UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Gulnar Mammadli

Compiling GitHub Actions Into Datalog

Master's Thesis (30 ECTS)

Supervisor(s): Bruno Rucy Carneiro Alves de Lima, MSc
Mykhailo Dorokhov, MSc

Tartu 2024

Compiling GitHub Actions Into Datalog

Abstract:

Datalog is a declarative programming language that defines arbitrary relationships with rules and facts. With this paper, we aim to evaluate using it as an alternative syntax to write continuous integration pipelines over the GitHub Actions platform. These pipelines are written in YAML and denote dependency chains. Our key contribution is experimenting with a syntax closer to the semantics being outlined, potentially solving the common problems developers have while working with these files.

Keywords:
GitHub Actions, Datalog, DevOps, CI/CD, Compiler

CERCS: P170 Computer science, numerical analysis, systems, control

GitHub Actions toimingute kompileerimine Datalogi

Lühikokkuvõte:

Datalog on deklaratiivne programmeerimiskeel, mis defineerib seoseid reeglite ja faktidega. Selle artikliga püüame hinnata selle kasutamist alternatiivse süntaksina, et kirjutada pidevintegratsioon konveier GitHub Actions'i platvormi kaudu. Need konveierid on kirjutatud YAML-is ja tähistavad sõltuvusahelaid. Meie peamine panus on süntaksi katsetus, mis on lähedasem kirjeldatavale semantikale, potentsiaalselt lahendades levinumaid probleemid, mida arendajad nende failidega töötamisel kogevad.

Võtmesõnad:

GitHub Actions, Datalog, DevOps, CI/CD, Kompilaator

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Acknowledgements

I want to express my gratitude to my supervisor, Bruno Rucy Carneiro Alves de Lima, for his constant support and always guiding me in the right direction. His supportive nature helped me be more motivated in my work. With our regular meetings, he always checked the progress that I had made and encouraged me to explore new methods. I also want to thank my manager and the "Industrial Master's Programme in IT" program supervisor at Pipedrive, Mykhailo Dorokhov, for his constant support. Additionally, I want to express my genuine gratitude to my teammates and team manager, Andrii Rozumnyi at Pipedrive, who always helped me and let me spend more time working on my thesis. Moreover, I am thankful to my teammate, Martin Kisand, for helping me translate the abstract of this thesis into Estonian and to my colleague, Taaniel Saarnik, for helping me with the thesis structure. Most importantly, I am grateful to my family, who not only always asked about the progress I made in my thesis but also how I was doing emotionally and supported me in every way I could pursue my dreams.

Contents

List of Figures

1 Introduction

1.1 Motivation

GitHub[1] is one of the famous platforms where developers can create, manage, and share their codes. It allows multiple developers to work on the same projects. GitHub provides access controls in addition to distributed version controls. GitHub Actions[2] as one of GitHub's key features is becoming more popular among businesses, which has resulted in many companies switching and starting to use it as a part of their development and deployment lifecycles. Much research has been done on GitHub Actions, its benefits, features, and common issues that developers face. GitHub Actions (GHA) has many features that aid developers in automating all their software workflows with a robust continuous integration and development process (CI/CD). Besides quickly creating and utilizing workflows, it offers built-in features in its marketplace to make the process quicker.

Having a reliable CI/CD tool is one of the critical factors for companies in developing and publishing their software products. CI/CD targets accelerating the software development lifecycle while reducing errors and development and delivery costs. It automates the process so that manual human intervention is decreased as much as possible. By applying DevOps's best practices, developers can increase their productivity. To automate the process, developers need more workflows triggered by various actions or events. For example, one workflow file can be configured to build and test the code, and another can check linting and formatting. Therefore, over time, these files get bigger, and at some time, one workflow file can depend on other files. This can cause some dependency issues among files when triggered. Besides dependency problems, making mistakes while configuring the file is easy. A typical example is formatting issues. Sometimes, it is hard to be sure whether the file is in the correct format. This paper investigates whether declarative language is a good fit for easily building dependency relationships.

[1] https://github.com/

[2] https://github.com/features/actions

1.2 Contribution

In this thesis, we aimed to write a program both to compile GitHub Actions into Datalog and Datalog to GitHub Actions to answer the following research questions then:

1. **What is necessary to compile Datalog to GitHub Actions? Does it make things easier or harder?**
2. **How about compiling GitHub Actions to Datalog? One of the biggest problems with GitHub Actions is propagating changes to upstream GitHub Actions to those downstream. Could having GitHub Actions as a Datalog program make it easier to deal with this problem?**

To answer these questions, we divided our investigation into two sections. Firstly, we investigated how to compile Datalog to GitHub Actions and outlined the associated issues. Secondly, we implemented a program that converts GitHub Actions to Datalog rules and facts. As a next step, we decided to "run" that datalog program as a form of linting. This allowed us to detect whether the provided file was in the correct form or whether there were dependency issues or missing data.

1.3 Thesis Outline

This thesis consists of five chapters. Chapter 1 introduces the thesis topic, research questions, and our contribution. Chapter 2 is a background section overviewing CI/CD, DevOps, Git, GitHub Actions, Datalog, and Rust. Additionally, related works on these topics have been discussed as well. Chapter 3 discusses how to compile GHA to Datalog. Chapter 4 explains how we compiled GHA to Datalog. In Chapter 5, we describe the results and conclude the thesis by answering research questions and summarizing what was achieved and the expected influence of how the proposed method can be used to find dependency problems to increase maintainability.

2 Background

Here, topics such as CI/CD, DevOps, its core principles, issues it resolves, Git, GitHub Actions and its components, Datalog, and related works are discussed.

2.1 CI/CD

CI/CD stands for continuous integration and continuous delivery. Increasing application complexity makes maintaining, developing, and delivering effectively and efficiently challenging. With the help of a formal CI/CD process, it becomes easier to deliver software products while reducing the development complexity and delivery time. If there are new code changes, developers want to ensure the existing code will stay intact when integrated with the new code. In this case, the CI pipeline runs, and it is ensured that everything works correctly. Additionally, the CI pipeline builds the codes and runs the existing tests. On the other hand, the CD pipeline safely enables developers to deploy the code to the production stage. It is an automated process that ships the code to environments like testing and production. It guarantees that the software product can be released reliably at a high pace. The examples of the tools used for CI/CD are Jenkins[3], CircleAI[4], TeamCity[5], GitLab[6], Bamboo[7][1].

