù hợp với cấu trúc lưu trữ của hệ thống Boomerang. Hình D.10 minh họa lại cấu trúc đó:



Hình D.10. Cấu trúc chương trình của hệ thống Boomerang

Trong hình D.10, 'Prog' đại diện cho một chương trình lớn, là chương trình sẽ được dịch ngược bởi Boomerang. 'Prog' gồm nhiều 'Proc', mỗi 'Proc' đại diện cho một hàm hay thủ tục. Trong mỗi 'Proc' lại gồm có một hoặc nhiều 'Instruction', tức là câu lệnh.

Như vậy, sau khi phân tích và rút trích thông tin ở nhiệm vụ trước. Những thông tin cần được chọn lọc cho sao phù hợp với cơ chế hoạt động giải mã. Các thông tin thiết yếu như các

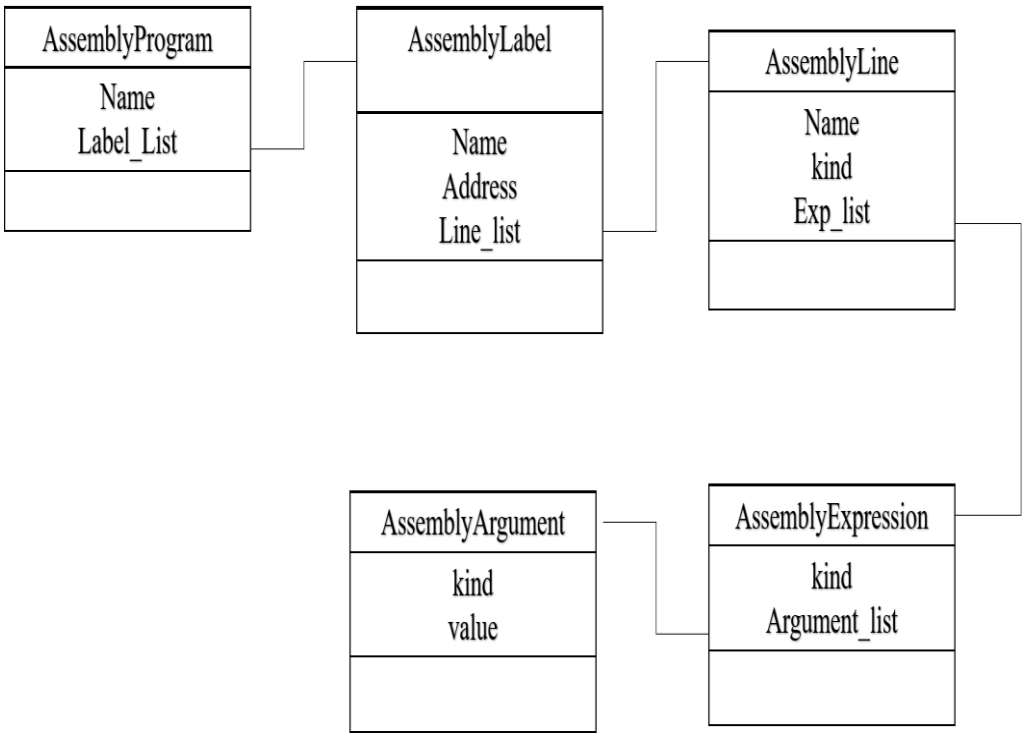
câu lệnh, nhãn, và một số *điều khiển* (ví dụ như khai báo biến toàn cục) sẽ được lưu trữ. Các thông tin phụ, không có ý nghĩa có quá trình giải mã như *chú thích* được bỏ qua.

a. Lựa chọn cách hiện thực

Cấu trúc lưu trữ được hiện thực bằng các lớp (Class) và được bổ sung trực tiếp ở giai đoạn Load. Cách hiện thực này có ưu điểm là dễ dàng chỉnh sửa, bổ sung và nhất là có khả năng thừa kế cao cho những mục đích sử dụng sau này.

b. Sơ đồ mô hình Class

Sơ đồ mô hình Class cho cấu trúc lưu trữ được mô tả ở hình D.11.



Hình D.11. Mô hình sơ đồ lớp lưu trữ thông tin chương trình Assembly.

Trong hình D.11, lớp AssemblyProgram chứa thông tin chương trình chính, gồm có tên chương trình (name) và danh sách các nhãn (Label_List) liên kết đến lớp AssemblyLabel. AssemblyLabel chứa thông tin của các nhãn gồm có tên nhãn (name), địa chỉ của nhãn (address), và danh sách các câu lệnh (Line_List) liên kết đến AssemblyLine. AssemblyLine thì đại diện cho thông tin của một câu lệnh như tên và biểu thức các tham số (Exp_List). AssemblyExpression chứa thông tin về các biểu thức gồm kiểu (kind) và danh sách cách tham số (argument_list). Kiểu của biểu thức gồm có: một ngôi (unary), hai ngôi (binary), chuỗi ký tự (literal). AssemblyArgument chứa thông tin về một tham số gồm có kiểu (kind) và giá trị của nó (value). Kiểu của tham số có thể gồm: kiểu số nguyên, số thực, chuỗi,…

Có thể thấy, cấu trúc lưu trữ này phù hợp với cơ chế hoạt động giải mã ở hình D.10. AssemblyProgram như là một 'prog', chứa nhiều 'proc' lưu trữ trong AssemblyLabel. AssemblyLabel lại chứa danh sách các 'instruction' (được lưu trữ trong AssemblyLine).

III. Hiện thực việc chuyển đổi mã Assembly thành RTL

Nội dung chương này đề cập đến việc hiện chuyển đổi mã Assembly thành mã trung gian RTL. Bao gồm các công việc như xây dựng bộ công cụ Boomerang Toolkit và Xử Lý Front End**.** Các công việc này đều diễn ra ở giải mã Giải Mã của hệ thống Boomerang.

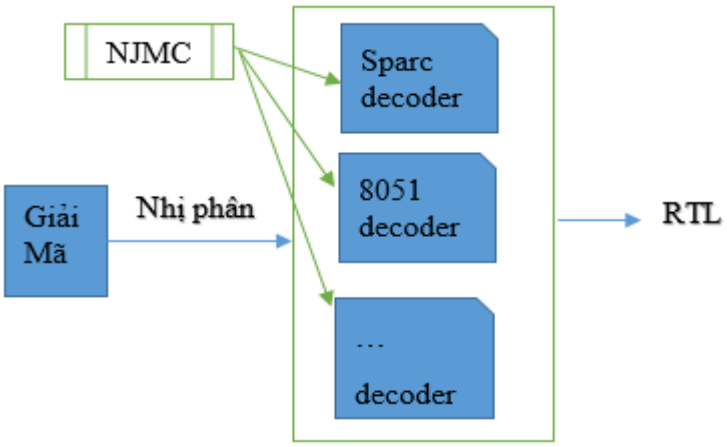
Boomerang ToolKit được xây dựng để thay thế bộ công cụ mà hệ thống Boomerang sử dụng trước đây là New Jersey Machine Code ToolKit (NJMC). Cơ chế hoạt động của NJMC, chức năng và chi tiết hiện thực của Boomerang ToolKit sẽ được đề cập đến trong phần đầu tiên của chương này.

Xử Lý Front End là quá trình xử lý thông tin ở trước, trong và sau giai đoạn Giải Mã. Những thông tin này sẽ tạo thành kết quả cuối của việc giãi mã và được sử dụng cho phần Back-end. Phần cuối của chương sẽ trình bày chi tiết về những thông tin đó cũng như các công đoạn cần phải xử lý.

1. Xây dựng Boomerang Toolkit

a. NJMC và Mục đích xây dựng Boomerang ToolKit

NJMC (New Jersey Machine Code Toolkit) là bộ công cụ hỗ trợ việc giải mã tập lệnh máy cho nhiều kiến trúc máy khác nhau. Trong giai đoạn Giãi Mã (Decode), hệ thống Boomerang nguyên bản sử dụng các kết quả của công cụ này (là các tập tin giả mã decoder), để tạo nên mã RTL. Vai trò của NJMC được minh họa ở hình D.12:

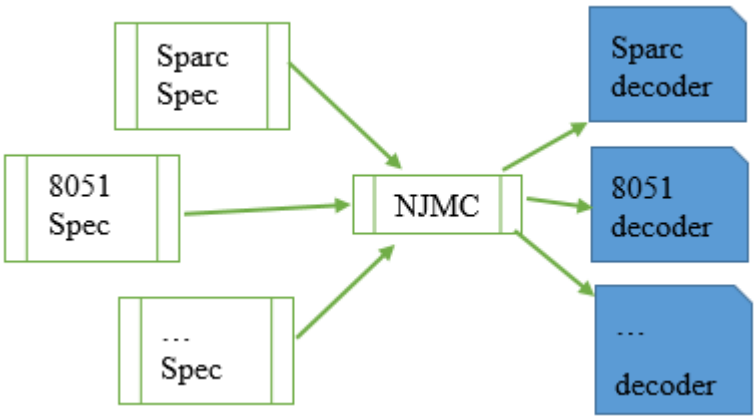


Hình D.12. Vai trò của NJMC trong giai đoạn giải mã.

Kỹ thuật dịch ngược

Trong hình D.12, giai đoạn Giải Mã sẽ sử dụng các tập tin giải mã (decoder) của các hệ máy khác nhau (như Sparc, 8051, …) để chuyển đổi mã nhị phân sang mã trung RTL tương ứng. Còn NJM**C** có nhiệm vụ là tạo ra các tập tin giải mã đó.

Cơ chế hoạt động của NJMC được khái quát hóa ở hình sau:

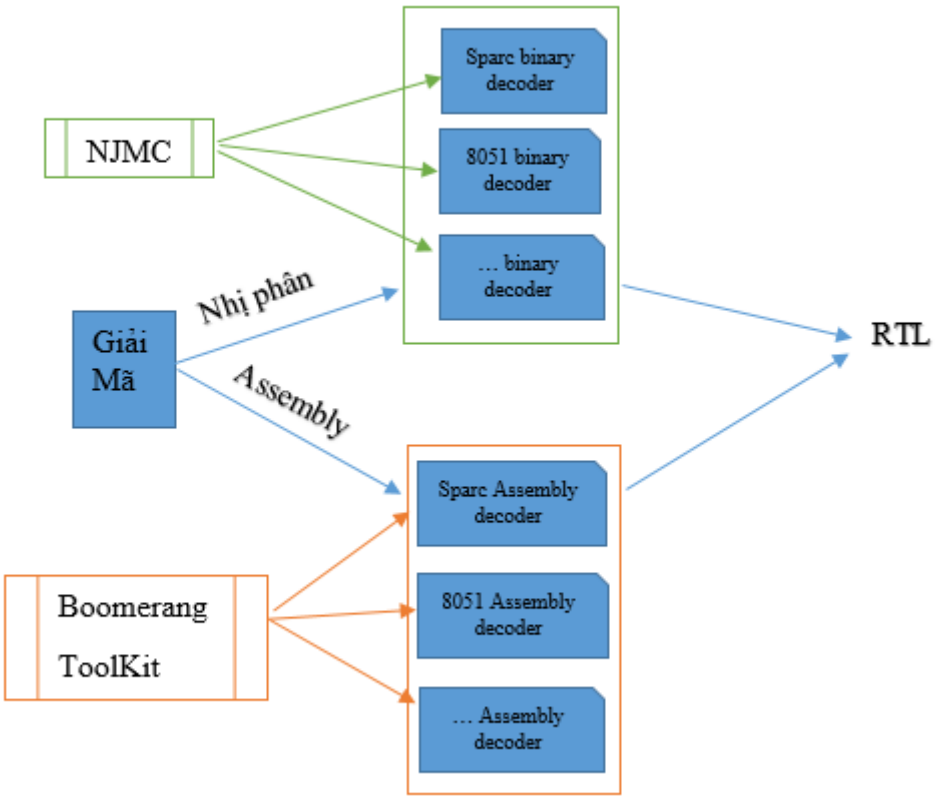


Hình D.13. Cơ chế hoạt động của NJMC.

NJMC ban đầu nhận vào các tập tin đặc tả tập lệnh kiến trúc máy (như Sparc Spec, 8051 Spec). Đặc tả này định nghĩa về việc chuyển đổi giữa dạng biểu diễn ký hiệu và dạng biểu diễn nhị phân của một câu lệnh. NJMC xử lý những định nghĩa trong đặc tả đó để tạo ra những tập tin giải mã tương ứng (như Sparc decoder, 8051 decoder). Chức năng chính của những tập tin giải mã là nhận dạng được dạng biểu diễn ký hiệu của các câu lệnh từ đoạn mã nhị phân đầu vào, để từ đó có thể xây dựng nên những câu lệnh RTL tương ứng.

NJMC là công cụ độc lập với hệ thống Boomerang nhưng có vai trò rất hữu dụng. Vì nó có khả năng tự động sinh ra các tập tin giải mã khác nhau chỉ dựa vào đặc tả của từng kiến trúc máy. Tính năng này giúp Boomerang giảm thiểu được nhiều chi phí cho quá trình Giải Mã, đồng thời có thể dễ dàng mở rộng cho đa nền tảng kiến trúc máy.

Mục đích xây dựng Boomerang Toolkit xuất phát từ lý do Hệ thống Boomerang cần phải có một công cụ mới để dành cho việc giải mã tập lệnh Assembly. Nguyên nhân là vì những tập tin giải mã được tạo ra khi sử dụng NJMC chỉ dành cho việc giải mã tập lệnh mã nhị phân, cho nên đối với quá trình giải mã tập lệnh Assembly, Boomerang không thể tận dụng lại tập tin đó.

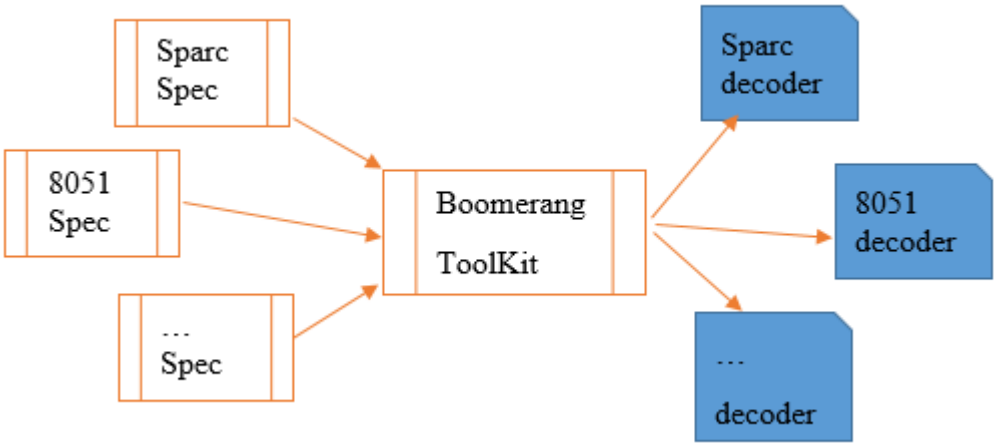


Hình D.14. Giai đoạn Giải Mã sử dụng những tập tin giải mã mới dành cho mã Assembly.

Boomerang Toolkit có chức năng chính là sinh ra những tập tin giải mVã mới dành cho mã Assemby của từng kiến trúc máy (như Sparc Assembly Decoder, 8051, Assembly Decoder, …). Như trong hình D.14, Giai đoạn Giải Mã có một luồng đi mới sau song song với luồng đi của việc giải mã mã nhị phân. Khi dịch ngược mã Assembly, hệ thống giải mã sẽ sử dụng những tập tin giải mã dành cho Assembly để tạo ra mã trung gian RTL.

Phương pháp tạo ra 2 hướng đi song song này có ưu điểm là nó tạo ra sư độc lập trong việc xử lý 2 loại mã, vừa tiết kiết chi phí lại dễ dàng sửa đổi so với phương án khả thi khác là trực tiếp sửa đổi lại những tập tin giải mã cũ vốn dành cho binary để chúng tương thích với Assembly.

Về cơ chế hoạt động, Boomerang Toolkit được phát triển tương tự như NJMC để kế thừa sự hữu dụng của công cụ cũ. Hình D.15 minh họa cơ chế hoạt động Boomerang Toolkit:



Hình D.15. Cơ chế hoạt động của Boomerang ToolKit.

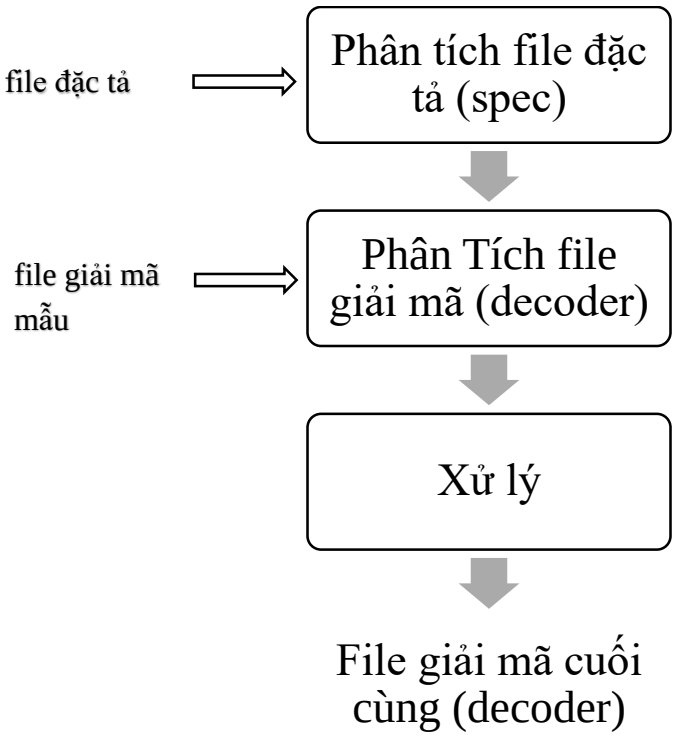
Công cụ cũng nhận đầu vào là các tập tin đặc tả tập lệnh kiến trúc máy và sinh ra các tập tin giải mã tương ứng. Và cũng giống như NJMC, nó là một công cụ tự động và có khả năng mở rộng cho nhiều kiến trúc máy khác nhau.

