Search-Based Programming

Bixin Li(李必信)
School of Computer Science and Engineering
Southeast University

Outline

1. Summary
2. State-of the art
3. Our research plan
4. Our initial work
5. Conclusion

1. Overview

- What is *search*-based programming?
  - A aggregated way of finding and reusing existing code
  - Search-based Programming is ***more than*** code search!
- It helps software engineer to do programming (or software development) by
  - searching code repositories
  - predicating possible code candidates
  - synthesizing code scripts
  - …

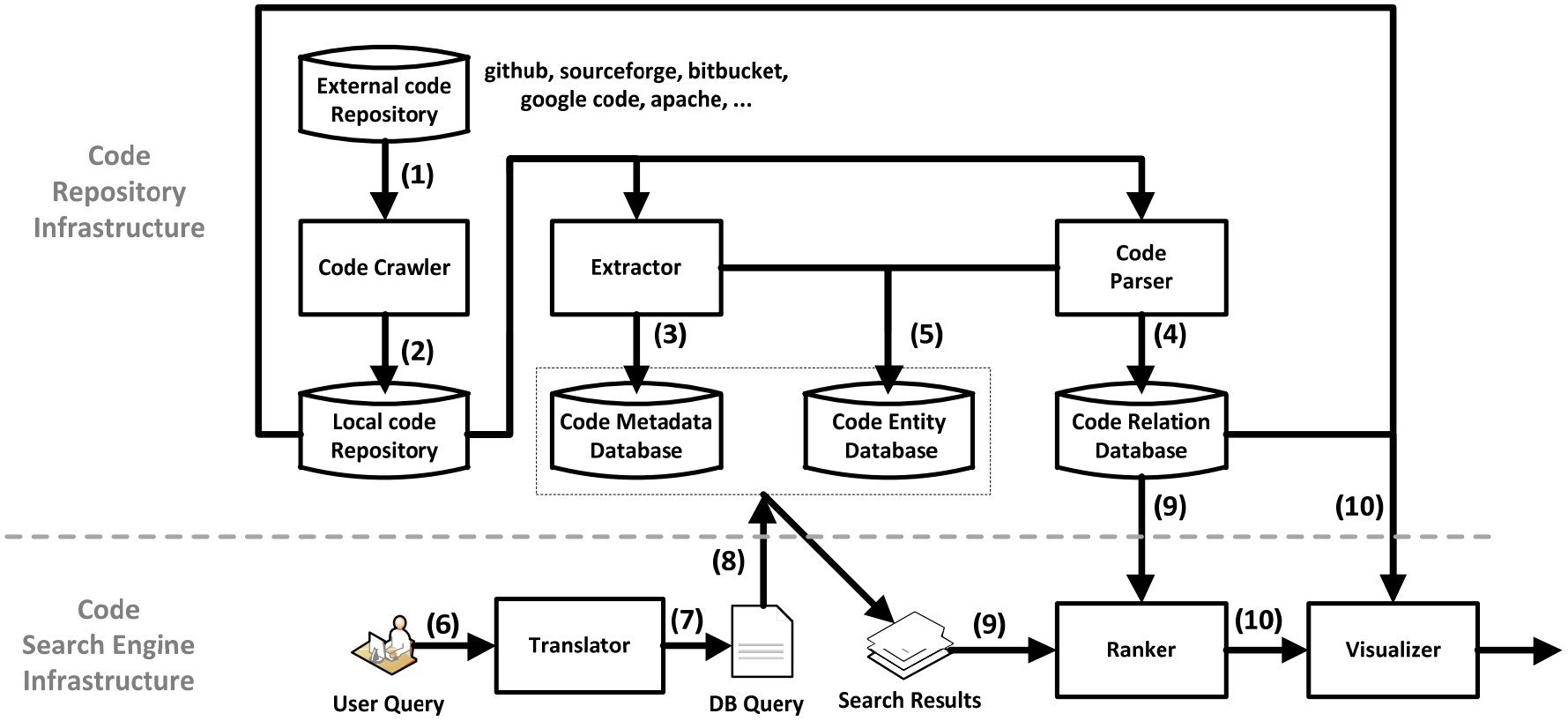
2. State of the art

- Code search
  - store and index code
  - queries
  - filter and rank results
- Code Recommendation
- Program Synthesis
  - Jungloid
  - SmartSynth
- Code completion

3. Research Plan

- Develop a Search Engine to aid the *code search*, *code rank* and *result visualization*.
- Synthesizing code scripts by studying the *lexical distinguishability*, *syntax rule usage*, *API usage analysis*.