Its Benefits

Automated testing in CI/CD ensures software quality and security for businesses. The CI/CD pipeline also reduces the time needed to release new product features. With these, an organization can boost its profit and competitive edge. Moreover, automation helps developers focus on the more critical tasks, enabling them to be more productive [2].

2.2 DevOps and Its Principles

The notion of DevOps appeared in the early 2000s and was proposed by Patrick Debois. *DevOps* is a methodology highlighting collaboration, automation, integration, and quick

[3]https://www.jenkins.io/
[4]https://www.clientcircle.com/features/circleai
[5]https://www.jetbrains.com/teamcity
[6]https://about.gitlab.com/
[7]https://www.atlassian.com/software/bamboo

feedback cycles. In other words, DevOps combines development (Dev) and IT operations (Ops), enabling faster and more secure software development and delivery. It is a smooth process that establishes fast delivery and deployment and decreases the risks of mistakes. This methodology has principles that help achieve its purpose. It has four fundamental principles [3]:

1. **Culture** - DevOps methodology emphasizes the significance of collaboration and communication between teams. Having shared goals, values, and a culture of trust is a key to success in DevOps. Development and operations teams keep each other informed by communicating and collaborating during the development and deployment cycle.
2. **Automation of The Software Development** - Automation has a vital role in DevOps. It is a process of developing and delivering software with minimum human interference. Developers try to automate as much as they can. Automation in DevOps has many gains for businesses. With the help of automation tools and systems, organizations lower operational costs, software delivery time, and manual errors while boosting productivity, efficiency, and reliability.
3. **Measurement** - Understanding how well the applied practices and principles work together is necessary. The companies can make crucial decisions by analyzing the essential metrics, such as lead time, deployment frequency, and mean time to recovery [3].
4. **Feedback Loop** - Building a continuous process to get customer feedback helps to see weak points and creates new opportunities to improve. Feedback loops are essential since they aid in advancing software development quality. There are two types of feedback loops:
   (a) Reinforcing feedback loop
   (b) Balancing feedback loop

   The first feedback loop type is a positive feedback loop where the output expands the input while the balancing feedback loop decreases it [4]. Balancing the feedback loop is considered a negative loop. An example of a negative loop is that when the production team discovers bugs, the code is not deployed to production. Instead, this information is shared with the development team. The code is

deployed once the development team fixes the bugs. Unlike positive feedback, it slows the process between the development and operations teams to make the software reliable. Collaboration and giving feedback are two essential processes that often exist together. When teams collaborate, giving and receiving feedback is easier and faster. Additionally, getting continuous feedback is a crucial factor in having solid collaborations.

Solved Issues

Continuous feedback is critical in fixing errors more easily and quickly. It fixes the errors or issues in earlier stages. Eventually, companies benefit from this by reducing the response time to customer problems and increasing their business ranges. In contrast to traditional techniques, DevOps is a method that continuously grows and develops. This means that with this method, businesses always utilize state-of-the-art technologies. By using Distributed Version Control, all team members participate in providing a smooth workflow of actions to satisfy customer needs and requirements. Continuous testing in all development and delivery stages makes the software more robust and reliable. In addition to these, automation significantly reduces manual work. All of these decrease the development and delivery costs quite a lot. Therefore, companies can deliver a lot of new features to their customers. Thus, adopting DevOps methodology and principles lets companies make high-quality products faster and more efficiently while decreasing security risks.

2.3 Git

Git[8] is an open-source distributed version control management system that enables multiple developers to work together on the same code base. A *version control system* is a source control system that tracks and manages modifications to software code. It keeps track of each change made to the code. Therefore, if developers make mistakes in code, they can easily compare previous versions of the code to fix the error. Below, we explain some of the fundamental notions in Git.

Branch

A Git branch is a pointer to a snapshot of the changes that a developer did [5]. Whenever

[8]https://git-scm.com/

a new feature is to be implemented, or an error is to be fixed, a developer creates a new branch that holds all the latest changes.

Pull Request

A pull request (PR) is a mechanism that a developer uses to notify other developers in the team that the feature they were working on is done and ready for review. A PR makes it easier to develop and collaborate with other developers.

Commit

A Git commit is a core building block unit of a Git project timeline. It can be considered as a snapshot along the timeline of the project [5].

Git Workflow

A Git workflow consists of three states [6]:

1. Working directory - A developer modifies code in the local directory.
2. Staging area - A developer stages the files. It stores data about what will go into the next commit.
3. Git directory (Repository) - In this stage, developers make a commit that stores the snapshots in a Git repository or directory.

One of the main benefits of Git is that it helps developers to have a copy of the code on their local systems. Additionally, when a developer works with other developers on the same project, others can see changes made to the source code. It helps to find and fix the errors easily.

2.4 GitHub Actions

Social code platforms have enabled practitioners to work together, communicate, and collaborate on projects effectively. These platforms allow developers to share and manage their code repositories easily. One of the famous such platforms is GitHub. Around 2019, the platform introduced its new feature, GitHub Actions. Before this feature, developers used Jenkins and Travis CI[9] for CI/CD. It has been replacing traditional CI/CD tools rapidly. GitHub Actions is a CI/CD platform offered by GitHub that lets one automate

[9]https://www.travis-ci.com/

the build, test, and deployment pipeline. It is essential to mention that the entire platform is called GitHub Actions. The GitHub Actions have some components, such as events, workflows, jobs, actions, and runners. Using GitHub Actions (GHA), one can configure the workflow for the pull request that opened to the repository and check some conditions. For instance, for every new pull request, a developer can check whether all tests pass with the newly added code. GHA is deeply integrated with GitHub. It can be used for running tests, code reviews, and monitoring.

2.4.1 GHA Components

Let us dive into the details of GHA components.

