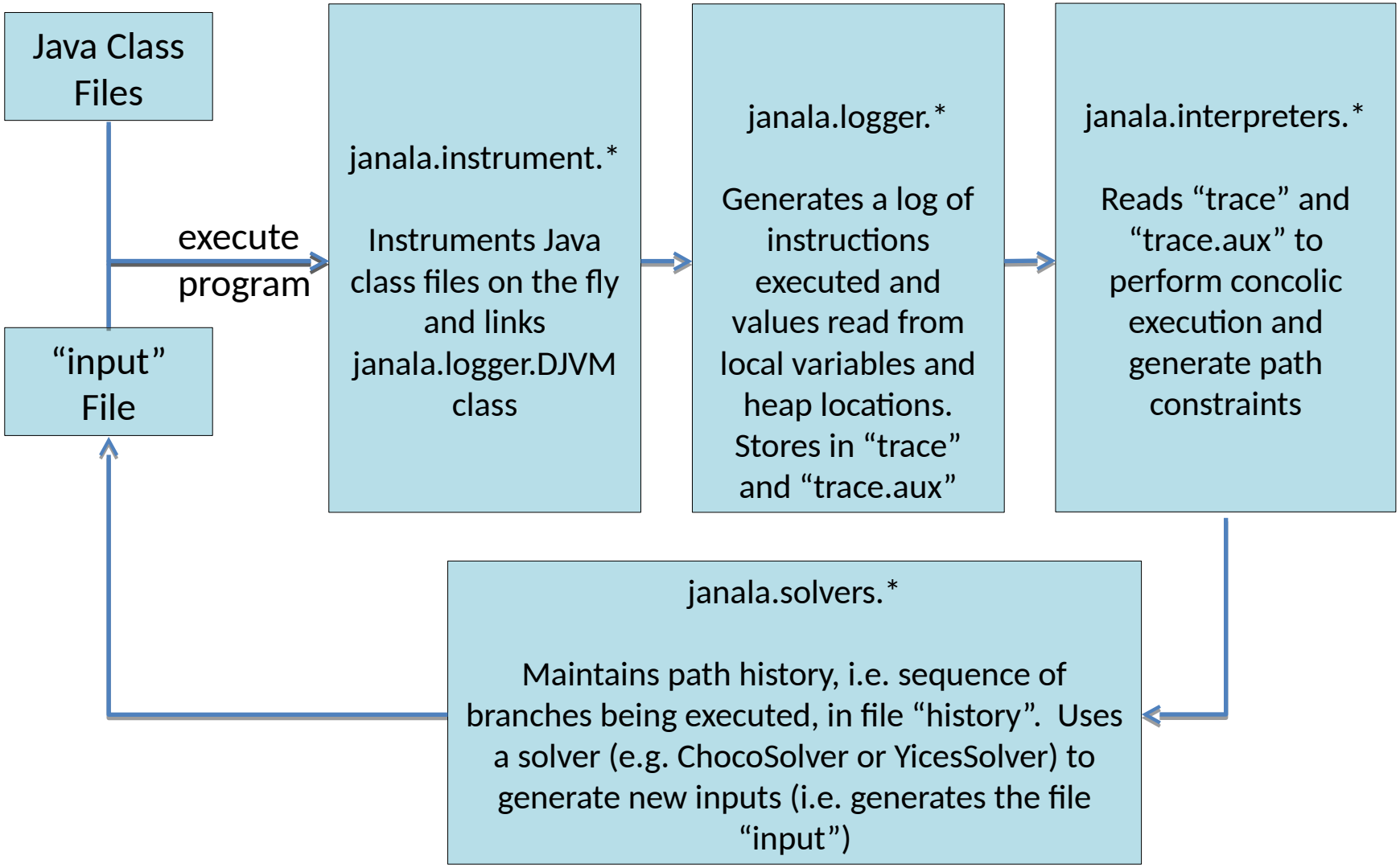
Java Concolic Testing

https://github.com/ksen007/janala2

Architecture

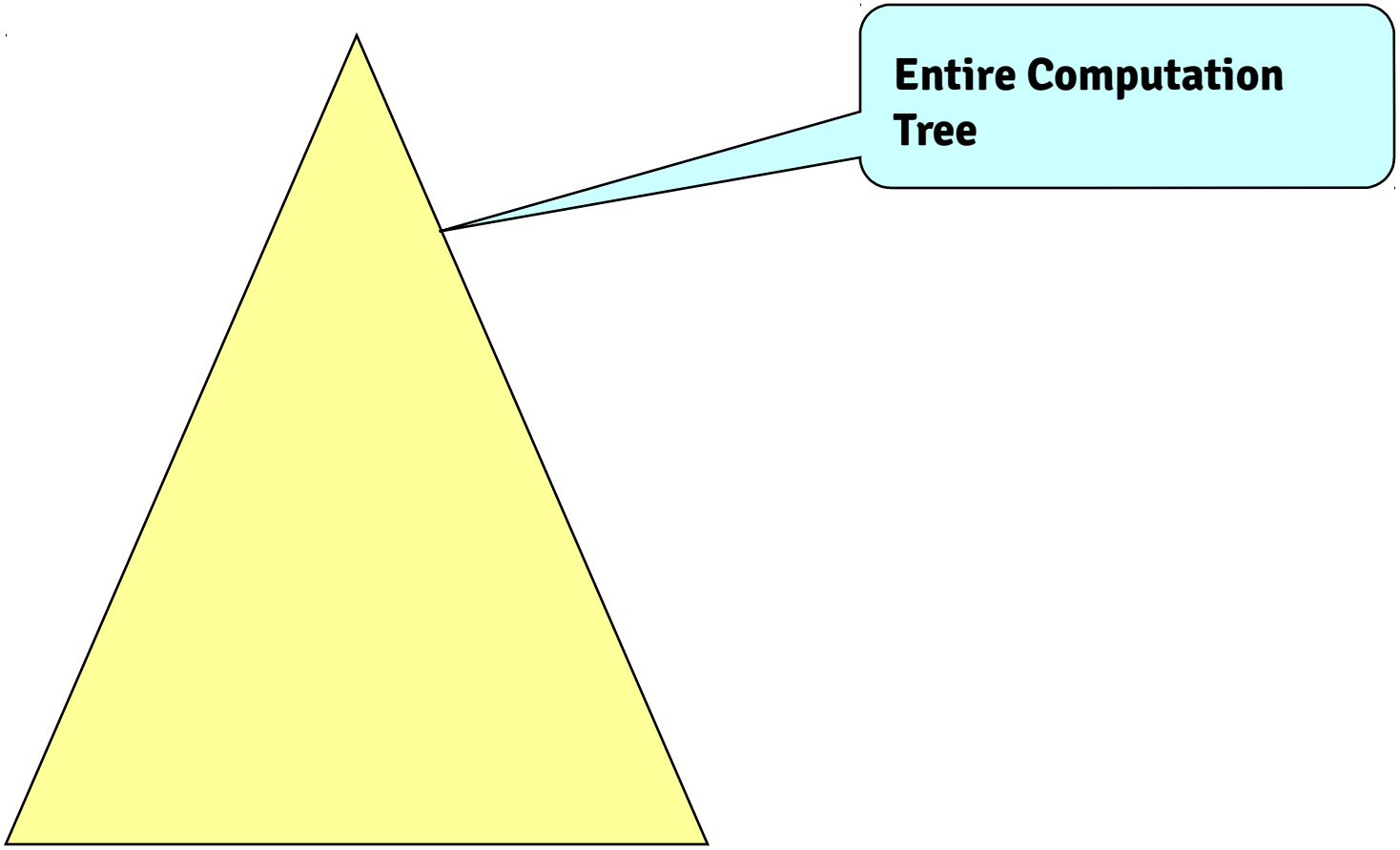


Details

- Handles integral types (int, char, boolean, byte, short, long)
- Handles all bytecode instructions (~200 bytecode instructions) except TABLESWITCH, LOOKUPSWITCH, and MULTIANEWARRAY)
- Handles Exceptions (~ 43 bytecode instructions) and uninstrumented methods robustly
- Handle String equals.
- Extend janala.solvers.Solver to implement custom solvers
  - Yices is much more faster than ChocoSolver
- Extend janala.solvers.Strategy to implement custom search strategies
- database.table.* has libraries for modeling SQL queries and creating symbolic databases

