ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Công Nghĩa Hiếu

XÂY DỰNG CÔNG CỤ PHÂN TÍCH MÃ NGUỒN CHO NGÔN NGỮ RUST

KHOÁ LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

HÀ NỘI - 2024

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Công Nghĩa Hiếu

XÂY DỰNG CÔNG CỤ PHÂN TÍCH MÃ NGUỒN CHO NGÔN NGỮ RUST

KHOÁ LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

Cán bộ hướng dẫn: PGS. TS. Võ Đình Hiếu

HÀ NỘI - 2024

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Cong Nghia Hieu

DEVELOPING A SOURCE CODE ANALYSIS TOOL FOR THE RUST PROGRAMMING LANGUAGE

BACHELOR'S THESIS

Major: Information technology

Supervisor: Assoc. Prof., Dr. Vo Dinh Hieu

HANOI - 2024

Lời cảm ơn

Lời đầu tiên, tôi xin bày tỏ lòng biết ơn sâu sắc tới thầy TS. Võ Đình Hiếu, người đã tận tình chỉ bảo tôi trong suốt quá trình học tập tại trường và đặc biệt là thời gian thực hiện khóa luận tốt nghiệp. Thầy đã không chỉ cung cấp những kiến thức chuyên môn quý báu mà còn động viên và hỗ trợ tôi vượt qua những khó khăn trong quá trình thực hiện khóa luận.

Tôi cũng xin cảm ơn ThS. Trần Mạnh Cường, những người đã luôn giúp đỡ tôi trong suốt thời gian thực tập. Sự hỗ trợ nhiệt tình của thầy đã giúp tôi có thêm tự tin và khả năng áp dụng kiến thức vào thực tiễn.

Ngoài ra, tôi xin gửi lời cảm ơn đến các thầy cô, anh chị, và các bạn trong phòng thí nghiệm bộ môn Công nghệ phần mềm, những người đã giúp đỡ tôi rất nhiều trong việc mở rộng kiến thức và kỹ năng thực hành.

Chúc mọi người luôn luôn vui vẻ và gặt hái được nhiều thành công trong cuộc sống.

Lời cam đoan

Tôi là Công Nghĩa Hiếu, sinh viên lớp QH-2021-I/CQ-I-IT2 khóa K66 theo học ngành Công nghệ thông tin tại trường Đại học Công Nghệ - Đại học Quốc gia Hà Nội. Tôi xin cam đoan khoá luận "Xây dựng công cụ phân tích mã nguồn cho ngôn ngữ Rust" là công trình nghiên cứu do bản thân tôi thực hiện. Các nội dung nghiên cứu, kết quả trong khoá luận là xác thực.

Các thông tin sử dụng trong khoá luận là có cơ sở và không có nội dung nào sao chép từ các tài liệu mà không ghi rõ trích dẫn tham khảo. Tôi xin chịu trách nhiệm về lời cam đoan này.

Hà Nội, ngày 16 tháng 12 năm 2024

Sinh viên

Công Nghĩa Hiếu

Tóm tắt

Tóm tắt: Rust đang trở thành một ngôn ngữ lập trình vô cùng nổi bật và được sử dụng rộng rãi trong vài năm gần đây. Nhờ vào các tính năng nổi bật về đảm bảo an toàn bộ nhớ và hiệu suất vượt trội, mã nguồn của hàng ngàn dự án trên thế giới đã được viết lại hoặc viết mới bằng Rust, từ các hệ thống nhúng cho đến các ứng dụng web hiện đại. Rust đã được chọn làm ngôn ngữ phát triển của các dự án cấp độ doanh nghiệp như Servo - web engine của trình duyệt Mozilla, 1 phần của hệ điều hành Windows 11, và nhiều dự án khác.

Những dự án này cho thấy Rust không chỉ là một xu hướng mà còn là một ngôn ngữ có tiềm năng phát triển lâu dài. Với sự phổ biến của Rust, nhu cầu về các công cụ phân tích mã nguồn, kiểm thử và bảo mật cũng ngày càng tăng. Khi các dự án lớn chuyển sang sử dụng Rust, việc đảm bảo mã nguồn an toàn và không có lỗi trở thành một thách thức. Tuy nhiên hệ sinh thái và các công cụ cho phân tích mã nguồn Rust hiện tại vẫn chưa đáp ứng đủ. Rust chỉ mới được sử dụng phổ biến trong vài năm gần đây, vì vậy việc phát triển các công cụ phân tích vẫn còn trong giai đoạn đầu. Nhằm nâng cao độ ti n cậy của mã nguồn, các phương pháp kiểm thử phần mềm tự động đang nhận được đông đảo sự quan tâm và đang được áp dụng rộng rãi trong cả cộng đồng nghiên cứu lẫn các công ty phần mềm. Một trong những phương pháp sử dụng trong kiểm thử là phân tích mã nguồn hay phân tích tĩnh. Quá trình này là việc phân tích, đánh giá chất lượng mã nguồn và tìm ra các lỗi lập trình, lỗ hổng bảo mật mà không cần phải thực thi chương trình.

Khóa luận này trình bày phương pháp xây dựng công cụ phân tích mã nguồn dành cho ngôn ngữ Rust. Đầu vào của công cụ là các tệp mã nguồn Rust và đầu ra là đồ thị thuộc tính mã nguồn (CPG). Công cụ sử dụng đồ thị thuộc tính mã nguồn tuân theo chuẩn đặc tả và nền tảng có sẵn của Joern. Đầu ra có thể được lưu trữ trong cơ sở dữ liệu đồ thị, trực quan hóa, truy vấn, phân tích thủ công hoặc tự động nhằm phục vụ cho mục đích phân tích mã nguồn khác nhau. Ngoài ra CPG này có thể được áp dụng cho các kĩ thuật học máy để phát hiện lỗ hổng bảo mật như graph neural networks (GNN).

Từ khóa: Phân tích mã nguồn, phân tích tĩnh, ngôn ngữ lập trình Rust

Abstract

Abstract: Rust has become an incredibly prominent and widely-used programming language in recent years. Thanks to its outstanding features of ensuring memory safety and superior performance, the source code of thousands of projects worldwide has been rewritten or newly developed using Rust, from embedded systems to modern web applications. Rust has been chosen as the development language for enterprise-level projects such as Servo, the web engine of the Mozilla browser, part of the Windows 11 operating system, and many other projects.

These projects demonstrate that Rust is not just a trend but a language with long-term potential for development. As Rust becomes more popular, the demand for source code analysis, testing, and security tools is also increasing. When major projects switch to using Rust, ensuring that the source code is safe and error-free becomes a challenge. However, the current ecosystem and tools for Rust source code analysis are still insufficient. Since Rust has only become widely used in recent years, the development of analysis tools is still in its early stages. To enhance the reliability of source code, automated software testing methods are receiving widespread attention and are being widely applied in both the research community and software companies. One of the methods used in testing is source code analysis or static analysis. This process involves analyzing, evaluating the quality of the source code, and identifying programming errors and security vulnerabilities without needing to execute the program.

This thesis presents a method for building a source code analysis tool for the Rust language. The tool's input is Rust source code files, and the output is a Code Property Graph (CPG). The tool utilizes the Code Property Graph following the specification and existing platform of Joern. The output can be stored in a graph database, visualized, queried, manually or automatically analyzed to serve various source code analysis purposes. Additionally, this CPG can be applied to machine learning techniques to detect security vulnerabilities such as graph neural networks (GNN).

Keywords: Source code analysis, static analysis, Rust programming language

Mục lục

Danh sách hình vẽ

Danh sách bảng

Danh sách đoạn mã

Danh mục các từ viết tắt

STT	Từ viết tắt	Cụm từ đầy đủ	Cụm từ tiếng Việt
1	AST	Abstract Syntax Tree	Cây cú pháp trừu tượng
2	CFG	Control Flow Graph	Đồ thị dòng điều khiển
3	PDG	Program Dependence Graph	Đồ thị phụ thuộc chương trình
4	CDG	Control Dependence Graph	Đồ thị phụ thuộc điều khiển
5	CPG	Code Property Graph	Đồ thị thuộc tính mã nguồn
6	IDE	Integrated Development Environment	Môi trường phát triển tích hợp

Chương 1

Đặt vấn đề

Hiện nay, cùng với sự phát triển của công nghệ và độ phủ sóng của internet, ngành công nghiệp phần mềm đang trải qua giai đoạn phát triển mạnh mẽ và đa dạng hóa không ngừng, đi kèm với đó là hàng tỷ người sử dụng. Các hệ thống phần mềm phải luôn đảm bảo tính sẵn sàng và ổn định bởi vì chỉ một sai xót xảy ra cũng sẽ gây ra những hậu quả khôn lường. Trong quá trình xây dựng phần mềm, các đội ngũ phát triển phần mềm thường dùng là sử dụng các công cụ phân tích mã nguồn cho việc phát triển, kiểm thử và đảm bảo chất lượng phần mềm. Các công cụ phân tích mã nguồn sẽ giúp phát hiện các lỗi lập trình, vấn đề về bảo mật hay việc sử dụng tài nguyên kém hiệu quả mà không cần phải thực thi chương trình. Đồng thời, các công cụ này sẽ giúp cải thiện chất lượng mã nguồn, đảm bảo lập trình viên sẽ tuân thủ các quy tắc và tiêu chuẩn khi viết mã nguồn. Điều này sẽ giúp giảm thiểu thời gian và công sức cho quá trình kiểm thử và đánh giá mã nguồn.

Hiện tại, trên thị trường đã có nhiều công cụ cung cấp khả năng phân tích mã nguồn như SonarQube, ReSharper hay CodeClimate. Những công cụ này cung cấp khả năng phân tích mã nguồn cho nhiều ngôn ngữ lập trình khác nhau như C/C++, Java, Javascript, C... Bên cạnh đó, mỗi ngôn ngữ lập trình lại có thêm nhiều công cụ phân tích mã nguồn khác nhau, nếu xét riêng cho ngôn ngữ lập trình Rust, ta có một số công cụ tiêu biểu như gosec1 , staticcheck2 hay govulncheck3 . Đây là các công cụ cung cấp khả năng phân tích cú pháp và tìm lỗ hổng trong mã nguồn. Điểm chung của các công cụ này là đều sử dụng dạng cây cú pháp trừu tượng được định nghĩa sẵn của Rust nên chúng chỉ dừng lại ở tìm các lỗi dựa trên cú pháp mã nguồn mà không khai thác sâu hơn về luồng điều khiển hay luồng dữ liệu của mã nguồn. Do vậy, những công cụ này thường cung cấp nhiều cảnh báo giả và bỏ qua nhưng lỗi nghiêm trọng khi phân tích mã nguồn.