- **Event** – An activity that triggers a workflow in a repository, as shown in Figure 1. Examples of events include creating a new pull request, opening an issue, or publishing a software release. One can compose external triggers using a dispatch webhook.
- **Runner** – A server that runs the workflows and can run one job simultaneously.
- **Job** – A job consists of steps in a workflow. These steps depend on each other and run in order.
- **Workflow** – An automated process that can run one or more jobs [7]. A workflow will run when an event triggers it. It consists of events, steps, actions, runners, and jobs. It will run jobs specified in the file when triggered. Developers can have or use as many workflow files as needed for a repository. For instance, a developer can create a workflow to build and test every pull request and another one to deploy. Additionally, a workflow can be reutilized in another one. Each workflow can have one or more actions.
- **Action** – An action is the smallest building block used in workflows. The goal of actions is to decrease the amount of repetition, which results in saving time. It is mainly used as a step in a job to configure a workflow. For instance, an action can pull a Git repository from GitHub. In addition, developers can configure custom actions in an existing workflow or use actions from GitHub Marketplace. They can build JavaScript, Docker containers, and composite actions. Actions need metadata files to define their inputs, outputs, and entry points for the action. After

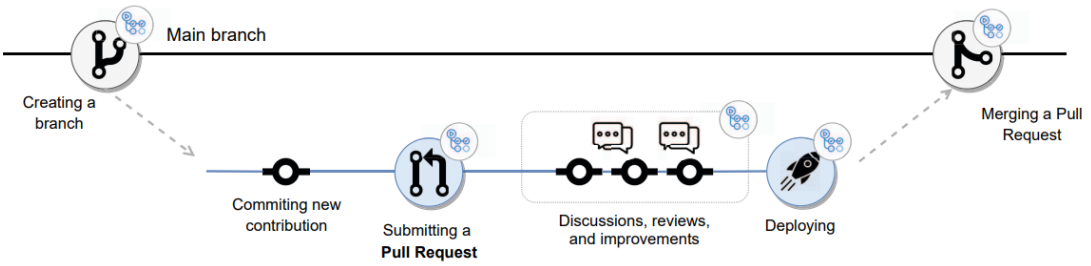

Figure 1. GitHub Workflow Automation with GitHub Actions(adapted from [8]).

the workflow runs, the outputs can be displayed through the GitHub Action Bot [8].

- **Artifact** - Artifacts are the files generated when building or testing software projects. They can have binary packages needed for your project. They can be created for one job and used for another.
- **Step** - It is an action or command. A job can have one or more steps running in the same runner.

2.4.2 GHA Workflow File

After having the example workflow file, as shown in Figure 2, this workflow file will be triggered whenever someone pushes something to the "main" branch in a repository or creates a pull request.

The file in Figure 2 is a simple example of building, testing, and deploying a Node.js[10] code. It can be updated to meet the specific project's needs and requirements. As seen from this workflow file, the workflow has a name (line 1), conditions (lines 3-7) for when to be triggered, and two jobs called "Build/Test" (lines 10-32) and "Deploy" (lines 33-42). These jobs have multiple steps that use different actions and run scripts. The first "build" job in line 10 is triggered whenever code changes occur. After completing all steps in lines 19-32, the deployment job will be triggered. This example shows that the deploy job can not run before the first job. This has been achieved by adding "needs: build" in line 35 to the "deploy" job.

Developers can see a list of all the workflows under the "actions" section on the GitHub

[10]https://nodejs.org/

```yaml
name: Node.js CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    name: Build & Test
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./src
    strategy:
      matrix:
        node-version: [20.x]
    steps:
    - name: Checkout repository
      uses: actions/checkout@v4
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - name: Install dependencies
      run: npm ci
    - name: Run the tests
      run: npm test --if-present
    - name: Build
      run: npm run build --if-present
  deploy:
    name: Deploy
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.ACTIONS_DEPLOY_ACCESS_TOKEN }}
          publish_dir:  src/build
```

Figure 2. GitHub Actions Workflow File.

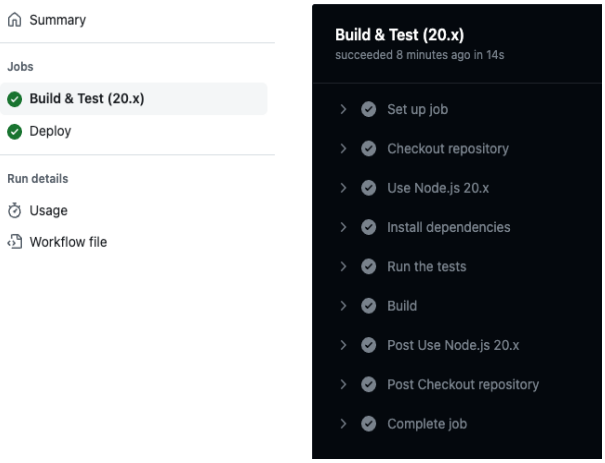

Figure 3. GitHub Actions Jobs.

platform. By clicking on any workflow file, they can see the run details, see if everything goes smoothly or fails, and see where and why. Figure 3 shows that both jobs successfully run, and all steps in these jobs are also shown on the right side. So, depending on the requirements, different workflow files can be created and set up for any software.

The previous research [8] on GHA shows that the following are the five most frequently used categories of actions: Continuous Integration, Utilities, Deployment, Publishing, and Code Quality, with “actions/checkout” as the most popular action. Chen et al. [9] show positive findings about using GHA in popular projects. The researchers in the paper find that projects with many developers tend to adopt GHA, and the average number of workflows per project is almost three [9].

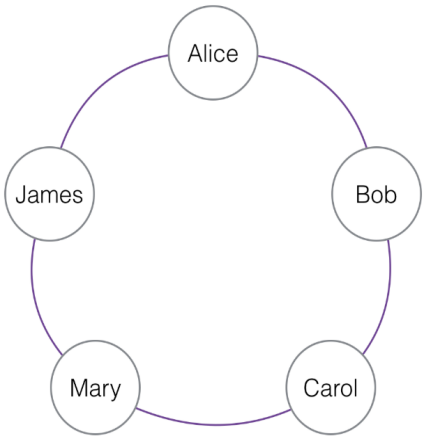

Figure 4. Social Network Example (adapted from [10]).