- Code Search Engine



- 该结构主要分为两个部分：Code Repository Infrastructure和Code Search Engine Infrastructure。
  - Code Repository Infrastructure主要用于收集源代码并对其进行处理;
  - Code Search Engine Infrastructure主要对收集的代码进行搜索、排序，并输出搜索结果。

- Code Repository Infrastructure工作流程
  - Code crawler从External code repository自动搜索项目源代码
  - Code crawler自动下载搜集到的项目源代码，并自动抓取项目相关辅助信息(Metadata)，存入Local code repository
  - Extractor从Local code repository中抽取项目Metadata信息，并存入Code metadata database
  - 利用Extractor和Code Parser抽取代码实体并存入Code entity database
  - 利用Code Parser抽取代码之间的关系并存入Code relation database

- Code Search Engine Infrastructure的工作流程
  - 用户输入查询语句
  - Translator对用户查询语句进行解析转换，并生成数据库查询语句
  - 数据库查询语句对Code entity database及Code metadata database进行查询并返回结果
  - 综合查询结果与Code relation database的内容，通过Ranker对查询结果进行排序
  - 综合排序结果、Code relation database及原始代码库(Local code repository)，使用Visualizer对搜索结果进行显示

- 外部代码仓库【**External code repository**】
  - 可下载的开源代码仓库包括Github，Sourceforge，Bitbucket，Google code，Apache等，通常情况下Code crawler负责寻找类似开源代码库。
- 本地代码仓库【**Local code repository**】
  - 此仓库主要用于存储Code crawler收集的源代码(source code)及源代码相关辅助信息，例如API文档(API documents)，测试用例(test cases)，故障报告(bug reports)，版本控制信息(version control information)等。
- 代码实体数据库【**Code entity database**】
  - 根据代码搜索中设定的粒度，此仓库主要用于存储对应代码实体（例如：代码片段，方法，类，组件等）。

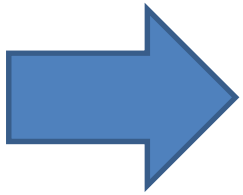
- 代码元数据数据库【Code metadata database】
  - 此仓库主要用于存储代码相关的辅助信息（例如：从版本控制系统中可以抽取项目的源代码的开发者，Commit提交日期，项目的介绍信息等）。
- 代码关系数据库【Code relation database】
  - 此仓库主要用于存储代码之间的相互关系（例如：调用图，控制流图，依赖图等）。
- 代码爬虫【Code crawler】
  - 自动收集网络中的项目源代码及相关Metadata信息，并下载至Local code repository。

- 信息收集器【Extractor】
  - 一般包含两个部分：(1) Metadata信息收集器；(2) 代码筛选器。Metadata信息收集器主要使用自然语言处理技术，从Local code repository中提取“有意义”的辅助信息；代码筛选器根据代码搜索粒度对源代码进行过滤。例如以方法为搜索对象，则代码筛选器需要过滤除方法以外的无用代码。通常情况下代码筛选器和Code parser配合使用。
- 代码解析器【Code parser】
  - 主要负责从代码中抽取关系。例如可以将代码解析成AST树并构建方法之间的调用关系。

- 查询转换器【Query Translator】
  - 主要负责将用户的查询语句转换为数据库查询语句。如何有效准确地理解用户的查询语句是主要难点。
- 对查询结构进行排序【Ranker】
  - 排序算法可采用通用算法，如PageRank算法；也可结合Extractor提取的代码关系，提高排序的精度。
- 代码搜索结果可视化【Visualizer】
  - 主要负责显示代码搜索的结果。除了显示正常的搜索结果，如何有效的帮助程序员更好的理解搜索结果至关重要。

4. Our initial work

Our Angle of View



- Understand the languages of
  - their usage
    - e.g. Popularity, Contextual dependency, …
  - their usage characteristics
- Investigate how search helps programming based on
  - the language characteristics
  - the language usage data

- 为此，我们针对language从如下三个方面开展一些初步研究：

  4.1 Lexical

  - lexical distinguishability of source code

  4.2 Syntactical

  - independent and dependent syntax usage
  - syntax rule usage evolution

  4.3 API

  - API usage analysis

4.1 On the Lexical Distinguishability of Source Code

- Natural language is robust against noise, source code of programming language has also much noise.
- **Wheat** and **Chaff** Hypothesis:
  - Units of code consists of “wheat”, important lexical features that preserve meaning, and “chaff”
  - The “wheat” is small compared to “chaff”

The meaning of many sentences survives the loss of words. Sometimes, many of them, some words in a sentence, however, cannot be lost without changing the meaning of the sentence. We call these words “wheat” and the rest “chaff”. The word “not” in the sentence “I do not like rain” is wheat and “do” is chaff.

