Lessons learned from implementing a language-agnostic dependency graph parser

Francesco Refolli Darius Sas Francesca Arcelli Fontana

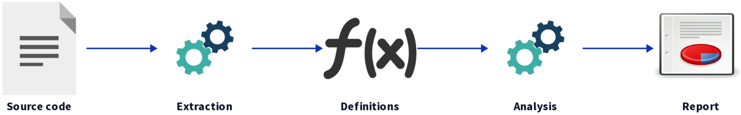
UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

ENASE 2025

Outline

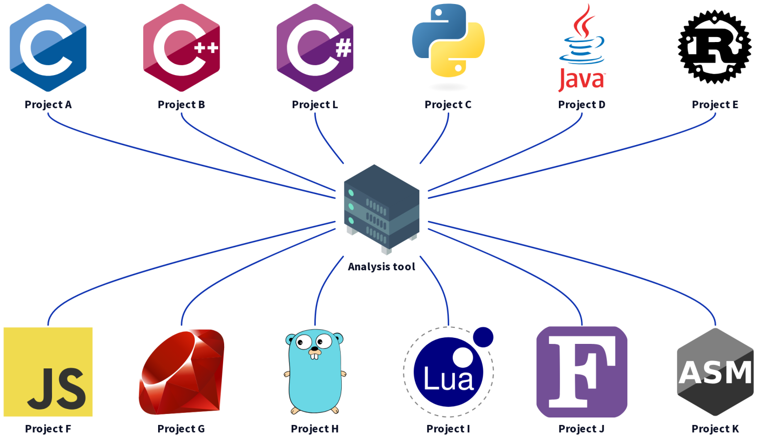
Introduction

How do software analysis tools work?



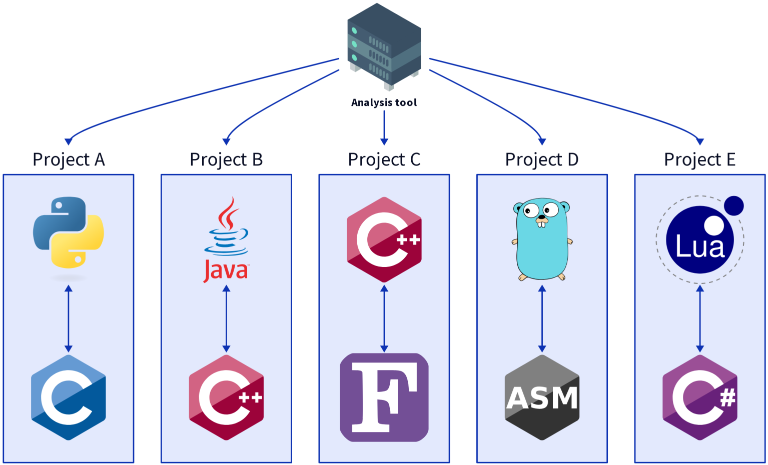
A tedious detail ...

Not every project is written in X language!

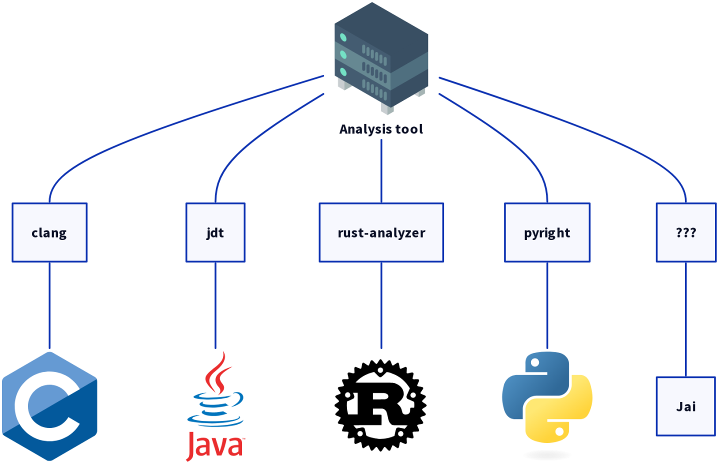


A tedious detail ...

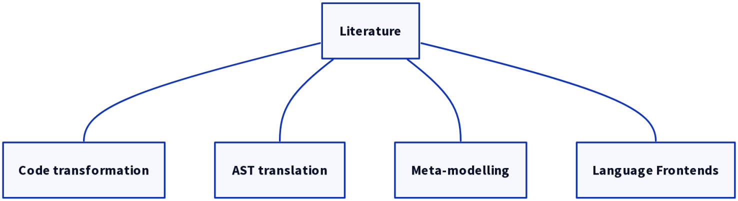
Some projects are even blends of languages!



