Programming Paradigms

Logic Programming

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Overview

- Introduction
- Prolog
- Datalog and CodeQL

Logic Programming

- **Declarative style of programming**
- **Based on logical deduction**
- **Program states ...**
  - What to compute
  - Not how to compute it
- **Implementations of logic PLs: Based on automated theorem proving**

Example: Sorting

- **Goal: Algorithm for sorting a list**
- **Imperative PLs**
  - Describe step-by-step how to rearrange elements of a list
- **Logic PLs**
  - Provide a constructive proof:
    For every list, there exists a sorted list composed of the same elements

Core Concepts

- **Programmer states axioms**
  - Typically as Horn clauses:

    $$H \leftarrow B_1, B_2, ..., B_3$$

  - Means: If $B_1, B_2, ..., B_3$ are true, then H is true
- **User states a theorem, i.e., the goal**
- **PL implementation tries to find a proof**
  - Axioms, inference steps, and choices of values for variables that prove the theorem

History

- **Popular for AI (artificial intelligence) programming in the 1970s and 1980s**
  - Idea: Declarative representation of knowledge
  - AI clearly has taken another path
- **Prolog language: Since 1972**
  - Position #30 in Tiobe PL popularity index
- **Datalog and CodeQL**
  - PLs for querying deductive databases
  - Applications, e.g., in program analysis

Overview

- Introduction
- Prolog ←
- Datalog and CodeQL

Example

```
has_exam(X) :- is_course(X), gives_grade(X).
is_course(pp).
gives_grade(pp).

?- has_exam(pp).
```

Example

Means "implication"

```
has_exam(X) :- is_course(X), gives_grade(X).
is_course(pp).
gives_grade(pp).

?- has_exam(pp).
```

Means "and"

Example

Means "implication"

```
has_exam(X) :- is_course(X), gives_grade(X).
is_course(pp).
gives_grade(pp).
```

Means "and"

```
?- has_exam(pp).
```

Evaluates to "true"

Clauses

- **Program runs in the context of a database of clauses assumed to be true**
- **General form: `<term>* :- <term>*`**
  - Both sides given: Rule
  - Only left side given: Fact
  - Only right side given: Goal (or query)
    - Usually written with `?-` instead of `:-`

Terms

- **A term is one of these three:**
  - Constant
    - An atom (must start with lower-case letter)
    - A number or a string
  - Variable (must starts with upper-case letter)
  - Structure: Logical predicate of the form

```
<functor>(<arg1>, ...  <argN>)
```

Example (Again)

```
has_exam(X) :- is_course(X), gives_grade(X).
is_course(pp).
gives_grade(pp).

?- has_exam(pp).
```

Example (Again)

Four clauses

One rule

```
has_exam(X) :- is_course(X), gives_grade(X).
is_course(pp).
gives_grade(pp).

?- has_exam(pp).
```

Two facts

Goal

Example (Again)

Variable

```
has_exam(X) :- is_course(X), gives_grade(X).
is_course(pp).
gives_grade(pp).

?- has_exam(pp).
```

Structure

Constant

Quiz: Prolog Syntax

How many occurrences of constants, variables, and structures are there?

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).

?- snow(C).
```

Quiz: Prolog Syntax

How many occurrences of constants, variables, and structures are there?

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).

?- snow(C).
```

3x constant

4x variable

6x structure

Answering Queries

- **How to answer a query (i.e., satisfy a goal)?**
- **Two key ideas**
  - Resolution: Replace terms based on already known clauses
  - Unification: “Pattern matching” to determine two terms to be the same

Resolution Principle

- **Given: Clauses C_1 and C_2**
- **If head of C_1 matches a term t in the body of C_2:**
  Can replace t with body of C_1

Example

takes (anna, theo3).
takes (anna, pp). (*)
takes (paul, ase).
takes (paul, pp).
classmates (X, Y) :- takes (X, Z), takes (Y, Z).