Khác với các công cụ vừa nêu, trên thị trường hiện nay còn xuất hiện một công cụ là Joern. Joern là một nền tảng mã nguồn mở cung cấp khả năng phân tích mã nguồn, mã bytecode và mã nhị phân. Một công cụ phân tích mã nguồn sẽ gồm ba thành phần chính: phân tích cú pháp mã nguồn (parser), biểu diễn cấu trúc của

mã nguồn và phân tích cấu trúc biểu diễn đó [3]. Joern đóng vai trò là một công cụ giúp xử lý hai bước đầu của quá trình phân tích mã nguồn. Thay vì chỉ sử dụng cây cú pháp trừu tượng, Joern biểu diễn mã nguồn dưới dạng đồ thị thuộc tính mã nguồn và cung cấp chức năng truy vấn khai thác đồ thị này bằng các câu lệnh truy vấn viết bằng ngôn ngữ Scala. Joern được xây dựng với mục tiêu cung cấp chức năng tìm kiếm lỗ hổng trong mã nguồn và cung cấp các công cụ để xây dựng các trình phân tích tĩnh sử dụng dữ liệu phân tích mà Joern cung cấp. Điểm đặc biệt của Joern là nó cung cấp dạng đồ thị biểu diễn duy nhất cho tất cả các ngôn ngữ mà nó hỗ trợ (bao gồm C, C++, Java, Javascript, Python...), điều này đem đến khả năng phân tích đa ngôn ngữ, giúp giảm thiểu việc phải xử lý riêng cho từng ngôn ngữ khi chúng ta tìm kiếm lỗi trong mã nguồn hay khi xây dựng các trình phân tích tĩnh dựa trên Joern.

Mặc dù Joern đã cung cấp khả năng phân tích mã nguồn đối với ngôn ngữ Rust tuy nhiên vẫn còn rất hạn chế do chưa xử lý hết toàn bộ các thành phần mã nguồn có trong Rust và thiếu các thông tin về kiểu dữ liệu. Vì vậy khóa luận này xây dựng một công cụ phân tích mã nguồn dành cho ngôn ngữ lập trình Rust dựa trên mã nguồn của Joern. Công cụ này sẽ xử lý các thành mã nguồn mà Rust cung cấp một cách chi tiết và đầy đủ hơn, đồng thời cũng xử lý triệt để kiểu dữ liệu để cung cấp đồ thị mã nguồn chi tiết nhất có thể.

Khóa luận sẽ được trình bày theo cấu trúc như sau. Đầu tiên, Chương 2 thảo luận một số kiến thức cơ sở liên quan đến phân tích mã nguồn, cụ thể cho ngôn ngữ lập trình Rust. Chương 3 trình bày chi tiết quy trình xây dựng công cụ phân tích. Tiếp theo, Chương 4 sẽ trình bày kiến trúc, cài đặt công cụ và một số kết quả thực nghiệm đánh giá. Cuối cùng, tóm tắt những kết quả và kết luận sau quá trình phát triển công cụ sẽ được trình bày ở Chương 5.

Chương 2

Kiến thức cơ sở

Chương này sẽ trình bày các kiến thức về ngôn ngữ lập trình Rust, đặc trưng và các tính năng làm Rust trở nên tiêu biểu so với các ngôn ngữ khác. Chương cũng sẽ trình bày các dạng biểu diễn đồ thị của mã nguồn như cây cú pháp trừu tượng, đồ thị thuộc tính mã nguồn, ...Ngoài ra, một số công cụ được sử dụng trong phân tích mã nguồn Rust cũng sẽ được giới thiệu ở phần này.

2.1 Ngôn ngữ lập trình Rust

2.1.1 Giới thiệu tổng quan

Bổ sung thêm vào đoạn văn phía trước vừa trả lời Rust là một ngôn ngữ lập trình được phát triển bởi Mozilla và chính thức ra mắt vào năm 2010. Một trong những chức năng nổi bật của Rust là khả năng quản lý bộ nhớ an toàn mà không cần sử dụng garbage collection, giúp ngăn chặn các lỗi tràn bộ nhớ (memory leaks) và các lỗi truy cập vùng nhớ sai (segmentation faults). Rust cũng được biết đến với hiệu suất cao và khả năng biên dịch nhanh chóng. Những đặc điểm này làm cho Rust trở thành lựa chọn lý tưởng cho các ứng dụng low-level như hệ điều hành, trình biên dịch, và các hệ thống nhúng (embedded systems). Bên cạnh đó, Rust còn được sử dụng trong lĩnh vực Internet of Things (IoT), nơi mà sự an toàn và hiệu suất là yếu tố quyết định.

Ngoài ra, Rust cũng đang dần trở thành một ngôn ngữ phổ biến trong phát triển web, với các framework như Rocket và Actix giúp dễ dàng xây dựng các ứng dụng web hiệu suất cao. Rust không chỉ thu hút sự chú ý của các lập trình viên cá nhân mà còn được các cộng đồng lập trình và các doanh nghiệp lớn quan tâm. Ví dụ, Microsoft đã sử dụng Rust để phát triển một số thành phần trong Windows, trong khi Amazon Web Services (AWS) cũng đã tích hợp Rust trong các dịch vụ của mình.

Dự án Servo của Mozilla là một minh chứng rõ ràng cho sức mạnh của Rust trong

việc phát triển các ứng dụng lớn. Servo là một trình duyệt web engine được viết hoàn toàn bằng Rust, nhằm cung cấp hiệu suất và độ an toàn cao. Ngoài ra, Rust cũng được sử dụng trong các dự án nguồn mở lớn khác trên GitHub, cho thấy sự tin tưởng của cộng đồng lập trình vào khả năng của ngôn ngữ này. Với sự phổ biến ngày càng tăng, Rust đang chứng tỏ mình là một công cụ mạnh mẽ và linh hoạt, phù hợp cho nhiều lĩnh vực phát triển phần mềm hiện đại.

Việc sử dụng Rust trong các dự án lớn không chỉ giới hạn ở các công ty công nghệ mà còn mở rộng ra các lĩnh vực khác như tài chính, game, và khoa học dữ liệu. Khả năng của Rust trong việc đảm bảo an toàn và hiệu suất làm cho nó trở thành lựa chọn lý tưởng cho các hệ thống đòi hỏi độ tin cậy cao. Với sự hỗ trợ mạnh mẽ từ cộng đồng và các tổ chức lớn, Rust đang ngày càng khẳng định vị thế của mình trong làng lập trình quốc tế.

Cộng đồng Rust đã phát triển mạnh mẽ và đa dạng trong những năm gần đây. Sự gia tăng đáng kể số lượng lập trình viên sử dụng Rust trong công việc, từ 16% vào năm 2020 và 2021 lên tới 18% vào năm 2022, cho thấy sự chấp nhận và ứng dụng của Rust trong các dự án thực tế đang tăng lên. Cộng đồng Rust cũng đã mở rộng ra nhiều ngôn ngữ khác nhau, với việc tổ chức các bản dịch tự động của khảo sát hàng năm, giúp người dùng từ các quốc gia khác nhau có thể tham gia và cung cấp phản hồi.

Cộng đồng Rust không chỉ phát triển mạnh mẽ mà còn đa dạng hóa, với sự tham gia của cả lập trình viên cá nhân và các nhóm phát triển công ty. Các lập trình viên mới cũng đang gia nhập cộng đồng, và có sự tăng trưởng về số lượng lập trình viên có kinh nghiệm dài hạn. Điều này cho thấy sự phát triển tự nhiên và bền vững của cộng đồng Rust, cũng như sự quan tâm và đầu tư vào việc đào tạo và hỗ trợ cho các nhóm phát triển.

Tóm lại, cộng đồng Rust đã và đang phát triển mạnh mẽ và đa dạng trong những năm gần đây, với sự tham gia của cả lập trình viên cá nhân và các công ty lớn, tạo điều kiện cho sự phát triển và ứng dụng của ngôn ngữ này trong nhiều lĩnh vực khác nhau.

2.1.2 Đặc trưng

NOTE: Highlight đầu dòng các đặc trưng khác biệt của Rust so với các ngôn ngữ khác (lấy ví dụ với C++). Rust là low-level, không coi trọng OOP (sử dụng đan xen FP), không có garbage collection (sử dụng borrow checker). Sử dụng lifetime, macro, borrow checker đảm bảo an toàn bộ nhớ cho cả single thread và multithread programming

Rust là một ngôn ngữ lập trình nổi bật với nhiều đặc trưng tiêu biểu, trong đó nổi bật nhất là khả năng đảm bảo an toàn bộ nhớ, hỗ trợ lập trình song song và cú pháp hiện đại.

Đầu tiên, Rust được thiết kế với mục tiêu chính là đảm bảo an toàn bộ nhớ. Điều này được thực hiện thông qua mô hình sở hữu (ownership model) nghiêm ngặt, giúp quản lý bộ nhớ mà không cần sử dụng bộ thu gom rác (garbage collector). Hệ thống sở hữu của Rust bao gồm các khái niệm như mượn (borrowing) và vòng đời (lifetimes), cho phép tham chiếu đến dữ liệu mà không cần sở hữu nó, giúp ngăn chặn các cuộc đua dữ liệu (data races). Thêm vào đó, vòng đời đảm bảo rằng các tham chiếu luôn hợp lệ trong suốt thời gian chúng được sử dụng, giảm thiểu nguy cơ xảy ra các con trỏ treo (dangling pointers). Trong khi đó, các ngôn ngữ như C/C++ thiếu các tính năng đảm bảo an toàn bộ nhớ tích hợp, dẫn đến những vấn đề phổ biến như tràn bộ đệm (buffer overflows) và sử dụng sau khi giải phóng (use-after-free).

Thứ hai, Rust nổi bật với khả năng hỗ trợ lập trình song song một cách an toàn và hiệu quả. Lập trình song song trong Rust được đảm bảo an toàn nhờ mô hình sở hữu, ngăn chặn các cuộc đua dữ liệu ngay từ khi biên dịch, làm cho việc lập trình song song trở nên an toàn và đáng tin cậy hơn. Ngôn ngữ này cung cấp các luồng nhẹ (lightweight threads) được gọi là "tasks" có thể chạy song song mà không tốn nhiều tài nguyên như các luồng truyền thống, giúp tối ưu hóa việc sử dụng tài nguyên. Thư viện chuẩn của Rust còn bao gồm các tính năng như kênh truyền thông điệp (channels) để hỗ trợ việc truyền thông giữa các task, nâng cao hiệu suất tổng thể của ứng dụng. Thêm vào đó, các từ khóa async và await cho phép lập trình không đồng bộ, giúp viết mã không bị khóa khi xử lý các hoạt động I/O một cách hiệu quả, cải thiện khả năng phản hồi của ứng dụng. Mô hình song song của Rust khuyến khích các nhà phát triển suy nghĩ về quyền sở hữu và mượn dữ

liệu, dẫn đến mã nguồn bền vững và dễ bảo trì hơn, cuối cùng giảm chi phí bảo trì dài hạn.

Cuối cùng, cú pháp hiện đại của Rust mang lại nhiều lợi ích cho lập trình viên. Rust có cú pháp hiện đại và biểu cảm hơn so với C++, một ngôn ngữ có cú pháp phức tạp hơn. Rust có bộ tính năng nhỏ hơn so với C++, giúp tăng năng suất của lập trình viên. Trong khi C++ có thể mang lại sự linh hoạt nhờ vào nhiều tính năng, điều này cũng làm tăng độ phức tạp của mã nguồn. Rust, với cú pháp hiện đại và bộ tính năng tối giản, giúp lập trình viên tập trung vào việc viết mã hiệu quả và dễ hiểu hơn[1].

Tóm lại, Rust nổi bật với các đặc trưng tiêu biểu như đảm bảo an toàn bộ nhớ, hỗ trợ lập trình song song an toàn và cú pháp hiện đại, làm cho nó trở thành lựa chọn lý tưởng cho nhiều ứng dụng từ hệ thống nhúng, Internet of Things, đến phát triển web và các dự án phần mềm lớn. Rust không chỉ giúp lập trình viên viết mã an toàn và hiệu quả mà còn đóng góp vào việc xây dựng các ứng dụng hiệu suất cao và đáng tin cậy.