2.5 Datalog

Datalog is a declarative programming language for logical inference. It is a simple version of general Logical Programming and a subset of Prolog[11]. Let us have a look at the following example [10] :

People sometimes receive some recommendations if they use social media, such as Facebook or LinkedIn. One of these kinds of recommendations is a friend recommendation. There are connections that they are friends with; some are their friends or those they do not know. While using this feature, most people need help understanding how it works. Let us assume we have a small social network, as shown in Figure 4. Each edge represents a friendship relationship between the nodes at the ends of an edge. This means that Alice and Bob are friends, and Bob is Carol's friend. Therefore, Alice and Carol have a connection. We can generalize the connection between two people as below:

$$connection(X, Z) \leftarrow friends(X, Y), friends(Y, Z) \quad [10];$$

This connection above is an example of Horn clauses, and it means: "There is an association between X and Z if and only if there is a friendship between X and Y, and Y and Z."

[11] https://en.wikipedia.org/wiki/Prolog

2.5.1 Syntax

A Datalog program consists of rules and facts [11]. Facts are statements held to be true, while the rules say how to deduce new facts from known facts. For instance, "John is the father of Harry" is a fact. An example of a rule is " If A is a parent of B, and B is a parent of C, then A is a grandparent of C." The rules are more general and contain variables such as A, B, and C in this example.

A Datalog program is a finite collection of rules and facts. In Datalog, both facts and rules are Horn clauses in the form of

$$L_0 \leftarrow L_1, ..., L_n \quad [11]$$

where each L_i is a literal form of

$$p_i(t_1, ..., t_{k_i}) \quad [11]$$

such that p_i is a predicate symbol and t_j are terms. A term is either a variable or a constant. In Datalog, there are two types of schemas: extensional and intensional schemas [12]. Extensional schema has extensional relations (EDBs), which are seen only on the right side of the Datalog rules, while intensional schemas consist of intensional relations (IDBs). Intensional relations are at least once on the Datalog rules' left side and considered "output".

A clause is a head followed by a body. The left-hand side of the clause is called the head, and the right-hand side of it is the body. The arrow notation ($\leftarrow$) separates the head from the body. A body is a comma-separated list of literals. The body here is optional. This means that there can be clauses without a body. This kind of clause is called facts. A fact can be defined as an atom with no variables. For a clause to be considered a rule, the body should have at least one literal. An atom in Datalog is a predicate. Therefore, the head is an atom, and the body is one or more atoms. A fact or rule that does not contain any variables is called ground.

Safety Conditions: A rule is considered a safe rule if the following conditions are met [13]:

1. Each fact is a ground.

2. A variable in the rule's head must also occur in the body.

Having safe rules prevents infinite results. Let us have a look at this rule:

$$R(x, y) \leftarrow Parent(" Hasan", x)$$

The rule will produce an infinite set of facts by having *y* not be bound anywhere in the body and assuming Prolog semantics. Therefore, this rule is unsafe.

2.5.2 Herbrand Universe and Model

Herbrand Universe

Assuming we are given an alphabet A, the set of all ground terms constructed from A's constant and function symbols is called the Herbrand Universe [14]. Let us consider the following program:

$$Grandparent(a, b) \leftarrow Parent(a, c), Parent(c, b) \quad [14]$$

This program means that *a* is the grandparent of *b* only and only if *a* is the parent of *c*, and *c* is the parent of *b*. If we consider the following "relations":

$$\textbf{Parent}(James, Tom), \textbf{Parent}(Tom, Emma)$$

Then, the Herband Universe of the program will be expressed as below:

$$U_A = \{James, Tom, Emma\}$$

Herbrand Model

A **Herbrand Interpretation** of a program P is an interpretation I that satisfies the following conditions [14]:

1. The domain of I: |I| = U_P
2. For every constant a: a_I = *a*
3. For every function symbol: $f_I(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$

4. For every predicate symbol: $P_I \subseteq (U_P)^n$

A **Herbrand Model** is an interpretation that satisfies the program, and the least Herbrand model is the smallest interpretation that satisfies the program.

2.5.3 Evaluation Methods

There are two approaches to evaluating a Datalog program: Bottom-up and Top-down. The top-down evaluation requires a query besides the data and a program. In this section, we will focus on bottom-up evaluation techniques.

The most common approaches are the Gauss-Seidel method [11], using arithmetic constraints [15], [16], the push method [17], using negation [18], and a linear algebraic approach [19]. We will explore some of the well-known methods for bottom-up evaluation.

The Gauss-Siedel method [11] is an iterative technique for solving a system of equations. The Gauss-Siedel method is prevalent in mathematics for finding an iterative solution to systems of equations. In this paper, the authors apply the following process: Initially, they take the system of algebraic equations and an extensional Database EDB. Then, they take the variable relations R_i as an empty set. Later, for (i=1,...,n) iteration, the following computation has been iterated:

$$R_i = E_i(R_1, ..., R_n)$$

In the end, the values of the variable relations R_i are the solution of the given system of equations. This method has two main inefficiencies [11]:

1. Some tuples are computed many times during iteration.
2. It produces all of the result relations.

Taisuke Sato [19] proposes a new technique for Datalog evaluation in which he evaluates logic programs in vector spaces. For that purpose, he considers a class of Datalog programs written using N constants and binary predicates. Then, he converts if-and-only-if completions of programs into matrix equations in N-dimensional Euclidean space with a non-linear operation and continues to translations with purely linear matrix equations. Ultimately, he gets the least Herbrand model as a set of adjacency matrices and computes

it by solving previous linear matrix equations in the vector space. Later, he verifies the validity of the new approach by conducting two different experiments. As a result of his experiments, he concludes that his linear algebraic approach runs around 10^1 ~10^4 times faster than the symbolic systems [19].

Ullman in [20] shows that for any safe Datalog program and any query, another safe Datalog program produces the answer to the given query, which takes less time when evaluated by semi-naive evaluation than evaluated top-down. A semi-naive evaluation is an algorithm that computes the least fixed point of rules. Instead of using all derived facts as input in each iteration, the semi-naive evaluation computes only new facts. Before Ullman, Ramakrishnan [21] independently shows something similar to the previous statement.