Let X = anna and Z = pp

classmates (anna, Y) :- takes (anna, pp), takes (Y, pp)

takes (anna, pp) matches (*)

new rule by replacing it w/ empty body of (*)

classmates (anna, Y) :- takes (Y, pp)

Unification Rules

- **A constant unifies only with itself**
- **Two structures unify if and only if**
  - Same functor
  - Same arity
  - Arguments unify recursively
- **A variable unifies with anything**
  - .. with a value: Variable is instantiated
  - .. with another variable: Both take same value

Equality

- **Equality is defined via unifiability:**
  Goal `A = B` succeeds if and only if `A` and `B` can be unified

Examples

```
?- a = a.
true.

?- a = b.
false.

?- foo(a, b) = foo(a, b).
true.

?- X = a.
X = a.

?- foo(a, b) = foo(X, b).
X = a.
```

Lists

- **Own syntax, as commonly used**
  - Empty list: `[]`
  - List with elements: `[a, b, c]`
  - Delimiter before tail of list: `|`
    - `[a, b | [b, c]]` means `[a, b, b, c]`
    - `[a, b, c | []]` means `[a, b, c]`

Examples

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

```
sorted([]).
sorted([_]).
sorted([A, B | T]) :- A =< B, sorted([B | T]).
```

Examples

Variable that isn't needed anywhere

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
```

Built-in predicate that operates on numbers

```
sorted([]).
sorted([_]).
sorted([A, B | T]) :- A =< B, sorted([B | T]).
```

Predicates vs. Functions

- **Functions distinguish between inputs and outputs**
  - In imperative or functional PL: Apply function to arguments to generate a result
- **Predicates don't distinguish between inputs and outputs**
  - In logic PL: Search values for which a predicate is true

Example

```
append([], A, A)
append([H|T], A, [H|L]) :- append(T, A, L)
% appending first & second arg. yields third arg.

?- append([a,b,c], [d,e], L).
L = [a,b,c,d,e].

?- append(X, [d,e], [a,b,c,d,e]).
X = [a,b,c].

?- append([a,b,c], Y, [a,b,c,d,e]).
Y = [d,e].
```

Quiz: Prolog Programs

What do the three queries evaluate to?

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).

?- member([a], [a, b]).
?- member(d, [a, b | [c, d]]).
?- member(X, [a, b | [c]]).
```

Quiz: Prolog Programs

What do the three queries evaluate to?

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).

