**Idea 1: Evolutionary Synthesis-based Program Repair from Refined Types**

1. Given a program with a bug automatically repairs it using genetic programming.
2. A test suite is provided by the user to declare its intention towards a function.
3. With the test suite, similarly to Daikon (<https://plse.cs.washington.edu/daikon/>), obtain the logical predicates of the function.
4. These logical predicates are either Refined Types or Non-Restricted Refined Types.
   1. Refined Types: Are types that are statically verifiable with a SMT-Solver. These are used to synthesize valid programs.
   2. Non-Restricted Refined Types: These types, also known as dependent types, are not verifiable statically and will be used to evaluate valid programs to ensure that they are also correct.
5. Convert the non-restricted refined types (logical predicates) deduced from the program traces into continuous fitness functions.
6. Use an evolutionary approach to obtain a valid and correct program.
   1. Each generation, use a non-deterministic synthesizer to generate expressions from the refined types.
   2. These expressions are evaluated with the fitness functions. The test suite provided is used in the fitness functions. To prevent overfitting the patch, randomly tests from the refinement types, similarly to the metamorphic testing (<https://www.sciencedirect.com/science/article/pii/S0164121216300206>) are generated.
   3. If a program passes all the tests, the repair is done.

Use asserts of unit tests to encaminhar a sintese.

Risks:

* Cannot assume refined types in real-world open-source programs
* Workaround: Provide libraries with refined types (i.e., we can ensure “good” API calls). [Could provide refinement types for rospy and roscpp?]
* Could potentially use Daikon to learn assertions -> could use assertions to learn refined types -> profit?
* Daikon produces spurious invariants (i.e., overfitting)
* Daikon involves instrumentation -- slows down execution of the program slightly, which can adversely affect real-time systems.
* Need to find robotics bugs: look at ROBUST, mine bugs in ROS projects
* How to deal with mutable objects and void methods (?)

Use cases:

* Repair for robots! Learn refined types offline using a cloud computing platform (in combination with a good test generator and oracle). Provide a “witness”/proof of vulnerability that exemplifies a bug in the robot. Localise and repair the bug!

Related work:

* Chris will add papers on using Daikon to learn invariants + Daikon for program repair + contract-based program repair + contract-based program repair without contracts (\*\*) + S3: Syntax-guided program repair
* Paulo: infer semantics on programs: <https://homepages.dcc.ufmg.br/~fernando/publications/papers/Leandro_POPL18.pdf> https://homepages.dcc.ufmg.br/~fernando/publications/papers\_pt/Marcus17SBLP.pdf

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Meeting 1**

Check JFix and the assertions generated with Java Path Finder:

<https://xuanbachle.github.io/semanticsrepair/>

<https://github.com/javapathfinder>

<https://github.com/Spirals-Team/IntroClassJava>

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Meeting 2**

How are we going to deal with the different programming languages we want to repair the programs in? Let's say we want to repair a program in Java, there are two ways of approaching this:

**1:** We source-to-source translate from Java to Aeon and reverse. Any behaviour that

we are not able to handle, we launch an exception.

**2:** Work directly in Java, convert everything to Java

**3:** We create best effort bindings from the Java language to an intermediate language

that will be able to translate to python. *Call Aeon from Java*

<http://spoon.gforge.inria.fr/>

<https://www.srcml.org/#download>

*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*

*Other info and interesting tools:*

*Annotate holes so they are binded (example):* <https://github.com/comby-tools/comby>

*Useful for testing multiple SMT:* <https://github.com/pysmt/pysmt>