Có 2 giải pháp để phát triển Boomerang Toolkit. Một là sửa đổi lại NJMC sao cho công cụ tương thích thêm với mã Assembly, hai là phát triển một công cụ hoàn toàn mới. Giải pháp thứ nhất tốn nhiều chi phí do NJMC vốn được phát triển chỉ dành việc giải mã mã nhị phân. Việc chỉnh sửa tiềm ẩn nhiều rủi ro nguy hại đến chức năng ban đầu cũng như chức năng được bổ sung. Hơn nữa những thông tin trong tập tin đặc tả kiến trúc máy có thể được tận dụng lại được bởi chúng không phụ thuộc vào bất kỳ loại mã nào (nhị phân hay Assembly). Do đó, giải pháp thứ hai được lựa chọn.

b. Chi tiết Boomerang Toolkit

Boomerang Toolkit là công cụ giả lập bộ New Jersey Machine Code ToolKit (NJMC) với chức năng là tạo ra các tập tin giải mã tập lệnh mã Assembly được sử dụng trong giai đoạn Giải Mã của Boomerang Decompiler.

Các giai đoạn của Boomerang Toolkit được mô tả ở hình sau:



Hình D.16. Các giai đoạn hoạt động của Boomerang Toolkit

Phân tích tập tin đặc tả: đây là giai đoạn mà công cụ sẽ phân tích *tập tin đặc tả* tập lệnh kiến trúc máy. Những tập tin này thường có định dạng đuôi *.spec*. Kết thúc giai đoạn này công cụ sẽ rút trích được thông tin chứa trong tập tin đặc tả.

Phân tích tập tin giải mã: ở giai đoạn này, Boomerang Toolkit sẽ phân tích và rút trích thông tin trong tập tin giải mã mẫu. Tập tin giải mã mẫu là bộ khung cho tập tin giải mã cuối cùng và thường có định dạng đuôi *.m*. Tập tin giải mã mẫu về cơ bản có đầy đủ cấu trúc của tập tin giải mã cuối cùng, chỉ còn thiếu những phần cần được bổ sung bởi những thông tin chứa trong tập tin đặc tả.

Xử lý: đây là giai đoạn mà Boomerang Toolkit sẽ sử dụng như thông tin có trong tập tin đặc tả để bổ sung vào cấu trúc của tập tin giải mã mẫu. Kết thúc quá trình này, công cụ sẽ tạo ra tập tin giải mã hoàn chỉnh, có thể sử dụng được bởi hệ thống Boomerang.

Các tập tin đặc tả tập lệnh máy (Spec): Các tập tin đặc tả chứa thông tin đặc tả về tập lệnh của một hệ máy cụ thể (ví dụ như Sparc, Windows, st20, mips, …). Các thông tin này bao gồm thông tin về địa chỉ câu lệnh, độ rộng bit câu lệnh, tên thanh ghi, tên dưới dạng Assembly của câu lệnh, các tham số của câu lệnh, các phép tính toán với các câu lệnh. Từ những thông tin này, đặc tả sẽ quy định về sự tương quan giữa dạng biểu diện ký hiệu và dạng biểu diễn nhị phân của một câu lệnh.

Ví dụ một phần về tập tin đặc tả tập lệnh máy Sparc được mô tả ở bảng D.2:

Bảng D.2. Ví dụ một phần đặc tả tập lệnh máy Sparc

fields of instruction (32)
inst 0:31 op 30:31 disp30 0:29 rd 25:29 op2 22:24 imm22 0:21
[rd rs1 rs2] is [names ["%g0" "%g1" "%g2" "%g3" "%g4" "%g5" "%g6" "%g7" "%o0" "%o1" "%o2" "%o3" "%o4" "%o5" "%sp" "%o7" "%l0" "%l1" "%l2" "%l3" "%l4" "%l5" "%l6" "%l7" "%i0" "%i1" "%i2" "%i3" "%i4" "%i5" "%fp" "%i7"]]

Trong bảng D.2, dòng 1 đặc tả về độ chiều dài của một câu lệnh là 32 bit. Dòng 2 đặc tả về vị trí của các thanh ghi trong một câu lệnh (như *inst* bắt đầu từ bit 0 kết thúc ở bit 31). Dòng 2 đặc tả về những cái tên (mảng bên phải) tương ứng với các thanh ghi (mảng bên trái): như thanh ghi *rd* có những cái tên đại diện như "%g0", "%g1", …

Đặc tả của những tập tin đặc tả được mô tả bởi những thành phần dưới đây:

- Đặc tả chính

  Một đặc tả chính (specification) thì gồm tập hợp các đặc tả nhỏ hơn (spec). Các đặc tả nhỏ hơn gồm có: đặc tả về từ tố (tokens) và trường dữ liệu (field), đặc tả về thông tin trường dữ liệu (fieldinfo), đặc tả về mẫu (pattern), đặc tả về hàm tạo (constructor), cùng một số đặc tả phụ khác…

  Biểu thức chính quy:

  Specification => {spec}

  Hình D.17. Biểu thức chính quy của Đặc Tả Chính.

- Từ tố và Trường dữ liệu

  Một câu lệnh là một chuỗi của một hay nhiều từ tố (token). Ví dụ như câu lệnh máy Pentium có 8 bit *tiền tố* (prefix), 8 bit *mã toán tử* (opcode), 8 bit *định dạng (format byte)*, 16 bit *toán hạng (operand)*. Những *tiền tố, mã toán tử, định dạng* hay *toán hạng* là những từ tố.

  Những từ tố được phân chia thành các *trường dữ liệu (field)*. Một *trường dữ liệu* là một dãy các bit trong một từ tố. Trường dữ liệu gồm có *mã toán tử (opcode), toán hạng*

(operands), *chế độ* (modes), và các thông tin khác.

Đặc tả về từ tố (token) và trường dữ liệu (field) xác định các phân chia từ tố thành những trường dữ liệu. Bắt đầu bằng từ khóa "fields of", đặc tả này mô tả thông tin về tên từ tố (token-name), độ dài bit (width) và đặc tả trường dữ liệu (field-specs) thuộc từ tố đó. Đặc tả trường dữ liệu gồm có tên trường (field-name), bit bắt đầu (low-bit) trong từ tố, bit kết thúc (high-bit) trong từ tố.

Biểu thức chính quy:

```
spec => fields of token-name (width) field-specs

field-specs => {field-name low-bit:high-bit}
```

Hình D.18. Biểu thức chính quy của Từ tố và Trường dữ liệu.

Ví dụ với đoạn đặc tả sau:

```
fields of instruction (32)
inst 0:31 op 30:31 disp30 0:29 rd 25:29
```

Hình D.19. Ví dụ về đặc tả từ tố và trường dữ liệu.

Từ tố (token) instruction có độ dài 32 bit, được chia thành các trường dữ liệu gồm có: inst, op, disp30, rd. Trong đó thanh ghi rd có bit bắt đầu 25, bit kết thúc 29.

- Thông tin trường dữ liệu

Đặc tả thông tin trường dữ liệu (Fieldinfo) đặc tả chi tiết hơn về các trường dữ liệu. Bắt đầu bắt từ từ khóa "fieldinfo", đặc tả gồm có chuỗi các đặc tả về thông tin trường dữ liệu (fieldinfo-spec). Đặc tả thông tin trường dữ liệu bắt đầu bằng định danh trường (field-specifier) kết hợp với từ khóa "is" và phần tử trường dữ liệu (field-item). Định danh trường gồm một hoặc một chuỗi các tên trường (field-name). Các phần tử bắt đầu bằng những từ khóa: "checked", "unchecked", "guaranteed" (dùng để đặc tả mức độ kiểm tra), "sparse", "names" (dùng để xác định tên của trường dữ liệu). Phần từ bắt đầu bằng "sparse" hoặc "names" được nối tiếp bởi chuỗi các tên (item-name).

Biểu thức chính quy:

```
spec => fieldinfo {fieldinfo-spec}

fieldinfo-spec => field-specifier is [ {field-item}]

field-specifier => field-name | [ {field-name}]

field-item => checked
              | unchecked
              | guranteed
              | sparse [item-name]
              | names [ item-name]
```

Hình D.20. Biểu thức chính quy đặc tả thông tin trường dữ liệu.

Ví dụ cho đặc tả về thông tin trường dữ liệu dưới dây:

```
fieldinfo
[ rd rs1 rs2 ] is [
names [ "%g0" "%g1" "%g2" "%g3" "%g4" "%g5" "%g6" "%g7"
      "%o0" "%o1" "%o2" "%o3" "%o4" "%o5" "%sp" "%o7"
      "%l0" "%l1" "%l2" "%l3" "%l4" "%l5" "%l6" "%l7"
      "%i0" "%i1" "%i2" "%i3" "%i4" "%i5" "%fp" "%i7" ]
                                    ]
```

Hình D.21. Ví dụ về đặc tả thông tin trường dữ liệu.

Các trường dữ liệu rd, rs1, rs2 có danh sách những cái tên như "%g0" "%g1" "%g2" "%g3" "%g4" "%g5" "%g6" "%g7"…

- Mẫu

Đặc tả về mẫu (Pattern) quy định sự ràng buộc về giá trị của những trường dữ liệu (field) trong một từ tố (token). Bắt đầu bằng từ khóa “patterns”, đặc tả gồm chuỗi các ràng buộc mẫu (pattern-binding). Ràng buộc mẫu bắt đầu bằng một hoặc nhiều tên mẫu (pattern-name), kết hợp với các từ khóa như “i” hoặc “is any of”, “which is”, và các mẫu (pattern). Các mẫu (Pattern) có thể kết hợp với nhau bở các phép toán như phép liên kết (&), phép nối (;), phép chia cắt (|).

Biểu thức chính quy:

```
spec => patterns {pattern-binding}

pattern-biding => pattern-name is pattern

                | [{pattern-name}] is pattern

                | [pattern-name] is any of [{pattern-name}], which is
pattern

pattern =>  pattern

            | pattern & pattern

            | pattern; pattern

            | pattern | pattern
```

Hình D.22. Biểu thức chính quy đặc tả mẫu.

Ví dụ với đoạn đặc tả sau của máy Sparc:

```
patterns call is op = 1
```

Hình D.23. Ví dụ đặc tả mẫu máy Sparc.

Cái tên "call" được ràng buộc vào mẫu tương ứng với mã toán tử (opcode) CALL của máy Sparc. Mẫu "op = 1" là điều kiện để nhận dạng toán tử Call trong một từ tố 32 bit, bit 31 phải có giá trị 0 và bit 30 phải có giá trị là 1.

Ví dụ sau đặc tả một ràng buộc khác của máy Sparc:

```
patterns

float2s is FMOVs | FNEGs | FABSs | FSQRTs
```

Hình D.24. Ví dụ đặc tả mẫu máy Sparc

Mẫu float2s là mẫu đại diện cho những cái tên FMOVs, FNEGs, FABSs, FSQRTs. Đây cũng là những cái tên câu lệnh của mã Assembly của máy Sparc.

- Hàm Tạo

Hàm tạo (Constuctor) kết nối giữa dạng biểu diễn ký hiệu và dạng biểu diễn nhị phân của câu lệnh. Hàm tạo chứa thông tin về một câu lệnh cụ thể như mã toán tử (opcode), các tham số (operand). Những mã toán tử được biểu diễn dưới dạng mã Assembly của một câu lệnh và được quy định ràng buộc bởi những mẫu (pattern).

Đặc tả của hàm tạo bắt đầu từ từ khóa "constructors" kèm dành sách các hàm tạo (constructor). Hàm tạo gồm có mã toán tử (opcode) và danh sách các toán hạng

(operand). Ngoài ra có có thể có thêm kiểu tham số (type-name) và các nhánh rẻ (branches).

Biểu thức chính quy của hàm tạo được mô tả ở hình D.25:

```
spec => constructor {constructor}

constructor => opcode {operand} [: type-name] * [branches]
```

Hình D.25. Biểu thức chính quy đặc tả hàm tạo.

Ví dụ đặc tả hàm tạo của máy Sparc được mô tả ở hình D.26:

```
constructors
  float2s fs2s, fds
  FSQRTd  fs2d, fdd
```

Hình D.26. Ví dụ về đặc tả hàm tạo.

Hàm tạo float2s có mã toán tử float2s (vốn gồm các câu lệnh FMOVs | FNEGs | FABSs | FSQRTs như ở ví dụ ở hình 24) và có tham số là 2 thanh ghi fs2s, fds. Constructor thứ 2 có opcode FSQRTd, cũng là tên dưới dạng Assembly và 2 tham số là 2 thanh ghi fs2d, fdd.

- Các đặc tả khác

Ngoài ra còn có những đặc tả ở trong tập tin đặc tả nhưng Boomerang Toolkit không sử dụng những thông tin của những đặc tả này.

Tập tin giải mã mẫu: Tập tin giải mã mẫu chứa bộ khung của tập tin giải mã cuối cùng mà Boomerang ToolKit sẽ sinh ra. Tập tin giải mã mẫu được viết trên ngôn ngữ lập trình C++. Nó gồm có 2 thành phần: phần code C++ của bộ khung và phần quan trọng nhất là Matching Statement.

Matching Statement là phần dùng để nhận dạng câu lệnh từ mã nhị phân (đối với NJMC) hoặc từ mã Assembly (đối với Boomerang Toolkit). Boomerang Toolkit sẽ sử dụng thông tin từ tập tin đặc tả (các mẫu và hàm tạo) để thay thế Matching Statement bằng những câu lệnh điều kiện (if, else) của ngôn ngữ lập trình C++. Kết thúc quá trình này, công cụ sẽ sinh ra tập tin giải mã hoàn chỉnh trên ngôn ngữ lập trình C++.

Biểu thức chính quy của Matching Statemnet được mô tả bởi hình sau:

match code **to**
{ pattern [{operands}] => code} [**else** code]
endmatch

Hình D.27. Biểu thức chính quy đặc tả Matching Statement.

Matching Statement bắt đầu bằng từ khóa **"**match" và kết thúc ở từ khóa "endmatch". Trong bảng biểu thức chính quy trong hình D.27, ở dòng 1, "code" là một biểu thức tính toán giá trị của địa chỉ (address) bằng ngôn ngữ C++. NJMC Toolkit sẽ bắt đầu từ địa chỉ này để giải mã tập tin mã nhị phân. Đối với Boomerang Toolkit, biểu thức này không hữu dụng. Dòng thứ hai là chuỗi các phần nhận dạng câu lệnh. Chuỗi này có thể kết thúc bởi cú pháp từ khóa "else" và đoạn mã (code) C++. Phần nhận dạng gồm có mẫu (pattern) để nhận dạng, danh sách các toán hạng liên quan (operands), ký hiểu =>, và đoạn mã (code) C++ tương ứng.

Ví dụ một Matching Statement trong tập tin giải mã mẫu được mô tả ở hình D.28:

```
match [nextPC] hostPC to
  | call__(addr) =>
                ADDRESS nativeDest = addr - delta;
                newCall->setDest(nativeDest);
  | float2s (fs2s, fds) [name] =>
                stmts = instantiate(pc,  name, DIS_FS2S, DIS_FDS);
  else
    stmts = NULL;
    result.valid = false;
endmatch
```

Hình D.28.. Ví dụ mẫu về Matching Statements.

Các mẫu cần được nhận dạng gồm có "call__" (có 1 toán hạng "addr"), "float2s" (có 2 toán hạng "fs2s", "fds"). Kết thúc quá trình xử lý, Boomerang sẽ thay thế các mẫu này bởi những câu lệnh điều kiện của ngôn ngữ lập trình C++.

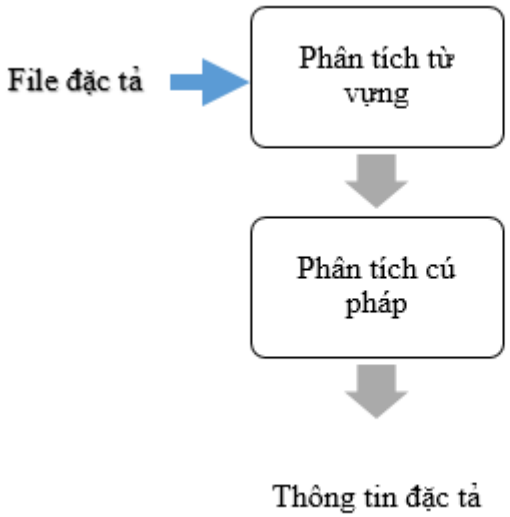
c. Hiện thực Boomerang Toolkit

Boomerang Toolkit có 3 giai đoạn cần hiện thực: Phân tích tập tin đặc tả, Phân tích tập tin giải mã mẫu và Xử lý.

Phân Tích Tập Tin Đặc Tả: Quá trình này sẽ phân tích văn bản của tập tin đặc tả để rút trích thông tin trong đó. Những thông tin quan trọng cần rút trích đó là tên dưới dạng

Assembly của các câu lệnh. Những tên này xác định ở trong hàm tạo (constructor) và các mẫu (pattern).

Các bước xử lý của giai đoạn phân tích tập tin đặc tả được thể hiện ở hình sau:



Hình D.29. Các bước xử lý của giai đoạn phân tích đặc tả

Phân tích từ vựng: Một số từ tố thông dụng trong giai đoạn phân tích từ vựng như:

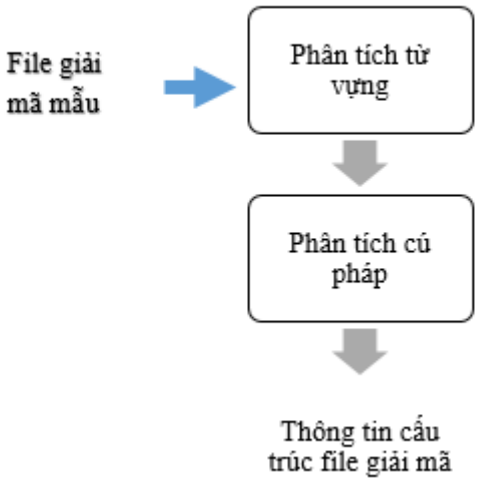
- Định danh: biểu diễn tên của mã toán tử, toán hạng, tên mẫu …
  Biểu thức chính quy: [A-Za-z_\$\@\#] [A-Za-z_$0-9\-]*
- Số thực: dành cho toán hạng số thực.
  Biểu thức chính quy: [0-9]+\.[0-9]+
- Số nguyên: dành cho toán hạng số nguyên
  Biểu thức chính quy: [0-9]+
- Từ khóa: Những từ được dùng làm từ khóa đặc tả, không thể làm định danh.
  Biểu thức chính quy:
  (constructors | patterns | pattern | constructor | syntax | to | is)

Phân tích cú pháp: Cú pháp cú tập tin đặc tả được xây dựng dựa trên đặc tả đề cập ở phần *các tập tin đặc tả tập lệnh máy* (gồm các hình D.17, D.18, D.22, D.25).