?- member([a], [a, b]).
?- member(d, [a, b | [c, d]]).
?- member(X, [a, b | [c]]).
```

false, because [a] is not in [a, b]

true

X = a (i.e., first solution found)

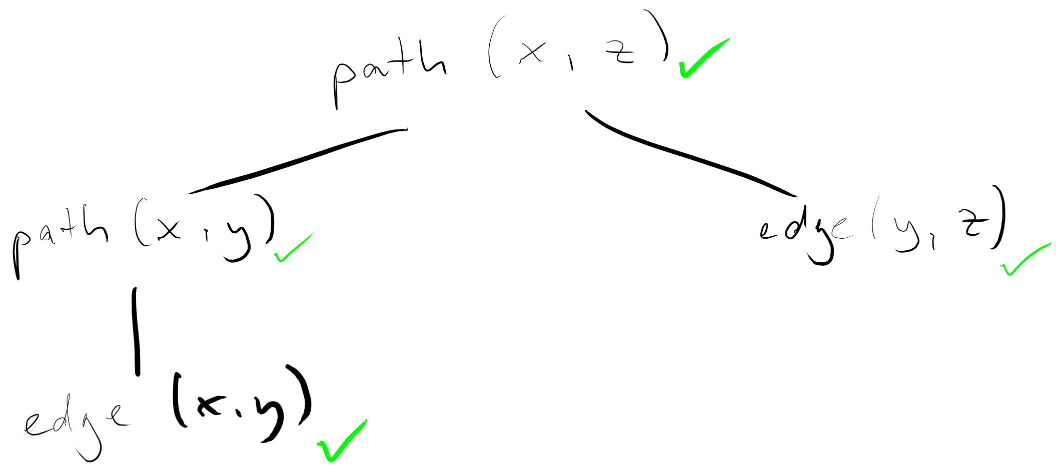
Searching for a Proof

- **Given a query/goal, how to answer it?**
  - Want: Sequence of resolution steps that build the goal out of known clauses
  - Or: Proof that no such sequence exists
- **Proof tree**
  - Root node: Goal
  - Other nodes: Subgoals

Example

edge (x, y).
edge (y, z).
path (A, B) :- edge (A, B).
path (A, C) :- path (A, B), edge (B, C).

?- path (x, z).



Forward vs. Backward Chaining

Two options for finding a proof:

- Forward chaining
  - Start with existing clauses and attempt to derive goal
  - I.e., build proof tree bottom-up
- Backward chaining