2.6 Programming Language

Rust[12] is a programming language that enables developers to write and build reliable and efficient software. It is a fast and memory-efficient language, meaning there is no runtime or garbage collector. Rust has a lot of packages that can be used in different tasks.

The following Listing 1 is used to explain the Rust language. The example shows how custom types are created and used besides primitive variable types. There are two custom data types (*struct*s) called *Person* and *Location* in lines 1-8. The *Person* has a name and age while *Location* stores city and country data. All of them have a string type, but age is an integer. The function *get_info* in lines 12-15 is defined to get full information about people and their locations. *Macros* in Rust are used to define reusable codes. For instance, the already existing *format!* macro in this method is used to format given arguments into the desired structure. In lines 17-20, the compiler internally inferred the types of *Person*s and *Location*s.

[12]https://www.rust-lang.org/

```rust
struct Person {
    name: String,
    age: i32,
}

struct Location{
    city: String,
    country: String,
}

fn main() {

  fn get_info(person: Person, location: Location) {
     let formatted = format!("{} is {} years old and from
     {}, {}.", person.name, person.age,
     location.city, location.country );

     println!("{}", formatted);
  }

    let person1 = Person { name: "Andres".to_string(), age:
   40};
    let person2 =Person { name: "Maryam".to_string(), age: 35};
    let location1 = Location {city: "Tartu".to_string(),
   country: "Estonia".to_string()};
    let location2 = Location {city: "Baku".to_string(), country
   : "Azerbaijan".to_string()};

    get_info(person1, location1);
    get_info(person2, location2);
    }
}
```

Listing 1. Rust Code Example.

Listing 2 shows the outputs when the code runs.

```
Andres is 40 years old and from Tartu, Estonia.
Maryam is 35 years old and from Baku, Azerbaijan.
```

Listing 2. Rust Output.

It shows two formatted outputs because the *get_info* function was called twice with different arguments.

3 Compiling Datalog to GHA

This section covers the preliminaries, the setting, and the experiments to compile Datalog to GHA.

3.1 Preliminaries

This section covers the data structures and algorithms used in the experiments while compiling GHA to Datalog.

3.1.1 Graphs

Graph data structures are commonly used to represent social networks, such as a social connection of friends on social media applications or recommendation systems, the connections between different places in a transportation network [22], and so on. These are some of the real-life situations in which graphs can be used. Graphs are used in compilers significantly. They can be used for type inference, query optimization in database languages, etc. They are an advanced tool to represent relationships between complex entities.

A graph is created from nodes and edges which connect nodes. In other words, graphs are composed of a set of vertices and a set of edges. Nodes and edges are the fundamental components of a graph. Nodes, known as vertices or vertexes, are basic units of graphs. They can be either labeled or unlabeled. Let us assume there are two nodes, and they are connected by a line. This line that connects them is called an edge. These nodes can be ordered pairs as well. There are two types of graphs based on direction:

1. Directed Graph - In a directed graph, edges have a certain direction, which means that nodes are ordered pairs.
2. Undirected Graph - It is a graph whose edges do not have any direction.

3.1.2 Trees

Trees are graphs with more rules. This means that every tree is a graph; however, not all graphs are trees. They are non-linear data structures with a root and one or more subtrees. Being a non-linear data structure means the elements connect hierarchically, making

traversing all elements in a single run difficult. The relationship or direction between nodes is parent ← children. Therefore, they are considered as hierarchical structures.

Having this kind of data structure is essential for complex entities. For instance, in our work, the data are from a file in YAML format, meaning data is presented as key-value pairs or a parent ← children relationship.

3.1.3 DFS

The depth-first search algorithm is a kind of traversal algorithm that uses backtracking. It is an algorithm that traverses or searches graph or tree data structures. It starts traversing from the root node and goes deeper as far as possible before backtracking. It explores the entire graph by examining each node until the leaf nodes. The leaf nodes are the nodes that have no children.

3.1.4 Simple Paths

A simple path in a graph is a path that does not have repetitive vertices or nodes. This means each node is visited only once and appears once in a sequence. Additionally, no edge is repeated as well. We used an existing library to get all simple paths from the graph in our work. To simplify it, we considered the simple paths from leaf nodes to the parent, in other words, with the bottom-up principle. The size of data we passed to the program during the experiments decreased by using simple paths since they do not hold repetitive paths. Additionally, it made it easier to experiment with the data.

3.2 The Setting

Pipedrive[13] is a company that offers customer relationship management (CRM) software. This software is designed to help businesses grow by providing one of the best CRMs to salespeople. With CRM application, Pipedrive offers the possibility of tracking your sales pipeline, managing the leads, and automation.

The mission of the DevOps team in Pipedrive is to facilitate efficient software development by providing and maintaining an optimized development environment and related tooling. They work as a team to eliminate waste from the development process, enhance

[13]https://www.pipedrive.com/en/about

productivity, and enable Piepdrive teams to focus on creating high-quality software. The teams in Pipedrive used Jenkins as a CI/CD tool. However, Pipedrive has recently begun an initiative among teams to implement and use GHA as a complete CI/CD. For this purpose, the DevOps team implemented various workflows to satisfy different needs, such as building, testing, and deploying. Then, every team reused these created workflows and adapted them for the specific needs of their services.

3.3 The Implementation

During the experiment, different workflow files have been analyzed and used. Figure 5 represents a part of one of these kinds of files. In the experiment, we decided to consider some of the data as facts, such as "secrets", "inputs", and "env" (environment variables) and other static data. Usually, these "secrets" and "env" values are not stored in the same file. As seen from Listing 3, we used variables in Datalog rules. Writing the rules for the simple relationships is easier, as seen in Listing 3. However, it becomes difficult to handle when expressing complex relations.

```
1 WITH("harbor_username",?x) <- [VARS("HARBOR_USERNAME", ?x)],
2 WITH("harbor_password",?x) <- [SECRETS("HARBOR_PASSWORD", ?x)],
3 WITH("docker_hub_username_write",?x) <- [SECRETS("
    DOCKER_HUB_USERNAME_WRITE", ?x)],
4 WITH("latest",true) <- [GITHUB("ref_name", "master")],
5 WITH("latest",true) <- [INPUTS("ref","master")]
```

Listing 3. Compiling Datalog Example.