2.1.3 Tính hướng hàm

NOTE: Highlight đầu dòng các tính năng của Functional Programming trong Rust

Rust là một ngôn ngữ lập trình nổi bật với nhiều tính năng hỗ trợ lập trình hướng hàm (Functional Programming), giúp cải thiện đáng kể độ an toàn và hiệu quả của mã nguồn. Expresion over Statement

Một trong những nguyên tắc cốt lõi của Rust là tính bất biến (immutability). Rust khuyến khích việc sử dụng các biến không thể thay đổi sau khi đã gán giá trị. Cách tiếp cận này giúp tăng cường độ tin cậy của mã nguồn bằng cách ngăn chặn các hiệu ứng phụ ngoài ý muốn và giúp dễ dàng suy luận về hành vi của chương trình.

```
1 fn main() {
2   let x = 5;  // Immutable variable
3 }
```

Đoạn mã 2.1: Ví dụ tính immutability trong Rust

Tiếp theo, một tính năng mạnh mẽ của Rust là pattern matching. Tính năng này cho phép lập trình viên phân rã và khớp các cấu trúc dữ liệu phức tạp một cách dễ dàng. Pattern matching nâng cao khả năng biểu đạt và đọc hiểu mã nguồn, làm cho việc lập trình trở nên trực quan và rõ ràng hơn, đặc biệt có giá trị trong lập trình hướng hàm.

```
fn match_example(value: Option<i32>) {
  match value {
      Some(x) => println!("Received a value: {}", x),
      None => println!("Received None"),
  }
}
```

Đoạn mã 2.2: Ví dụ pattern matching trong Rust

Rust cũng hỗ trợ các hàm bậc cao (higher-order functions HOC), cho phép các hàm được xem như những thực thể hạng nhất. Điều này có nghĩa là các hàm có thể được truyền làm tham số cho các hàm khác và được trả về như những giá trị. Các hàm bậc cao thúc đẩy phong cách lập trình hướng hàm, cho phép tạo ra mã nguồn trừu tượng và tái sử dụng được nhiều lần.

```
fn apply_operation<F>(value: i32, operation: F) -> i32
where
    F: Fn(i32) -> i32,
{
    operation(value)
}

fn main() {
    let result = apply_operation(5, |x| x * 2);  // 10
}
```

Đoạn mã 2.3: Ví dụ Higher-order Function trong Rust

Closure, còn được biết đến là các biểu thức lambda, cho phép tạo ra các hàm ẩn danh. Closure của Rust rất linh hoạt và có thể nắm bắt các biến từ môi trường xung quanh, cung cấp một công cụ mạnh mẽ cho lập trình hướng hàm.

```rust
fn main() {
  let multiplier = |x| x * 3;
  let result = multiplier(4);  // 12
}
```

Đoạn mã 2.4: Ví dụ Closure trong Rust

Loại dữ liệu Option và Result trong Rust rất quan trọng để xử lý sự vắng mặt của giá trị và lỗi tương ứng. Những loại này phù hợp với các nguyên tắc của lập trình hướng hàm, nơi việc xử lý sự vắng mặt hoặc thất bại là một kịch bản phổ biến.

```rust
fn safe_divide(dividend: i32, divisor: i32) -> Option<i32> {
  if divisor != 0 {
      Some(dividend / divisor)
  } else {
      None
  }
}

fn main() {
  let result = safe_divide(10, 2);
  match result {
      Some(value) => println!("Result: {}", value),
      None => println!("Cannot divide by zero"),
  }
}
```

Đoạn mã 2.5: Ví dụ Monad Design pattern trong Rust

Hệ thống đặc điểm (traits) của Rust cung cấp một cơ chế mạnh mẽ để định nghĩa các hành vi chung giữa các loại, thúc đẩy tổ chức mã nguồn và tái sử dụng. Các đặc điểm trong Rust tương tự như các lớp kiểu (type classes) trong các ngôn ngữ lập trình hướng hàm, cho phép lập trình viên đóng gói chức năng và đạt được tính đa hình.

```rust
trait Printable {
  fn print(&self);
}

struct Book {
  title: String,
  author: String,
}

impl Printable for Book {
  fn print(&self) {
      println!("Book: {} by {}", self.title, self.author);
  }
}
```

Đoạn mã 2.6: Ví dụ Trait trong Rust

Hệ thống kiểu của Rust kết hợp các loại dữ liệu đại số, cho phép lập trình viên mô hình hóa dữ liệu một cách biểu đạt hơn. Các kiểu liệt kê (enum) trong Rust, đặc biệt khi kết hợp với pattern matching, giống như các loại dữ liệu đại số được tìm thấy trong các ngôn ngữ lập trình hướng hàm. Điều này cho phép biểu diễn các cấu trúc dữ liệu một cách ngắn gọn và rõ ràng.

```rust
enum Option<T> {
  Some(T),
  None,
}

fn main() {
  let some_value: Option<i32> = Option::Some(42);
  let no_value: Option<i32> = Option::None;

  match some_value { // Pattern matching
      Option::Some(value) => println!("Got a value: {}", value),
      Option::None => println!("No value"),
  }
}
```

Đoạn mã 2.7: Ví dụ ADT trong Rust

Cuối cùng, trong Rust, hầu hết mọi thứ đều là biểu thức, điều này giúp tạo ra mã nguồn sạch hơn và có thể kết hợp dễ dàng hơn. Ví dụ đơn giản về biểu thức if-else minh họa rõ ràng cho đặc điểm này của Rust.

```
fn main() {
  let x = 5;
  let value = if x < 0 {
      -1
  } else {
      1
  }
}

```

Đoạn mã 2.8: Ví dụ Expression trong Rust

Tất cả các tính năng liên quan đến lập trình hướng hàm của Rust tạo nên sự khác biệt so với các ngôn ngữ như C/C++ và Java. Trong khi C/C++ và Java thiên về lập trình hướng đối tượng và quản lý bộ nhớ thủ công, Rust nổi bật với hệ thống kiểu an toàn và mô hình sở hữu, giúp ngăn chặn các lỗi bộ nhớ và tăng cường độ an toàn của mã nguồn. Những tính năng này không chỉ giúp lập trình viên viết mã dễ bảo trì hơn mà còn cải thiện hiệu quả phân tích mã nguồn, giúp phát hiện sớm các lỗi tiềm ẩn và tối ưu hóa hiệu suất chương trình. Rust không chỉ mang lại lợi ích về mặt kỹ thuật mà còn thúc đẩy tư duy lập trình hướng hàm, làm cho mã nguồn trở nên rõ ràng và dễ hiểu hơn.

2.2 Cây cú pháp trừu tượng

Cây cú pháp trừu tượng (Abstract Syntax Tree - AST) là một cấu trúc dữ liệu quan trọng trong ngôn ngữ lập trình Rust, biểu diễn cấu trúc mã nguồn của chương trình. Khi trình biên dịch Rust phân tích mã nguồn, nó tạo ra một AST để hiểu và xử lý cú pháp của chương trình. AST trong Rust giúp trừu tượng hóa các phần chi tiết của mã nguồn và chỉ giữ lại những thông tin cần thiết để trình biên dịch hiểu cấu trúc của chương trình. Điều này bao gồm các thông tin về khai báo biến, hàm, khối lệnh, và các biểu thức khác.

AST trong Rust được sử dụng rộng rãi trong nhiều công cụ và ứng dụng khác nhau, bao gồm trình biên dịch (compiler), trình dịch ngược (decompiler), và các công cụ phân tích mã nguồn (SAST - Source Code Analysis Tool). Trong Rust, AST không chỉ giúp trình biên dịch hiểu cấu trúc của mã nguồn mà còn hỗ trợ quá trình tối ưu hóa và kiểm tra lỗi. Ví dụ, khi Rust kiểm tra các quy tắc về an toàn bộ nhớ, nó sử dụng AST để xác minh rằng các biến được sử dụng đúng cách và không gây ra các lỗi bộ nhớ như tràn bộ đệm hay truy cập ngoài giới hạn.

Đối với các nhà phát triển Rust, việc hiểu và sử dụng AST có thể giúp họ tạo ra các công cụ tùy chỉnh để phân tích và tối ưu hóa mã nguồn. AST cũng hỗ trợ việc thực hiện các thao tác như refactoring, giúp thay đổi cấu trúc mã nguồn một cách tự động mà không làm thay đổi hành vi của chương trình. Ngoài ra, AST còn đóng vai trò quan trọng trong việc tạo ra các công cụ kiểm thử tự động và phân tích bảo mật, giúp phát hiện sớm các lỗi và lỗ hổng tiềm ẩn trong mã nguồn.

Cấu trúc của AST trong Rust có thể khác nhau tùy thuộc vào cách triển khai của trình biên dịch. Mỗi nút trong AST đại diện cho một phần tử cụ thể của mã nguồn, chẳng hạn như một khai báo biến hay một biểu thức điều kiện. Các nút này có thể có các thuộc tính và con trỏ đến các nút con, tạo thành một cây cấu trúc phân cấp.

Một trong những ưu điểm lớn của AST trong Rust là khả năng tích hợp với các công cụ xử lý ngôn ngữ tự nhiên (NLP) và các công cụ mã nguồn khác. Bằng cách sử dụng AST, các nhà phát triển có thể tạo ra các công cụ phân tích mã nguồn mạnh mẽ và hiệu quả, giúp cải thiện chất lượng và an toàn của mã nguồn Rust. Nhờ vào tính linh hoạt và khả năng mở rộng của AST, Rust đã trở thành một ngôn ngữ lý tưởng cho việc phát triển các ứng dụng yêu cầu độ tin cậy và hiệu suất cao.

2.3 Đồ thị dòng điều khiển

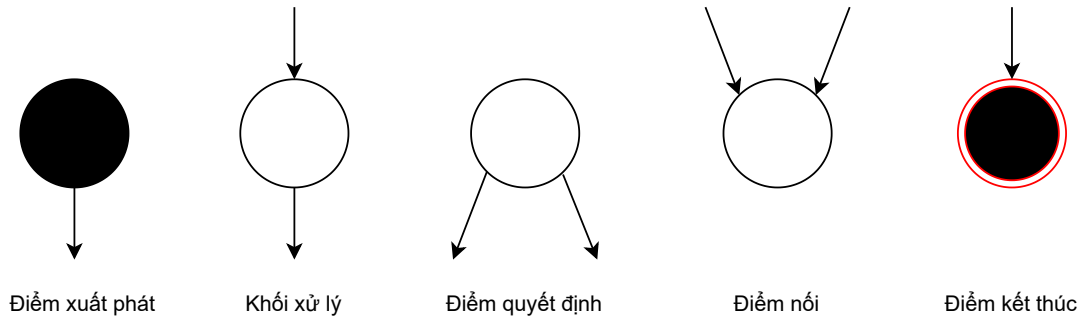
Như đã đề cập ở trên, phương pháp đề xuất trong khóa luận này áp dụng hướng tiếp cận kiểm thử tượng trưng động - kỹ thuật kiểm thử dựa trên dòng điều khiển. Tổng quan của phương pháp này là quá trình phân tích mã nguồn, xây dựng đồ thị dòng điều khiển, sau đó thực hiện kỹ thuật thực thi giá trị tượng trưng trên mỗi đường đi của đồ thị. Việc sinh dữ liệu kiểm thử dựa trên phân tích mã nguồn

sẽ gặp rất nhiều khó khăn nếu chỉ thao tác với mã nguồn ở dạng văn bản đơn thuần. Vì vậy, chúng ta cần có một cấu trúc dữ liệu khác không chỉ có thể mô tả mã nguồn mà còn tốn ít chi phí để phân tích. Đồ thị dòng điều khiển là một cấu trúc dữ liệu hỗ trợ giải quyết vấn đề này.

