Property probes

Source code based exploration of program analysis results

Anton Risberg Alaküla, anton.risberg_alakula@cs.lth.se
PhD student @ department of Computer Science, Lund University

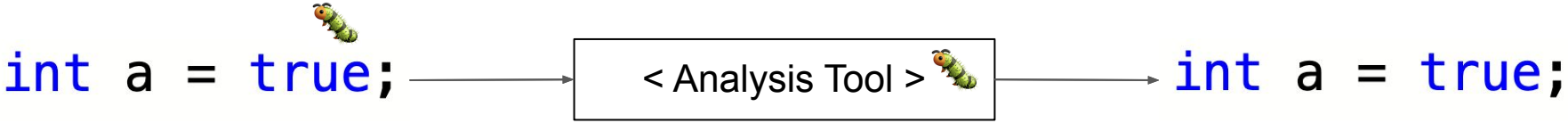
Program Analysis

Program Analysis

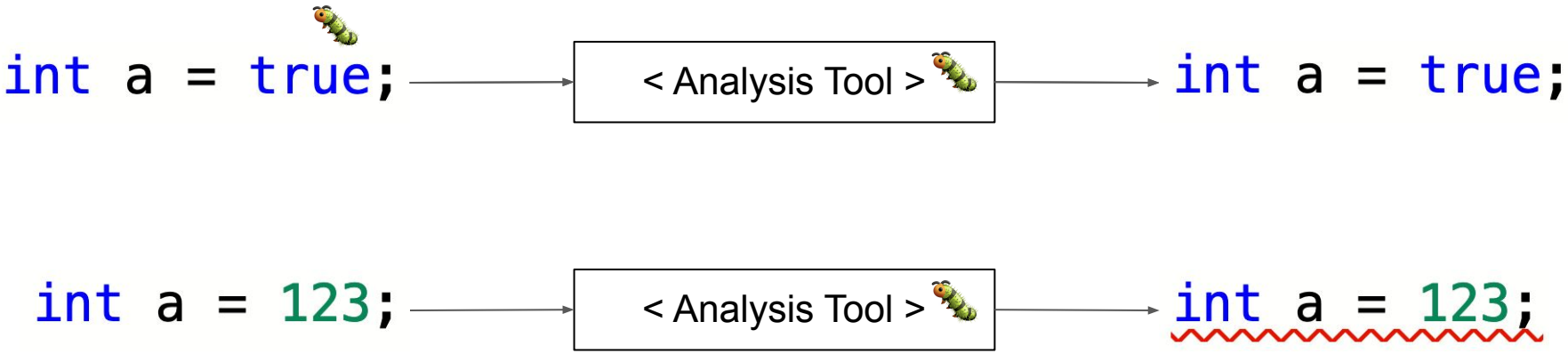
Type checking, dataflow analysis, linting, pretty printing, code generation, [...]