Thông tin sau khi được rút trích được lưu trữ trong một cấu trúc dạng Hash để sử dụng trong giai đoạn Xử Lý.

Phân Tích Tập Tin Giải Mã Mẫu: Giai đoạn này sẽ phân tích và rút trích thông tin tập tin giải mã mẫu. Có 2 phần riêng được rút trích. Đó là phần mã C++ dùng để dựng lại tập tin giải mã cuối cùng và phần Matching Statement dùng cho giai đoạn Xử Lý kế tiếp.

Các bước xử lý của giai đoạn phân tích tập tin giải mã mẫu được thể hiện ở hình sau:

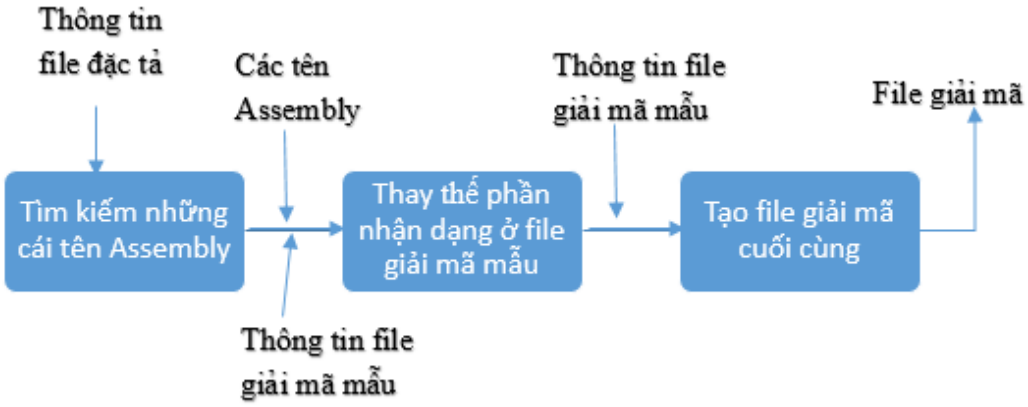


Hình D.30. Các bước xử lý của giai đoạn phân tích đặc tả

Phân tích từ vựng: Các từ tố thông dụng được phân tích giống như giai đoạn Phân Tích Tập tin Đặc Tả.

Phân tích cú pháp: Cú pháp phần Matching Statement được đặc tả ở phần *Chi tiết về các tập tin giải mã mẫu* đã đề cập ở phần trên (hình D.27)

Xử lý: Đây là giai đoạn mà Boomerang Toolkit sẽ tìm kiếm những cái tên ở dạng Assembly của các câu lệnh (từ thông tin ở tập tin đặc tả), rồi thay thế những phần nhận dạng ở tập tin giải mã mẫu bằng những câu lệnh điều kiện tương ứng. Cuối cùng, công cụ sẽ sinh ra tập tin giải mã cuối cùng (dựa vào tập tin giải mã mẫu đã được xử lý). Hình D.30 mình họa luồng hoạt động của giai đoạn Xử Lý.



Hình D.31. Luồng hoạt động của giai đoạn Xử Lý.

Tìm Kiếm Tên Assembly là giai đoạn đầu tiên dùng để tìm ra những câu lệnh ở dạng

Assembly (gồm tên và tham số). Phần lớn phần nhận dạng của Matching Statement ở trong giải mã mẫu là khớp với những thông tin mà hàm tạo (constructor) đặc tả trong tập tin đặc tả.

Ví dụ, trong tập tin đặc tả có hàm tạo "alu" ở hình D.32 (khung trên) khớp với phần nhận dạng của Matching Statement trong tập tin giải mã mẫu ở khung dưới do cùng mã toán tử (opcode) và có số lượng tham số bằng nhau.

```
alu rs1, reg_or_imm, rd
```

```
| alu (rs1, roi, rd) [name] =>
    stmts = instantiate(pc,       name, DIS_RS1, DIS_ROI, DIS_RD);
```

Hình D.32. Ví dụ phần nhận dạng trong tập tin giải mã mẫu khớp với thông tin của hàm tạo

Như vậy bước đầu đã xác định được mã toán tử bằng cách so trùng. Việc tiếp theo là nhận dạng được những cái tên dưới mã Assembly mà toán tử này đại diện. Có hai trường hợp xảy ra, một là mã toán tử này có thể trực tiếp là một tên dưới dạng Assembly, hai là nó là một mẫu (pattern) chứa những cái tên Assembly. Dù ở trường hợp nào thì Boomerang Toolkit cũng sẽ xác định bằng cách lặp đệ quy tất cả các mẫu để tìm kiếm cái tên cuối cùng.

Như ở ví dụ ở hình D.33 dưới đây, mã toán tử alu được xác định được bởi các mẫu là "logical", "shift" và "arith". Tiếp tục lặp lại các mẫu trên cho đến khi không còn mẫu nữa, kết quả sẽ thu được các cái tên cuối cùng: AND, ANDcc, ANDN, …

```
logical is AND | ANDcc
 shift   is SLL | SRL   | SRA
 arith   is ADD | ADDcc


 alu     is logical | shift | arith
```

Hình D.33. Ví dụ về mẫu quy định mã toán tử.

Ngoài ra còn trường hợp phần nhận dạng của một số Matchig Statement không khớp với thông tin của các hàm tạo. Tuy nhiên chúng lại chứa thông tin về tên dạng Assembly chứa trong các các mẫu. Do đó, công cụ Boomerang cũng sẽ quét lần lượt các mẫu để tìm cái tên tương ứng.

Thay thế Matching Statement là giai đoạn kế tiếp sau khi có được những cái tên khả thi, Boomerang Toolkit sẽ thay thế phần maching statement bằng những dòng mã C++.

Những dòng mã này là những câu lệnh điều kiện nhằm nhận dạng các câu lệnh Assembly. Với ví dụ ở hình D.32, tập tin giải mã mẫu sẽ được sẽ được chỉnh sửa thành:

```
if (opcode == “ADD” || opcode  == “SSL” ||  opcode == “ADD”)
{
        stmts = instantiate (pc, name, DIS_RS1, DIS_ROI, DIS_RD);
}
```

Hình D.34. Ví dụ tập tin giải mã mẫu đứa chỉnh sửa.

Có thể thấy, phần nhận dạng ban đầu được thay bằng câu điều kiện mã C++. Câu điều kiện này nhận dạng câu lệnh Assembly bằng tên câu lệnh.

Sau khi xử lý được những cái tên Assembly, Boomerang Toolkit sẽ xử lý những tham số để đưa về dạng mà Boomerang Decompiler có thể xử lý được (mã C++).

Ví dụ một tập tin đầu vào Assembly có lệnh ADD như sau:

```
add     %g1, 1, %g1
```

Hình D.35. Ví dụ về một câu lệnh trong tập tin Assembly.

Những tham số như %g1 là tên của các thanh ghi được đặc tả ở phần trường dữ liệu (fieldinfo) trong tập tin đặc tả (như trong hình D.36). Nó tương ứng với tham số r1 được dùng trong giai đoạn Biên Dịch Ngược của Boomerang (1 là số chỉ mục của %g1 trong mảng names).

```
fieldinfo
[ rd rs1 rs2 ] is [
names [ "%g0"  "%g1"  "%g2"  "%g3"  "%g4"  "%g5"  "%g6"  "%g7"
      "%o0"  "%o1"  "%o2"  "%o3"  "%o4"  "%o5"  "%sp"  "%o7"
      "%l0"  "%l1"  "%l2"  "%l3"  "%l4"  "%l5"  "%l6"  "%l7"
      "%i0"  "%i1"  "%i2"  "%i3"  "%i4"  "%i5"  "%fp"  "%i7" ]
                                        ]
```

Hình D.36. Đặc tả thông tin trường dữ liệu.

Kết quả sau khi xử tham số ở ví dụ hình D.34 là đoạn mã C++ sau:

```
if (name == "%g0") return 0;
 else if (name == "%g1") return 1;
 else if (name == "%g2") return 2;
 else if (name == "%g3") return 3;
 else if (name == "%g4") return 4;
 else if (name == "%g5") return 5;
 else if (name == "%g6") return 6;
 else if (name == "%g7") return 7;
```

Hình D.37. Đoạn mã C++ kết quả

Trong hình D.37, ứng với tên tham số sẽ trả về chỉ mục của tham số đó trong mảng names ở hình D.36.

Tạo tập tin giải mã cuối cùng là giai đoạn kết thúc. Sau cùng, Boomerang Toolkit sẽ tạo ra tập tin giải mã dưới dạng ngôn ngữ C++ hoàn chỉnh dựa vào bộ khung giải mã mẫu đã được chỉnh sửa ở giai đoạn trên.

Ví dụ về một phần đoạn mã tập tin giải mã cuối cùng:

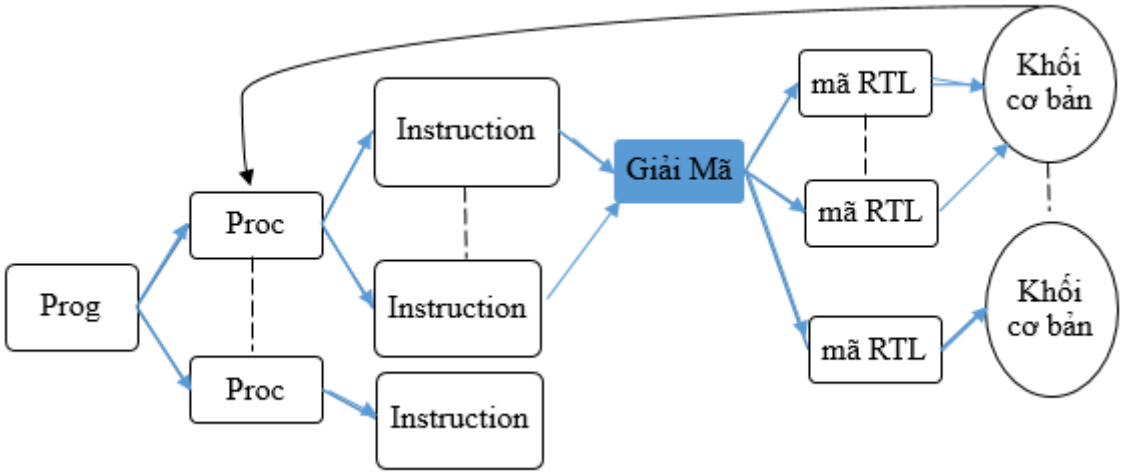
```
if (opcode == “ADD” || opcode  == “SSL” ||  opcode == “ADD”)
{
        stmts = instantiate (pc, name, DIS_RS1, DIS_ROI, DIS_RD);
}
else {
         result.numBytes = nextPC - hostPC;
}

return result;
```

Hình D.38. Ví dụ một phần đoạn mã tập tin giải mã cuối cùng.

2. Xử lý Front End

Xử lý Front End là giai đoạn xử lý các thông tin trước, trong và sau khi quá trình Giải Mã diễn ra. Mục đích của giai đoạn này đó là giúp cho quá trình xử lý mã trung gian RTL sinh ra sau khi giải mã (decode) mã Assembly diễn ra phù hợp với cơ chế giải mã. Mô hình D.39 minh họa cơ chế giải mã đó:



Hình D39. Cơ chế hoạt động ở giai đoạn *Giải Mã.*

Hình D.39 mô tả về cấu trúc một chương trình. 'Prog' đại diện cho một chương trình lớn, là chương trình sẽ được dịch ngược bởi Boomerang. 'Prog' gồm nhiều 'Proc', mỗi 'Proc' đại diện cho một hàm hay thủ tục (trong mã Assembly, mỗi Proc bắt đầu bởi một nhãn). Trong mỗi 'Proc' lại gồm có một hoặc nhiều 'Instruction', tức là câu lệnh (là mã Assembly hoặc mã nhị phân).

Cơ chế giải mã diễn ra như sau: hệ thống sẽ lần lượt đi vào từng 'Proc' để giải mã từng 'Instruction'. Sau khi giải mã xong, hệ thống sẽ thực hiện công việc tương tự cho những 'proc' khác cho đến khi hoàn tất mọi 'Proc'. Đối với mỗi 'Proc', các Instruction của nó sau khi được giải mã sẽ chuyển thành những đoạn mã trung gian RTL tương ứng. Các đoạn mã RTL lại được phân chia vào các khối cơ bản (Basic Block). Mỗi khối cơ bản giống như một đoạn chương trình trong một chương trình lớn hơn. Ví dụ như đoạn mã trong một hàm thì thuộc một khối cơ bản, trong một vòng lặp hay một câu lệnh rẽ nhánh If, Else thì thuộc một khối cơ bản khác. Kết thúc quá trình giải mã, các 'Proc' sẽ chứa các khối cơ bản gồm có những đoạn mã RTL. Chương trình lớn 'Prog' sẽ chứa những 'proc' đã được giải mã. Phần Back-End của Boomerang sẽ sử dụng 'Prog' để tiến hành quá trình dịch ngược sau đó.

Đối với việc giải mã mã nhị phân, tất cả các quá trình xử lý mã RTL đều dựa vào địa chỉ của tập lệnh. Điều này gây ra nhiều khó khan cần phải giải quyết khi giải mã mã Assembly bởi tập lệnh mã Assembly được biểu diễn ở dưới dạng văn bản.

Cụ thể những vấn đề mà việc giải mã mã Assembly có thể gặp bao gồm:

- Gọi hàm: Với câu lệnh gọi hàm, Boomerang sẽ lưu địa chỉ hàm được gọi vào một cấu trúc để sau nay sử dụng nó cho việc giải mã hàm đó. Đối với mã Assembly, cấu trúc này là không phù hợp vì đa số hàm được gọi được biểu diễn dưới dạng tên chứ

không phải địa chỉ. Ví dụ: CALL printf.

- Câu lệnh rẽ nhánh: Câu lệnh rẽ nhánh tạo thành những khối cơ bản cũng sử dụng một địa chỉ để đánh dấu sự bắt đầu của đoạn mã trong khối đó.

- Giải mã tuần tự: Việc giải mã đối với mã nhị phân trong một 'Proc' được diễn ra tuần tự từ trên xuống kể cả đối với câu lệnh rẽ nhánh hay lệnh nhảy. Đối với mã Assembly, việc giải mã tuần tự diễn ra khó khăn do các nhãn có thể nằm ở vị trí bất kỳ. Ví dụ trong đoạn mã Assembly sau:

```
ABCD:
        ADD A, B
MAIN:
        ADD A, C
        JMP ABCD
        RET
```

Hình D.40. Ví dụ đoạn mã Assembly 8051

Câu lệnh JMP ABCD biểu diễn lệnh nhảy đến nhãn ABCD nằm ở một vị trí ngoài khu vực nhãn MAIN. Trong quá trình giải mã đoạn mã ở nhãn MAIN Boomerang không tự xác định được ví trí của nhãn ABCD.

Như vậy các công việc cần phải xử lý trong giai đoạn này gồm có: xử lý các câu lệnh liên quan đến địa chỉ (gọi hàm, rẽ nhánh) và xử lý việc giải mã tuần tự cho mã Assembly.

Giải pháp thực hiện gồm các công việc được mô tả dưới đây:

- Xử lý các câu lệnh liên quan đến địa chỉ**:**

  Giải pháp để xử lý cho những câu lệnh gọi hàm, rẽ nhánh và nhảy của mã Assembly là giả lập địa chỉ cho chúng. Đây là phương pháp đơn giản mà không cần phải sửa đổi cấu trúc lưu trữ vốn có của Boomerang.

  Một địa chỉ ngẫu nhiên sẽ được gán ở câu lệnh đầu tiên của 'proc' đầu tiên. Địa chỉ này cũng là địa chỉ 'proc' đó. Địa chỉ của những Proc tiếp theo được tính toán dựa vào số lượng câu lệnh của Proc kế tiếp. Ví dụ việc giả lập địa chỉ cho đoạn mã Assembly 8051:

```
ABCD:                   // địa chỉ 1000
        ADD A, B        // địa chỉ 1004
        ADD A, 0        // địa chỉ 1008
MAIN:                   // địa chỉ 1012
        ADD A, C        // địa chỉ 1012
        CALL ABCD       // địa chỉ 1016
        RET             // địa chỉ 1020
```

Hình D.41. Ví dụ giả lập địa chỉ cho đoạn mã Assembly 8051

Tham số của các câu lệnh gọi hàm, rẽ nhánh được chuyển về dạng địa chỉ. Như ở ví dụ hình D.41, câu lệnh "CALL ABCD" được chuyển về thành "CALL 1000".

- Xử lý việc giải mã tuần tự:

Đối với câu lệnh rẽ nhánh, những câu lệnh trong khối cơ bản tạo bởi câu lệnh rẽ nhánh đó sẽ được nối vào ngay sau nó. Đồng thời độ dời bước nhảy đến cuối khối cơ bản được tính toán và lưu lại. Ví dụ với đoạn mã Assembly 8051 được xử lý ở hình D.42 dưới đây (Đoạn mã ban đầu nằm ở cột bên trái, đoạn mã sau khi được xử lý nằm ở cột bên phải):

ABCD: ADD A, 0 MAIN BNE ACC.5, ABCD JMP ABCD RET	ABCD: ADD A, 0 MAIN BNE ACC.5, ABCD ADD A, 0 JMP ABCD RET

Hình D.42. Đoạn mã ví dụ của mã Assembly 8051

Câu lệnh "BNE ACC.5, ABCD" biểu diễn câu điều kiện: "nếu ACC.5 != 1 thì rẽ nhánh tới nhãn ABCD". Sau khi được xử lý, câu lệnh "ADD A, 0" được thêm vào ngay sau câu lệnh rẽ nhánh "BNE ACC.5, ABCD". Độ dời bước nhảy được tính và lưu trong câu lệnh "BNE" là 1.