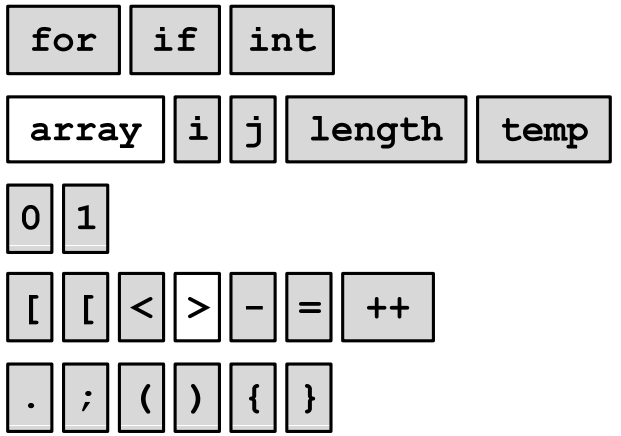
- **Methodology**
  - Map source code (method) to a set of lexical features
    - bag-of-words model
    - lexicons

Source Code

```
private static void bubbleSort(int array[]){
  int length = array.length;
  for (int i = 0; i < length; i++) {
    for (int j = 1; j > length - i; j++) {
      if (array[j-1] > array[j]) {
        int temp = array[j-1];
        array[j-1] = array[j];
        array[j] = temp;
}}}}
```

apache/core/src/test/java/org/apache/logging/log4j/core/SimplePerfTest.java

Bag-of-words



- **Methodology**
  - Find MINSET (Minimum Distinguishing Subset) to distinguishes a method lexically
    - MINSET is **wheat**

MINSET problem

Given a finite set S, and a finite collection of finite sets $\mathcal{C}$, S' is a distinguishing subset of S if and only if

(P1) $S' \subseteq S$, S' is a subset of S

(P2) $\forall C \in \mathcal{C}, S' \nsubseteq C$, S' is only a subset of S

- **Interesting results**
  - Methods are **lexically distinguishable**
    - which can be identified by their MINSETs
  - All MINSETs are **small**,
    - MINSET has 1.56 words in average (none exceeds 6 words)
    - MINSET only comprise 4% of each method
  - MINSETs are **meaningful**

- **Applications**
  - keyword-based programming
    - Using wheat to generate the **smartphone script** from a natural language description
  - Code Search Engine
    - Using wheat to rank the results
  - Code Summarizer
    - Wheat can effectively summarize code

4.2 Qualifying the Syntactic Rule Usage in Java

- **Motivation**
  - Language designers and users usually have limited knowledge on how programmers actually use a language
  - Language syntax, remain a significant barrier to novice programmers
  - Help validate or disprove the many popular “theories” of language feature usage
  - Enable guiding pedagogy

- **Methodology**
  - Use language **grammars** to characterize language features,…
  - Study over *5,000* open-source Java projects (over *150* million SLoC) on syntactic rules in JLS1~JLS4(JAVA语言规范)
    - *5,000* open-source Java projects containing over 13,000 versions
  - Perform analysis including
    - Syntax rule usage independent ly and usage over time
    - Syntax rule usage dependently

- **Interesting Results**
  - The usage of syntax rules is **Zipfian (**frequency**)**
    - 85% usage belongs to 20% of the rules
    - 65% of the rules are used < 5%, 40% only < 1%
  - Part of the rules are unpopular (popularity)
    - 16.7% of the rules are adopted in < 25% of the projects
  - Most projects use only a **subset** of syntactic rules
  - Project size impacts the adoption of syntactic rules
  - Some of the syntactic rules were losing their attraction over time
  - Most newly-introduced syntactic rules (e.g. annotation in JAVA) were adopted by programmers gradually and some have been widely used in projects

- **Interesting Results** (cont.)
  - The newly-added rules did impact the use of the existing relevant rules
  - Syntactic rules exhibit nontrivial dependency
    - 6% of rule combinations show strong dependency with > 50% probability
  - Syntactic rule usage is contextual and helps
    - identify potential syntactic sugar to simplify a language use
    - guide syntactic (rather than lexical) refactoring or code completion and suggestion.

- **Applications**
  - Syntax rule design and use restriction
  - Automatic syntactic sugaring
  - Code completion and suggestion

4.3 Qualifying the Language API Usage

- **Motivation**
  - Find hotspot and coldspot of the API
  - Help validate or disprove the many popular “theories” of API usage and API design
  - Help guiding teaching

- **Methodology**
  - Gather the **Corpus**
    - contain 5,185 open-source Java projects from Github
  - Analyze and capture dependencies
    - capture project dependencies automatically
  - Collect API usage
    - by traversing resolved ASTs with binding information
  - Calculate Metrics
    - Frequency
    - Coverage
    - Popularity

- **Interesting results**
  - API usage origin distribution

	Core	Third-Party	Self-Defined
Class Use	40.27%	15.25%	44.42%
Method invocation	26.19%	21.16%	50.38%
Field Use	5.49%	6.36%	87.97%

- **Interesting results**
  - The API usage obey the power-law; They fits the power-law distribution
    - The top 1% of the packages account for 80% of all API usage
  - Core API is **not** fully used
    - **15.3%** of the classes, **41.2%** of the methods and **41.6%** of the fields in core API are never used
    - **In 9.5%** of the packages, all belonging methods are never used
    - **In 29.2%** of the classes, all belonging methods are never used

- **Applications**
  - API design and optimization
  - API recommendation and use
  - Library recommendation
  - Language API education

5. Conclusion

- Search-based programming is a feasible way to increase the efficiency of programming by reusing existed code.
- Understanding language characteristics in search-based programming is important
- Good work can better programming languages
- Good work can better programming models