  - Start with goal and “unresolve” it into a set of existing clauses
  - I.e., build proof tree top-down

Forward vs. Backward Chaining

Two options for finding a proof:

- Forward chaining
  - Start with existing clauses and attempt to derive goal
  - I.e., build proof tree bottom-up
- Backward chaining
  - Start with goal and "unresolve" it into a set of existing clauses
  - I.e., build proof tree top-down

Prolog uses this

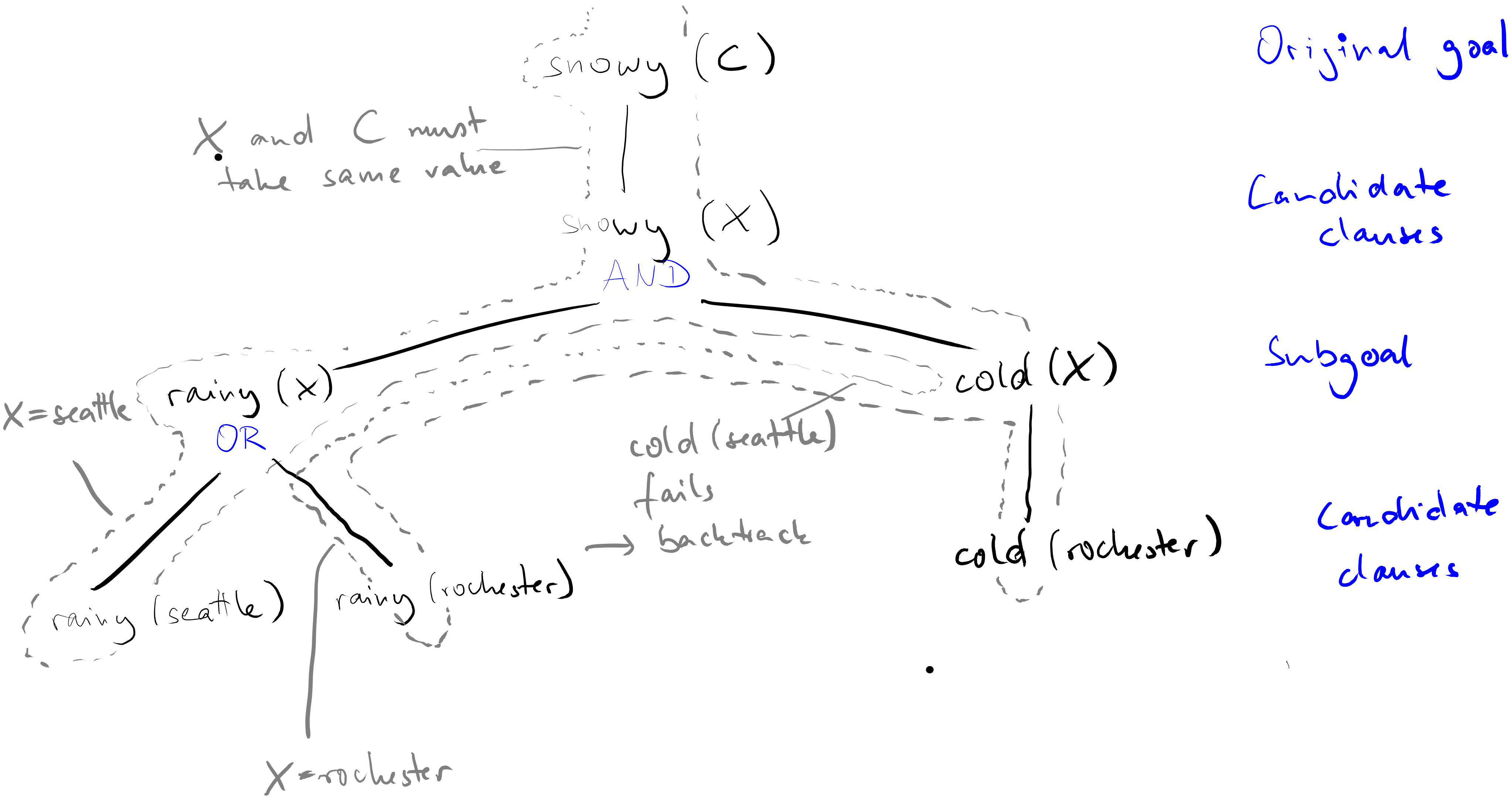
Backtracking Search

- **Prolog explores the tree depth-first, left-to-right**
  - Search for rule R whose head can be unified with current goal
  - Terms in body of R become new subgoals
- **Backtrack if a subgoal fails**

Example

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).

?- snowy(C).
```