Intended behavior

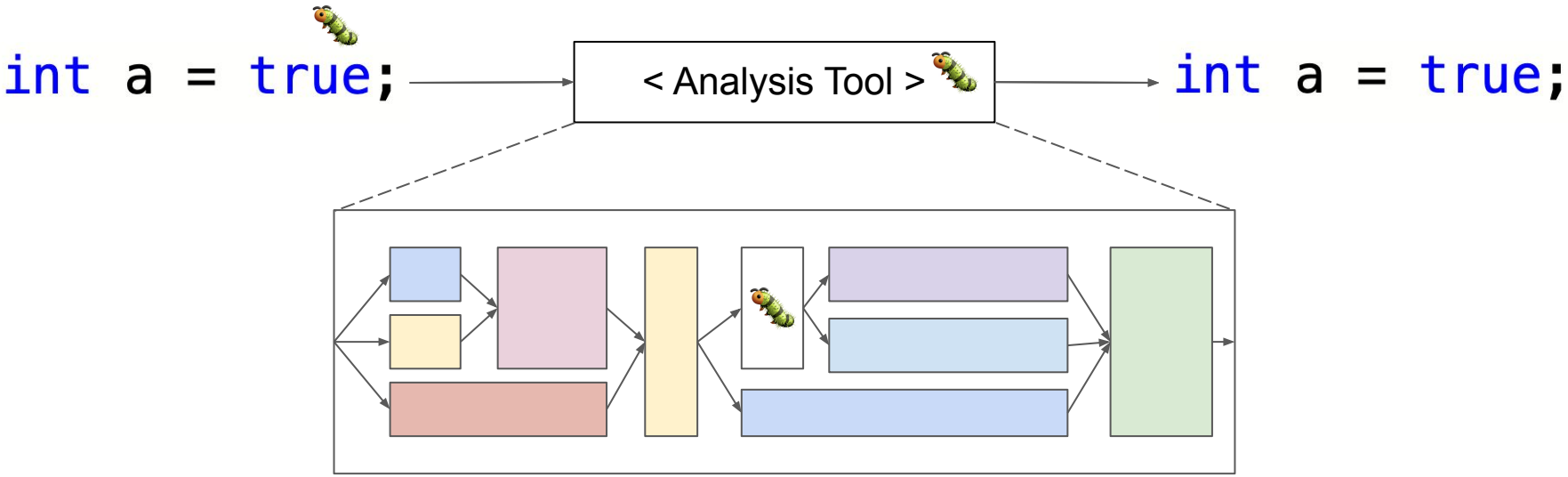
What if…



What if…


int a = true;
< Analysis Tool >
int a = true;
int a = 123;
< Analysis Tool >
int a = 123;

Breaking down the analysis



Property probes

Property probe definition

- A live observer of a property on an AST node

```
int x = 123;

return x;
```

Property: type()

Result: int

Property probe definition

- A live observer of a <u>property</u> on an AST node
  - Evaluate a property and present the results (ref: watch expressions)

```
int x = 123;

return x;
```

Property: type()

Result: int

Property probe definition

- A <u>live observer</u> of a property on an AST node
  - Evaluate a property and present the results (ref: watch expressions)
  - Live / up-to-date (ref: live programming)

```
int x = 123;

return x;
```

Property: type()

Result: int

Property probe definition

- A live observer of a property on an AST node
  - Evaluate a property and present the results (ref: watch expressions)
  - Live / up-to-date (ref: live programming)
    - Across multiple versions of a source file

```
int x = 123;

return x;
```

Property: type()

Result: int

Property probe definition

- A live observer of a property on an AST node
  - Evaluate a property and present the results (ref: watch expressions)
  - Live / up-to-date (ref: live programming)
    - Across multiple versions of a source file

```
int x = 123;

return x;
```

Property: type()

Result: int

```
5   int x = 123;
6
7   return x;
8
```

VarAccess.type [7:8→7:8]

[0:0→0:0] IntType

Demo - CodeProber

```
int a = 123;
```

VariableDeclarator.isFinal [5:9→5:15]

```
false
```

```
int a = 123;

int b = a * 2;
```

MulExpr.constant [7:11→7:15]

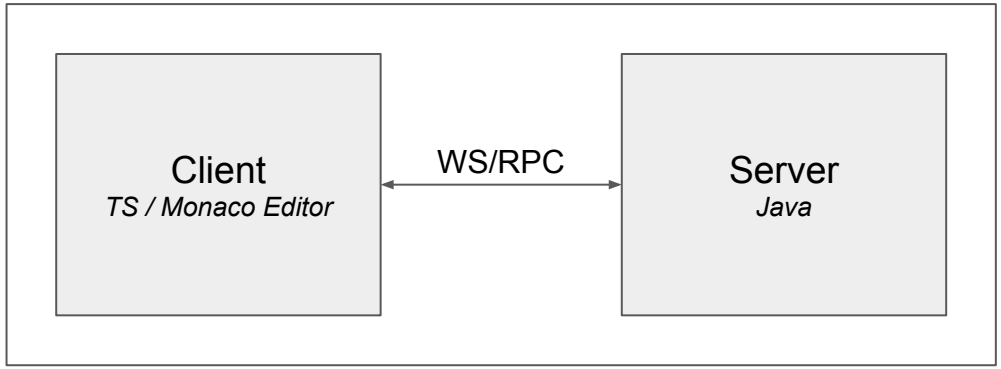
```
246
```

```
void f(boolean b) {
  String s = null;
  if (b) {
    s = "Hello";
  }
  s.hashCode();
}
```

MethodDecl.bytecodes [10:3→16:3]