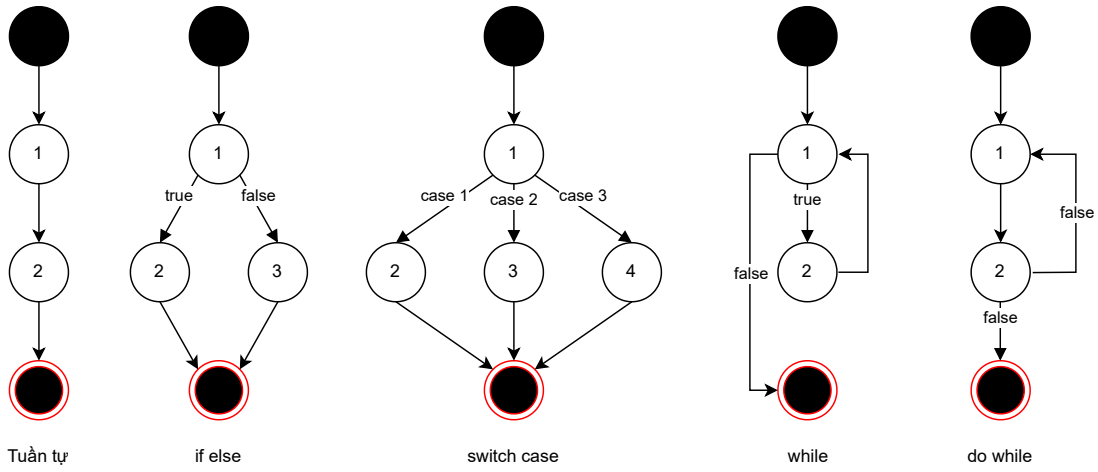
Đồ thị dòng điều khiển (Control Flow Graph - CFG) mô tả kịch bản thực thi của chương trình một cách trực quan. Đồ thị này được xây dựng từ mã nguồn của chương trình/đơn vị chương trình. Cụ thể, đồ thị dòng điều khiển được định nghĩa trong Định nghĩa 2.1.

Định nghĩa 2.1 [1]: "Đồ thị dòng điều khiển là một đồ thị có hướng gồm các điểm tương ứng với các câu lệnh/nhóm câu lệnh và các cạnh là các dòng điều khiển giữa các câu lệnh/nhóm câu lệnh. Nếu i và j là các điểm của đồ thị dòng điều khiển thì tồn tại một cạnh từ i đến j nếu lệnh tương ứng với j có thể được thực hiện ngay sau lệnh tương ứng với i."

Tất cả các đồ thị dòng điều khiển đều có điểm xuất phát và điểm kết thúc đại diện cho trạng thái bắt đầu và trạng thái kết thúc của chương trình. Các cạnh là các mũi tên có hướng thể hiện thứ tự thực hiện của câu lệnh/nhóm câu lệnh. Về cơ bản, CFG bao gồm các thành phần chính là điểm xuất phát, khối xử lý, điểm quyết định, điểm nối và điểm kết thúc được mô tả trong Hình 2.1.



Hình 2.1: Các thành phần cơ bản trong đồ thị dòng điều khiển.



Hình 2.2: Các cấu trúc điều khiển phổ biến trong các ngôn ngữ lập trình.

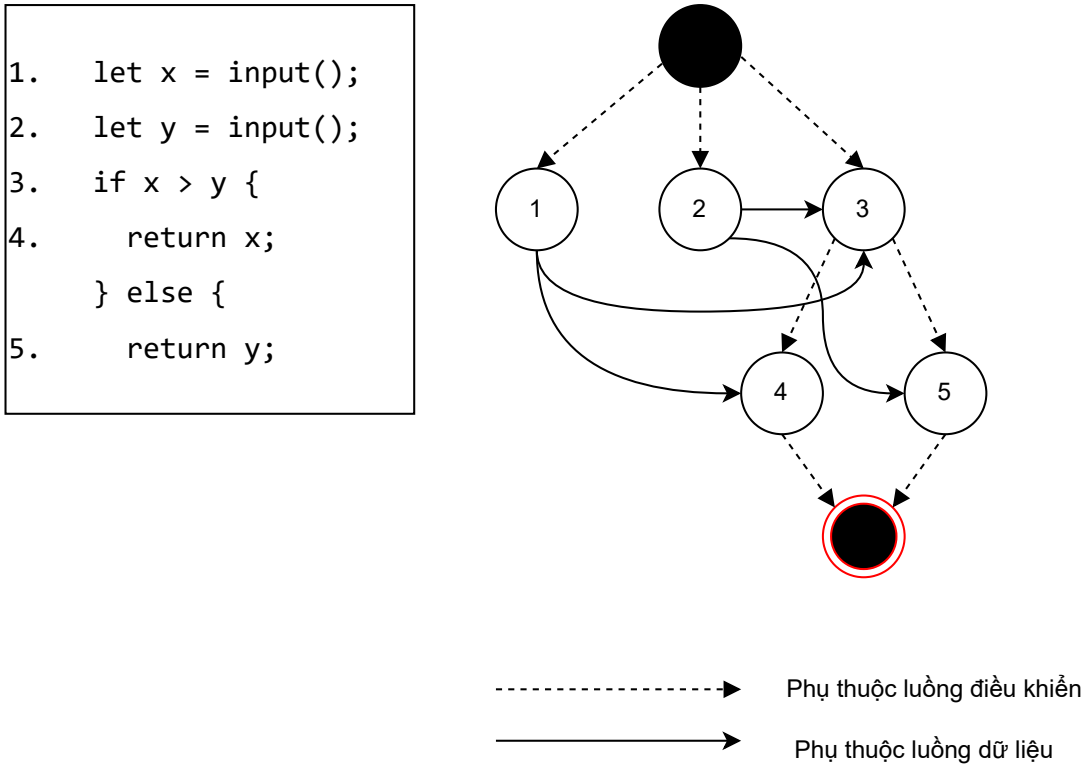
- **Điểm xuất phát:** Đánh dấu thời điểm xuất phát của chương trình, được thể hiện bằng hình tròn đặc
- **Khối xử lý:** Đại diện cho các câu lệnh gán, khai báo và khởi tạo, được thể hiện bằng hình tròn rỗng;
- **Điểm quyết định:** Đại diện cho câu lệnh điều khiển trong khối lệnh điều khiển rẽ nhánh, được thể hiện bằng hình tròn rỗng và có nhiều cạnh đi ra từ điểm;
- **Điểm nối:** Đại diện cho câu lệnh được thực hiện ngay sau các lệnh rẽ nhánh, có nhiều hơn một điểm trỏ đến, được thể hiện bằng hình tròn rỗng;
- **Điểm kết thúc:** Đánh dấu thời điểm kết thúc của hàm, được thể hiện bằng hình tròn đặc có viền.

Hình 2.2 mô tả các cấu trúc điều khiển chính có trong mã nguồn C/C++ được mô phỏng dưới dạng các điểm của CFG, bao gồm có cấu trúc điều khiển tuần tự, rẽ nhánh if-else, switch-case, vòng lặp while c do ... và vòng lặp do ... while c.

2.4 Đồ thị phụ thuộc chương trình

Đồ thị phụ thuộc chương trình (Program Dependence Graph) biểu diễn chương trình dưới dạng một đồ thị mà trong đó các nút là các câu lệnh và biểu thức

(expression) hoặc các toán tử và toán hạng. Các cạnh của đồ thị biểu diễn sự phụ thuộc dữ liệu (data dependence) và điều kiện điều khiển (control condition) mà việc thực thi nút đó phụ thuộc vào hay có thể hiểu đơn giản hơn là phụ thuộc điều khiển (control dependence) [2].

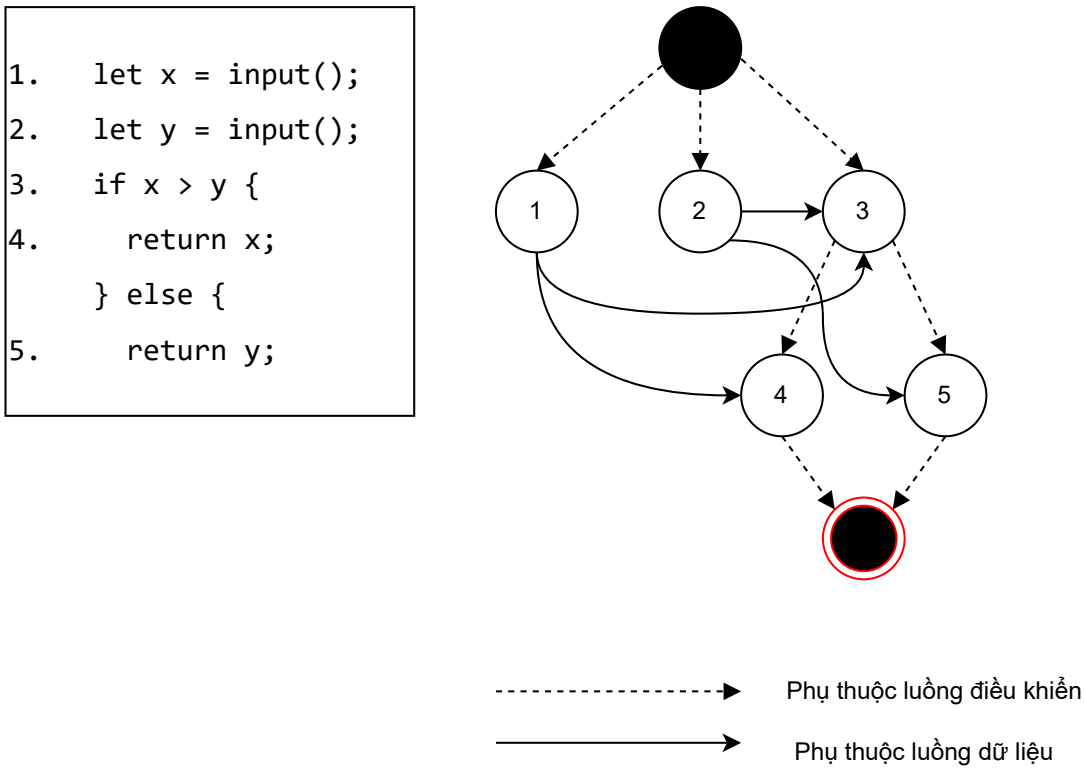


Hình 2.3: Ví dụ về đồ thị phụ thuộc chương trình

Hình 2.4 biểu diễn ví dụ về một đồ thị phụ thuộc chương trình với nút ENTRY là điểm bắt đầu của chương trình, các cạnh nét đứt biểu diễn phụ thuộc điều khiển và các cạnh nét liền biểu diễn phụ thuộc dữ liệu.