snowy (C)
X and C must take same value
snowy (X)
AND
rainy (X)
X=seattle
OR
rainy (seattle)
rainy (rochester)
X=rochester
cold (X)
cold (seattle) fails
backtrack
cold (rochester)
Original goal
Candidate clauses
Subgoal
Candidate clauses

Overview

- Introduction
- Prolog
- Datalog and CodeQL ←

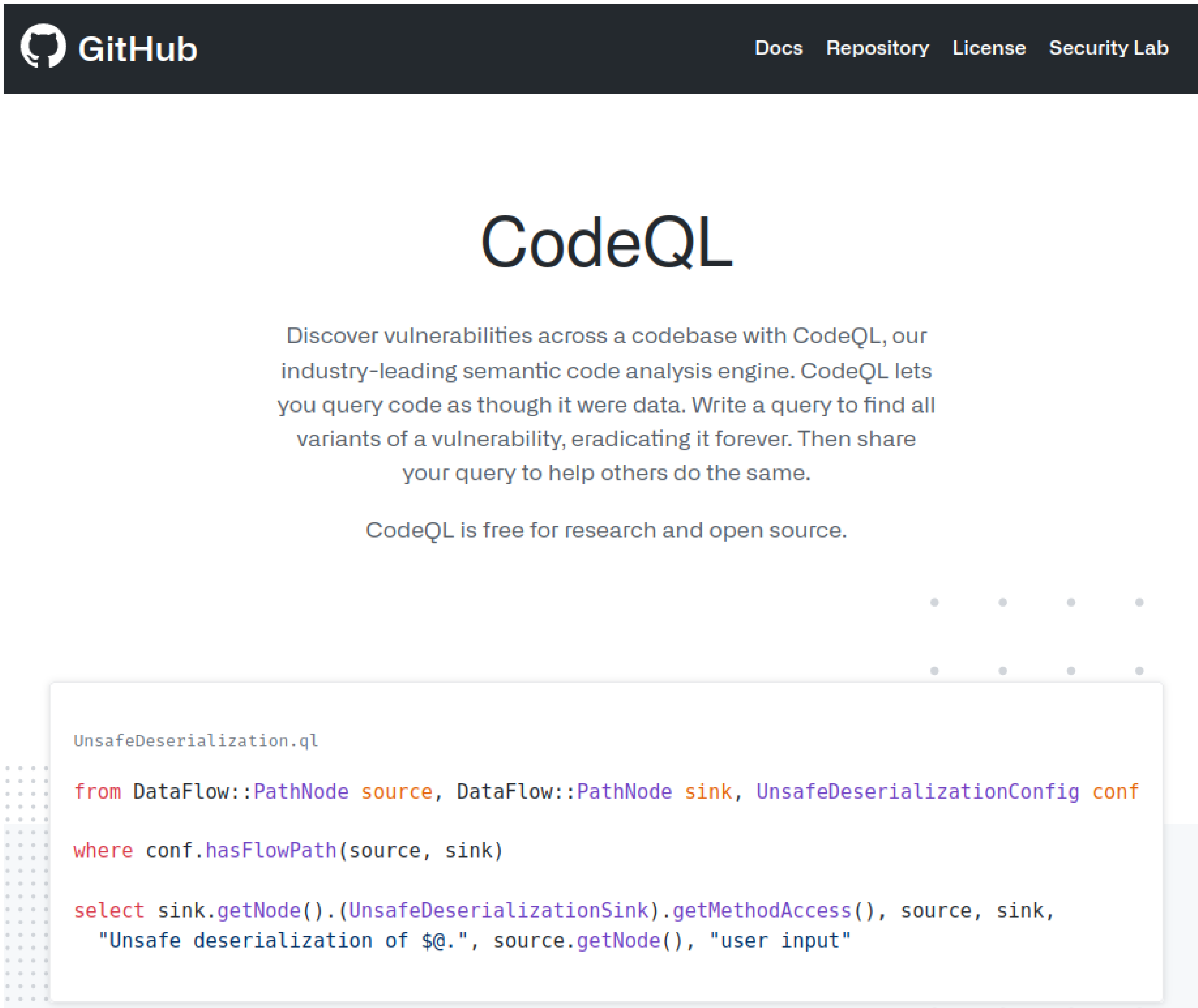
Datalog

- **Variant of Prolog**
- **Used as query language for deductive databases**
  - Set of known facts
  - Rules to derive new facts
- **E.g., used for reasoning about code**
  - Fact: `y` is assigned to `x`
  - Rule: If `y` is assigned to `x` and `y` points to object `o`, then `x` also points to object `o`

CodeQL

- **Static analysis engine by GitHub**
- **Goal: Find vulnerabilities and other bugs in the source code**
- **CodeQL language**
  - Variant of Datalog
  - One query per bug pattern
    - E.g., deserialization of unsanitized user input

CodeQL


GitHub
Docs
Repository
License
Security Lab
CodeQL
Discover vulnerabilities across a codebase with CodeQL, our industry-leading semantic code analysis engine. CodeQL lets you query code as though it were data. Write a query to find all variants of a vulnerability, eradicating it forever. Then share your query to help others do the same.
CodeQL is free for research and open source.
UnsafeDeserialization.ql
from DataFlow::PathNode source, DataFlow::PathNode sink, UnsafeDeserializationConfig conf
where conf.hasFlowPath(source, sink)
select sink.getNode().(UnsafeDeserializationSink).getMethodAccess(), source, sink,
"Unsafe deserialization of $@.", source.getNode(), "user input"

CodeQL Language

- Syntax **resembles SQL**
- But actually a **declarative logic PL**

Example:

```
from Class c
where c.declaresMethod("equals") and
      not(c.declaresMethod("hashCode")) and
      c.fromSource()
select c.getPackage(), c
```

CodeQL Language

- Syntax **resembles SQL**
- But actually a **declarative logic PL**

Example:

```
from Class c
where c.declaresMethod("equals") and
      not(c.declaresMethod("hashCode")) and
      c.fromSource()
select c.getPackage(), c
```