A "naive" solution



Other techniques



What we want

A few key principles:

- An efficient intermediate representation
- Avoid source code / syntax tree translations
- Avoid language-specific dependencies
- A Maintainable, extensible and generic approach
- Easily add support for a new language
- Build it right and built it fast

The tools

Tree Sitter

- Is a parser generator
- Parses code into a Concrete Syntax Tree (CST)

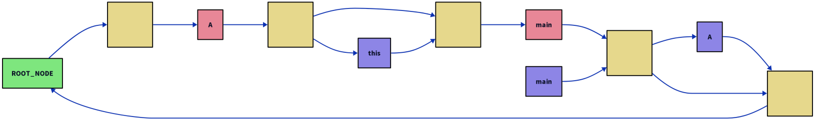
```
class A {
  public static void main() {
  }
}
```

```
program [0, 0] - [4, 0]
  class_declaration [0, 0] - [3, 1]
    name: identifier [0, 6] - [0, 7]
    body: class_body [0, 8] - [3, 1]
      method_declaration [1, 2] - [2, 3]
        modifiers [1, 2] - [1, 15]
        type: void_type [1, 16] - [1, 20]
        name: identifier [1, 21] - [1, 25]
        parameters: formal_parameters [1, 25] - [1, 27]
        body: block [1, 28] - [2, 3]
```

Stack Graphs

- Composable graph
- Represents identifiers and scopes of source code
- Enables language agnostic reference resolution

```java
class A {
  public static void main() {
  }
}
```



Tree Sitter Stack Graphs

- A "TSSG" Grammar is composed of pairs
- Each pair matches a section of the CST with some procedural code
- Definition of nodes, edges and labels

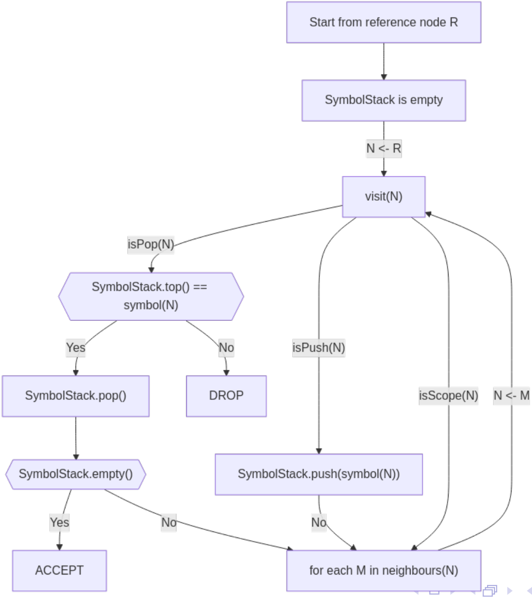
```
(field_access object: (_)@object field: (_)@field)@this {
  node @this.scope
  node @this.pop_start
  node @this.pop_end
  edge @field.push_end -> @object.push_start
  let @this.push_start = @field.push_start
  let @this.push_end = @object.push_end
  edge @this.push_end -> @this.scope
  edge @this.pop_start -> @this.push_start
  attr (@field.push_start)
    is_reference, refkind = "accessField"
}
```

Building the dependency graph

How is the Stack Graph built?

- Definition identifiers get a *pop* node
- Reference identifiers get a *push* node
- *Scope* nodes appear at various levels to allow navigation
- Chaining nodes with edges allow to resolve complex references. Examples:
  - "$java \leftarrow util \leftarrow Scanner$" defines a sequential search for its identifies.
  - "$point \leftarrow x$" employs type resolution and field access.
- Try to avoid cycles to reduce reference resolution time
- Apply *refkind/defkind* labels to capture fine-grain relationships and component types

How are references resolved?



How is the dependency graph built?

- Depth-first visit of the Stack Graph from root
- Add each definition node to the component graph
- Keep track of structural dependencies
- Decorate the graph with references resolved previously
- Serialization with JSON, GraphML ... etc

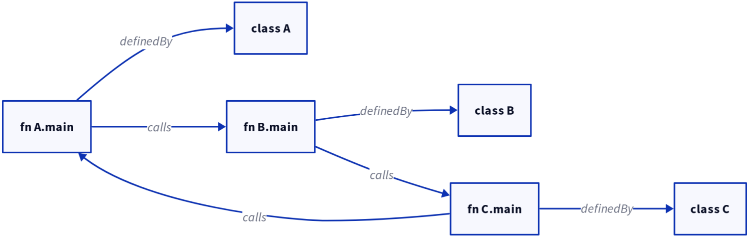
Another basic example / 1

```
class A {
  public static void main() {
    B.main();
  }
}

class B {
  public static void main() {
    C.main();
  }
}

class C {
  public static void main() {
    A.main();
  }
}
```