2.5 Đồ thị thuộc tính mã nguồn

NOTE: Tự chạy 1 đoạn code sinh ra CPG rồi paste ảnh và đoạn code vào đây (thay cho ảnh và đoạn code bên dưới)



Hình 2.4: Đồ thị thuộc tính mã nguồn biểu diễn đoạn mã nguồn 2.9

```
1 fn main() {
2   let x = input();
3   let y = input();
4   if x > y {
5     return x;
6   } else {
7     return y;
8   }
9 }
```

Đoạn mã 2.9: Mã nguồn đầy đủ cho đồ thị thuộc tính mã nguồn 2.4

Đồ thị thuộc tính mã nguồn (Code Property Graph) là dạng đồ thị biểu diễn mã nguồn của chương trình. Nó được hợp thành cây cú pháp trừu tượng, đồ thị dòng điều khiển và đồ thị phụ thuộc chương trình. Đồ thị chứa các thông tin về cấu trúc cú pháp, luồng điều khiển và phụ thuộc dữ liệu trong chương trình. CPG được sử

dụng để tìm kiếm lỗ hổng trong mã nguồn, phát hiện sao chép mã nguồn, đo lường khả năng kiểm thử mã nguồn và sinh mã khai thác. [7, 8, 11]

Các nút đại diện cho các thành phần như hàm, biến, lớp, gói và các cạnh đại diện cho mối quan hệ giữa chúng như lời gọi hàm, sự gán giá trị, quan hệ cha con hay tham chiếu.

2.6 Joern

2.6.1 Công cụ Joern

NOTE: Nhấn mạnh Joern hiện tại chưa hỗ trợ Rust, chủ yếu hỗ trợ cho C/C++, Java là cực mạnh

Joern là một nền tảng mạnh mẽ dành cho việc phân tích mã nguồn, bytecode và mã nhị phân. Công cụ này tạo ra các đồ thị thuộc tính mã nguồn (code property graphs), một cách biểu diễn đồ thị của mã giúp cho việc phân tích mã nguồn đa ngôn ngữ trở nên dễ dàng hơn. Các đồ thị thuộc tính mã nguồn được lưu trữ trong một cơ sở dữ liệu đồ thị tùy chỉnh, cho phép khai thác mã nguồn thông qua các truy vấn tìm kiếm được xây dựng trong một ngôn ngữ truy vấn đặc thù dựa trên Scala. Joern được phát triển với mục tiêu cung cấp một công cụ hữu ích cho việc khám phá lỗ hổng bảo mật và nghiên cứu phân tích chương trình tĩnh.

Joern hỗ trợ các nhà phát triển và nhà nghiên cứu trong việc tìm kiếm và xác định các điểm yếu tiềm ẩn trong mã nguồn, giúp nâng cao chất lượng và bảo mật của phần mềm. Bên cạnh đó, Joern còn có khả năng phân tích mã nguồn đa ngôn ngữ, giúp các nhóm phát triển có thể làm việc với nhiều ngôn ngữ lập trình khác nhau mà không gặp trở ngại về công cụ. Với khả năng truy vấn mạnh mẽ và linh hoạt, Joern đã trở thành một công cụ quan trọng trong việc phân tích mã nguồn và phát hiện các lỗ hổng bảo mật. Bạn có thể tìm hiểu thêm về Joern tại địa chỉ https://joern.io/.

Joern hỗ trợ nhiều ngôn ngữ lập trình khác nhau. Các ngôn ngữ được hỗ trợ bao gồm C/C++, Java, JavaScript, Python, x86/x64, JVM Bytecode, Kotlin, PHP, Rust, Swift, Ruby và C#. Điều này cho thấy Joern có khả năng phân tích mã nguồn đa ngôn ngữ, giúp các nhà phát triển và nhà nghiên cứu có thể làm việc với nhiều ngôn ngữ lập trình khác nhau mà không gặp trở ngại về công cụ. Tuy nhiên,

Joern chưa hỗ trợ cho ngôn ngữ Rust.

2.6.2 Đặc tả đồ thị thuộc tính mã nguồn của Joern

NOTE: Thêm ý đặc tả CPG Joern ban đầu được viết cho ngôn ngữ C, sau đó mở rộng cho các ngôn ngữ khác như Java, PHP, ... Tuy nhiên đa phần các ngôn ngữ này đều là ngôn ngữ có cú pháp C-like và chủ yếu được code dưới OOP. Do vậy đối với ngôn ngữ có sự đan xen của các tính năng Functional Programming, thì bản đặc tả này tỏ ra có sự hạn chế

Đồ thị thuộc tính mã nguồn (Code Property Graph - CPG) là một cấu trúc dữ liệu được thiết kế để khai thác các cơ sở mã nguồn lớn nhằm tìm kiếm các mẫu lập trình. Những mẫu này được hình thành trong một ngôn ngữ đặc thù (DSL) dựa trên Scala. CPG đóng vai trò như một biểu diễn chương trình trung gian duy nhất cho tất cả các ngôn ngữ được Joern và phiên bản thương mại của nó là Ocular hỗ trợ.

Đồ thị thuộc tính là một trừu tượng chung được hỗ trợ bởi nhiều cơ sở dữ liệu đồ thị đương đại như Neo4j, OrientDB và JanusGraph. Trên thực tế, các phiên bản cũ của Joern đã sử dụng các cơ sở dữ liệu đồ thị mục đích chung làm nơi lưu trữ và ngôn ngữ truy vấn đồ thị Gremlin. Tuy nhiên, khi những hạn chế của cách tiếp cận này trở nên rõ ràng theo thời gian, chúng tôi đã thay thế cả hệ thống lưu trữ và ngôn ngữ truy vấn bằng cơ sở dữ liệu đồ thị OverflowDB của riêng mình.

Qwiet AI (trước đây là ShiftLeft) đã mã nguồn mở việc triển khai đồ thị thuộc tính mã nguồn và đặc tả của nó.

Các thành phần cấu thành đồ thị thuộc tính mã nguồn bao gồm:

- **Các nút và loại của chúng:** Các nút đại diện cho các thành phần của chương trình. Điều này bao gồm các cấu trúc ngôn ngữ cấp thấp như phương thức, biến, và cấu trúc điều khiển, cũng như các cấu trúc cấp cao hơn như điểm cuối HTTP hoặc các kết quả phân tích. Mỗi nút có một loại, loại này chỉ ra loại thành phần chương trình mà nút đó đại diện, ví dụ, một nút với loại METHOD đại diện cho một phương thức, trong khi một nút với loại LOCAL đại diện cho khai báo của một biến cục bộ.
- **Cạnh có nhãn:** Quan hệ giữa các thành phần chương trình được biểu diễn

thông qua các cạnh giữa các nút tương ứng của chúng. Ví dụ, để biểu thị rằng một phương thức chứa một biến cục bộ, chúng ta có thể tạo một cạnh với nhãn CONTAINS từ nút của phương thức đến nút của biến cục bộ. Bằng cách sử dụng các cạnh có nhãn, chúng ta có thể biểu diễn nhiều loại quan hệ khác nhau trong cùng một đồ thị. Hơn nữa, các cạnh có hướng để biểu thị, ví dụ, rằng phương thức chứa biến cục bộ nhưng không phải ngược lại. Nhiều cạnh có thể tồn tại giữa cùng hai nút.

- **Cặp khóa-giá trị:** Các nút mang các cặp khóa-giá trị (thuộc tính), trong đó các khóa hợp lệ phụ thuộc vào loại nút. Ví dụ, một phương thức có ít nhất tên và chữ ký, trong khi một khai báo biến cục bộ có ít nhất tên và loại của biến được khai báo.

Tóm lại, đồ thị thuộc tính mã nguồn là các đồ thị có hướng, được gán nhãn cạnh, và chứa các thuộc tính, và chúng tôi khẳng định rằng mỗi nút mang ít nhất một thuộc tính chỉ ra loại của nó. Điều này giúp cho việc phân tích mã nguồn trở nên dễ dàng và hiệu quả hơn, đồng thời mở ra nhiều khả năng cho việc tìm kiếm và phát hiện các mẫu lập trình trong các cơ sở mã nguồn lớn.

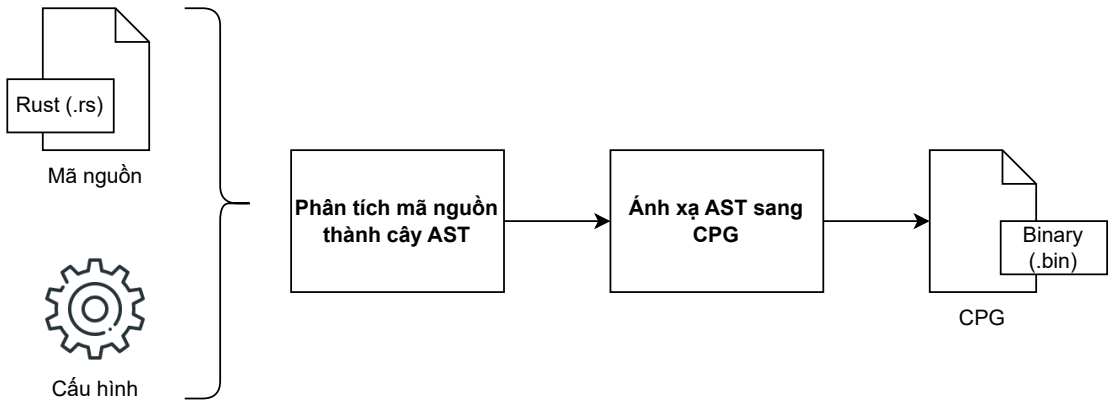
Chương 3

Phân tích mã nguồn Rust

Chương này sẽ trình bày quy trình phân tích mã nguồn cho ngôn ngữ lập trình Rust từ việc xây dựng cây cú pháp trừu tượng, xử lý kiểu dữ liệu đến ánh xạ cây cú pháp trừu tượng sang đồ thị thuộc tính mã nguồn. Đồng thời, chương này cũng cung cấp cái nhìn chi tiết về hai dạng biểu diễn mã nguồn được sử dụng là cây cú pháp trừu tượng và đồ thị thuộc tính mã nguồn.

3.1 Quy trình tổng quan

Mục tiêu của công cụ là phân tích mã nguồn Rust và xây dựng đồ thị thuộc tính mã nguồn (CPG) biểu diễn mã nguồn đó. Hình 3.1 thể hiện các bước xây dựng cây CPG từ các tệp mã nguồn Rust.



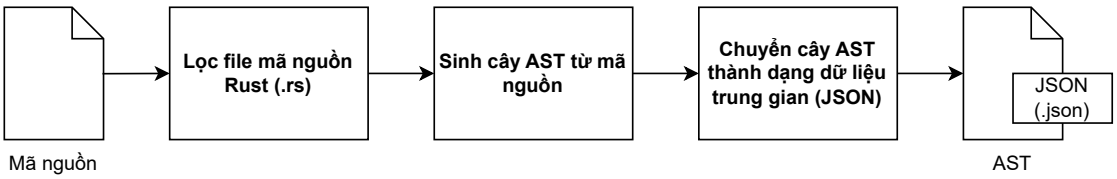
Hình 3.1: Quy trình phân tích mã nguồn Rust

Mã nguồn Rust sẽ được trình phân tích go/parser phân tích và sinh ra cây AST tương ứng với từng tệp mã nguồn. Sau đó, công cụ go/package sẽ phân tích các tệp mã nguồn và sinh ra các thông tin về kiểu dữ liệu và ta sẽ thực hiện bổ sung các thông tin về kiểu dữ liệu và một số thông tin khác như chữ ký (signature), tên đầy đủ (fullyQualifiedName) và kiểu dữ liệu vào các nút tương ứng trong cây cú pháp trừu tượng. Ở bước thứ ba, cây cú pháp trừu tượng sẽ được ánh xạ sang đồ

thị thuộc tính mã nguồn. Cuối dùng, đồ thị CPG sẽ được lưu dưới dạng cơ sở dữ liệu đồ thị phục vụ mục đích của người dùng.