Đối với câu lệnh nhảy, những câu lệnh trong khối cơ bản tạo bởi câu lệnh nhảy sẽ thay thế chính câu lệnh đó. Với ví dụ ở hình D.42, câu lệnh "JMP ABCD" biểu diễn lệnh nhảy

đến nhãn ABCD sẽ bị thay thể bởi câu lệnh "ADD, 0".

Với các lệnh rẽ nhánh hay nhảy mà được dùng để biểu diễn vòng lặp thì chúng không được xử lý bằng những cách trên. Khối cơ bản tạo bởi câu lệnh rẽ nhánh sẽ không được nối vào sau câu lệnh đó, chỉ có việc tính toán bước nhảy là được thực hiện. Câu lệnh nhảy cũng không được thay thế bởi khối cơ bản của nó, thay vào đó độ dời bước nhảy đến nhãn được tính toán và lưu lại.

IV. Kết quả hiện thực

1. Kết quả thử nghiệm các mẫu thử

Đối với việc dịch ngược mã Assembly máy 8051, Boomerang Decompiler đã giải mã được hầu hết tập lệnh của kiến trúc máy này (thiếu các lệnh DA và XCHD). Quá trình dịch ngược cũng tạo ra kết quả thành công đối với các mẫu thử tự viết cũng như những mẫu thử có sẵn trong thực tế. Kết quả chi tiết đối với việc dịch ngược mã Assembly 8051 được biểu diễn ở bảng D.3:

Bảng D.3: Kết quả thử nghiệm dịch ngược mã Assembly 8051

Trường hợp	Số mẫu thử đúng
Tính toán (cộng, trừ, nhân, chia) và trả về kết quả	10/10
Tính toán logical (XOR, AND, …) và trả về kết quả	5/5
Gọi hàm không tham số	2/2
Gọi hàm một tham số	2/2
Gọi hàm nhiều tham số	2/2
Biểu diễn cấu trúc Union và Struct	3/3
Câu lệnh rẽ nhánh	5/5
Câu lệnh vòng lặp	2/2
Biểu diễn con trỏ	3/3

Đối với máy Sparc, Boomerang Decompiler giải mã thử nghiệm đối với một số lệnh thông dụng. Kết quả thử nghiệm chi tiết đối với việc dịch ngược mã Assembly được biểu diễn ở bảng D.4:

Bảng D.4: Kết quả thử nghiệm dịch ngược mã Assembly Sparc

Trường hợp	Số testcase đúng
Tính toán (cộng, trừ, nhân, chia) và trả về kết quả	5/5
Gọi hàm thư viện	1/1
Gọi hàm một tham số	1/1
Gọi hàm hai tham số	1/1

2. Nhận xét

Việc dịch ngược mã Assembly sử dụng nền tảng của Boomerang Decompiler tạo ra kết quả tốt. Với việc xây dựng được cấu trúc lưu trữ mã Assembly và công cụ Boomerang Toolkit, Boomerang Decompiler có thể được tận dụng để mở rộng chức năng dịch ngược mã Assembly cho nhiều hệ máy khác nhau với chi phí được giảm thiểu đáng kể.

CHƯƠNG E
HIỆN THỰC CẢI TIẾN CHỨC NĂNG XÁC ĐỊNH HÀM NGUYÊN MẪU

Chương này sẽ trình bày chi tiết về quá trình cải tiến chức năng xác định hàm nguyên mẫu. Phần đầu tiên sẽ tập trung vào việc tìm ra điểm yếu trong quá trình xác định hàm nguyên mẫu hiện tại của Boomerang. Phần tiếp theo sẽ nói về cách thay đổi mô hình hiện tại của Boomerang và thiết kế các mô hình hệ thống nhằm phục vụ cho việc cải tiến chức năng xác định hàm nguyên mẫu.Phần thứ ba sẽ đề cập đến các công việc được thực hiện nhằm hiện thực giải pháp đã xác định ở phần đầu tiên dựa trên các mô hình hệ thống của Boomerang. Phần cuối cùng sẽ trình các kết quả của việc thử nghiệm.

I. Xác định điểm yếu giải thuật xác định hàm nguyên mẫu của Boomerang

1. Thiết lập thí nghiệm

Để tìm được các vấn đề còn tồn tại trong việc xác định hàm nguyên mẫu của hệ thống Boomerang hiện tại, chúng ta phải kiểm tra kết quả của Boomerang trong các trường hợp có thể có. Các mẫu thử(testcase) theo các trường hợp đó đã được tạo ra(tại test/testcases) và có kết quả khi chạy thử trên hệ thống Boomerang theo bảng các mẫu thử E.1 sau đây:

Bảng E.1 : Bảng mẫu thử kiểm tra kết quả xác định hàm nguyên mẫu của Boomerang

Mục	Test cases	Số testcase đúng
Main gọi hàm	long add1(int) long add2(int, int) long add10(int, int, int, int, int, int, int, int, int, int) float add1(float) float add2(float, float)	9/9

	float add10(float, float, int, int, float, float, int, int, float, float) long sumarray(int[],int) long sumarray(int[], int, int[], int)	
Main gọi hàm A, hàm A gọi hàm B	long add1(int) -> long add2(int) long add1(int) -> long add2(int, int) float add2(float, float)-> float add1(float) long add1(int) -> void printscreen(char*) long sumarray(int[], int) -> long add2(int)	5/5
Main gọi hàm A, hàm A gọi hàm B, hàm B gọi hàm C	long add1(int)-> long add2(int, int) -> long add3(int, int, int) float add1(float)-> float add2(float, float) -> float add3(float, float, float) long sumarray1(int[], int)-> long sumarray2(int[], int)-> long sumarray3(int[], int) long add1(int)-> long add2(int, int)-> void printscreen(char*)	4/4
Main gọi hàm A đệ quy	int fibonaci(int) int factor(int) int factor2(int) long sum2recur(int, int) int plusandminus(int)	2/5
Main gọi hàm A và B đệ quy vòng	int add1(int) -> int add2(int) -> int add1(int) float add1(float) -> float add2(float) -> float add1(float) int add2(int, int)->int add1(int) -> int add2(int, int) float add2(float, float)->float add1(float) -> int add2(float, float) long sumarray1(int[], int)-> long sumarray2(int[], int)-> long sumarray1(int[], int)	5/5

Kỹ thuật dịch ngược

Viết chương trình dưới dạng assembly(Chia theo cách truyền tham số)	Sử dụng thanh ghi chuẩn, cách xa lệnh gọi Sử dụng thanh ghi chuẩn, 1 tham số xa và 1 tham số gần lệnh gọi Sử dụng 1 thanh ghi không chuẩn, cách xa lệnh gọi Sử dụng 1 thanh ghi chuẩn và 1 thanh ghi không chuẩn, cách xa lệnh gọi Chỉ truyền vào stack, cách xa lệnh gọi Truyền 1 tham số từ hàm gọi qua hàm trung gian Truyền 2 tham số từ hàm gọi qua hàm trung gian Testcases tổng hợp	8/8

Bảng E.1 trên là kết quả kiểm thử việc xác định hàm nguyên mẫu của Boomrerang trong các trường hợp. Cột bên trái là các nhóm các mẫu thử, các mẫu thử sẽ được phân chia thành các nhóm theo cách gọi hàm(độ sâu của hàm được gọi), tính chất của hàm(có phải hàm đệ quy không?) và cách sinh ra mã(dùng trình biên dịch hay viết từ mã assembly). Cột ở giữa là thông tin của từng mẫu thử. Trong trường hợp mã sinh ra dùng trình biên dịch, thì thông tin này là hàm nguyên mẫu và cách gọi hàm, còn trong trường hợp mã được viết từ assembly, thì thông tin là cách truyền tham số từ hàm gọi vào hàm được gọi. Cột bên phải là kết quả kiểm tra (số mẫu thử được xác định hàm nguyên mẫu đúng/số mẫu thử). Qua kiểm tra, chúng ta có nhận xét như sau:

- Đối với trường hợp gọi hàm đơn giản (từ main gọi hàm A, main gọi hàm A->B, main gọi hàm A->B->C), Boomerang xác định đúng đối với tất cả các trường hợp số nguyên, số thực, mảng, con trỏ, và trường hợp có nhiều tham số.

- Đối với các trường hợp đệ quy, Boomerang xác định sai 3/5 trường hợp. Các trường hợp Boomerang xác định đúng là các trường hợp chỉ gọi đệ quy một lần trong một hàm. Còn 3 trường hợp Boomerang xác định sai là các trường hợp được gọi đệ quy 2 lần trong cùng 1 hàm.

- Đối với các trường hợp đệ quy vòng chỉ có một lệnh gọi đệ quy trong một hàm, Boomerang cũng xác định đúng các tham số.

- Đối với các trường hợp chương trình được viết dưới dạng assembly, và truyền tham số vào hàm không theo quy ước gọi hàm, Boomerang cũng xác định chính xác các tham số.

2. Phân tích và tìm nguyên nhân của điểm yếu

Như kết quả có được từ việc kiểm tra chức năng xác định hàm nguyên mẫu của Boomerang theo bảng mẫu thử ở trên, việc xác định hàm nguyên mẫu ở các hàm đệ quy được gọi lại nhiều lần không được tốt. Và để tìm hiểu xem chính xác là Boomerang đã sai ở điểm nào, chúng ta sẽ tiến hành phân tích mẫu thử có kết quả sai theo kỹ thuật xác định hàm nguyên mẫu dựa trên giải thuật phân tích dòng dữ liệu mà Boomerang tuân theo.

Hình E.1 dưới đây là mã C của hàm Fibonacci mà Boomerang xác định sai hàm nguyên mẫu:

```
int Fibonacci(int n)
{
  if ( n == 0 )
    return 0;
  else if ( n == 1 )
    return 1;
  Else
    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Hình E.1: Mã nguồn của hàm Fibonacci

Mã nguồn của hàm Fibonacci ở trên rất đơn giản, chỉ bao gồm 2 lần rẽ nhánh và 2 lệnh gọi đệ quy. Vì lý do mã nhị phân rất khó đọc, và mã assembly tương đương với mã nhị phân về ngữ nghĩa, nên chúng ta sẽ không đi từ mã nhị phân mà sẽ đi từ mã assembly của hàm Fibonacci này. Mã assembly của hàm Fibonacci được sinh ra từ trình biên dịch và thể hiện trong bảng E.2 sau, trong đó cột bên trái là các lệnh assembly, còn cột bên phải là ý nghĩa của từng dòng lệnh đó.

Bảng E.2: Lệnh assembly của hàm Fibonacci và ý nghĩa

Lệnh Assembly	Ý nghĩa của dòng lệnh
Fibonacci:	
save %sp, -96, %sp	Thực hiện chuyển đổi CWP
st %i0, [%fp+68]	Lưu giá trị thanh ghi i0 vào địa chỉ
ld [%fp+68], %g1	%fp+68
cmp %g1, 0	Lấy giá trị từ địa chỉ %fp+68 vào %g1
bne .L6	So sánh %g1 và 0
nop	

	mov	0, %g1	Nếu kết quả không bằng nhau, rẽ nhánh
	b	.L7	L6
	nop		
.L6:			%g1 = 0
	ld	[%fp+68], %g1	Rẽ nhánh xuống L7
	cmp	%g1, 1	
	bne	.L8	
	nop		Lưu giá trị từ địa chỉ %fp+68 vào %g1
	mov	1, %g1	So sánh %g1 và 1
	b	.L7	Không bằng nhau thì rẽ nhánh L8
	nop		
.L8:			%g1 = 1
	ld	[%fp+68], %g1	Rẽ nhánh xuống L7
	add	%g1, -1, %g1	
	mov	%g1, %o0	
	call	Fibonacci, 0	Lưu giá trị từ địa chỉ %fp+68 vào %g1
	nop		%g1 = %g1 – 1
	mov	%o0, %i5	%o0 = %g1
	ld	[%fp+68], %g1	Gọi hàm Fibonacci
	add	%g1, -2, %g1	
	mov	%g1, %o0	%i5 = %o0
	call	Fibonacci, 0	Lưu giá trị từ địa chỉ %fp+68 vào %g1
	nop		%g1 = %g1 -2
	mov	%o0, %g1	%o0 = %g1
	add	%i5, %g1, %g1	Gọi hàm Fibonacci
.L7:			
	mov	%g1, %i0	%g1 = %o0
	restore		%g1 = %i5 + %g1
	jmp	%o7+8	
	nop		%i0 = %g1
			Thực hiện chuyển đổi CWP
			Đến lệnh tiếp theo sau lệnh gọi hàm này

Kỹ thuật dịch ngược

Sau giai đoạn giải mã và dựa vào đặc tả SSL, đoạn mã assembly sẽ được chuyển về mã RTL và xử lý tiếp như bảng E.3 sau(ở dưới đây chỉ lấy những đoạn có liên quan đến việc xác định mã nguyên hàm):

Bảng E.3. Quá trình chuyển đổi từ mã Assembly của hàm Fibinacci về dạng SSA.

Mã Assembly	Mã RTL sau khi được đánh số dòng	Mã RTL sau giai đoạn SSA
save %sp, -96, %sp	1 tmp := r14 - 96 18 r24 := r8 19 r25 := r9 20 r26 := r10 21 r27 := r11 22 r28 := r12 23 r29 := r13 24 r30 := r14 25 r31 := r15 26 r14 := tmp	1 tmp := r14{-} – 96 18 r24 := r8{-} 19 r25 := r9{-} 20 r26 := r10{-} 21 r27 := r11{-} 22 r28 := r12{-} 23 r29 := r13{-} 24 r30 := r14{-} 25 r31 := r15{-} 26 r14 := tmp{1}
st %i0, [%fp+68] …	27 m[r30 + 68] := r24	27 m[r30{24} + 68] := r24{18}
call Fibonacci, 0	46 call Fibonacci	46 call Fibonacci
mov %o0, %i5	47 r29 := r8	**47 r29 := r8{46}**
call Fibonacci, 0	52 call Fibonacci	52 call Fibonacci
add %i5, %g1, %g1	54 tmp := r29 55 r1 := r29 + r1	**54 tmp := r29{52}** **55 r1 := r29{52} + r1{53}**
mov %g1, %i0	56 r24 := r1	56 r24 := r1{86}
.L7: Restore	84 RET	84 RET

Bảng E.3 trên là quá trình chuyển đổi từ mã assembly về mã RTL ở dạng SSA:

- Cột đầu tiên là các dòng lệnh Assembly. Ở đây chúng tôi chỉ lấy các dòng lệnh assembly có liên quan đến việc xác định hàm nguyên mẫu
- Cột thứ 2 là đoạn mã assembly đã được chuyển về mã RTL dựa trên đặc tả SSL(Specification Semantic Language). Đặc tả SSL chứ các thông tin chuyển đổi từ

mỗi dòng lệnh Assembly thành các lệnh RTL tương ứng trong giai đoạn giải mã. Mỗi lệnh assembly được chuyển đổi thành các lệnh RTL ở cột bên cạnh. Sau khi được chuyển đổi thành lệnh RTL, các lệnh này được đánh số thứ tự để tiện cho việc xử lý sau này. Ở trên ví dụ, lệnh RTL được đánh số từ 1, 18, 19, 20… Các lệnh còn lại như lệnh số 2, 3, 4, 5 … được bỏ qua để dễ đọc.

- Cột cuối cùng là kết quả của việc chuyển đổi từ mã RTL về dạng SSA. Các định nghĩa cho thanh ghi, ô nhớ trong bộ nhớ sẽ được đánh số và phân biệt sao cho không có trường hợp một thanh ghi, ô nhớ được định nghĩa hai lần. Như ở dòng số 47 r29 := r8 sẽ được chuyển đổi thành 47 r29:= r8{46}. Lệnh ở dòng số 55 r1:=r29 + r1 sẽ được chuyển đổi thành 55 r1 := r29{52} + r1{53}, lệnh này có thể hiểu là "giá trị của r1 ở dòng 55 bằng tổng giá trị của r29 ở dòng 52 cồng với giá trị của r1 ở dòng 53". Trong các trường hợp các lệnh gán mà vế phải chưa được định nghĩa trong hàm, thì dấu - được sử dụng để biểu thị điều này. Ví dụ như ở dòng số 18 r24 := r8, giá trị của thanh ghi r8 chưa được định nghĩa trước lệnh này, vì vậy, lệnh này sẽ được chuyển đổi thành 18 r24 := r8{-}. Ở đây cần lưu ý thêm là lệnh gọi hàm Fibonacci là lệnh gọi hàm đệ quy. Tại thời điểm này, các tham số cũng như vị trí trả về của lệnh gọi hàm này chưa được xác định rõ ràng, vì vậy chúng ta giải sử các vị trí dữ liệu được định nghĩa đến thời điểm này có thể đều là tham số, và các vị trí sống đều có thể là vị trí trả về. Do đó, chương trình sẽ hiểu là thanh ghi r29 có thể đã được thay đổi giá trị ở dòng lệnh gọi hàm số 52 call Fibonacci, vậy nên ở dòng lệnh số 54 tmp := r29{52}, giá trị của thanh ghi r29 được xem là lấy từ dòng lệnh số 52 chứ không phải dòng lệnh số 47.

Sau khi chuyển về dạng SSA form, Boomerang tiếp tục thực hiện quá trình lan truyền biểu thức và loại bỏ mã chết như sau:

Bảng E.4 Mã RTL của hàm Fibonacci ở dạng SSA sau quá trình lan truyền biểu thức và loại bỏ mã chết..

Mã RTL ở dạng SSA	Lan truyền biểu thức	Loại bỏ mã chết
18 r24 := r8{-}	18 r24 := r8{-}	**0 r8 :=** -
19 r25 := r9{-}	19 r25 := r9{-}	0 r13 := -
20 r26 := r10{-}	20 r26 := r10{-}	0 r14 := -
21 r27 := r11{-}	21 r27 := r11{-}	18 r24 := r8{-}
22 r28 := r12{-}	22 r28 := r12{-}	**23 r29 := r13{-}**
23 r29 := r13{-}	**23 r29 := r13{-}**	
24 r30 := r14{-}	24 r30 := r14{-}	
25 r31 := r15{-}	25 r31 := r15{-}	

27 m[r30{24} + 68] := r24{18}	27 m[r14{-}+ 68] := r14{-}	27 m[r30{24} + 68] := r14{-}
46 call Fibonacci	46 call Fibonacci	46 call Fibonacci
47 r29 := r8{46}	**47 r29 := r8{46}**	**47 r29 := r8{46}**
52 call Fibonacci	52 call Fibonacci	52 call Fibonacci
54 tmp := r29{52} 55 r1 := r29{52} + r1{53}	**54 tmp := r29{52}** 55 r1 := r29{52} + r1{53}	**54 tmp := r29{52}** 55 r1 := r29{52} + r1{53}
56 r24 := r1{86}	56 r24 := m[r14{-} - 8]{107}	56 r24 := m[r14{-} - 8]{107}
84 RET	84 RET	84 RET

Bảng E.4 ở thể hiện quá trình biến đổi mã RTL ở dạng SSA qua hai quá trình lan truyền biểu thức và loại bỏ mã chết. Các quá trình này được thực hiện như sau:

- Cột đầu tiên mà mã RTL ở dạng SSA được lấy từ giai đoạn trước.
- Cột ở giữa là kết quả có được sau quá trình lan truyền biểu thức, giá trị của các vị trí dữ liệu sẽ được lan truyền xuống dưới. Ví dụ như ở dòng lệnh số 27 m[r30{24} + 68] := r24{18}, lệnh này có ý nghĩa là gán giá trị của thanh ghi r24 ở dòng lệnh số 18 vào ô nhớ có địa chỉ là tổng của giá trị thanh ghi r30 ở dòng lệnh số 24 cộng thêm 6. Để ý thấy giá trị của r30 ở dòng lệnh số 24 (24 r30 := r14{-}), và giá trị của r24 ở dòng lệnh số 18 (18 r24 := r8{-}) có thể được lan truyền đến dòng lệnh số 27. Sau khi thay các vị trí dữ liệu r30{24} bằng r14{-} và r24{18} bằng r8{-}, chúng ta được dòng lệnh số 27 sau giai đoạn lan truyền dữ liệu là 27 m[r14{-} + 68] = r14{-}. Việc gọi hàm đệ quy, vốn chưa có các thông tin rõ ràng ở dòng lệnh số 46 và 52 khiến cho giá trị các thanh ghi r8 và r29 không được thay thế.
- Cột cuối cùng là kết quả sau giai đoạn loại bỏ mã chết. Trong giai đoạn này, các đoạn mã RTL, các phép gán không còn được sử dụng sẽ được loại bỏ. Ví dụ như ở dòng lệnh số 25 r31 := r15{-}, tất cả các dòng lệnh còn lại không sử dụng đến định nghĩa của thanh ghi r31 ở dòng lệnh này, nên dòng lệnh này được xem là mã chết, và được loại bỏ. Ngoài ra, kết thúc quá trình này, các thanh ghi được sử dụng mà không có định nghĩa khởi đầu như thanh ghi r8 ở dòng lệnh 18(r24 := r8{-}), thanh ghi r14 ở dòng lệnh 23(r29 := r13{-}) hay thanh ghi r14 ở dòng lệnh 27(m[r30{24} + 68] := r14{-}) sẽ được đưa lên dòng lệnh số 0 với phép gán cho – (mang ý nghĩa là null).

Kỹ thuật dịch ngược

Sau các giai đoạn biến đổi, các biến được sử dụng mà không có định nghĩa từ đầu nằm ở vị trí dòng lệnh số 0. Trong quá trình xác định hàm nguyên mẫu dựa trên giải thuật phân tích dòng dữ liệu, Boomerang sẽ xem các vị trí dữ liệu này là các vị trí dữ liệu "live-on-entry".

Như vậy, cuối cùng Boomerang sẽ nhận thấy các live-on-entry(Fibonacci) = {r18, r13, r14}. Các vị trí dữ liệu này đều không phải là các vị trí nằm trên bộ nhớ nên thỏa mãn yêu cầu para-filter.Trong đó r14 là thanh ghi chứa địa chỉ của con trỏ-chồng(stack-pointer) và là vị trí dữ liệu được bảo quản(preserved location), nên Boomerang sẽ quyết định danh sách các vị trí dữ liệu là tham số final-param(Fibonacci)={r8,r13}.

Sau quá trình phân tích này, Boomerang đã có thể sinh ra được đoạn mã nguồn như trong hình E.2 sau đây. Đối với đoạn mã nguồn này thì Boomerang đã xác định sai các tham số của hàm Fibonacci từ một thành hai tham số. Các dòng chú thích "Warning: also return in o5" và "WARNING: Also returning: o5:= i5" có nghĩa là ngoài kết quả trả về vào biến o0, Boomerang còn xác định là biến o5 là kết quả trả về, và kết quả đó được lấy từ biến i5 của hàm Fibonacci.