For instance, to write a rule for "build-allure-report-action-image", not only considering "steps" (which is a complex relation) is needed but also "name", "runs-on", and "timeout-minutes". Depending on the needs and requirements, the file's size may increase. This results in more complex relations and rules. Having complex relations makes it easy to make mistakes while writing complex rules.

```
jobs:
    build-allure-report-action-image:
        name: Allure-report action image
        runs-on: eks-runner
        timeout-minutes: 10
        steps:
            - name: Checkout ${{ github.repository }}
              uses: actions/checkout@v4

            - name: Build Image
              id: build-image
              uses: actions/build-container-image@master
              with:
                images: report-github-action
                harbor_username: ${{ vars.HARBOR_USERNAME }}
                harbor_password: ${{ secrets.HARBOR_PASSWORD }}
                docker_hub_username_write: ${{ secrets.DOCKER_HUB_USERNAME_WRITE }}
                docker_hub_token_write: ${{ secrets.DOCKER_HUB_TOKEN_WRITE }}
                dockerfile: actions/report/Dockerfile
                context: actions/allure-report/
                latest: ${{ github.ref_name == 'master' || (inputs && inputs.ref == 'master') }}
```

Figure 5. Pipedrive Jobs Example.

```
let f1: SkolemFunctionCall = |args| {
    let x_variable_value = args.get("x").unwrap();
    let y_variable_value = args.get("y").unwrap();

    return format!("{}={}", x_variable_value, y_variable_value)
   .into()
};

  let f2: SkolemFunctionCall = |args| {
    let x_variable_value = args.get("x").unwrap();
    let y_variable_value = args.get("y").unwrap();
    let z_variable_value = args.get("z").unwrap();
    let t_variable_value = args.get("t").unwrap();

    return format!("{}-{}/{}.{}", x_variable_value,
   y_variable_value, z_variable_value, t_variable_value).into()
};
```

Listing 4. Functions Example.

Some "jobs" data needed additional configurations or modifications while writing rules. The primary example is the name of the "step": "Checkout &{{github.repository}}" in line 7, Figure 6. This "step" name is created by concatenating the word "Checkout" and the name of a GitHub repository. These kinds of values are common in "jobs.". To address this, custom functions that take a different number of arguments as input and do the required modifications or formatting were implemented. Listing 4 shows such two custom functions. The function in lines 1-6 takes two arguments, while the second one in lines 8-15 takes four. Additionally, these functions format their arguments differently based on the given arguments. As seen from Lines 1 and 8, they are Skolem functions. Skolemization is a transformation that removes all existential quantifiers from logical formulas. Certain Skolem functions are permissible to occur in the head of a Datalog rule without making rules not to terminate. Most of the explored workflow files required similar templating, therefore needing custom functions. Another key requirement was importing other files.

```
1 micro_runtime.insert("WITH", vec!["images".into(), "report-
    github-action".into()]);
2 micro_runtime.insert("WITH", vec!["dockerfile".into(), "actions
    /report/Dockerfile".into()]);
3 micro_runtime.insert("WITH", vec!["context".into(), "actions/
    report/".into()]);
4 micro_runtime.insert("SECRETS", vec!["DOCKER_HUB_USERNAME_WRITE
    ".into(), "${{ secrets.DOCKER_HUB_USERNAME_WRITE }}".into()
    ]);
5 micro_runtime.insert("SECRETS", vec!["DOCKER_HUB_TOKEN_WRITE".
    into(), "${{ secrets.DOCKER_HUB_TOKEN_WRITE }}".into()]);
6
7 };
```

Listing 5. Testing.

Listing 5 shows how we tested the program with given values. As seen from Listing 5, we first insert the values to the corresponding atoms. After that, we decide on the key to which we want to build our query. For instance, if we compile **WITH** rules in this code block, the output will be only the corresponding rules. Of course, this approach could be

improved to get all the rules at once.

Overall, compiling Datalog to GHA is feasible, and there are two main takeaways from these experiments. Firstly, we found a common set of rules on what a fact and a rule could be. Secondly, extending the Datalog with arbitrary Skolem functions is needed. It could be limited to only string templating instead.

4 Compiling GHAs to Datalog

In this section, we investigate how to compile GHAs to Datalog. Therefore, here we talk about how we processed and experimented with our data and implemented our program.

4.1 Data Format

This section covers the data that we have used in our work.

Each service has workflow files that satisfy an organization's needs and requirements. These files are in YAML format, meaning data can be considered a set of key-value pairs. Each service or repository can have one or more workflow files. Usually, varying workflows are configured for different needs in a service. For instance, different workflow files can be created for functional testing, integration testing, building, turning auto-scaling on and off, deploying to other regions or live, and rollback.

Figure 6 shows the example workflow file used in Pipedrive. This file is used to restart the service when needed. As seen from line 30, this file uses another workflow file. This is common and makes these files reusable. Since this file is a YAML file, it has a specific format that defines the relationship between parents and children. For instance, this file's first key-value or parent-child pair is "name: Restart Service" (line 29). Here, "name" is a key, and "Restart Service" is a value in our data structure. In our solution, we added keys and values as nodes of the graph and then created an edge between them. These keys and values in our data represent different nodes or vertexes; each pair means an edge between them. With graph data structure, all relationships or paths from a starting node to all its child nodes can be easily defined. This will make it easier to traverse or search through the data and find all the simple paths we are trying to get in the experiment.

```yaml
name: Restart Service
run-name: Restart Service ${{ inputs.dispatch_id }}

on:
  workflow_dispatch:
    inputs:
      env:
        description: Name of the target env
        type: choice
        default: test
        options:
          - test
          - live
        required: true
      region:
        description: Name of the target region (management, us-east-1, ...)
        type: string
        required: true
      deployment:
        description: Name of the Kubernetes deployment to restart
        type: string
        required: false
      dispatch_id:
        description: An unique ID for for the workflow dispatch
        type: string
        required: false
jobs:
  restart:
    name: Restart ${{ inputs.deployment }}
    uses: .github/workflows/reusable_cicd-restart.yml@v1
    with:
      env: ${{ inputs.env }}
      region: ${{ inputs.region }}
      deployment: ${{ inputs.deployment }}
    secrets: inherit
```

Figure 6. Pipedrive GHA Workflow File.