3.2 Xây dựng cây cú pháp trừu tượng

NOTE: Nhắc đến việc sử dụng thư viện Syn, Rust chưa có đặc tả chính thức mà có reference. Thư viện syn có ngữ pháp tuân theo reference này. Tất cả các loại AST sẽ tuân theo định nghĩa của thư viện Syn.



Hình 3.2: Quy trình xây dựng cây cú pháp trừu tượng

Quy trình xây dựng cây cú pháp trừu tượng bao gồm ba bước được thể hiện trong Hình 3.2, chi tiết các bước được tiến hành như sau:

1. Từ thư mục của dự án, lọc lấy các tệp mã nguồn Rust (các tệp có đuôi là .rs)
2. Với mỗi tệp mã nguồn, sử dụng thư viện syn để phân tích thành cây cú pháp trừu tượng (AST) ứng với tệp đó.
3. Chuyển đổi cây cú phá p trừu tượng thu được sang dạng cây tự định nghĩa. Về bản chất dạng cây tự định nghĩa này hoàn toàn giống với dạng cây gốc, tuy nhiên cây chỉ lưu các thông tin cần thiết cũng như bổ sung thêm một số thông tin để xác định nút cha, chữ ký nút và một số thông tin khác để phục vụ pha phân tích tiếp theo.

Trong quá trình xây dựng cây cú pháp trừu tượng, các thành phần mã nguồn trong Rust sẽ được phân tích thành dạng cây tương ứng. Dưới đây là một số hình ảnh ví dụ về cây cú pháp trừu tượng ứng với các thành phần mã nguồn này.

Hình 3.3 biểu diễn một hàm được phân tích thành một nút định nghĩa hàm (FunctionDeclaration), nút này có các thuộc tính là chữ ký (signature), kiểu trả về (returnType) và tên đầy đủ (fullName). Nút có các nút con gồm nút định danh

(Identifier) biểu diễn định danh của hàm, nút kiểu hàm (FunctionType) biểu diễn kiểu dữ liệu của hàm và nút thân hàm (BlockStatement).

Ví dụ về biểu diễn một kiểu cấu trúc được thể hiện trong Hình 3.4. Nút TypeSpecification đại diện cho định nghĩa kiểu. Nút Identifier là nút định danh biểu diễn tên kiểu. Nút StructType biểu diễn kiểu cấu trúc. Nút FieldList có các nút con là Field biểu diễn các thuộc tính của kiểu này.

STT	**Tên nút**	**Mô tả**
1	Abi	The binary interface of a function: ‘extern "C"‘.
2	AngleBracketed GenericArguments	Angle bracketed arguments of a path segment: the ‘<K, V>‘ in HashMap<K, V>.
3	Arm	One arm of a ‘match‘ expression: ‘0..=10 => return true; ‘.
4	AssocConst	An equality constraint on an associated constant: ‘the PANIC = false in Trait<PANIC = false>‘.
5	AssocType	A binding (equality constraint) on an associated type: ‘the Item = u8 in Iterator<Item = u8>‘.
6	Attribute	An attribute, like ‘#[repr(transparent)]‘.
7	BareFnArg	An argument in a function type: ‘the usize in fn(usize) -> bool‘.
8	BareVariadic	The variadic argument of a function pointer like ‘fn(usize, ...)‘.
9	Block	A braced block containing Rust statements.
10	BoundLifetimes	A set of bound lifetimes: ‘for<’a, ’b, ’c>‘.
11	ConstParam	A const generic parameter: ‘const LENGTH: usize‘.
12	Constraint	An associated type bound: ‘Iterator<Item: Display>‘.
13	DataEnum	An enum input to a ‘proc_macro_derive‘ macro.
14	DataStruct	A struct input to a ‘proc_macro_derive‘ macro.
15	DataUnion	An untagged union input to a ‘proc_macro_derive‘ macro.
16	DeriveInput	Data structure sent to a ‘proc_macro_derive‘ macro.
17	Error	Error returned when a Syn parser cannot parse the input tokens.
18	ExprArray	A slice literal expression: ‘[a, b, c, d]‘.
19	ExprAssign	An assignment expression: ‘a = compute()‘.
20	ExprAsync	An async block: ‘async ... ‘.
21	ExprAwait	An await expression: ‘fut.await‘.
22	ExprBinary	A binary operation: ‘a + b, a += b‘.
23	ExprBlock	A blocked scope: ‘ ... ‘.
24	ExprBreak	A ‘break‘, with an optional label to break and an optional expression.

25	ExprCall	A function call expression: ‘invoke(a, b)‘.
26	ExprCast	A cast expression: ‘foo as f64‘.
27	ExprClosure	A closure expression: ‘[a, b] + a + b‘.
28	ExprConst	A const block: ‘const ... ‘.
29	ExprContinue	A ‘continue‘, with an optional label.
30	ExprField	Access of a named struct field ‘obj.k‘ or unnamed tuple struct field ‘obj.0‘.
31	ExprForLoop	A for loop: ‘for pat in expr ... ‘.
32	ExprGroup	An expression contained within invisible delimiters.
33	ExprIf	An if expression with an optional else block: if expr ... else ... .
34	ExprIndex	A square bracketed indexing expression: vector[2].
35	ExprInfer	The inferred value of a const generic argument, denoted _.
36	ExprLet	A let guard: let Some(x) = opt.
37	ExprLit	A literal in place of an expression: 1, "foo".
38	ExprLoop	Conditionless loop: loop ... .
39	ExprMacro	A macro invocation expression: format!("", q).
40	ExprMatch	A match expression: match n Some(n) => , None => .
41	ExprMethodCall	A method call expression: x.foo::<T>(a, b).
42	ExprParen	A parenthesized expression: (a + b).
43	ExprPath	A path like std::mem::replace possibly containing generic parameters and a qualified self-type.
44	ExprRange	A range expression: 1..2, 1.., ..2, 1..=2, ..=2.
45	ExprRawAddr	Address-of operation: &raw const place or &raw mut place.
46	ExprReference	A referencing operation: &a or &mut a.
47	ExprRepeat	An array literal constructed from one repeated element: [0u8; N].
48	ExprReturn	A return, with an optional value to be returned.
49	ExprStruct	A struct literal expression: Point x: 1, y: 1 .
50	ExprTry	A try-expression: expr?.
51	ExprTryBlock	A try block: try ... .
52	ExprTuple	A tuple expression: (a, b, c, d).
53	ExprUnary	A unary operation: !x, x.
54	ExprUnsafe	An unsafe block: unsafe ... .
55	ExprWhile	A while loop: while expr ... .
56	ExprYield	A yield expression: yield expr.
57	Field	A field of a struct or enum variant.
58	FieldPat	A single field in a struct pattern.
59	FieldValue	A field-value pair in a struct literal.

60	FieldsNamed	Named fields of a struct or struct variant such as Point x: f64, y: f64 .
61	FieldsUnnamed	Unnamed fields of a tuple struct or tuple variant such as Some(T).
62	File	A complete file of Rust source code.
63	ForeignItemFn	A foreign function in an extern block.
64	ForeignItemMacro	A macro invocation within an extern block.
65	ForeignItemStatic	A foreign static item in an extern block: static ext: u8.
66	ForeignItemType	A foreign type in an extern block: type void.
67	Generics	Lifetimes and type parameters attached to a declaration of a function, enum, trait, etc.
68	Ident	A word of Rust code, which may be a keyword or legal variable name.
69	ImplGenerics	Returned by Generics::split_for_impl.
70	ImplItemConst	An associated constant within an impl block.
71	ImplItemFn	An associated function within an impl block.
72	ImplItemMacro	A macro invocation within an impl block.
73	ImplItemType	An associated type within an impl block.
74	Index	The index of an unnamed tuple struct field.
75	ItemConst	A constant item: ‘const MAX: u16 = 65535‘.
76	ItemEnum	An enum definition: ‘enum Foo<A, B> A(A), B(B) ‘.
77	ItemExternCrate	An extern crate item: ‘extern crate serde‘.
78	ItemFn	A free-standing function: ‘fn process(n: usize) -> Result<(), ... ‘.
79	ItemForeignMod	A block of foreign items: ‘extern "C" ... ‘.
80	ItemImpl	An impl block providing trait or associated items: ‘impl<A> Trait for Data<A> ... ‘.
81	ItemMacro	A macro invocation, which includes ‘macro_rules!‘ definitions.
82	ItemMod	A module or module declaration: ‘mod m or mod m ... ‘.
83	ItemStatic	A static item: ‘static BIKE: Shed = Shed(42)‘.
84	ItemStruct	A struct definition: ‘struct Foo<A> x: A ‘.
85	ItemTrait	A trait definition: ‘pub trait Iterator ... ‘.
86	ItemTraitAlias	A trait alias: ‘pub trait SharableIterator = Iterator + Sync‘.
87	ItemType	A type alias: ‘type Result<T> = std::result::Result<T, MyError>‘.
88	ItemUnion	A union definition: ‘union Foo<A, B> x: A, y: B ‘.
89	ItemUse	A use declaration: ‘use std::collections::HashMap‘.
90	Label	A lifetime labeling a for, while, or loop.

91	Lifetime	A Rust lifetime: ‘’a‘.
92	LifetimeParam	A lifetime definition: ‘’a: ’b + ’c + ’d‘.
93	LitBool	A boolean literal: ‘true or false‘.
94	LitByte	A byte literal: ‘b’f‘.
95	LitByteStr	A byte string literal: ‘b"foo"‘.
96	LitCStr	A null-terminated C-string literal: ‘c"foo"‘.
97	LitChar	A character literal: ‘’a’‘.
98	LitFloat	A floating point literal: ‘1f64 or 1.0e10f64‘.
99	LitInt	An integer literal: ‘1 or 1u16‘.
100	LitStr	A UTF-8 string literal: ‘"foo"‘.
101	Local	A local let binding: ‘let x: u64 = s.parse()?‘.
102	LocalInit	The expression assigned in a local let binding, including optional diverging else block.
103	Macro	A macro invocation: ‘println!("!", mac)‘.
104	MetaList	A structured list within an attribute, like ‘derive(Copy, Clone)‘.
105	MetaNameValue	A name-value pair within an attribute, like ‘feature = "nightly"‘.
106	Parenthesized GenericArguments	Arguments of a function path segment: the (A, B) -> C in ‘Fn(A, B) -> C‘.
133	PatConst	A pattern that contains a constant.
134	PatIdent	A pattern that matches an identifier or wildcard: ‘mut var @ pat‘.
135	PatLit	A pattern that matches a literal value.
136	PatMacro	A pattern that comes from an inline macro.
137	PatOr	A pattern that matches any one of a list of patterns.
138	PatParen	A pattern enclosed in parentheses.
139	PatPath	A pattern that matches a path to a constant.
140	PatRange	A pattern that matches a range of values.
141	PatReference	A pattern that matches a reference.
142	PatRest	A pattern that matches the rest pattern: ‘..‘.
143	PatSlice	A pattern that matches a slice of values.
144	PatStruct	A pattern that matches a struct.
145	PatTuple	A pattern that matches a tuple.
146	PatTupleStruct	A pattern that matches a tuple struct.
147	PatType	A pattern that matches a type.
148	PatWild	A wildcard pattern: ‘_‘.
149	Path	A path like ‘std::mem::replace‘, possibly containing generic parameters and a self-type.