```
__size32 Fibonacci(__size32 param1, __size32 param2){
  __size32 g1; // r1
  __size32 i5; // r29
  __size32 o0; // r8
  __size32 o0_1; // r8{52}
  __size32 o5; // r13
  i5 = param2;
   if (param1 != 0) {
      if (param1 != 1) {
         o0 = Fibonacci(param1 - 1, param2); /* Warning: also results in o5 */
         i5 = o0;
         o0_1 = Fibonacci(param1 - 2, o5);
         g1 = o0 + o0_1;
      } else {
         g1 = 1;
      }
  } else {
      g1 = 0;
  }
  return g1; /* WARNING: Also returning: o5 := i5 */
```

```
}
```

Hình E.2: Mã nguồn hàm Fibonacci sinh ra từ Boomerang

Tuy nhiên, sau quá trình phân tích ra được mã nguồn như bảng trên, Boomerang còn trái qua quá trình phân tích để loại bỏ các tham số dư thừa đối với hàm đệ quy. Trong quá trình phân tích này, đối với tham số trả về, Boomerang nhận thấy tham số trả về o5 trong lệnh gọi đệ quy đầu tiên là tham số đầu vào của lệnh gọi thứ hai, điều đó có nghĩa là tham số trả về này được sử dụng, nên đó không phải là tham số dư thừa. Cũng vì o5 không phải là tham số nên biến i5 được xác định là được sử dụng qua lệnh gán o5 := i5. Biến i5 được sử dụng nên các lệnh gán với i5 không bị xác định là mã chết và không bị loại bỏ. Do đó, trong quá trình phân tích đối với tham số đầu vào, ngoài lệnh gọi đệ quy "o0 = Fibonacci(param1 - 1, param2);" thì param2 còn được sử dụng ở lệnh "i5 = param2;", vậy nên param2 cũng không phải là tham số dư thừa. Tuy nhiên, nếu để ý kỹ, chỉ có param1 mới là tham số của hàm Fibonacci.

Nhận xét:

- Việc nhận dạng nhầm thanh ghi r13 (cũng là thanh thi %o5) là tham số bắt nguồn từ lệnh gán %i5 = %o5 và việc sử dụng thanh ghi %i5 tại câu lệnh "mov %o0, %i5" và "add %i5, %g1, %g1". Điều này dẫn đến việc xác định sai tham số trong bước đầu của việc phân tích.
- Trong trường hợp không gọi lệnh đệ quy(bỏ 2 lệnh gọi đệ quy), thì giá trị của các thanh ghi sẽ được lan truyền một cách tường minh, kết quả trả về cuối cùng chỉ là %r8 +%r8 – 3. Dẫn đến việc các lệnh gán liên quan đến thanh ghi %i5 là dư thừa, và được loại bỏ. Như vậy cuối cùng cũng chỉ có %r8 là tham số.
- Trong trường hợp chỉ gọi lệnh đệ quy 1 lần, thì việc xác định %o5 là tham số trả về được

  xác định là dư thừa, bởi vì tham số này không được sử dụng ở bất kì lệnh nào ở hàm gọi, trừ lệnh trả về. Với việc xác định này thì việc sử dụng thanh ghi %i5 làm tham số cũng

  được xác định là dư thừa và được loại bỏ khỏi danh sách tham số trong giai đoạn loại bỏ các tham số dư thừa đối với hàm đệ quy. Bởi vì thanh ghi này chỉ được sử dụng để truyền tham số vào lệnh gọi (giá trị của param2 ở trên chỉ được truyền vào lệnh gọi hàm Fibonacci).
- Chỉ với trường hợp gọi đệ quy 2 hoặc nhiều lần, thì tham số truyền vào lệnh gọi hàm sau sẽ chịu ảnh hưởng bởi lệnh gọi hàm trước. Trong quá trình phân tích nhằm loại bỏ tham số dư thừa ở hàm đệ quy, hai quá trình phân tích đối với tham số trả về và tham số đầu vào được thực hiện tách biệt nhau, từng tham số sẽ được phân tích. Vì vậy, quá trình

phân tích này không thể phát hiện trường hợp dư thừa có liên quan đến cả tham số đầu vào và trả về như trong trường hợp trên.

II. Thiết kế giải pháp khắc phục lỗi sai của Boomerang

1. Phân tích và lựa chọn giải pháp

Ở trên chúng ta đã tìm ra được trường hợp mà Boomerang có thể xác định sai hàm nguyên mẫu. Như đã phân tích, trường hợp sai sót này là vì giai đoạn loại bỏ các tham số dư thừa đối với hàm đệ quy của giải thuật xác định hàm nguyên mẫu dựa trên phân tích dòng dữ liệu chưa thể kết hợp được hai quá trình phân tích để loại bỏ tham số đầu vào và tham số trả về, mà để hai quá trình này tách biệt với nhau.

Tuy nhiên, trên thực tế, các trường hợp mã nguồn đệ quy chỉ thường xảy ra trong trường hợp mã máy được biên dịch từ ngôn ngữ cấp cao xuống nhờ trình biên dịch chứ không xảy ra trong trường hợp mã được viết bằng tay. Và trong trường hợp nếu trình biên dịch tuân theo quy ước gọi hàm thì việc xác định hàm nguyên mẫu dựa trên quy ước gọi hàm rất chính xác. Để có thể lựa chọn được giải pháp, chúng ta sẽ so sánh hai kỹ thuật xác định hàm nguyên mẫu trong bảng E.6 sau đây:

Bảng E.6: So sánh hai kỹ thuật xác định hàm nguyên mẫu

Phân tích dòng dữ liệu	Quy ước gọi hàm
- Dựa trên giải thuật xác định hàm nguyên mẫu	- Dựa trên các quy ước được đặt ra
- Xác định đúng hàm nguyên mẫu cho phần lớn trường hợp	- Chỉ xác định được khi mã đầu vào tuân theo quy ước gọi hàm
- Xác định sai đối với các trường hợp phức tạp như gọi đệ quy nhiều lần	- Xác định chính xác khi đầu vào đã tuân theo quy ước gọi hàm.
- Áp dụng không phụ thuộc vào tất cả kiến trúc máy	- Mỗi kiến trúc máy và trình biên dịch có các quy ước gọi hàm khác nhau, cần áp dụng đúng tập quy ước cho các trường hợp khác nhau

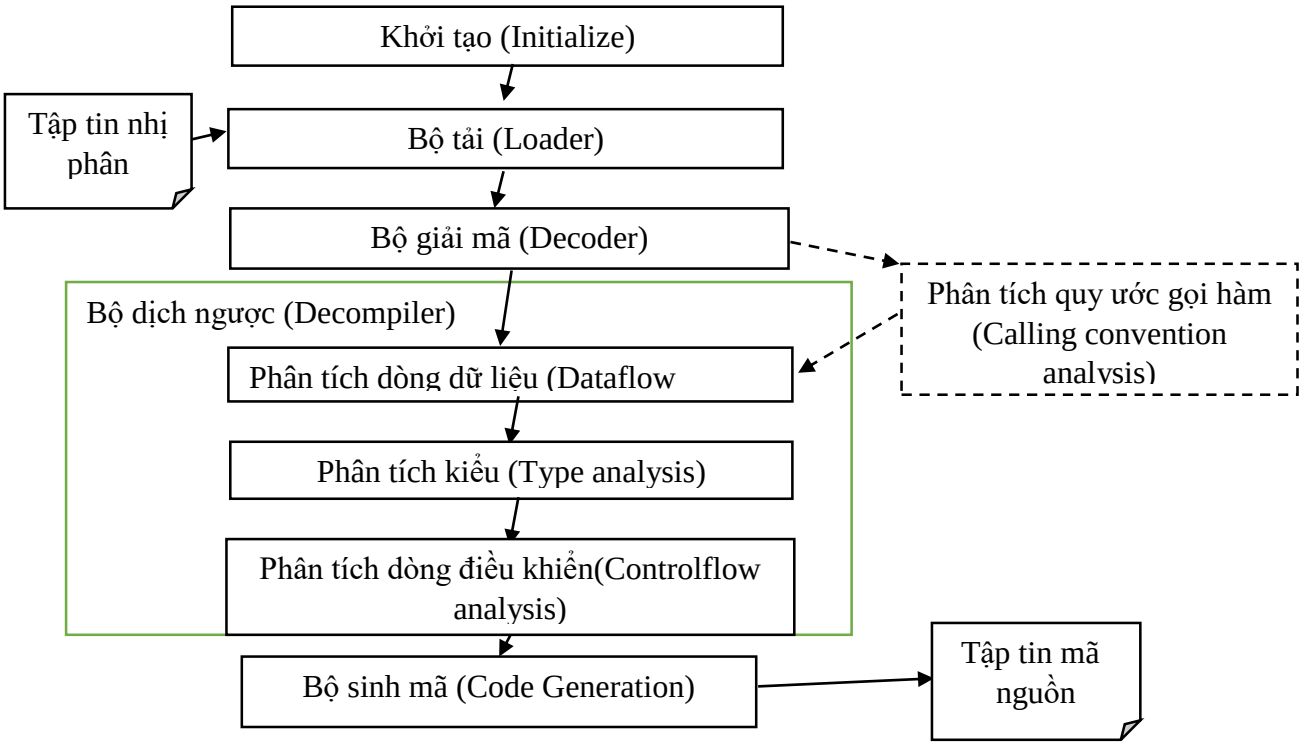
Như kết quả so sánh trong bảng trên, mỗi kỹ thuật xác định hàm nguyên mẫu đều có các điểm mạnh và điểm yếu riêng của mình. Xác định hàm nguyên mẫu dựa trên quy ước gọi hàm cần có mã đầu vào tuân theo quy ước gọi hàm, và khi mã đầu vào đã tuân theo quy ước gọi hàm

thì kết quả hàm nguyên mẫu sinh ra rất chính xác. Xác định hàm nguyên mẫu dựa trên giải thuật phân tích dòng dữ liệu thì có thể áp dụng rộng rãi trên các mã đầu vào. Tuy nhiên trong một số trường hợp cá biệt thì giải thuật này sẽ cho kết quả sai.

Để có thể áp dụng được cả ưu điểm của cả hai kỹ thuật trên, kỹ thuật xác định hàm nguyên mẫu dựa trên phân tích dòng dữ liệu vẫn sẽ được giữ lại để không phá vỡ cấu trúc hiện tại của Boomerang, song song kỹ thuật xác định hàm nguyên mẫu dựa trên quy ước gọi hàm sẽ được hiện thực. Như vậy, người dùng có thể xác định được đúng hàm nguyên mẫu cho tất cả các trường hợp chương trình được viết bằng tay (sử dụng giải thuật phân tích dòng dữ liệu) và được sinh ra từ trình biên dịch(sử dụng quy ước gọi hàm).

2. Mô hình xác định hàm nguyên mẫu dựa trên quy ước gọi hàm hiện thực trên Boomerang

Để có thể hiện thực chức năng xác định hàm nguyên mẫu dựa trên quy ước gọi hàm trên Boomerang, trước hết chúng ta sẽ thiết kế mô hình cải tiến dựa trên mô hình dịch ngược của Boomerang hiện tại. Mô hình cải tiến đó được trình bày trong hình E.3:



Hình E.3: Mô hình hệ thống Boomerang sau khi cải tiến chức năng xác định hàm nguyên mẫu

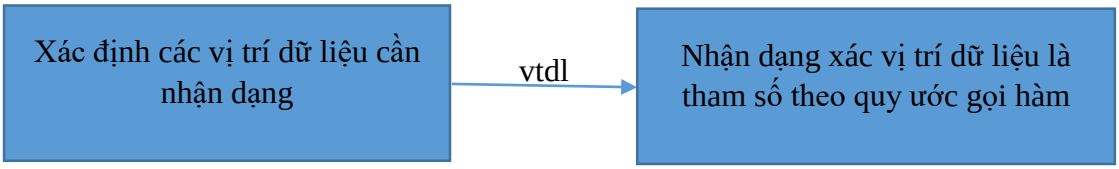
Trong hình E.3 trên, các phần nét liền là hệ thống đã có sẵn của Boomerang, phần nét đứt là nội dung sẽ được thêm vào. Trong đó module "phân tích quy ước gọi hàm" sẽ thực hiện

nhiệm vụ phân tích mã và dựa trên các chuẩn có sẵn để có thể tìm được hàm nguyên mẫu. Sau giai đoạn này, chúng ta sẽ có được thông tin về tham số của hàm nguyên mẫu. Chúng ta có được mô hình như trên vì các lý do sau:

- Chúng ta sẽ giữ lại luồng xử lý của Boomerang (phân tích dòng dữ liệu, phân tích kiểu và phân tích dòng điều khiển), bởi vì như vậy thì kiến trúc của Boomerang sẽ không bị phá vỡ. Các giai đoạn phân tích kiểu và phân tích dòng điều khiển đều có thể tận dụng được. Chỉ có kết quả của việc xác định hàm nguyên mẫu trong giai đoạn phân tích dòng dữ liệu được bỏ qua và thay thế bằng kết quả của giai đoạn phân tích quy ước gọi hàm. Ngoài ra, chúng ta còn có thể giữ được hai phương pháp xác định hàm nguyên mẫu trên hệ thống cho người dùng lựa chọn.
- Module "phân tích quy ước gọi hàm" sau khi Boomerang đã thực hiện giải mã. Vì giai đoạn giải mã, chương trình của chúng ta đang ở dạng nhị phân (mã máy), mỗi kiến trúc máy có một tập lệnh và mã nhị phân khác nhau, vì vậy rất khó để xử lý. Sau giai đoạn decode, mã nhị phân của các kiến trúc máy khác nhau đều được đưa về mã RTL, và phân chia vào các hàm, khối căn bản phù hợp. Lợi dụng sự thống nhất về ngôn ngữ cũng như việc được cấu trúc một phần vào các đối tượng dữ liệu của Boomerang, công việc xử lý phân tích quy ước gọi hàm sẽ nhẹ nhàng hơn.
- Module "phân tích quy ước gọi hàm" nằm trước giai đoạn phân tích dòng dữ liệu. Trong giai đoạn phân tích dòng dữ liệu này có hàng loạt các phép biến đổi. Việc áp dụng các quy ước gọi hàm không dựa trên các phép biến đổi này. Ngoài ra, các phép biến đổi này còn làm biến đổi vị trí, cũng như việc sử dụng các vị trí dữ liệu. Vì vậy, quá trình phân tích quy ước gọi hàm nên được thực hiện trước giai đoạn này.