Another basic example / 2



Evaluation

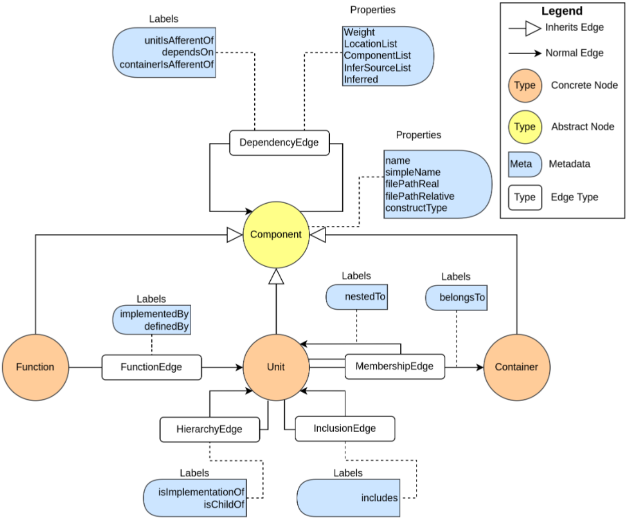
How is the prototype evaluated?

Describe assertion on files, nodes and edges

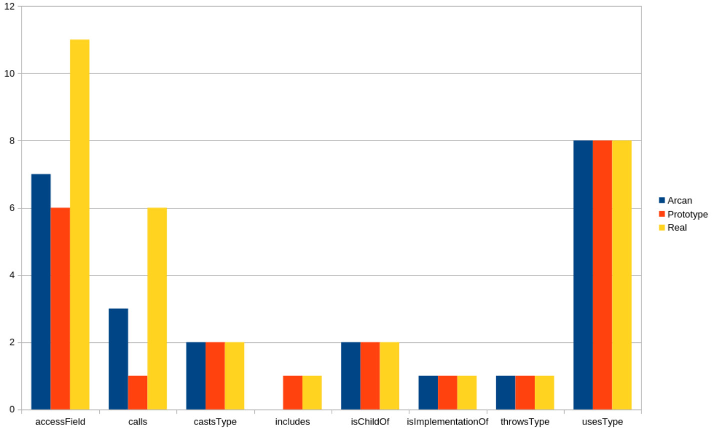
```yaml
filepaths:
  - main.java
nodes:
  - name: ClassName
    kind: class
  - name: ClassName.MethodName
    kind: function
edges:
  - source: ClassName.MethodName
    sink: ClassName
    kind: definedBy
```

- node types: class, function, enum ...
- edge types: definedBy, includes, usesType, calls ...

The tool Arcan



The Pruijt et al's benchmark



1

[1] The accuracy of dependency analysis in static architecture compliance checking, Pruijt et Al, Softw. Pract. Exp. 2017

Running on real-world scenarios

Project	Language	Version	Size (in LOC)
JUnit4	Java	4	30K
JUnit5	Java	5	100K
ANTLR	Java	4	180K
Fastjson	Java	1	50K

Project	Tool	Min	Max	Mean execution time
JUnit4	prototype	65,82	69,07	65,88
JUnit4	Arcan	13,850	17,079	14,611
JUnit5	prototype	132,87	134,75	134,44
JUnit5	Arcan	42,542	47,613	44,186
ANTLR	prototype	171,65	175,49	172,64
ANTLR	Arcan	19,222	20,140	19,691
Fastjson	prototype	N/A	N/A	N/A
Fastjson	Arcan	66,932	71,094	69,071

Lessons learned

Advantages

- Reference resolution and dependency inference are not tied to third party dependencies
- Achieves decent precision (comparable with the tool Arcan)
- Approach exportable to other languages (ex: Python, C++)
- Writing a TSSG grammar is easier than implementing a language frontend
  (but it can be harder than writing bindings to those)

Shortcomings

- Poor performance
- Generally large Stack Graphs
- Lack of support for dependencies to external libraries
- Limited resolution for complex references
- The TSSG DSL is fairly limited (can lead to labeling conflicts)
- TSSG Grammars are long and not easy to read

Conclusion

Conclusions

Summarizing:

- The presented approach works for small-to-medium projects
- Further work is needed to scale project sizes

Future work:

- Experiment with automatic graph construction
- Build abstractions upon Tree Sitter CSTs

Thank you for the attention