150	PathSegment	A segment of a path: ‘std‘, ‘std::mem‘, ‘std::mem::replace‘.
151	PreciseCapture	Specifies precise capture behavior for closures.
152	PredicateLifetime	A lifetime predicate in a where clause.
153	PredicateType	A type predicate in a where clause.
154	QSelf	The portion of a path before the ‘::‘, if the path is qualified.
155	Receiver	The self argument of an associated method: ‘&self‘ or ‘&mut self‘.
156	Signature	A function signature in a trait or impl block.
157	StmtMacro	A macro invocation as a statement: ‘println!("Hello, world!")‘.
158	TraitBound	A trait bound on a type parameter or associated type.
159	TraitItemConst	An associated constant in a trait.
160	TraitItemFn	An associated function in a trait.
161	TraitItemMacro	A macro invocation in a trait.
162	TraitItemType	An associated type in a trait.
163	TurboFish	A turbo-fish, e.g., ‘collect::<Vec<_»()‘.
164	TypeArray	An array type: ‘[T; n]‘.
165	TypeBareFn	A bare function type: ‘fn(usize) -> bool‘.
166	TypeGenerics	A generic type parameter in a type definition.
167	TypeGroup	A group of types.
168	TypeImplTrait	An ‘impl Trait‘ type in return position.
169	TypeInfer	An inferred type: ‘_‘.
170	TypeMacro	A macro in the type position.
171	TypeNever	The never type: !.
172	TypeParam	A generic type parameter: T: Into<String>.
173	TypeParen	A parenthesized type equivalent to the inner type.
174	TypePath	A path like std::slice::Iter, optionally qualified with a self-type as in <Vec<T> as SomeTrait>::Associated.
175	TypePtr	A raw pointer type: const T or mut T.
176	TypeReference	A reference type: &’a Tor &’a mut T.
177	TypeSlice	A dynamically sized slice type: [T].
178	TypeTraitObject	A trait object type dyn Bound1 Bound2 + Bound3 where Bound is a trait or a lifetime.
179	TypeTuple	A tuple type: (A, B, C, String).
180	UseGlob	A glob import in a use item: .
181	UseGroup	A braced group of imports in a use item: A, B, C.
182	UseName	An identifier imported by a use item: HashMap.
183	UsePath	A path prefix of imports in a use item: std::.

184	UseRename	An renamed identifier imported by a use item: HashMap as Map.
185	Variadic	The variadic argument of a foreign function.
186	Variant	An enum variant.
187	VisRestricted	A visibility level restricted to some path: pub (self) or pub (super) or pub (crate) or pub (in some::module).
188	WhereClause	A where clause in a definition: where T: Deserialize<'de>, D: 'static.

Bảng 3.1: Các nút trong cú pháp mã nguồn của Rust

3.3 Xây dựng đồ thị thuộc tính mã nguồn (CPG)

NOTE: Nói về lý do lựa chọn sử dụng Joern (Joern Backend và Joern Frontend) để xây dựng đồ thị thuộc tính mã nguồn.

Mục này sẽ trình bày quá trình công cụ phân tích ánh xạ cây cú pháp trừu tượng thành cây đồ thị thuộc tính mã nguồn.

3.3.1 Joern Frontend và Joern Backend

3.3.2 Các loại đỉnh và cạnh của đồ thị thuộc tính mã nguồn CPG

Đồ thị thuộc tính mã nguồn là dạng cấu trúc dữ liệu được kết hợp giữa cây cú pháp trừu tượng, đồ thị luồng điều khiển và đồ thị phụ thuộc chương trình. Đồ thị này biểu diễn các đỉnh là các nút trong cây cú pháp trừu tượng và các cạnh của đồ thị biểu diễn mối quan hệ giữa chúng. Đồ thị này giúp theo dõi các luồng điều khiển, các phụ thuộc trong mã nguồn.

STT	**Tên nút**	**Mô tả**
1	META_DATA	Nút này chứa siêu dữ liệu (metadata) của đồ thị. Một đồ thị CPG phải có và chỉ có duy nhất một nút META_DATA.
2	FILE	Nút biểu diễn một tệp mã nguồn. Với mỗi tệp tin trong mã nguồn, đồ thị sẽ có chính xác một nút FILE để biểu diễn tệp mã nguồn đó.
3	NAMESPACE	Nút này biểu diễn một không gian tên (namespace) hoặc một gói (package), đối với Go, nút này biểu diễn các gói trong mã nguồn.

4	NAMESPACE_BLOCK	Nút này biểu diễn một tham chiếu đến không gian tên (namespace) tương ứng. Nó chứa các khối mã nguồn được định nghĩa dưới không gian tên (namespace) tương ứng.
5	METHOD	Biểu diễn các hàm, phương thức hoặc biểu thức lambda.
6	PARAMETER_IN	Nút đại diện cho một tham số đầu vào của hàm.
7	PARAMETER_OUT	Nút đại diện cho một tham số đầu ra của hàm.
8	METHOD_RETURN	Nút này biểu diễn kiểu trả về của một hàm.
9	MEMBER	Biểu diễn thuộc tính của một kiểu cấu trúc (struct).
10	TYPE	Nút đại diện cho một thể hiện (instance) của một kiểu dữ liệu.
11	TYPE_ARGUMENT	Biểu diễn đối số của một kiểu dữ liệu, có thể hình dung giống như các kiểu dữ liệu generic trong JAVA.
12	TYPE_DECL	Biểu diễn một khai báo kiểu dữ liệu ví dụ như khai báo một kiểu cấu trúc (struct) hay khai báo một giao diện (interface).
13	BLOCK	Biểu diễn một khối mã nguồn.
14	CALL	Biểu diễn một lời gọi hàm.
15	CONTROL_STRUCTURE	Biểu diễn một cấu trúc điều khiển ví dụ như câu lệnh if else, vòng lặp for, cấu trúc switch case.
16	IDENTIFIER	Nút định danh biểu diễn tham chiếu đến một biến.
17	JUMP_LABEL	Biểu diễn các cấu trúc nhảy như BREAK, CONTINUE, GOTO.
18	LITERAL	Biểu diễn một hằng số như số nguyên hoặc chuỗi.
19	LOCAL	Biểu diễn một biến cục bộ.
20	METHOD_REF	Biểu diễn một tham chiếu đến hàm khi hàm đó được dùng làm tham số cho một lời gọi.
21	MODIFIER	Biểu diễn một bộ chỉnh sửa (modifier) như static, public hay private.
22	RETURN	Biểu diễn một chỉ thị trả về (return instruction).
23	TYPE_REF	Biểu diễn tham chiếu đến một kiểu dữ liệu.

Bảng 3.2: Các đỉnh trong đồ thị thuộc tính mã nguồn (CPG)

STT	**Tên nút**	**Mô tả**
1	SOURCE_FILE	Quan hệ biểu diễn một nút là tệp gốc của một nút khác. Quan hệ cha con (biểu diễn quan hệ trong cây cú pháp mã nguồn (AST)).
2	AST	Cây cú pháp trừu tượng (Abstract Syntax Tree).
3	CALL	Quan hệ gọi, dùng để biểu diễn quan hệ giữa hàm/phương thức với nút gọi hàm/phương thức đó.

4	CONTAINS	Quan hệ bao gồm.
5	CONDITION	Quan hệ điều kiện, dùng để biểu diễn quan hệ giữa khối điều khiển như if else, switch hay for với biểu thức điều kiện của chúng.
6	ARGUMENT	Quan hệ đối số kết nối các điểm gọi (kiểu nút CALL) với các đối số của chúng, cũng như các nút RETURN với các biểu thức trả về.
7	RECEIVER	Quan hệ kết nối các điểm gọi (CALL) tới đối số nhận (receiver argument) của chúng. Một đối số nhận là một đối tượng mà phương thức đó thuộc về (có thể hiểu đối số nhận ở đây là con trỏ 'this' trong Java).
8	CFG	Quan hệ này biểu diễn luồng điều khiển từ nút nguồn đến nút đích.
9	DOMINATE	Quan hệ biểu diễn tính kiểm soát (dominate) ngay lập tức từ nút nguồn đến nút đích.
10	POST_DOMINATE	Quan hệ biểu diễn nút nguồn kiểm soát (dominate) ngay lập tức sau nút đích.
11	CDG	Quan hệ biểu diễn nút đích phụ thuộc điều khiển vào nút nguồn.
12	REACHING_DEF	Quan hệ biểu diễn một biến được tạo ra từ nút nguồn và không bị gán lại giá trị trên đường đi tới nút đích.
13	CONTAINS	Quan hệ này kết nối một nút tới hàm/phương thức chứa nó.
14	EVAL_TYPE	Quan hệ kết nối một nút tới nút kiểu dữ liệu của nút đó.
15	PARAMETER_LINK	Quan hệ kết nối một nút tham số đầu vào với nút tham số đầu ra tương ứng của hàm/phương thức đó.

Bảng 3.3: Các cạnh trong đồ thị thuộc tín h mã nguồn (CPG)

3.3.3 Chuyển hóa cây cú pháp trừu trượng sang đồ thị thuộc tính mã nguồn

Với các thông tin có được về cây cú pháp mã nguồn được lưu trong các tệp JSON, đồ thị thuộc tính mã nguồn sẽ được dựng lên từ những thông tin đó. Sau đây tôi sẽ mô tả việc ánh xạ các thành phần mã nguồn chính của Rust từ cây cú pháp mã nguồn sang đồ thị thuộc tính mã nguồn.

Ánh xạ tệp (file) và gói (package)

Hình 3.5 biểu diễn ánh xạ từ mã nguồn sang cây cú pháp trừu tượng và ánh xạ sang đồ thị thuộc tính mã nguồn. Cây cú pháp trừu tượng của Rust không có nút biểu diễn gói, thông tin về gói sẽ được lưu trong nút tệp định nghĩa gói đó. Khi ánh xạ nút tệp sang đồ thị thuộc tính mã nguồn

sẽ biểu diễn được nút FILE và NAMESPACE tương ứng.

Nút NAMESPACE_BLOCK biểu diễn thân của gói, nút này đóng vai trò làm nút cha của các định nghĩa bên trong thân của gói.