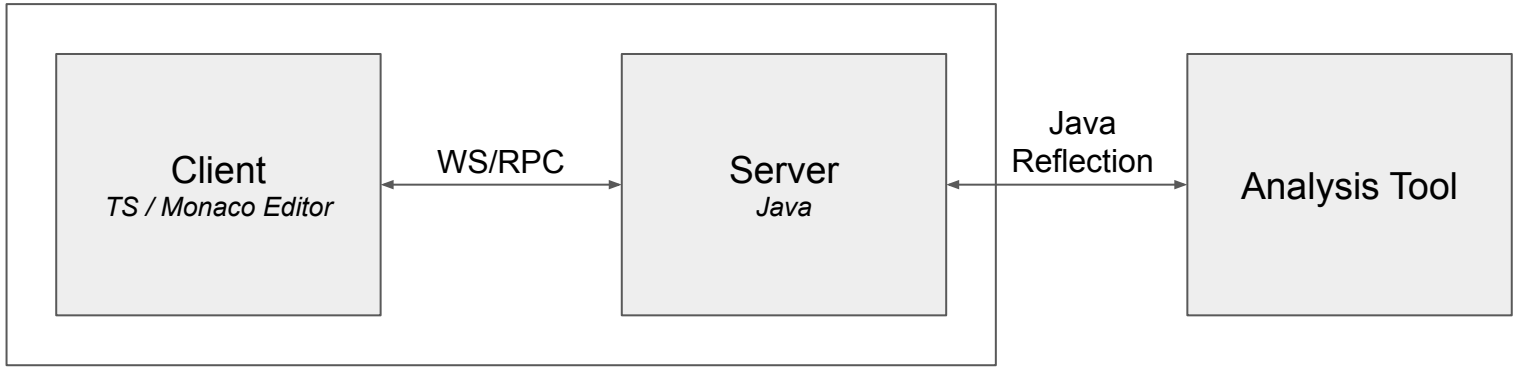
0	1	aconst_null
1	2	astore_2
2	3	iload_1
3	4	ifeq 6
6	5	ldc 8
8	6	astore_2
9	7	aload_2
10	8	invokevirtual 14
13	9	pop
14	10	return

CodeProber Architecture



Show UI, Traverse AST, invoke properties,
handle changes (source code & underlying tool), [..]

CodeProber Architecture



Show UI, Traverse AST, invoke properties,
handle changes (source code & underlying tool), [..]

Parse (Text->AST)

Example uses

```
class Foo {
    String bar(boolean b) {
        return b ? "Hello" : "World";
    }
}
```

MethodNode.getCfg [4:0→6:4095]

```
BB0
    -> BB1
BB1
  0  LocalLoad(I,1)
  1  Constant(I,0)
  2  ConditionalBranch(I,eq,5)
    -> BB3
    -> BB2
BB2
  3  Constant(Ljava/lang/String;,"Hello")
  4  Goto(6)
    -> BB4
BB3
  5  Constant(Ljava/lang/String;,"World")
    -> BB4
BB4
  6  Return(Ljava/lang/Object;)
    -> BB5
BB5
```

WALA

```
class Foo {
    void bar() {
        new Object().finalize();
    }

    int Foo;
}
```

PmdProgram.getProblems [3:1→24:1]

```
[5:9→5:31] ASTPrimaryExpression
Violated AvoidCallingFinalizeRule

[8:5→8:12] ASTFieldDeclaration
Violated AvoidFieldNameMatchingTypeNameRule
```

PMD

```
class Foo {
    int bar() {
        String s = null;
        return s.hashCode();
    }
}
```

SpotbugsProgram.getProblems [0:1→999:0]

```
[8..8]: NP: Null pointer dereference
```

SpotBugs

More example uses

```
def f(var:int, var: int) -> int:
```

IdDecl.isMultiDeclared [3:16→3:18]

```
true
```

ChocoPy

```
1  class A extends Object {
2      A() { super(); }
3  }
4  class B extends A {
5      B() { super(); }
6  }
```

TypeUse.lookupType("A") [4:17→4:17]

```
[1:1→3:1] ClassDecl
```

Featherweight Java

```
1
2  int f() {
3      return true;
4  }
```

Program.errors [2:1→4:1]

```
Error at line 3: Expected 'int', got 'boolean'
```

SimpliC

```
void f(boolean b) {
    if (b) {
        g();
    }
}
```

MethodDecl.exit [24:3→28:3]

```
[28:3→28:3] Exit
```

IntraJ (Java Compiler with CFG analysis)

< Your analyzer here >

CodeProber requirements

- Properties must be represented as methods on nodes.

CodeProber requirements

- Properties must be represented as methods on nodes.
- Must have source code location in normal nodes

CodeProber requirements

- Properties must be represented as methods on nodes.
- Must have source code location in normal nodes
- Must be able to traverse the tree
  - (To support the “exploration” aspect of clicking the output of the probe)

CodeProber requirements

- Properties must be represented as methods on nodes.
- Must have source code location in normal nodes
- Must be able to traverse the tree
  - (To support the “exploration” aspect of clicking the output of the probe)

- JastAdd is a meta-compiler supporting reference attribute grammars (RAGs)
- JastAdd fulfills these requirements!
  - https://jastadd.cs.lth.se/web/

CodeProber requirements

- Properties must be represented as methods on nodes.
- Must have source code location in normal nodes
- Must be able to traverse the tree
  - (To support the “exploration” aspect of clicking the output of the probe)