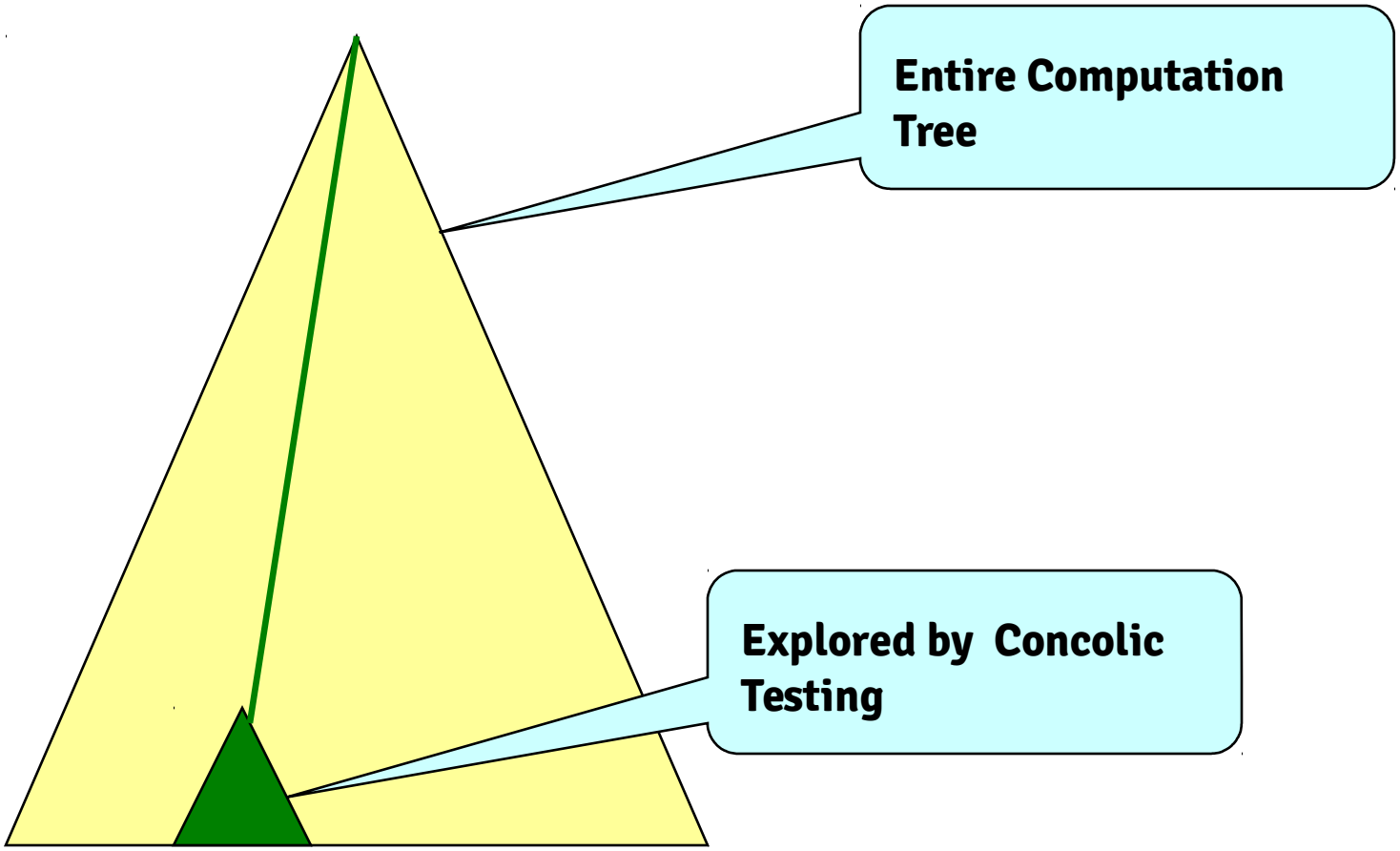
Limitations

- Path Space of a Large Program is Huge
  - Path Explosion Problem



Limitations

- Path Space of a Large Program is Huge
  - Path Explosion Problem



Limitations

- Simple Depth-First Search is not effective in quickly generating tests
  - exponential blow-up in search space
- Various heuristics to guide search
  - Control flow graph based
  - Data slicing based
  - Summary based
- None seems to work well for initial database generation

Annotations

- User understands the code
  - user can guide us how to search effectively
- Goal: provide a simple to use annotation mechanism so that user can guide search by annotating code
  - Number of annotations should be minimal
  - Annotations should be easy to understand
- Results:
  - Control Annotation
  - Data Annotation

Control Annotation: assertIfPossible

- CATG.assertIfPossible(int pathId, boolean predicate)
- The function can be inserted throughout the code under test
- CATG will try to take paths such that "predicate" passed as second argument of any assertIfPossible along the path evaluates to true.
- assertIfPossibles whose "pathId"s are not equal to catg.pathId in catg.conf are ignored along the path
  - Enables to insert different sets of independent annotations
  - A set of annotations with common “pathId” can be activated by setting catg.pathId in catg.conf
- See src/tests/DTEST1 and src/tests/ManyColumnsOrRecords for examples.
- Usually, before returning "ret" from any where clause, you can insert CATG.assertIfPossible(1,ret). This will force Where clause to return true.

assertIfPossible: Example

```