4.2 Data Processing and Experiments

This section covers how the data in the files, such as in Figure 6, are processed and used in the work. Additionally, the experiments and issues that occurred are discussed.

The initial approach was to read the provided file from the user and store them in a corresponding data structure, a directed graph. For this purpose, we implemented a custom function to read the data and convert it to a graph using the DFS. In the given file, there might be a call to other files, which the program detects and integrates all of them into the data. While iterating over the data, the nodes' positions were stored to determine which were leaf nodes. All simple paths from the leaf to the root nodes were found during the experiment. Finding simple paths helped to exclude the unnecessary paths. It also helped to decrease the size of paths. Figure 7 represents a small part of the

Figure 7. Simple Paths.

output of the simple path for the workflow file in Figure 6. The simple paths in Figure 7 are the unique paths from a starting node to its leaf nodes. For simplicity, when we find the simple paths, we start from leaf nodes as a starting node.

As seen from Figure 6, over time, the size of a workflow file can get larger and more complex. This example file uses another workflow file in line 30. This means that this second workflow should compile and run, then continue to the following stages. Having one or more workflows integrated makes the work more difficult. After reading each file, we added or mapped the "jobs" from the file (line 30) to the first workflow file. While testing with different files, this step revealed that some jobs might have duplicate keys while mapping them from the other files. We cannot map the new jobs if they have the same key. Instead, for the same key, we still had the previous value. This happened because the data type we mapped was "map". To address this issue, the name of the job key in the first file was updated by concatenating some random numbers. This solution also required us to update the values if this job is "needed" or "called" in other steps. As a result of these operations, the initial data grew considerably, depending on the number of workflows each file calls or uses.

```
impl From<IntermediateProgramRepresentation> for GroundProgram {
    fn from(value: IntermediateProgramRepresentation) -> Self {
        return GroundProgram { inner: value.inner };
    }
```

Figure 8. Ground Program.

Problems Encountered

The main issues while mapping the data from different files:

1. Different files may use or call the same files.
2. Jobs from different files can have the same keys.

Unique workflow paths were stored to address the first issue and then kept iterating over them. Non-unique paths were excluded not to read and map the "jobs" of these files more than once. To do so, we switched from using vectors to hash sets. Vectors in Rust language represent arrays. Hash sets were used since they are sets, meaning they will store unique elements. Otherwise, it would be a waste of time since it would unnecessarily increase the data size and cause inconsistencies. After having the final data, the next step is to convert them to Datalog rules.

4.3 Datalog Program

Since the data in all files were mapped into one workflow, there was no longer a need to worry about "file" dependencies. Therefore, all needed was to implement the program as a next step. This section will explain how GHA is written into Datalog rules using the Datalog program.

The "GroundProgram " represents compiled GitHub Actions, presented in Figure 8. The data passed to the program is an array of simple paths like those in Figure 7. Then, the crucial nodes and values were extracted. We first extract "name", "inputs", "outputs", "steps", "with", "id", and "uses" nodes and their corresponding values from the data. Here, while working with "steps" nodes' values, "input" and "step" dependencies were considered there as well. After getting and formatting these nodes and the values into the desired shape, "steps" are converted into Datalog rules.

After these procedures, the following steps were applied.

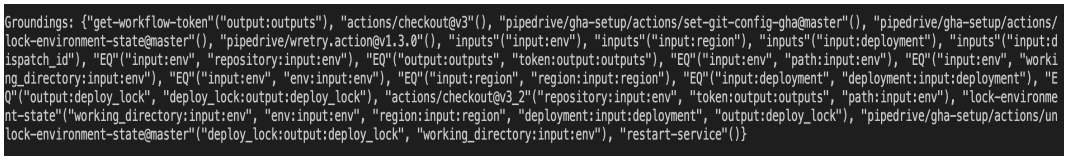

Figure 9. Actual Solution.

```
Time elapsed for running program is: 300.873759ms

Time elapsed for running ground program solver is: 384.853

Number of problems: 0
```

Figure 10. Duration.

1. *run_program*()
2. passing data to *GroundProgramSolver*
3. *compute_error_free_solution*()
4. *solve*()

The first function, *run_program*(), was used to get all ground rules. Secondly, these rules were passed to a ground program called *GroundProgramSolver*. A ground program is a program that consists of ground atoms, meaning no variables are involved. The third function was used to get the "perfect" solution, which assumes that all of its rules and facts are true, and lastly, the *solve*() function was run to get the actual solution. The "actual" solution that was achieved is shown in Figure 9.

Figure 10 shows how long it takes to compute these methods for the file in Figure 6. The time measurements there are in milliseconds. According to Figure 10, computing a "perfect" solution takes around three seconds, while running a ground program solver takes around four seconds. It is essential to mention that these times differ for each file, meaning the bigger the size of the data, the more time it takes to compute. After that, we evaluate the issues and inform the user of any problems. This information will help users to know where the issues are.

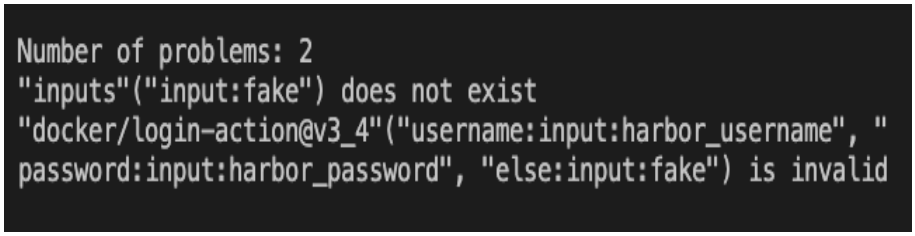

Figure 11. Compiler Output with Faulty Files.