- JastAdd is a meta-compiler supporting reference attribute grammars (RAGs)
- JastAdd fulfills these requirements!
  - https://jastadd.cs.lth.se/web/
- Non-JastAdd tools can work too

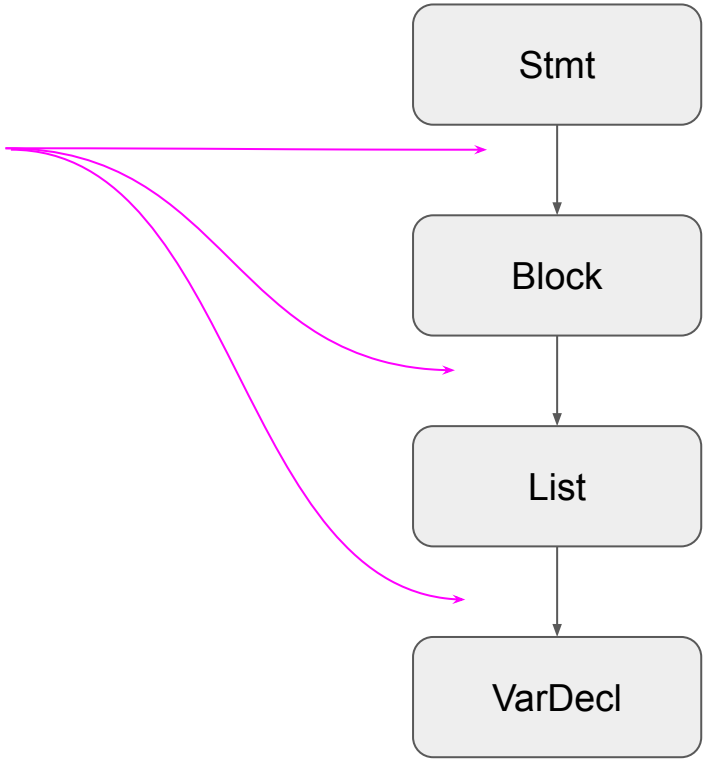
Node Locators

How we track AST nodes across multiple edits to a source file

Terminology

A step connects parent and child nodes.

A list of steps is called a “node locator”.



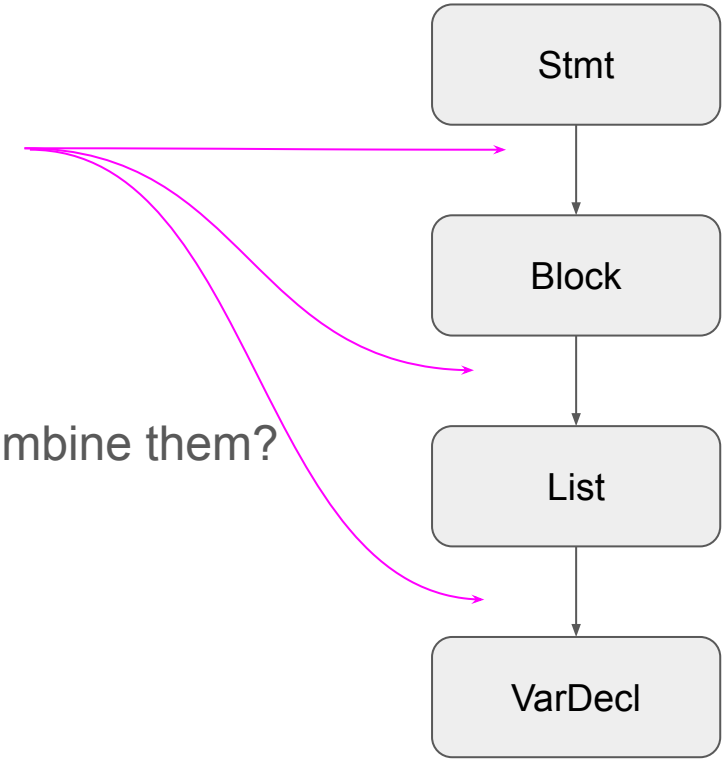
Terminology

A step connects parent and child nodes.

A list of steps is called a “node locator”.

What step types exist, and how do you combine them?