Find classes with an `equals` but no `hashCode` method

CodeQL Demo

[DEMO]

import of database from GitHub

view AST of some file

run a query

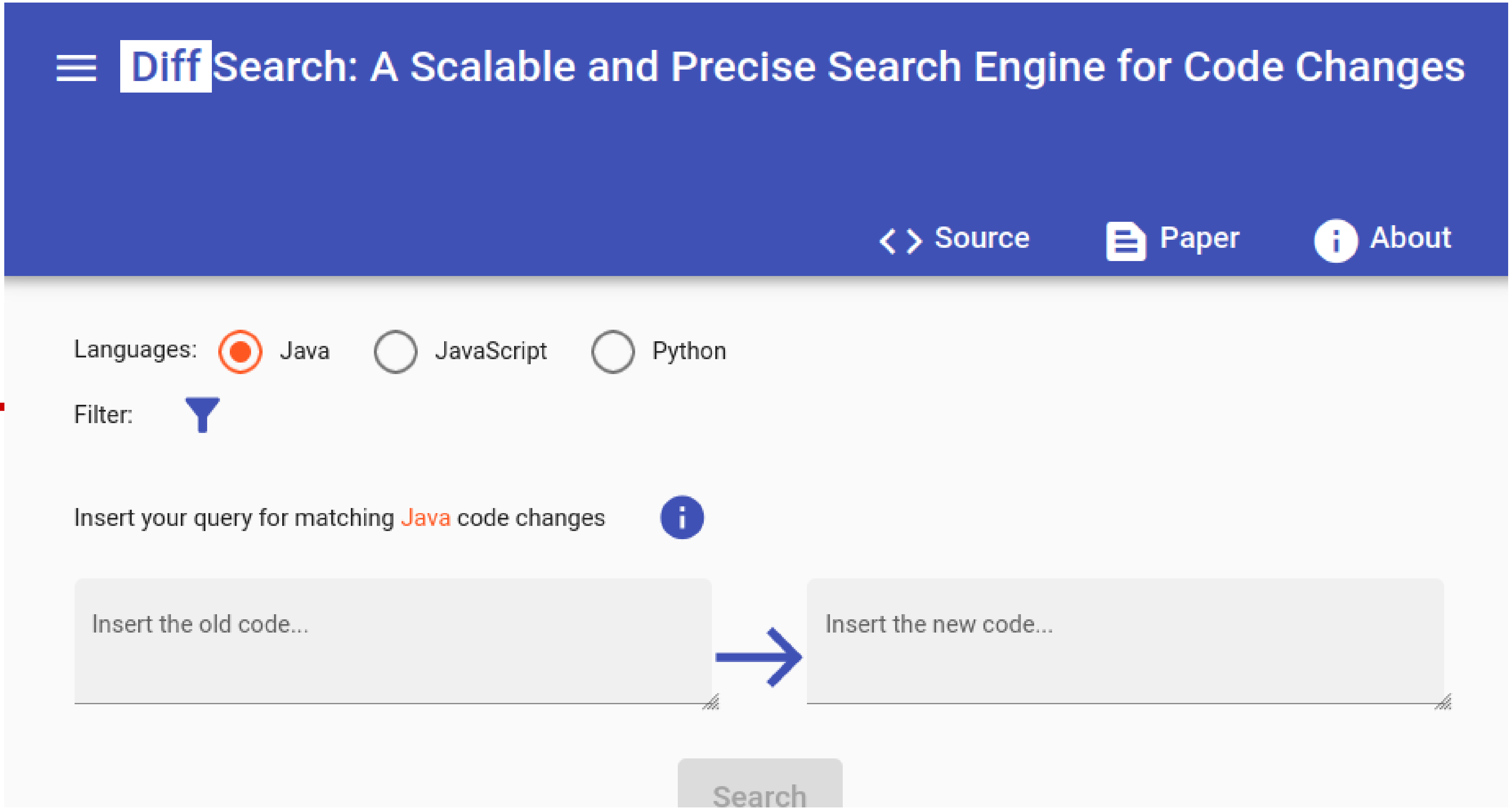
Overview

- Introduction
- Prolog
- Datalog and CodeQL ✔

Outlook & Opportunities

- **Master-level courses**
  - Program Analysis
  - Analyzing Software using Deep Learning
  - Seminar: Machine Learning for Programming
- **Do research with us**
  - Bachelor and Master theses