3. Mô hình cơ chế nhận dạng tham số dựa trên quy ước gọi hàm

Ở trên, chúng ta đã có được mô hình cải tiến chức năng xác định hàm nguyên mẫu của hệ thống Boomerang, trong đó có module phân tích quy ước gọi hàm. Và ở phần này, mô hình của module phân tích quy ước gọi hàm đó sẽ được thiết kế như hình E.4 sau:

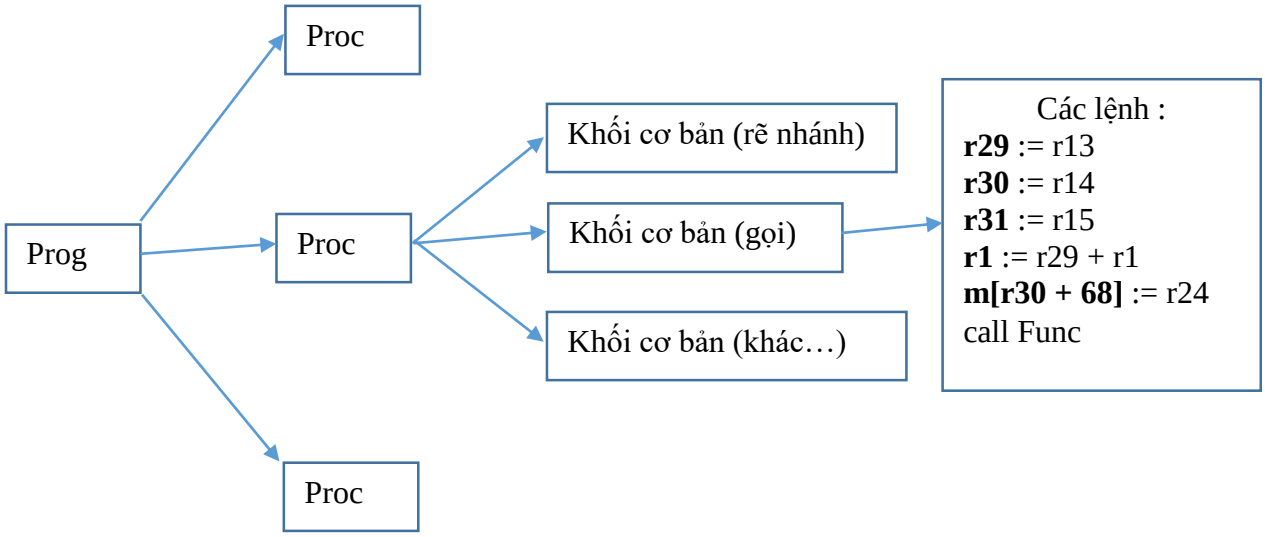


Hình E.4. Mô hình module phân tích quy ước gọi hàm

Hình E.4 phía trên thê hiện module phân tích quy ước gọi hàm. Đâu tiên, chúng ta cân xác định xem vị trí dữ liệu nào là các vị trí dữ liệu liên quan đến hàm gọi. Một chương trình có thể là rất lớn và sử dụng rất nhiều vị trí dữ liệu, nên chúng ta phải có cách lựa chọn phù hợp. Tiếp

theo, các vị trí dữ liệu được vào phần nhận dạng nhằm xác định tham số theo các quy tắc của quy ước gọi hàm.

Các vị trí dữ liệu cần nhận dạng sẽ được xác định dựa trên cấu trúc dữ liệu của Boomerang như hình E.5 sau:



Hình E.5: Mô hình xác định các vị trí dữ liệu liên quan đến lệnh gọi

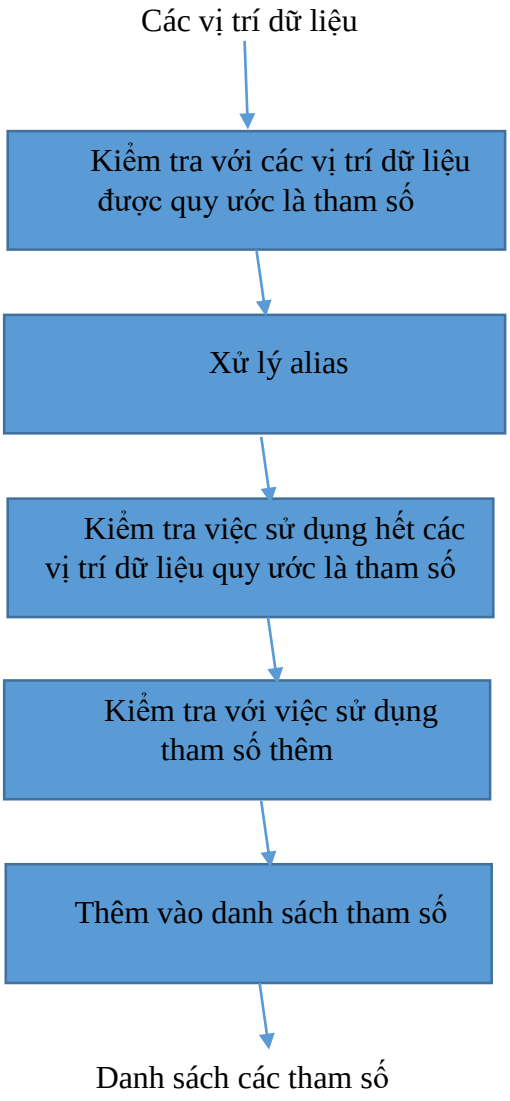
Như trong hình E.5 mà chúng ta có thể thấy, sau khi thực hiện quá trình giải mã, tất cả các lệnh RTL sẽ được phân chia về các khối cơ bản (basic block) thuộc proc. Đối với quá trình nhận dạng tham số dựa trên quy ước gọi hàm, chúng ta chỉ quan tâm đến các khối căn bản có lệnh gọi. Sau đó, trong khối căn bản này, chúng ta sẽ xét đến vế trái của các lệnh gán. Bởi vì vế trái của các lệnh gán này là các đối tượng sẽ có giá trị thay đổi và có thể là tham số của lệnh gọi phía sau. Như ví dụ trên hình, thì các vị trí dữ liệu được in đậm (r29, r30, r31, r1, m[r30 + 68]) sẽ được đưa vào quá trình nhận dạng tham số dựa trên quy ước gọi hàm.

Quá trình nhận dạng các tham số dựa theo quy ước gọi hàm sẽ được hiện thực theo mô hình trong hình E.6 sau đây:

- Các vị trí dữ liệu từ tập danh sách đã có từ trên sẽ được kiểm tra với tập tham số được quy ước sẵn.
- Đối với các trường hợp vị trí dữ liệu có alias, thì chúng ta cũng xử lý để có thể đưa các vị trí được alias này vào danh sách tham số
- Số lượng vị trí dữ liệu được quy ước là tham số có giới hạn. Chúng ta sẽ kiểm tra xem danh sách đó đã được sử dụng hết chưa, để có thể đưa các trường hợp sử dụng tham số thêm vào (như con trỏ-chồng) vào xử lý.
- Đối với trường hợp đã sử dụng hết tham số có sẵn, chúng ta sẽ kiểm tra với trường hợp các “tham số thêm”

Kỹ thuật dịch ngược

- Sau khi xác định được các vị trí dữ liệu là tham số, chúng ta đưa các vị trí dữ liệu này vào danh sách tham số để có thể xử lý, đưa vào hệ thống Boomerang.

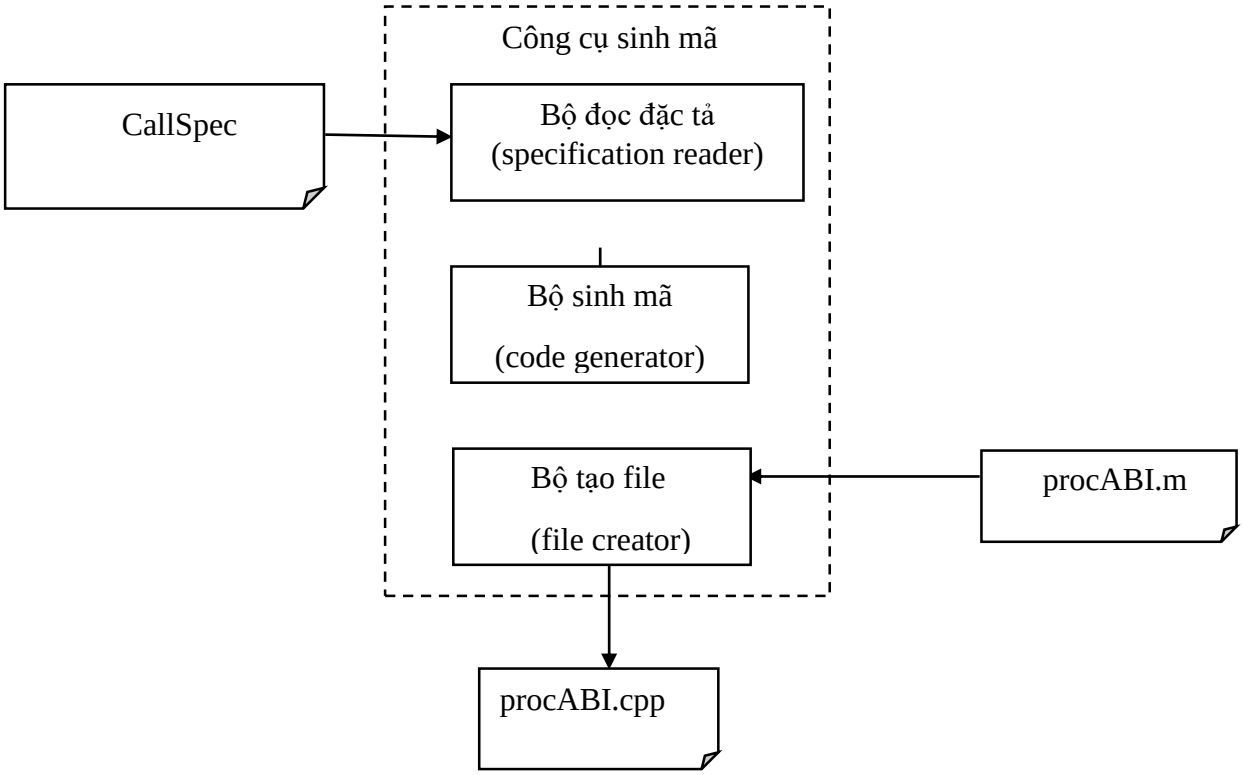


Hình E.6: Mô hình quá trình nhận dạng các vị trí dữ liệu là tham số

4. Mô hình công cụ tự động sinh mã nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm

Như chúng ta đã thấy, ở mô hình trên, quy trình xác định các tham số dựa trên quy ước gọi hàm là tương tự đối với các kiến trúc máy khác nhau, và chỉ khác nhau ở chi tiết của các quy ước này. Ngoài ra, việc viết mã cho mỗi quy ước gọi hàm rất tốn công và khó thừa kế, vậy nên chúng ta sẽ phát triển công cụ tự động sinh mã của module phân tích quy ước gọi hàm dựa trên đặc tả của quy ước gọi hàm.

Để có thể viết được công cụ tự động sinh mã trên, chúng ta sẽ thiết kế mô hình hệ thống của công cụ này. Hình E.7 phía dưới thể hiện mô hình hệ thống đó:



Hình E.7. Mô hình công cụ tự sinh mã nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm

Mô hình công cụ tự động sinh mã này gồm có các thành phần như sau:

- **CallSpec**: Tập tin đặc tả chứa các thông tin về quy ước gọi hàm của các hệ kiến trúc máy. Các đặc tả này phải tuân theo định dạng sẽ được trình bày ở phần sau.
- **procABI.m**: Tập tin khung sường sẽ được dùng để sinh ra tập tin procABI.cpp. Tập tin này sẽ chứa các đoạn mã không thay đổi và các dấu hiệu để công cụ sinh mã nhận dạng được vị trí cần sinh mã.
- **procABI.cpp**: Tập tin mã nguồn được sinh ra, chứa module cần thiết để nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm.
- **Công cụ sinh mã**: công tự động sinh mã của module phân tích quy ước gọi hàm, thực chất là sinh ra tập tin procABI.cpp. Công cụ này bao gồm các thành phần là **bộ đọc đặc tả**, **bộ sinh mã** và **bộ tạo tập tin**.
- **Bộ đọc đặc tả:** Có nhiệm vụ đọc các đặc tả về quy ước gọi hàm từ tập tin **CallSpec,** xử lý và lưu trữ vào cấu trúc dữ liệu dưới dạng cấu trúc mà công cụ tự định nghĩa.
- **Bộ sinh mã:** phần này sẽ dựa vào các thông tin đã xử lý từ **bộ đọc đặc tả**, sinh ra các đoạn mã xử lý dựa trên đặc tả của từng kiến trúc máy.
- **Bộ tạo tập tin:** phần này sẽ đọc tập tin khung sường **procABI.m,** nhận dạng vị trí cần sinh mã và dựa vào các đoạn mã có được từ **bộ sinh mã** để sinh ra tập tin **procABI.cpp**

III. Chi tiết hiện thực

1. Đặc tả quy ước gọi hàm

Để có thể sinh ra mã của module phân tích quy ước gọi hàm, chúng ta phải có tập tin đặc tả quy ước gọi hàm của các kiến trúc máy khác nhau. Công cụ của chúng ta sẽ đọc tập tin đặc tả này để có thông tin về quy ước gọi hàm của các kiến trúc máy, rồi dựa vào đó để sinh ra các đoạn mã nhận dạng tương ứng.

Tuy nhiên, hiện nay chưa có ngôn ngữ đặc tả chuẩn nào cho quy ước gọi hàm mà chỉ có các khuyến cáo về ngôn ngữ đặc tả này. Các khuyến cáo này không được tuân thủ bởi các tổ chức, công ty sản xuất vi xử lý các kiến trúc máy(Intel, Oracle, Apple…) và trình biên dịch. Mà hầu hết các đặc tả sẽ được thể hiện bằng ngôn ngữ thông dụng. Vì vậy nên chúng tôi sẽ tìm hiểu quy ước gọi hàm và tự xây dựng cấu trúc của tập tin đặc tả. Tập tin đặc tả này không chứa toàn bộ thông tin của quy ước gọi hàm, mà chỉ chứa các thông tin vừa đủ để nhận dạng hàm nguyên mẫu, phục vụ cho mục đích của đề tài.

Tập tin đặc tả CallSpec sẽ có định dạng được quy định như sau:

1. Tập tin CallSpec sẽ chứa các đặc tả gọi hàm của các platform
2. Các Platform sẽ được cách nhau bằng dấu xuống dòng nếu nhiều hơn một platform
3. Platform được đặc tả theo thứ tự là dấu thăng (#), tên platform, dấu mở ngoặc nhọn {, các phép gán, và kết thúc bằng dấu đóng ngoặc nhọn }
4. Các phép gán sẽ được cách nhau bằng dấu xuống dòng nếu nhiều hơn một phép gán
5. Phép gán sẽ có định dạng theo thứ tự là thuộc tính, dấu bằng (=), các giá trị
6. Các giá trị có thể là các vị trí dữ liệu hoặc hai số 0 hoặc 1(đại diện cho biến bool)
7. Các vị trí dữ liệu sẽ được cách nhau bằng dấu phẩy(,) nếu nhiều hơn 1 vị trí dữ liệu
8. Vị trí dữ liệu có thể là thanh ghi hoặc ô nhớ
9. Ô nhớ được đặc tả dưới dạng chữ m, theo sau là địa chỉ
10. Địa chỉ có thể là thanh ghi, hoặc là thanh ghi cộng thêm offset
11. Offset là một số nguyên.
12. Thanh ghi được đặc tả là r, theo sau là số nguyên. Ví dụ r9, r18
13. Tên platform là tên các kiến trúc máy. Tên này phải tương ứng với tên frontend mà Boomerang đang xử lý. Hiện tại có thể dùng các tên PLAT_SPARC (kiến trúc máy SPARC), PLAT_PENTIUM (kiến trúc máy PENTIUM), PLAT_PPC (kiến trúc máy POWERPC), PLAT_MIPS (kiến trúc máy MIPS), PLAT_ST20 (Kiến trúc máy ST20).
14. Thuộc tính là các quy ước gọi hàm. Các tên thuộc tính thực chất là chuỗi này được đặc theo kiểu gợi nhớ để có thể xử lý ở phần sau.

Kỹ thuật dịch ngược

Các quy ước đó sẽ được mô tả bằng biểu thức chính quy trong bảng E.5 sau:

Bảng E.5: Bảng định nghĩa chính quy mô tả quy ước gọi hàm

Số quy định	Tên chính quy	Biểu thức chính quy
14	Thuộc_Tính	parameters|add_params|righttoleft|return|alias
13	Tên_Platform	PLAT_SPARC|PLAT_PENTIUM|PLAT_PPC|PLAT_MIPS|PLAT_ST20
12	Thanh_Ghi	r[0-9]+
11	Offset	[0-9]+
10	Địa_Chỉ	Thanh_Ghi (+ Offset)?
9	Ô_Nhớ	m[Địa_Chỉ]
8	Vị_Trí_Dữ_Liệu	Thanh_ghi|Ô_Nhớ
7	Các_Vị_Trí_Dữ_Liệu	Vị_Trí_Dữ_Liệu(,Vị_Trí_Dữ_Liệu)*
6	Các_Giá_Trị	Các_Vị_Trí_Dữ_Liệu|(1|0)
5	Phép_Gán	Thuộc_Tính = Các_Giá_Trị
4	Các_Phép_Gán	Phép_Gán(\n[\r]Phép_Gán)*

3	Platform	#Tên_Platform{
		Các_Phép_Gán
		}

2	Các_Platform	Platform(\n[\r]Platform)*
1	CallSpec	Các_Platform

Sau khi xây dựng được cấu trúc của tập tin đặc tả và tìm hiểu về quy ước của các kiến trúc máy Sparc, Pentium và PowerPC, tập tin CallSpec được xây dựng lên có nội dung theo hình E.8 sau:

```
#PLAT_SPARC{
        parameters = r8, r9, r10, r11, r12, r13
        add_params = r14
        righttoleft = 1
        return = r24
}
#PLAT_PENTIUM{
```

```
        righttoleft = 1
        return = r24
        add_params = r28



}
#PLAT_PPC{
        return = r24
        parameters = r3,r4,r5,r6,r7,r8,r9,r10
        alias  = m[r1 + 24], m[r1 + 28], m[r1 + 32], m[r1 + 36], m[r1 + 40], m[r1 +
44], m[r1 + 48], m[r1 + 52]
        righttoleft = 1
        add_params = r1
}
```

Hình E.8: Nội dung của tập tin đặc tả quy ước gọi hàm CallSpec

Hình E.8 ở trên là nội dung của tập tin CallSpec chứa đặc tả của của các kiến trúc máy Sparc, Pentium và Power PC. Một số các thuộc tính được sử dụng có ý nghĩa như sau:

- Return: vị trí dữ liệu được quy ước sẽ mang giá trị trả về.
- Parameter: các vị trí dữ liệu được quy ước dùng để truyền tham số vào
- Add_params: thanh ghi được sử dụng để lấy truyền tham số vào trong trường hợp có nhiều tham số hơn số vị trí dữ liệu ở thuộc tính parameter. Thanh ghi này thường là thanh ghi con trỏ chồng
- Righttoleft: Các tham số có được truyền vào theo thứ tự từ phải qua trái không
- Alias: các vị trí dữ liệu tương đương với các vị trí dữ liệu trong thuộc tính parameter

2. Xây dựng công cụ tự động sinh mã

a. Xây dựng module đọc đặc tả từ tập tin CallSpec

Việc xây dựng module đọc đặc tả từ tập tin CallSpec là cần thiết để có thể xây dựng công cụ tự đông sinh mã. Các thông tin trong tập tin CallSpec chỉ là dòng text, để có thể đưa các thông tin này vào công cụ của chúng ta phục vụ cho việc xử lý sau này, thì chúng ta cần phải phân tích ngữ pháp (parse) tập tin này.

Hình E.9 thể hiện việc phân tích cú pháp tập tin CallSpec. Trong giai đoạn này, chúng ta sẽ đọc tập tin đặc tả và đưa các thông tin cần thiết vào đối tượng của chương trình, chi tiết như sau:

Kỹ thuật dịch ngược

```
def read_CallSpec
content = File.read('CallSpec')

        machines = content.scan(/^\s*\#([^\#\{}]+)\{([^\#]+)\}/m )
        callSpecs = [] ……
        for machine in machines
 assignments=machine[1].scan(/[\n\r]*([a-zA-Z_]+)[ ]*=[ ]*([a-zA-Z0-9,\[\]\+\-
 ]+)[\n\r]+/m)

                for assignment in assignments
                        #print assignment[0]  + "123"
                        if assignment[0]=~/parameters/
                                p3 = assignment[1].gsub(/\s+/, "").split(',')
                        end
                        if assignment[0]=~/add_params/
                                p4 = assignment[1]#.gsub(/\s+/, "").split(',')
                        end
                        if assignment[0]=~/alias/
                                p5 = assignment[1].gsub(/\s*,\s*/, ",").split(',')
                        end
                        if assignment[0]=~/return/
                                p2 = assignment[1].gsub(/\s+/, "").split(',')
                        end
                        if assignment[0]=~/righttoleft/
                                p8 = assignment[1]#.gsub(/\s+/, "").split(',')
                        end
                end
……
```

Hình E.9: Hiện thực chức năng đọc thông tin từ tập tin CallSpec

Đầu tiên, nội dung tập tin CallSpec sẽ được phân tách thành các platform khác nhau cùng với tên platform dựa vào biểu thức chính quy như hình E.10:

```
machines = content.scan(/^\s*\#([^\#\{}]+)\{([^\#]+)\}/m )
```

Hình E.10: Đoạn mã xử lý với biểu thức chính quy

Biểu thức chính quy này sẽ giúp chúng ta làm các công việc sau đây:

- Phân tách các platform khác nhau. Mỗi platform sẽ thỏa mãn biểu thức `^\s*\#([^\#\{}]+)\{([^\#]+)\}`
- Tìm ra tên của mỗi platform, tên platform sẽ nằm ở vị trí của biểu thức chính quy `([^\#\{}]+)`
- Tìm ra nội dung của quy ước gọi hàm, nội dung này sẽ nằm ở vị trí của biểu thức chính quy `([^\#]+)`

Sau đó, trên mỗi platform, các dòng lệnh có phép gán sẽ được nhận dạng và phân tách thành thuộc tính cùng với các giá trị.

```
assignments=machine[1].scan(/[\n\r]*([a-zA-Z_]+)[ ]*=[ ]*([a-zA-Z0-9,\[\]\+\-
]+)[\n\r]+/m)
```

Hình E.11: Đoạn mã xử lý với biểu thức chính quy

Đoạn mã với biểu thức chính quy trong hình E.11 này sẽ giúp chúng ta làm các công việc sau đây:

- Phân tách các phép gán khác nhau. Mỗi dòng thỏa mãn biểu thức chính quy `(/[\n\r]*([a-zA-Z_]+)[ ]*=[ ]*([a-zA-Z0-9,\[\]\+\- ]+)[\n\r]+` là một phép gán.
- Tìm ra tên thuộc tính, tên thuộc tính sẽ nằm ở vị trí của biểu thức chính quy `([a-zA-Z_]+)`
- Tìm ra giá trị của thuộc tính, giá trị này sẽ nằm ở vị trí của biểu thức chính quy `([a-zA-Z0-9,\[\]\+\- ]+)`

Tiếp theo, tùy vào thuộc tính của phép gán mà ta phân tích và lưu trữ các giá trị này xuống các đối tượng riêng lẻ như trong hình E.12.

```
if assignment[0]=~/parameters/
        p3 = assignment[1].gsub(/\s+/, "").split(',')
End
if assignment[0]=~/add_params/
        p4 = assignment[1]#.gsub(/\s+/, "").split(',')
End
if assignment[0]=~/alias/
        p5 = assignment[1].gsub(/\s*,\s*/, ",").split(',')
End
if assignment[0]=~/return/
        p2 = assignment[1].gsub(/\s+/, "").split(',')
end
if assignment[0]=~/righttoleft/
        p8 = assignment[1]#.gsub(/\s+/, "").split(',')
End
```

Hình E.12: Đoạn mã nhận dạng từng thuộc tính của quy ước gọi hàm

Sau đó, chúng ta đưa các đối tượng này vào Struct Call Spec có cấu trúc nhất định đã được khai báo từ trước. Khi thực hiện xong việc phân tích ngữ pháp đối với các đặc tả, chúng ta sẽ có một mảng chứa các đặc tả quy ước gọi hàm của các máy.

b. Xử lý và sinh mã

Sau khi có được các thông tin từ tập tin CallSpec, các thông tin đó được đưa vào bộ sinh mã để có thể xử lý sinh ra các đoạn mã xử lý phù hợp, nhằm hiện thực module phân tích quy ước gọi hàm.

Để công việc hiện thực việc xử lý và sinh mã được dễ dàng, một số hàm biến đổi được hiện thực. Mặc dù các thông tin từ tập tin CallSpec đã được đưa vào chương trình, nhưng đây vẫn

không phải là thông tin mà Boomerang có thể hiểu được. Các thông tin về vị trí dữ liệu cần phải được đưa qua các hàm biến đổi, nhờ đó sinh ra các đoạn mã sử dụng cấu trúc dữ liệu hiện tại của Boomerang, nhờ đó Boomerang mới có thể hiểu được. Các hàm biến đổi đó sẽ được trình bày sau đây:

- Biến đổi về hằng số: hàm biến đổi sẽ nhận dạng số tự nhiên dựa vào biểu thức chính quy `(\d+)` và đưa thành đối tượng Const của Boomerang như hình E.14

```
def transform_const str
        temp = str.scan(/(\d+)/)
        code = ""
        code << "new Const((int)#{temp[0][0]})"
        return code
End
```

Hình E.14: Đoạn mã của hàm transform_const

- Biến đổi về thanh ghi: chương trình sẽ nhận dạng vị trí dữ liệu là thanh ghi dựa vào biểu thức chính quy `r(\d+)` và chuyển thành đối tượng Location:Regof của Boomerang như trong hình E.15.

```
def transform_reg str
        temp = str.scan(/r(\d+)/)
        code =""
        code <<"Location::regOf(#{temp[0][0]})"

        return code
End
```

Hình E.15: Đoạn mã của hàm transform_reg

- Biến đổi về biểu thức nhị phân: chương trình sẽ nhận dạng biểu thức nhị phân dựa vào biểu thức chính quy `(r\d+)\s*(\+|\-)\s*(\d+)` và chuyển đổi thành đối tượng Binary của Boomerang như trong hình E.16:

```
def transform_reg_op str
        temp = str.scan(/(r\d+)\s*(\+|\-)\s*(\d+)/)

        code = ""
        code<<"(new Binary("
        code<< (temp[0][1]=='+' ? "opPlus, " : "opMinus, ")
        code<< transform_reg(temp[0][0])
        code << ", "
        code <<transform_const(temp[0][2])
```

```
        code<<")"
        code<<")"
        return code
End
```

Hình E.16: Đoạn mã của hàm transform_reg_op

- Biến đổi về ô nhớ trên bộ nhớ: chương trình sẽ nhận dạng các vị trí dữ liệu là ô nhớ trên bộ nhớ dựa vào biểu thức chính quy `m\[(r\d+\s*[+-]\s*\d+)\]` và biến đổi về đối tượng Location:Memof của Boomerang như trong hình E.17.

```
def transform_mem_op str
        temp = str.scan(/m\[(r\d+\s*[+-]\s*\d+)\]/)

        code = ""
        code <<"Location::memOf("
        code << transform_reg_op(temp[0][0])
        code <<")"
        return code
End
```

Hình E.17: Đoạn mã của hàm transform_mem_of

Sau khi hiện thực các phép biến đổi để hỗ trợ, chúng ta tiến hành hiện thực phần xử lý và sinh mã. Hình E.18 thể hiện một phần trong giai đoạn xử lý và sinh ra mã.

```
def code_generation (specification)
        code = ""
        code<<"switch (prog->getFrontEndId()){\n" ##switch machine architecture
……
        for machine in specification
……..             for i in 1..machine[:param].length…
……..             for reg in machine[:param] ## process parameter…
……..             for aliasreg in machine[:alias]…
……..             if !machine[:additionParam].nil? …
……..                       if machine[:platform] == "PLAT_PENTIUM"….
```

Hình E.18: Hiện thực việc xử lý các thông tin và sinh ra mã

Trong Giai đoạn này, chúng ta sẽ sinh ra các đoạn code phụ thuộc vào từng kiến trúc máy. Đối với từng loại kiến trúc máy, chúng ta sẽ lần lượt dựa vào các thông tin về việc sử dụng các thanh ghi, truyền tham số có được từ trên, cộng với việc sử dụng các hàm biến đổi, qua đó xử lý để sinh ra được các đoạn mã C++ để nhận dạng các tham số dựa tên quy ước gọi hàm. Các đoạn mã C++ này sẽ được giải thích ở phần 3.

c. Xử lý sinh ra tập tin cpp

Sau khi xử lý và sinh ra các đoạn mã phù hợp, các đoạn mã đó được đưa vào giai đoạn này để có thể sinh ra tập tin cpp. Hình E.19 là quá trình nhận dạng và sinh ra tập tin cpp này:

```
def write_cpp_file
        mfile=""
        indent = 0;
       File.open("procABI.m").each do |line|
              mfile<<line
              if line =~ /.*{.*/
                     indent=indent+1
                     #print line
              end
              if line =~ /.*}.*/
                     indent=indent-1
              end
              if line =~ /\/\/##@@.+/
                     indentstr="";
                     #print indent
                     for temp in 1..indent
                            indentstr<<"\t"
                     end
                     @content.each_line do |linecode|
                            mfile<<indentstr
                            mfile<<linecode
                     end
              end
       end
       @content=mfile
       write_to_output "test.cpp"
end
```

Hình E.19: Hiện thực việc xuất mã ra tập tin cpp

Trong giai đoạn sinh ra tập tin cpp này, sẽ dựa vào tập tin mẫu có các đoạn mã xử lý chung (procABI.m), nhận dạng vị trí cần sinh mã, và đưa đoạn mã đã có được từ giai đoạn xử lý và sinh mã vào. Vị trí cần sinh mã được nhận dạng dựa vào biểu thức chính quy như hình E.20 sau:

```
if line =~ /\/\/##@@.+/
```

Hình E.20: Đoạn mã xử lý với biểu thức chính quy

Tập tin sườn procABI.m sẽ chứa các đoạn mã xử lý chung như hình E.21 sau đây:

```
#include "proc.h"
void UserProc::findABIParameters(){
       BB_IT it;
       for (PBB bb = cfg->getFirstBB(it); bb; bb = cfg->getNextBB(it)) {
              if (bb->getType() == CALL) {
```

```
                        CallStatement* call = (CallStatement*)bb->getRTLs()->back()-
>getHlStmt();
                        if (!call->isCall()) {
                                LOG << "bb at " << bb->getLowAddr() << " is a CALL
but last stmt is not a call: " << call << "\n";
                        }
                        assert(call->isCall());
                        UserProc* c = (UserProc*)call->getDestProc();
                        if (c == NULL || c->isLib()) continue;
                        BB_IT previous=it;
                        previous--;
                        std::cout<<"Get user call "<<c->getName()<<"from"<<
getName()<<"\n";

                        std::cout<<" current bb test remake";
                        StatementList stmts;
                        bb->getStatements(stmts);
                        StatementList::iterator stit;
        //##@@EACH TYPE OF MACHINE WILL BE PROCESSED HERE //DONT DELETE THIS LINE
                                                //FINISH PROCESSED
```

Hình E.21: Nội dung tập tin sườn procABI.m

Như hình trên, tập tin sườn sẽ có các công đoạn khai báo các thư viện liên quan, khai báo hàm, các đoạn mã tiền xử lý đối với từng loại máy, và dấu hiệu nhận biết vị trí sinh mã (//##@@).

3. Hiện thực cơ chế xử lý dựa trên quy ước gọi hàm

Để hiện thực được việc xử lý nhận dạng các tham số dựa trên quy ước gọi hàm trên hệ thống Boomerang, chúng ta sẽ kết hợp các đặc tả của quy ước gọi hàm, xử lý dựa trên các đối tượng có sẵn của Boomerang, để đưa các kết quả có được vào hệ thống. Cụ thể như sau:

- Chúng ta sẽ duyệt qua tất cả các khối cơ bản nằm trong luồng của proc, và chỉ xử lý đối với các khối cơ bản có chứa lệnh gọi hàm, và không phải là gọi thư viện. Khối căn bản là khối bao gồm các lệnh xử lý có liên quan với nhau. Trong khối căn bản bao gồm cả lệnh gọi hàm, sẽ có chứa các phép gán, trong đó có các phép gán để tuân theo quy ước gọi hàm. Các hàm thư viện đã có hàm nguyên mẫu sẵn nên không được xét đến trong quá trình này nữa. Quá trình này được hiện thực như đoạn mã trong hình E.21:

```
for (PBB bb = cfg->getFirstBB(it); bb; bb = cfg->getNextBB(it)) {
        if (bb->getType() == CALL) {
                ……
                if (!call->isCall()) { ……}
                if (c == NULL || c->isLib()) continue;
```

Hình E.21: Đoạn mã xác định các khối căn bản cần xử lý

- Sau khi tìm được các khối căn bản cần xử lý, chúng ta sẽ dùng switch có thể xử lý tách biệt đối với các kiến trúc máy khác nhau như trong hình E.22. Sau giai đoạn này, Các trường hợp xử lý đều dựa trên đặc tả quy ước gọi hàm, vậy nên chúng tôi sẽ giải thích theo các thuộc tính đặc tả đó.

```
bb->getStatements(stmts);
switch (prog->getFrontEndId()){
                case PLAT_SPARC:{ … }
                case PLAT_PENTIUM:{ … }
                case PLAT_PPC:{ … }
}
```

Hình E.22: Đoạn mã rẽ nhánh xử lý với các kiến trúc máy khác nhau

- Lần lượt duyệt qua các dòng lệnh thuộc khối cơ bản. Chúng ta chỉ xử lý đối với các lệnh là phép gán. Chúng ta sẽ bỏ qua các lệnh khác như lệnh rẽ nhánh, gọi hàm… bởi vì các lệnh này không liên quan đến đặc tả gọi hàm. Trong các phép gán, chúng ta sẽ tập trung vào vế trái (LHS – left hand side), bởi vì đây là các đối tượng có giá trị sẽ thay đổi, và có thể là tham số của lệnh gọi hàm. Các đối tượng này được xác định như hình E.23 sau:

```
for (stit = stmts.begin(); stit != stmts.end(); ++stit) {
 if(!s->isAssignment())
        continue;
  Exp *lhs = ((Assignment*)s)->getLeft();
```

Hình E.23: Đoạn mã nhận dạng các đối tượng dữ liệu cần xử lý

- Đối với các kiến trúc máy có quy ước về các thanh ghi được sử dụng làm tham số, chúng ta sẽ so sánh các giá trị vế trái này với các thanh ghi được quy ước đó. Trong trường hợp phép so sánh này cho kết quả bằng nhau, thì chúng ta sẽ bật cờ là tham số để xử lý sau. Việc so sánh được thực hiện như hình E.24:

```
if(((std::string)lhs->prints())==(std::string)Location::regOf(3)->prints()){
   ispara3 = true; …
}
```

Hình E.24: Đoạn mã ví dụ xử lý đối với thuộc tính parameters

- Sau khi kiểm tra việc sử dụng hết các thanh ghi được quy ước là tham số (số lượng tham số lớn, hoặc không có thanh ghi nào được quy ước sẽ là tham số), chúng ta sẽ xét đến trường hợp đưa tham số vào lệnh gọi hàm bằng cách sử dụng “addparams”. Đây thường là con trỏ chồng và nó thường được sử dụng để đưa giá trị vào chồng (stack). Tuy nhiên đối với máy Pentium thì offset sẽ được cộng thêm 4. Bởi vì sau

lệnh gọi hàm thì giá trị cũ của con trỏ chồng sẽ được đưa vào chồng nhằm mục đích lưu trữ, và giá trị mới sẽ được cộng thêm 4. Đoạn mã hiện thực được thể hiện trong hình E.25:

```
if(param11 && param12 && param13 && param14 && param15 && param16 &&
addparam1){…
 if(addparam1 &&(((std::string)lhs->prints()).find("r14")!=std::string::npos)){
…
  if(!lhs->getSubExp1()->isRegOf()){
     offset= ((Const*)lhs->getSubExp1()->getSubExp2())->getInt();
     regis = ((Const*)(lhs->getSubExp1()->getSubExp1()->getSubExp1()))-
  >getInt();
    }
  Else
     regis = ((Const*)(lhs->getSubExp1()->getSubExp1()))->getInt();
     offset = offset + 4;
  …
```

Hình E.25: Đoạn mã ví dụ xử lý thuộc tính add_params

- Đối với các máy có các trường hợp alias(như trong trường hợp của máy PPC). Khi phát hiện ra trường hợp các alias được sử dụng, chúng ta cũng đưa các thanh ghi được alias vào, bởi vì 2 giá trị này là tương đương với nhau:

```
if(((std::string)lhs->prints())==(std::string)Location::memOf((new
Binary(opPlus, Location::regOf(1), new Const((int)24))))->prints()){
     temp2 = Location::regOf(3);
```

Hình E.26: Đoạn mã ví dụ xử lý thuộc tính alias

- Trong trường hợp xác định được vế trái là tham số, chúng ta sẽ tạo bản sao cho nó và đưa vào danh sách tham số được xác định bởi quy ước gọi hàm. Việc tạo bản sao để đảm bảo là các giá trị này không thay đổi. Nếu đưa trực tiếp vế trái vào danh sách tham số này, thì trong quá trình xử lý của Boomerang, các giá trị này cũng có thể thay đổi theo.