public class AssertIfPossibleTest {
   public static boolean foo(){
      boolean ret;
      int x = CATG.readInt(0);
      int y = CATG.readInt(0);
      if (x == 100 && y == 200) {
         ret = true;
      } else {
         ret = false;
      }
      CATG.assertIfPossible(1, ret);
      return ret;
   }

   public static void main(String[] args){
      int sum = 0;

      if (foo()) sum++;
      if (foo()) sum++;
      if (foo()) sum++;
      if (foo()) sum++;

      if (sum > 2) {
         System.out.println("sum > 2");
      } else {
         System.out.println("sum < 2");
      }
   }
}
```

- Simple concolic testing explores $3^4 = 81$ paths
  - foo has 3 paths
  - foo is called 4 times
  - exponential blow-up
- Single control annotation explores 9 paths
  - $1 + 4*(3-1)$

Data annotation: abstractXXX()

API:

- void CATG.BeginScope()
- void CATG.EndScope()
- int CATG.abstractInt(int x)
- boolean CATG.abstractBool()
- long CATG.abstractLong(long x)
- char CATG.abstractChar(char x)
- byte CATG.abstractByte(byte x)
- short CATG.abstractShort(short x)
- String CATG.abstractString(String x)

See for examples:

- src/tests/AbstractionTest1.java
- src/tests/AbstractionTest2.java
- src/tests/ManyColumnsRecords2.java

Data annotation: abstract code block

- Annotate a code block, say x = foo(), to be abstract as follows:

```
CATG.BeginScope();
x = foo();
CATG.EndScope();
x = CATG.abstractInt(x);
```

- Surround the code block with CATG.BeginScope() and CATG.EndScope()
- Any variable that is written (or computed) by the code block is then abstracted by using CATG.abstractXXX()
- An abstract block can be nested within other abstract blocks