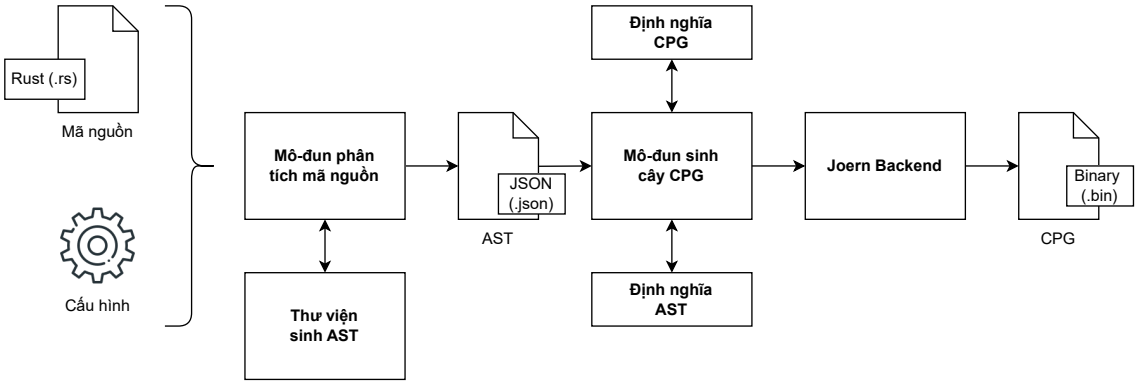
Chương 4

Cài đặt công cụ và thực nghiệm

Chương này sẽ tập trung trình bày về quá trình thực nghiệm và đánh giá phương pháp đánh giá điểm thưởng dựa trên thiết kế đã mô tả chi tiết trong phần trước. Phần thực nghiệm sẽ đi sâu vào việc áp dụng phương pháp để cải thiện hiệu quả công cụ ARAT-RL và so sánh hiệu suất của phương pháp mới với phương pháp đánh giá điểm thưởng mặc định của ARAT-RL. Dựa trên kết quả thu được, tôi sẽ đưa ra các nhận xét và kết luận về giải pháp đề xuất.

4.1 Cài đặt công cụ

Công cụ phân tích mã nguồn Rust được phát triển từ mã nguồn của công cụ Joern. Công cụ được cài đặt bằng ngôn ngữ Rust và Scala. Kiến trúc tổng quan của công cụ được mô tả ở Hình 4.1



Hình 4.1: Kiến trúc công cụ

Công cụ gồm hai mô-đun là GoParser và JoernParse trong đó JoernParse là mô-đun chính. Mô-đun GoParse được cài đặt bằng ngôn ngữ Rust. Mô-đun này sẽ nhận đầu vào là đường dẫn đến thư mục dự án chứa mã nguồn Rust và thực hiện việc phân tích mã nguồn thành cây cú pháp mã nguồn, xử lý kiểu dữ liệu và lưu cây cú pháp mã nguồn vào các tệp JSON. JoernParse được cài đặt bằng ngôn ngữ Scala và được phát triển từ mã nguồn của công cụ Joern. Mô-đun thực hiện nhận mã nguồn đầu vào, gọi mô-đun GoParser để dựng cây cú pháp mã nguồn và sau đó xây dựng đồ thị thuộc tính mã nguồn từ các thông tin trong cây cú pháp mã nguồn. Đầu ra của mô-đun này cũng là đồ thị thuộc tính mã nguồn được lưu dưới dạng tệp nhị phân (.bin). Đây cũng là đầu ra của cả công cụ. Chúng ta có thể dùng một số công cụ được Joern cung cấp sẵn để thao tác với đồ thị này như xuất đồ thị dưới nhiều định dạng khác nhau như neo4jcsv, graphml,

graphson, dot bằng công cụ JoernExport, thực hiện các câu lệnh truy vấn trên đồ thị hoặc quét đồ thị để tìm lỗ hổng trong mã nguồn bằng công cụ JoernScan.

4.1.1 Mô-đun xây dựng cây cú pháp trừu tượng Rust Parser

Chức năng chính của GoParser là nhận thông tin đường dẫn đến mã nguồn, lọc các tệp mã nguồn Rust, sinh cây cú pháp trừu tượng, xử lý kiểu dữ liệu cho các nút định danh (identifier) và lưu kết quả vào các tệp JSON. Các thành phần trong mô-đun này bao gồm cli, parser, resolver, ast và util. Chức năng chi tiết của từng thành phần được mô tả như sau:

- Thành phần cli làm nhiệm vụ xử lý và cung cấp giao diện dòng lệnh (CLI), nhận đầu vào là đường dẫn đến dự án Rust, nhận các cấu hình (ví dụ như đường dẫn lưu các tệp đầu ra hoặc danh sách các tệp mã nguồn cần bỏ qua) và gọi các chức năng phân tích.
- Thành phần parser thực hiện nhận thông tin từ thành phần cli, từ đó đọc các tệp mã nguồn của Rust có đuôi là ".rs" từ dự án đầu vào, phân tích các tệp tin này và sinh cây cú pháp trừu tượng.
- Thành phần resolver cung cấp các API phục vụ việc xử lý kiểu dữ liệu.
- Thành phần ast chứa các cấu trúc định nghĩa các loại nút và thuộc tính của từng loại nút của cây cú pháp trừu tượng.
- Thành phần util cung cấp các tiện ích chung bao gồm xử lý tệp tin và xử lý nhật ký (log).

Hình 4.2 biểu diễn các thành phần của mô-đun GoParser, các mũi tên biểu mối quan hệ sử dụng giữa các thành phần này.

4.1.2 Mô-đun xây dựng đồ thị thuộc tính mã nguồn Joern Rust

Mô-đun JoernParse thực hiện nhiệm vụ xây dựng đồ thị thuộc tính mã nguồn (CPG) từ cây cú pháp trừu tượng AST. Mô-đun sẽ thực hiện đọc thông tin từ các tệp JSON được tạo ra từ mô-đun GoParser và xây dựng đồ thị thuộc tính mã nguồn (CPG). Mô-đun này được phát triển từ mô-đun JoernParse của công cụ Joern nên sử dụng lại một số thành phần có sẵn của Joern bao gồm các thành phần console, semanticcpg, codepropertygraph và x2cpg. Dưới đây tôi sẽ mô tả chức năng của từng thành phần:

- Thành phần console thực hiện chức năng tương tự như thành phần cli của mô-đun GoParser.
- Thành phần rust-language-frontend đảm nhận nhiệm vụ nhận đường dẫn đến mã nguồn từ thành phần console, gọi mô-đun GoParser để sinh cây cú pháp trừu tượng sau đó đọc các tệp JSON được sinh ra và xây dựng thành đồ thị thuộc tính mã nguồn.
- Thành phần codepropertygraph chứa các lớp định nghĩa các đỉnh và cạnh của đồ thị thuộc tính mã nguồn.
- Thành phần x2cpg cung cấp các tiện ích để ánh xạ các thuộc tính từ các nút của cây cú pháp trừu tượng sang thuộc tính của các đỉnh trong đồ thị thuộc tính mã nguồn.

- Thành phần semanticcpg chứa các tiện ích cho phép vấn đồ thị thuộc tính mã nguồn. Thành phần này được sử để thực hiện các tác vụ hậu xử lý sau khi đồ thị thuộc tính mã nguồn được xây dựng (ví dụ như bổ sung thông tin hoặc tạo các đỉnh liên kết).

Hình 4.3 biểu diễn các thành phần của mô-đun GoParser, các mũi tên biểu mối quan hệ sử dụng giữa các thành phần này.

4.2 Thực nghiệm trên các tính năng đã hỗ trợ

4.2.1 fn

4.2.2 if let

4.2.3 let else

4.2.4 life time

4.3 Nhược điểm

4.3.1 Macro

4.3.2 Qself

4.3.3 Path

4.3.4 Proxy

Kết luận

Phân tích mã nguồn là một bước quan trọng trong quá trình phát triển phần mềm. Quá trình này sẽ giúp phát hiện các lỗi, lỗ hổng bảo mật và vấn đề tiềm ẩn tồn tại trong mã nguồn, đồng thời nó giúp lập trình viên phát hiện các lỗi logic, tối ưu mã nguồn và đảm bảo các quy tắc và tiêu chuẩn khi lập trình. Điều này giúp sớm phát hiện các lỗi nghiêm trọng, đảm bảo tính ổn định, dễ quản lý và dễ bảo trì cho mã nguồn về sau.

Hiện nay trên thế giới có khoảng 20 ngôn ngữ lập trình thông dụng và đi kèm với đó là hàng trăm công cụ phân tích mã nguồn khác nhau. Khóa luận này đã xây dựng thành công một công cụ phân tích mã nguồn dành cho ngôn ngữ lập trình Rust. Khác so với các công cụ phân tích mã nguồn hiện có, công cụ chọn phân tích mã nguồn thành đồ thị thuộc tính mã nguồn thay vì chỉ phân tích thành cây cú pháp trừu tượng bởi lẽ dạng đồ thị này là tổng hợp của cây cú pháp trừu tượng, đồ thị dòng điều khiển và đồ thị thuộc tính mã nguồn. Nó không chỉ cung cấp thông tin về cú pháp trong mã nguồn mà còn cung cấp cả thông tin về luồng điều khiển và phụ thuộc dữ liệu trong chương trình. Đồng thời công cụ cũng được phát triển dựa trên công cụ có sẵn là Joern nên nó kế thừa được những tiện ích mạnh mẽ sẽ giúp cho quá trình khai thác mã nguồn dựa trên đồ thị thuộc tính mã nguồn trở nên dễ dàng và chuyên sâu hơn. Qua quá trình thực nghiệm công cụ với bộ dự án mã nguồn Rust phổ biến, công cụ đã đáp ứng được những mục tiêu đề ra của khóa luận khi đã xử lý được thông tin về kiểu dữ liệu, phân tích được phần lớn các cú pháp mã nguồn Rust và đưa ra kết quả chi tiết và đầy đủ hơn so với công cụ Joern. Tuy nhiên, công cụ vẫn chưa xử lý được toàn bộ các cú pháp mã nguồn mà Rust cung cấp, việc phân tích còn tốn nhiều thời gian đặc biệt là với các dự án có sử dụng nhiều thư viện bên ngoài do việc xử lý kiểu dữ liệu đòi hỏi phải phân tích cả những thư viện đi kèm.

Trong tương lai, công cụ phân tích mã nguồn Rust sẽ hoàn thiện hơn khi xử lý được toàn bộ các cú pháp mã nguồn. Việc xử lý kiểu dữ liệu sẽ được tối ưu bằng một chiến thuật phân tích khác thay vì phải phân tích toàn bộ các thư viện đi kèm dự án, từ đó giúp thời gian phân tích nhanh chóng hơn. Đồng thời công cụ cũng sẽ hỗ trợ các hệ điều hành thông dụng như Windows hay MacOS thay vì chỉ hỗ trợ Ubuntu như hiện tại. Với độ chi tiết của đồ thị đầu ra và các tiện ích truy vấn mạnh mẽ đi kèm, công cụ này sẽ là một nền tảng mạnh mẽ để xây dựng nên các công cụ phân tích mã nguồn khác với các chức năng như tìm kiếm lỗ hổng trong mã nguồn, gợi ý mã nguồn, phát hiện lỗi cú pháp... Bên cạnh đó, đầu ra của công cụ cũng có thể sử dụng cho các bài toán về học máy hoặc học sâu.

Thông qua quá trình tìm hiểu và xây dựng công cụ, em đã có được những kiến thức sâu hơn về ngôn ngữ lập trình Rust. Bên cạnh đó, em cũng có thêm hiểu biết về các công cụ phân tích mã nguồn cũng như các dạng đồ thị biểu diễn mã nguồn. Ngoài ra, em cũng đã trau dồi thêm được các kỹ năng mềm sẽ giúp ích cho bản thân trong chặng đường sự nghiệp sắp tới.

Tài liệu tham khảo

[1] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 289–301, 2022.