4.4 Testing the Program

Different workflow files were used to test the program. If the file was in the correct format and there were no dependency issues, the number of problems detected was zero. This means that the program worked as expected and that GitHub Actions are consistent since there are no issues. As seen, the program acts like a *linter*, a term for a code analysis tool used to inform users about possible programming and formatting errors in the file or program.

However, if the provided file from the user needs to be in the correct format or there are dependency issues in the files, the program will inform users that there is something wrong. As seen in Figure 11, there are two problems with the file that the user provided. The first issue is that there is a missing input in the file. The second part of the output shows where this "fake" input has been used. This helps the user find the error's place easily to get fixed.

5 Conclusions

This chapter covers the results of the experiments done.

Conclusion about using Datalog to write GHA: After doing many experiments while using Datalog to write GHA, the key conclusion is that there are some requirements or "rules" to achieve it. Firstly, there is a need to decide which data are facts and which are rules. Secondly, arbitrary Skolem functions are required to extend the Datalog program.

Conclusion about linting GHA by compiling them to Datalog: Having many workflow files and integrating them results in the possibility of making mistakes while writing them or having dependency issues among files during run time. As shown in Figure 11, it is hard to be sure that all inputs (or other data) used exist in the files. These problems will occur in the actual run time unless GHA is consistent. Additionally, since in the experiment, the data of all related files are mapped into one workflow file, there are no "file" dependencies to be worried about. Overall, the main result is that compiling GHA to Datalog has the immediate benefit of making dependency resolution.

References

[1] Kadaskar H. (2024), Unleashing The Power of DevOps in Software Development. International Journal of Scientific Research in Modern Science and Technology. 3. 01-07. 10.59828/ijsrmst.v3i3.185.

[2] What is CI/CD?, https://www.synopsys.com/glossary/what-is-cicd.html (22.05.2024).

[3] Kaledio P., Lucas D., Agile DevOps Practices: Implement agile and DevOps methodologies to streamline development, testing, and deployment processes, https://www.researchgate.net/publication/378342213_Agile_DevOps_Practices _Implement_agile_and_DevOps_methodologies_to_streamline_development_testing _and_deployment_processes.

[4] How To Improve DevOps Feedback Loop,

https://www.browserstack.com/guide/devops-feedback-loop (22.05.2024).

[5] Git Tutorials, https://www.atlassian.com/git/tutorials (22.05.2024).

[6] What is Git?, https://www.simplilearn.com/tutorials/git-tutorial/what-is-git (22.05.2024).

[7] Understanding GitHub Actions, https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions (22.05.2024).

[8] T. Kinsman, M. Wessel, M. A. Gerosa and C. Treude, "How Do Software Developers Use GitHub Actions to Automate Their Workflows?," 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 2021, pp. 420-431, doi: 10.1109/MSR52588.2021.00054.

[9] T. Chen, Y. Zhang, S. Chen, T. Wang and Y. Wu, "Let's Supercharge the Workflows: An Empirical Study of GitHub Actions," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), Hainan, China, 2021, pp. 01-10, doi: 10.1109/QRS-C55045.2021.00163.

[10] Logic Programming,

https://cs.colby.edu/courses/F19/cs333/notes/10.LogicProgramming.pdf (22.05.2024).

[11] Ceri S., Gottlob G., Tanca L. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). Knowledge and Data Engineering, IEEE Transactions on 1 (04 1989), 146 – 166. https://doi.org/10.1109/69.43410.

[12] Koutris P, Lecture 8: Introduction to Datalog,

https://pages.cs.wisc.edu/~paris/lecture-notes/lecture8.pdf (22.05.2024).

[13] Tomy C., Barr E. T., Wang T., Mechtaev S., Modus: A Datalog Dialect for Building Container Images,

https://discovery.ucl.ac.uk/id/eprint/10159351/2/Mechtaev_Modus_AAM.pdf (22.05.2024).

[14] Fodor P., Definite Logic Programs: Models,

https://www3.cs.stonybrook.edu/~pfodor/courses/CSE595/L12_Definite_Logic_Programs_Models.pdf (22.05.2024).

[15] Fribourg L., Peixoto M., Bottom-up Evaluation of Datalog Programs with Arithmetic Constraints: The Case of 3 Recursive Rules.

[16] Fribourg L., Peixoto M., (1994). Bottom-up Evaluation of Datalog Programs with Arithmetic Constraints.. 311-325.

[17] Brass, S., Stephan, H. (2018). Pipelined Bottom-Up Evaluation of Datalog Programs: The Push Method. In: Petrenko, A., Voronkov, A. (eds) Perspectives of System Informatics. PSI 2017. Lecture Notes in Computer Science(), vol 10742. Springer, Cham. https://doi.org/10.1007/978-3-319-74313-4_4.

[18] She B., Zhou A., (1994). Bottom-up evaluation of datalog with negation. J. Comput. Sci. Technol.. 9. 229-244. 10.1007/BF02939504.

[19] Sato T., (2016). A Linear Algebraic Approach to Datalog Evaluation. Theory and Practice of Logic Programming. 17. 10.1017/S1471068417000023.

[20] Ullman J. D., A bottom-up beats top-down for Datalog,

https://dl.acm.org/doi/10.1145/73721.73736.

[21] Ramakrishnan R., Magic templates: a spellbinding approach to logic programs, The Journal of Logic Programming, Volume 11, Issues 3–4, 1991, Pages 189-216, ISSN 0743-1066, https://doi.org/10.1016/0743-1066(91)90026-L.

[22] Introduction to Graphs - Data Structure and Algorithm Tutorials - GeeksforGeeks, https://origin.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/?ref=lbp (22.05.2024).

Appendix

I. Acronyms

- CI/CD - Continuous Integration and Continuous Delivery
- CRM - Customer Relationship Management
- GHA - GitHub Actions

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Gulnar Mammadli**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Compiling GitHub Actions Into Datalog,

   supervised by Bruno Rucy Carneiro Alves de Lima and co-supervised by Mykhailo Dorokhov

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Gulnar Mammadli
23/05/2024