```
temp2 = lhs->clone();
if (ispara3){
   c->ABIparameters.insert(eit,temp2);
}
```

Hình E.27: Đoạn mã đưa vị trí dữ liệu vào danh sách tham số

4. Các xử lý ngoài việc nhận dạng tham số

a. Cơ chế nhận dạng kiến trúc máy

Đối với việc nhận dạng tham số dựa trên quy ước gọi hàm, thông tin về kiến trúc máy được của tập tin đầu vào cần được xác định rõ ràng để có thể áp dụng quy ước gọi hàm phù hợp.

Kỹ thuật dịch ngược

Các tập tin thực thi mà Boomerang đang quan tâm đến trong việc dịch ngược là dạng tập tin ELF(Executable and Linkable Format). May mắn là đối với dạng tập tin này, định dạng của nó đã được quy định sẵn. Trong số quy định về định dạng của tập tin ELF có quy định rằng 2 bytes thứ 18 và 19 kể từ đầu tập tin sẽ giữ thông tin về kiến trúc máy[xii]. Cụ thể theo bảng E.6 sau:

Bảng E.6: Quy ước về nhận dạng kiến trúc máy trên tập tin ELF

Giá Trị	Kiến trúc máy
0x02	SPARC
0x03	X86
0x08	MIPS
0x14	PowerPC
0x28	ARM
0x2A	Super-H
0x32	IA-64
0x3E	X86-64
0xB7	AArch64

Bảng E.6 trên là quy ước về các giá trị của 2 byte thứ 18 và 19 và kiến trúc máy của tập tin thực thi tương ứng. Ví dụ như giá trị của 2 byte này là 0x02, thì tập tin thực thi thuộc kiến trúc máy SPARC.

Trong giai đoạn tải lên(loader), Boomerang sẽ đọc 2 byte thứ 18 và 19 của tập tin và dựa vào các thông tin quy ước này để biết được kiến trúc máy phù hợp. Sau đó thời tạo ra các frontend tương ứng. Chúng ta sẽ dựa vào thông tin từ các frontend này để có cơ sở đối với việc lựa chọn quy ước gọi hàm phù hợp trong việc xác định các tham số.

b. Đưa kết quả thu được vào hệ thống Boomerang

Các kết quả mà chúng ta có được từ giai đoạn trên mới chỉ là kết quả rời rạc, nằm trong một đối tượng riêng biệt với hệ thống Boomerang(biến ABIparameters), vì vậy chúng ta phải có giai đoạn đưa các kết quả này vào trong hệ thống. Quá trình này được thực hiện như sau:

- Đầu tiên, chúng ta sẽ tạo ra một biến global trong hệ thống. Biến này sẽ có giá trị phụ thuộc vào tham số khi chạy chương trình. Ở đây, tham số "-ab" được sử dụng. Khi sử dụng tham số -ab, biến ABI_conv sẽ có giá trị là true (mặc định là false), và chương trình sẽ hiểu là chúng ta đang sử dụng quy ước gọi hàm để tìm tham số.

Kỹ thuật dịch ngược

- Tạo một danh sách tham số tạm để hàm tìm các tham số dựa trên quy ước có thể đưa kết quả vào. Danh sách này sẽ không liên quan trực tiếp đến luồng xử lý của Boomerang.
- Trong hàm Findfinalparameter của Boomerang, dựa vào biến global đã tạo từ đầu, thay vì đưa các vị trí dữ liệu được sử dụng trong hàm vào giải thuật để tìm xem vị trí dữ liệu nào là tham số, thì chúng ta sẽ so sánh vị trí dữ liệu đó với danh sách tham số được xác định bởi quy ước gọi hàm đã tạo ở trên. Như vậy, chúng ta vừa sử dụng kết quả từ quy ước gọi hàm, vừa có thể tận dụng luồng xử lý hiện tại để phân tích kiểu của biến số. Đoạn mã hiện thực được thể hiện trong hình E.28 sau:

```
if(ABI_ASS){..
   Exp* temp2 = e->clone();
   temp2= temp2->removeSubscripts(allZero);
  for(eit=ABIparameters.begin();eit != ABIparameters.end();eit++)
  {…
   if(((std::string)temp2->prints())==((std::string)temp->prints()))
   {…
      Type* ty = ((ImplicitAssign*)s)->getType();
      addParameter(e, ty);
      insertParameter(e, ty);
      break;
   }
  }
continue; }
```

Hình E.28. Đoạn mã hiện thực

IV. Kết quả thực hiện

1. Kết quả thử nghiệm

Sau khi hiện thực chức năng nhận dạng tham số dựa trên quy ước gọi hàm, chúng tôi tiến hành kiểm tra chức năng đó trên bộ mẫu thử đã kiểm tra với chức năng xác định hàm nguyên mẫu của boomerang. Và kết quả sinh ra như bảng E.7 sau:

Bảng E.7: Kết quả thử nghiệm nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm

Mục	Test cases	Số testcase đúng
Main gọi hàm	long add1(int) long add2(int, int) long add10(int, int, int, int, int, int, int, int, int, int) float add1(float)	9/9

	float add2(float, float) float add10(float, float, int, int, float, float, int, int, float, float) long sumarray(int[],int) long sumarray(int[], int, int[], int)	
Main gọi hàm A, hàm A gọi hàm B	long add1(int) -> long add2(int) long add1(int) -> long add2(int, int) float add2(float, float)-> float add1(float) long add1(int) -> void printscreen(char*) long sumarray(int[], int) -> long add2(int)	5/5
Main gọi hàm A, hàm A gọi hàm B, hàm B gọi hàm C	long add1(int)-> long add2(int, int) -> long add3(int, int, int) float add1(float)-> float add2(float, float) -> float add3(float, float, float) long sumarray1(int[], int)-> long sumarray2(int[], int)-> long sumarray3(int[], int) long add1(int)-> long add2(int, int)-> void printscreen(char*)	4/4
Main gọi hàm A đệ quy	int fibonaci(int) int factor(int) int factor2(int) long sum2recur(int, int) int plusandminus(int)	5/5
Main gọi hàm A và B đệ quy vòng	int add1(int) -> int add2(int) -> int add1(int) float add1(float) -> float add2(float) -> float add1(float) int add2(int, int)->int add1(int) -> int add2(int, int) float add2(float, float)->float add1(float) -> int add2(float, float)	5/5

	long sumarray1(int[], int)-> long sumarray2(int[], int)-> long sumarray1(int[], int)	
Viết chương trình dưới dạng assembly(Chia theo cách truyền tham số)	Sử dụng thanh ghi chuẩn, cách xa lệnh gọi Sử dụng thanh ghi chuẩn, 1 tham số xa và 1 tham số gần lệnh gọi Sử dụng 1 thanh ghi không chuẩn, cách xa lệnh gọi Sử dụng 1 thanh ghi chuẩn và 1 thanh ghi không chuẩn, cách xa lệnh gọi Chỉ truyền vào stack, cách xa lệnh gọi Truyền 1 tham số từ hàm gọi qua hàm trung gian Truyền 2 tham số từ hàm gọi qua hàm trung gian Testcases tổng hợp	2/8

Nhận xét: Như kết quả từ bảng E.7 trên, hàm nguyên mẫu sinh ra chính xác trong các trường hợp mã nguồn được sinh ra từ trình biên dịch. Như vậy, việc áp dụng nhận dạng quy ước gọi hàm dựa trên quy ước gọi hàm có thể giải quyết các khó khăn của trường hợp đệ quy mà Boomerang hiện đang mắc phải. Tuy nhiên, đối với các chương trình được viết dưới dạng assembly(hoặc mã máy) và không theo quy ước gọi hàm, thì nhận dạng hàm nguyên mẫu dựa trên quy ước không mang lại kết quả tốt(2 trường hợp đúng là 2 trường hợp đầu tiên, sử dụng thanh ghi chuẩn đúng quy ước), và điều này cũng nằm trong dự tính.

Chúng ta sẽ so sánh một trường hợp mà việc nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm mang lại kết quả tốt(test/testcases/fiborecur_sparc). Kết quả cụ thể nằm trong bảng E.8 sau:

Bảng E.8. So sánh kết quả xác định hàm nguyên mẫu của Boomerang dựa trên quy ước gọi hàm.

Mã nguồn	int Fibonacci(int n)
	{
	  if (n == 0)
	  return 0;
	  else if (n == 1)
	  return 1;
	  else
	  return (Fibonacci(n-1) + Fibonacci(n-2));

	}

Kết quả của Boomerang	__size32 Fibonacci(__size32 param1, __size32 param2) {
	__size32 g1; // r1
	__size32 i5; // r29
	__size32 o0; // r8
	__size32 o0_1; // r8{52}
	__size32 o5; // r13
	i5 = param2;
	if (param1 != 0) {
	if (param1 != 1) {
	o0 = Fibonacci(param1 - 1, param2); /* Warning: also results in o5 */
	i5 = o0;
	o0_1 = Fibonacci(param1 - 2, o5);
	g1 = o0 + o0_1;
	} else {
	g1 = 1; }
	} else {
	g1 = 0; }
	return g1; /* WARNING: Also returning: o5 := i5 */ }

Kết quả khi áp dụng quy ước gọi hàm	__size32 Fibonacci(__size32 param1) {
	__size32 g1; // r1
	__size32 o0; // r8
	__size32 o0_1; // r8{52}
	if (param1 != 0) {
	if (param1 != 1) {
	o0 = Fibonacci(param1 - 1);
	o0_1 = Fibonacci(param1 - 2);
	g1 = o0 + o0_1;
	} else {
	g1 = 1; }
	} else {
	g1 = 0; }
	return g1; }

Như kết quả có thể thấy từ bảng E.8 trên, việc sử dụng quy ước gọi hàm trong trường hợp này giúp xác định hàm nguyên mẫu tốt hơn giải thuật hiện có của Boomerang. Qua việc nhận dạng hàm nguyên mẫu tốt, chúng ta có thể loại bỏ được nhiều đoạn mã dư thừa, không cần thiết (như các biến i5, o5, các lệnh gán liên quan đến các biến này), nhờ đó đoạn code sinh ra cũng dễ đọc, dễ hiểu, chính xác hơn.

Trong quá trình kiểm tra thử các mẫu thử có sẵn của Boomerang, có trường hợp Boomerang mang lại kết quả tốt hơn là kết quả của việc áp dụng quy ước gọi hàm. Tuy nhiên, sau khi dùng trình biên dịch để dịch đoạn mã mà Boomerang sinh ra về mã máy, và áp dụng quy ước gọi hàm để nhận dạng lại, thì ra kết quả tương đương với giải thuật của Boomerang. Ở đây có thể giải thích là vì đoạn mã gốc ban đầu không tuân theo quy ước gọi hàm nên việc áp dụng quy ước gọi hàm không được như mong đợi. Kết quả kiểm tra cụ thể như bảng sau E.9: (test/sparc/paramchain).

Bảng E.9: Kết quả thử nghiệm cho tập tin test/sparc/paramchain

Kết quả của Boomerang	void passem(__size32 param1, __size32 param2, __size32 param3, __size32 *param4) {
	addem(param1, param2, param3, param4);
	return;
	}
	void addem(__size32 param1, __size32 param2, __size32 param3, __size32 *param4) {
	(__size32)param4 = param1 + param2 + param3;
	return;
	}

Kết quả khi áp dụng quy ước gọi hàm	void passem() {
	addem();
	return;
	}
	void addem() {
	__size32 o0; // r8
	__size32 o1; // r9
	__size32 o2; // r10
	(__size32)o3 = o0 + o1 + o2;
	return;

	}

Kết quả áp dụng quy ước gọi hàm đối với tập tin binary được biên dịch	void passem(__size32 param1, __size32 param2, __size32 param3, __size32 *param4) {
	  addem(param1, param2, param3, param4);
	  return;
	}
	void addem(__size32 param1, __size32 param2, __size32 param3, __size32 *param4) {
	  (__size32)param4 = param1 + param2 + param3;
	  return;
	}

2. Nhận xét chung về kết quả thử nghiệm

Việc thử nghiệm cơ chế nhận dạng hàm nguyên mẫu dựa trên quy ước gọi hàm mang lại kết quả tốt. Hàm nguyên mẫu được xác định đúng trên tất cả các mẫu thử được sinh ra từ trình biên dịch. Việc áp dụng quy ước gọi hàm không những xác định hàm nguyên mẫu đúng, mà qua đó còn giúp mã nguồn đơn giản, dễ hiểu hơn. Ngoài ra, trong quá trình thử nghiệm, còn có một số trường hợp hàm nguyên mẫu bị xác định sai. Nhưng với việc tìm hiểu về của các trường hợp sai đó, thì nguyên nhân là từ các lý do khách quan đã được lường trước (gọi hàm không theo quy ước gọi hàm). Tuy nhiên các trường hợp sai sót này có thể được xử lý tốt bằng cơ chế nhận dạng hàm nguyên mẫu hiện tại của Boomerang. Như vậy, hai cơ chế này có thể hỗ trợ, bổ sung các trường hợp còn thiếu của nhau để có được kết quả tốt nhất trong việc dịch ngược.

CHƯƠNG F

ĐÁNH GIÁ – KẾT LUẬN

Trong chương này, chi tiết về kết quả của việc giải quyết 2 bài toán được trình bày ở phần đầu tiên. Phần tiếp theo sẽ giới thiệu những hướng mà Framework Boomerang Decompiler có thể phát triển trong tương lai.

I. Đánh giá kết quả đề tài.

Kết quả của Luận Văn Tốt Nghiệp đã hoàn thành các mục tiêu đề ra:

- Với mục tiêu dịch ngược mã Assembly, hệ thống Boomerang Decompiler đã tiếp nhận và dịch ngược thành công mã Assembly của 2 kiến trúc máy Sparc và 8051. Bộ phân tích văn bản chương trình Assembly của máy 8051 được tích hợp vào Boomerang. Công cụ Boomerang Toolkit đã được phát triển và có khả sinh ra thành công các tập tin giải mã từ đặc tả có sẵn cũng như những đặc tả tự viết.
- Với mục tiêu cải tiến chức năng xác định hàm nguyên mẫu, chức năng xác định hàm nguyên mẫu dựa trên quy ước gọi hàm của kiến trúc máy Sparc, Pentium và PowerPC đã được hiện thực song song với chức năng xác định hàm nguyên mẫu của Boomerang. Công cụ tự động sinh mã cho module phân tích quy ước gọi hàm cũng được phát triển. Kết quả thử nghiệm cho thấy chức năng hoạt động tốt trên tập các mẫu thử được sinh ra từ trình biên dịch và tập các mẫu thử có sẵn của Boomerang.

II. Hướng Phát Triển Trong Tương Lai

Với những kết quả đã thu được, những cải tiến mà hệ thống Boomerang Decompiler có thể có gồm:

- Mở rộng chức năng dịch ngược mã Assembly cho nhiều kiến trúc máy khác.
- Phân tích và sửa lỗi sai của giải thuật phân tích dòng dữ liệu.
- Cải tiến chức năng nhận dạng kiểu của Boomerang.

Tài Liệu Tham Khảo

[i] Basic of Decompiler Design -Torben Ægidius Mogensen
[ii] A Decompilation Project - Mohsen Hariri
[iii] Boomerang Decompiler Homepage - http://boomerang.sourceforge.net/
[iv] Funcion Prototype - https://www.cs.auckland.ac.nz/references/unix/digital/AQTLTBTE/DOCU_055.HTM
[v] Static Single Assignment for Decompilation – Michael James Van Emmerik
[vi] Dataflow Analysis - http://www.cs.colostate.edu/~mstrout/CS553/slides/lecture03.pdf
[vii] Appication Binary Interface - https://en.wikipedia.org/wiki/Application_binary_interface
[viii] Calling Convention - https://en.wikipedia.org/wiki/Calling_convention
[ix] SPARC Calling Convention - http://ieng9.ucsd.edu/~cs30x/sparcstack.html
[x] X86 Architecture - https://msdn.microsoft.com/en-us/library/windows/hardware/ff561502(v=vs.85).aspx
[xi] 32-bit PPC Function calling convention - https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/LowLevelABI/100-32-bit_PowerPC_Function_Calling_Conventions/32bitPowerPC.html#//apple_ref/doc/uid/TP40002438-SW20
[xii] Executable and Linkable Format - http://wiki.osdev.org/ELF