- 🏃 Speed
- 🎯 Accuracy



AST Step types

1) **Child**

E.g “node.getChild(3)”

👍 Fast
👎 Overly specific (brittle)

AST Step types

1) **Child**

E.g “node.getChild(3)”

👍 Fast
👎 Overly specific (brittle)

2) **FN** (“Function”)

E.g “node.desugar()”

👍 Versatile
👎 Hard to (automatically) create

AST Step types

1) **Child**

E.g “node.getChild(3)”

👍 Fast
👎 Overly specific (brittle)

2) **FN** (“Function”)

E.g “node.desugar()”

👍 Versatile
👎 Hard to (automatically) create

3) **TAL** (“Type At Location”)

E.g “CallExpr at line 7, column 12”

👍 Resilient
👎 Sometimes slow and/or ambiguous

Node Locator samples

```
3    void f(boolean b) {
4      String x = "abc";
5      int y = b ? 10 : 20;
6    }
```

Node Locator samples

```
3   void f(boolean b) {
4     String x = "abc";
5     int y = b ? 10 : 20;
6   }
```

[Child(0), TAL(5:15,5:16,13,IntegerLiteral)]

Pick first file

Search within single file (fast)

Node Locator samples

(CFG Entry)
[Child(0), TAL(3:1,6:1,4,MethodDecl), FN(“entry”)]

```
3    void f(boolean b) {
4      String x = "abc";
5      int y = b ? 10 : 20;
6    }
```

[Child(0), TAL(5:15,5:16,13,IntegerLiteral)]

Node Locator samples

(CFG Entry)
[Child(0), TAL(3:1,6:1,4,MethodDecl), FN(“entry”)]

(String type)
[FN(“getLibCompilationUnit”, “java.lang.String”), TAL(0:0,0:0,2,ClassDecl)]

```
3    void f(boolean b) {
4      String x = "abc";
5      int y = b ? 10 : 20;
6    }
```

[Child(0), TAL(5:15,5:16,13,IntegerLiteral)]

Thank you for listening!

https://youtu.be/d-KvFy5h9W0

5min demo

https://git.cs.lth.se/an6308ri/code-prober

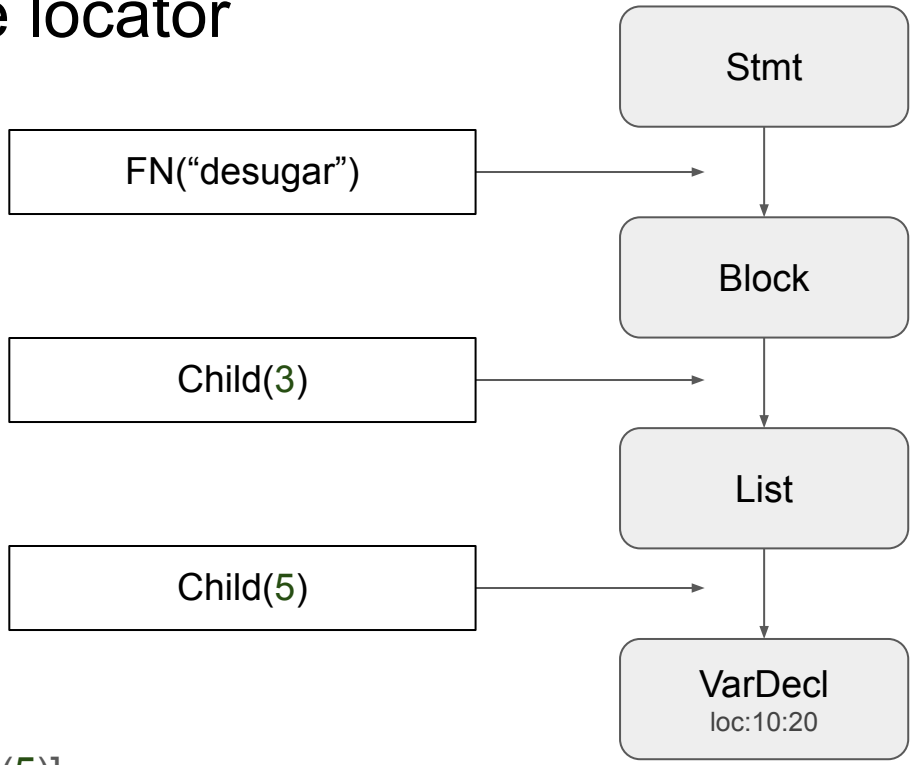
Source code

< Bonus slides >

(Not shown during presentation)

Creating Node Locators: 2 steps

Step 1 - construct naive locator



Result: [FN(“desugar”), Child(3), Child(5)]

Step 2 - Merge Child into TAL

Input: [FN(“desugar”), Child(3), Child(5)]

For each sequence of 1+ ‘Child’ steps:

```
if (sequence as ‘TAL’ is unambiguous) {
    replace sequence with a single TAL
}
```

Result: [FN(“desugar”), TAL(10:20,VarDecl))