Data annotation: example

```
public class AbstractionTest2 {
   public static boolean foo(){
      CATG.BeginScope();
      boolean ret;
      int x = CATG.readInt(0);
      int y = CATG.readInt(0);
     if (x == 100 && y == 200) {
         ret = true;
      } else {
         ret = false;
      }
      CATG.EndScope();
      ret = CATG.abstractBool(ret);
      return ret;
   }

   public static void main(String[] args){
      int sum = 0;

      CATG.BeginScope();
      if (foo()) sum++;
      if (foo()) sum++;
      if (foo()) sum++;
      if (foo()) sum++;
      CATG.EndScope();
      sum = CATG.abstractInt(sum);

      if (sum > 2) {
         System.out.println("sum > 2");
      } else {
         System.out.println("sum < 2");
      }
   }
}
```

- Simple concolic testing explores $3^4 = 81$ paths
  - foo has 3 paths
  - foo is called 4 times
  - exponential blow-up
- Code with data annotation explores 13 paths
- Gives full coverage as original code
  - assertIfPossible may miss coverage

Data annotation: exploration algorithm

- We explore the function under test using concolic testing, but
  - avoid exploring the block of code enclosed within CATG.BeginScope() and CATG.EndScope() (which we will call abstract blocks)
  - replace each variable x with an unconstrained input if we call x = CATG.abstractXXX(x) after a CATG.EndScope()
  - this creates an abstraction of the program: by ignoring paths in the abstract blocks of code, we may generate paths that are not concretely realizable.
- Abstraction reduces the complexity of interprocedural path exploration to intraprocedural path exploration
- concolic testing finds an abstract path PI in the abstract program
- Given a path PI in the abstraction,
  - we perform path refinement, that is, expand the ignored abstract blocks along the path to get a concretely realizable path whose projection on the function under test is PI
- Path refinement performs a backtracking search over the path PI,
  - finds concrete paths through each abstract code blocks that can be stitched together
- Refinement recursively invokes the AbstractRefineStrategy.java algorithm, expanding ignored abstract code blocks along the path
- Paths inside the nested abstract code blocks are expanded on demand to get a concretely realizable path
  - we only explore relevant parts of the program path space
  - this significantly prunes our search while retaining the relative soundness and completeness of concolic testing

Performance Improvement of CATG

- Improved performance of CATG by at least 2X
  - Previously each test generation execution of CATG had two phases
    - Recording of all instructions executed on a concrete test input
    - Replay and reinterpret logged instructions in a second execution
    - This was done because certain class information were not available during recording
  - CATG now generates a test input in a single phase
    - No separate record and replay phases
    - Concolic testing takes place during normal execution of a program
    - Handled missing class information by modifying instrumentation and by populating class information on demand
  - Single phase concolic testing is necessary in future if we want to avoid restarting JVM for each test execution
    - Avoiding JVM restart should give another 5X-100X speedup

Handling “Branch Prediction Failure”

- CATG used to restart the entire test generation process whenever there was a “branch prediction failure” warning, i.e. concolic testing is not taking the expected path on a generated input
- Created some adverse side-effects in data annotation
  - In data annotation, it is common to get “branch prediction failure” warning
- Modified CATG so that “branch prediction failure” state is handled properly
  - Backtrack to the parent branch instead of restart if a “branch prediction failure” happens at a branch
  - Data annotation should now work as expected

Python scripts for CATG

- Replaced non-portable shell scripts of CATG with portable python scripts
- Python concolic.py gives the usage documentation of the script

Future Work

- Improve data annotation to handle cases where value passed to a CATG.abstractXXX(v) could be symbolic.
- Avoid restarting JVM for each test input. This will take considerable effort, but it could speedup CATG by a factor of 5X – 100X.
- Once evaluation results of annotation mechanism are available, write a paper on the annotation mechanism