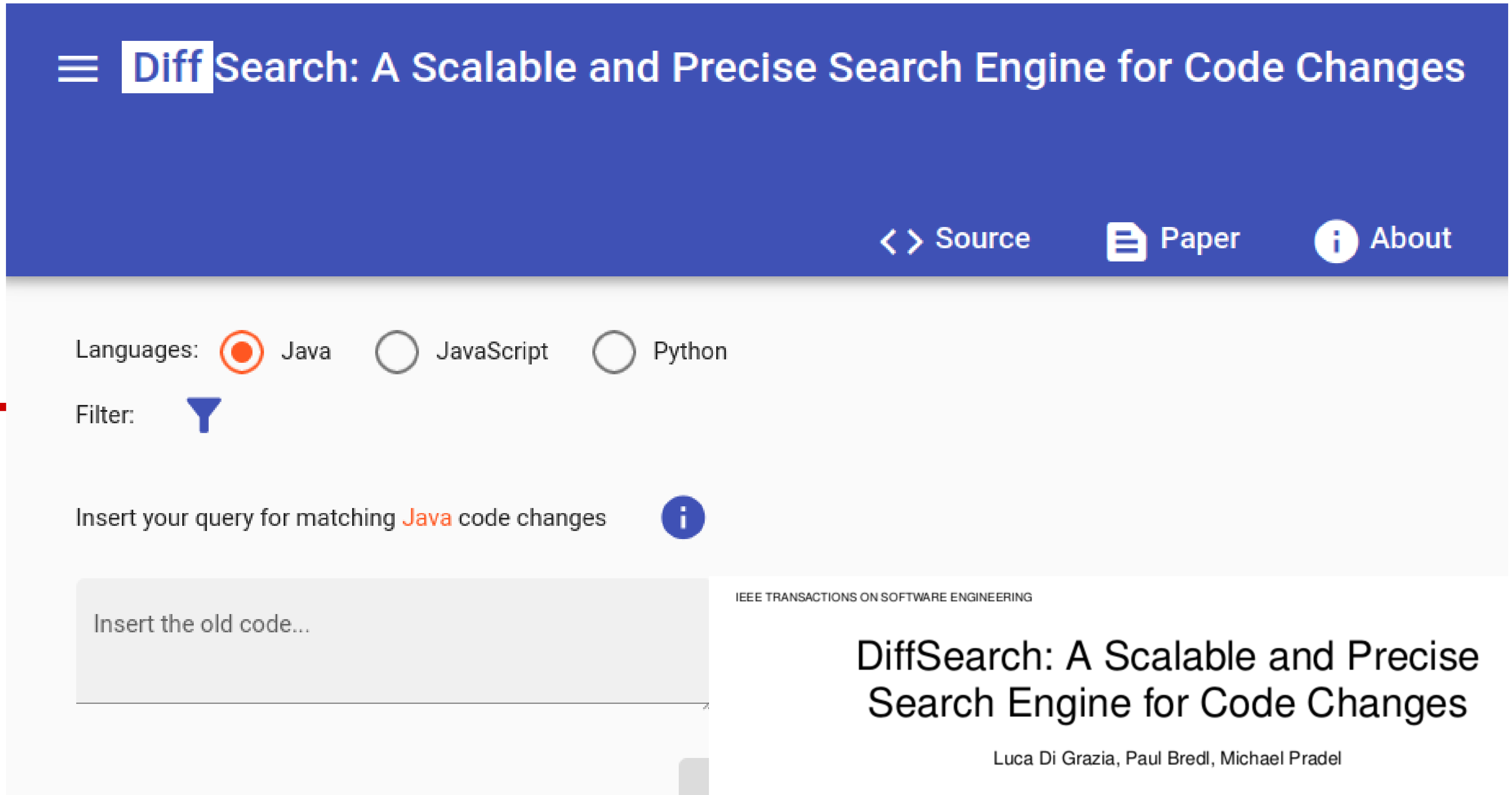
Outlook & Opportunities

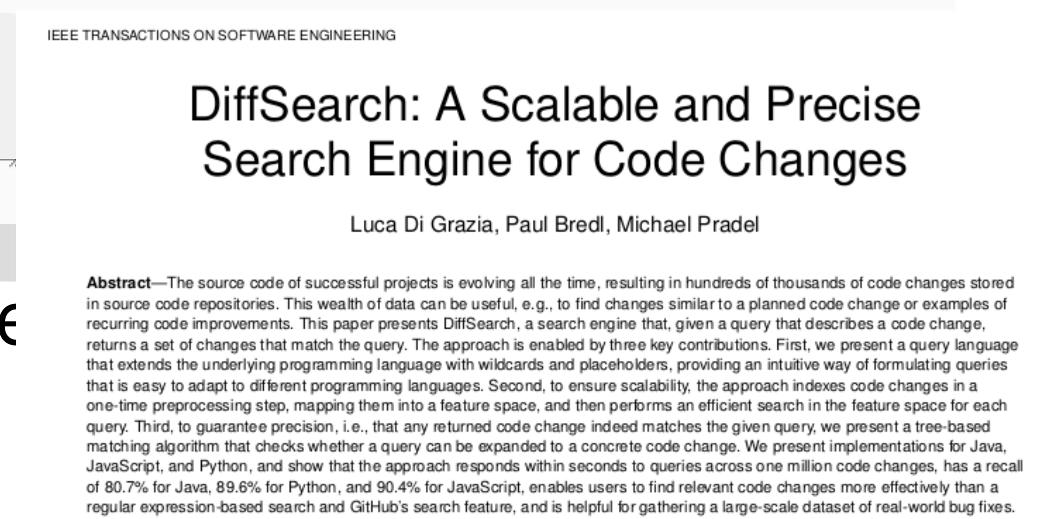


- Bachelor and Master theses

Paul Bredl (2021)

Outlook & Opportunities





DiffSearch: A Scalable and Precise Search Engine for Code Changes

Luca Di Grazia, Paul Bredl, Michael Pradel

Abstract—The source code of successful projects is evolving all the time, resulting in hundreds of thousands of code changes stored in source code repositories. This wealth of data can be useful, e.g., to find changes similar to a planned code change or examples of recurring code improvements. This paper presents DiffSearch, a search engine that, given a query that describes a code change, returns a set of changes that match the query. The approach is enabled by three key contributions. First, we present a query language that extends the underlying programming language with wildcards and placeholders, providing an intuitive way of formulating queries that is easy to adapt to different programming languages. Second, to ensure scalability, the approach indexes code changes in a one-time preprocessing step, mapping them into a feature space, and then performs an efficient search in the feature space for each query. Third, to guarantee precision, i.e., that any returned code change indeed matches the given query, we present a tree-based matching algorithm that checks whether a query can be expanded to a concrete code change. We present implementations for Java, JavaScript, and Python, and show that the approach responds within seconds to queries across one million code changes, has a recall of 80.7% for Java, 89.6% for Python, and 90.4% for JavaScript, enables users to find relevant code changes more effectively than a regular expression-based search and GitHub's search feature, and is helpful for gathering a large-scale dataset of real-world bug fixes.

- Bachelor and Maste

Paul Bredl (2021)

Outlook & Opportunities

Beware of the Unexpected: Bimodal Taint Analysis

Yiu Wai Chow
University of Stuttgart
Stuttgart, Germany
victorcwai@gmail.com

Max Schäfer
GitHub
Oxford, UK
max-schaefer@github.com

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

ABSTRACT

Static analysis is a powerful tool for detecting security vulnerabilities and other programming problems. Global taint tracking, in particular, can spot vulnerabilities arising from complicated data flow across multiple functions. However, precisely identifying which flows are problematic is challenging, and sometimes depends on factors beyond the reach of pure program analysis, such as con-

17–21, 2023, Seattle, WA, United States. ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3597926.3598050

1 INTRODUCTION

Taint analysis is a powerful technique for detecting various kinds of programming mistakes, including both security vulnerabilities and other kinds of bugs. A taint analysis tracks the flow of information

- **Do research with us**
  - Bachelor and Master theses

Yiu Wai Chow (2022)