# Refactoring with Synthesis

Authors: Veselin Raychev, Max Schäfer, Manu Sridharan, Martin Vechev

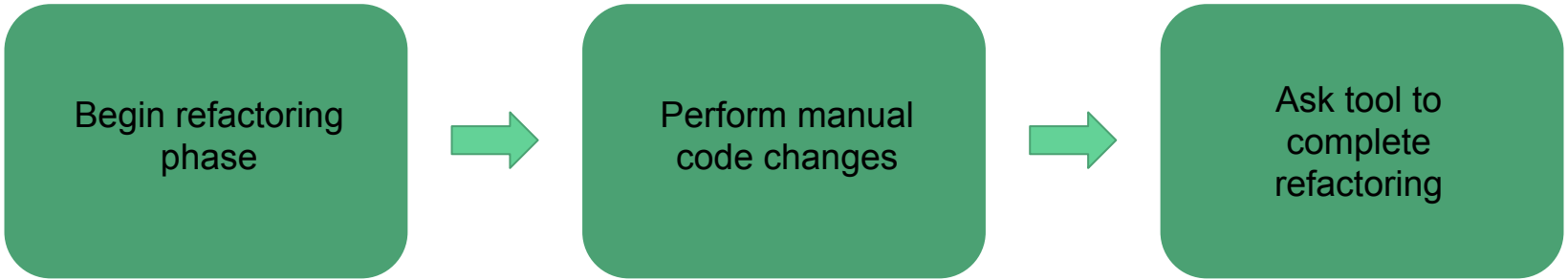
slide author names omitted for FERPA compliance

# Research Questions

- Can the refactoring process be simplified?
- Can the refactoring process be made to be more flexible?
- Can automated refactoring be used to apply refactorings to the whole program based on an example refactoring by the developer rather than forcing a developer to follow a pre-set refactoring?

# Solution

- Base the refactorings off of changes made by the user rather than having the user use refactorings presented to them by the IDE



# Main Contributions

1. A new interface for refactoring tools, where the user indicates the desired transformation with example edits and the tool synthesizes a sequence of refactorings that include the edits.
2. A novel technique for synthesizing refactoring sequences via heuristic search over pruned programs.
3. An implementation RESYNTH, whose architecture minimizes the effort required to add new refactorings.
4. An initial evaluation of RESYNTH, showing it can synthesize complex refactoring sequences for real examples.

# Resynth

- Same interface for each refactoring process
- A sequence of refactorings becomes much easier
- The refactorings are applied as a unit
- Adding a new refactoring is easier and cleaner

```
1 public class Account {
2   private String name;
3
4   void printOwing() {
5     printBanner();
6     System.out.println("name: " + name);
7     System.out.println("outstanding: " +
8                        getOutstanding());
9   }
10
11  private double getOutstanding() {
12    //...
13  }
14
15  private void printBanner() {
16    //...
17  }
18 }
```

(a)

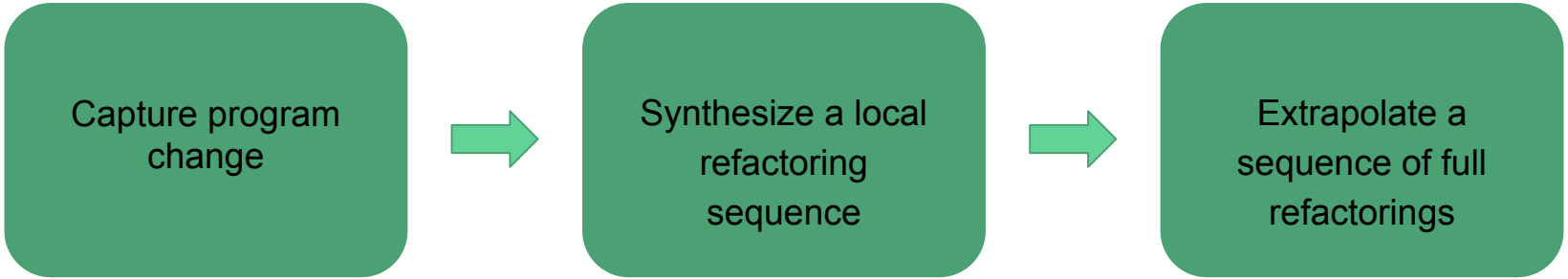
```
19 public class Account {
20   private String name;
21
22   void printOwing() {
23     printBanner();
24     printDetails( getOutstanding());
25   }
26
27   private void printDetails(double outstanding) {
28     System.out.println("name: " + name);
29     System.out.println("amount: " + outstanding);
30   }
31
32   private double getOutstanding() {
33     //...
34   }
35
36   private void printBanner() {
37     //...
38   }
39 }
```

(b)

**Figure 1.** Example of a complicated refactoring that cannot be achieved in a single step in current IDEs; changes highlighted.

# The Approach

The heart of RESYNTH is a search strategy for discovering appropriate refactoring sequences based on a small number of user edits. RESYNTH takes the following approach:



# Abstract Syntax Tree

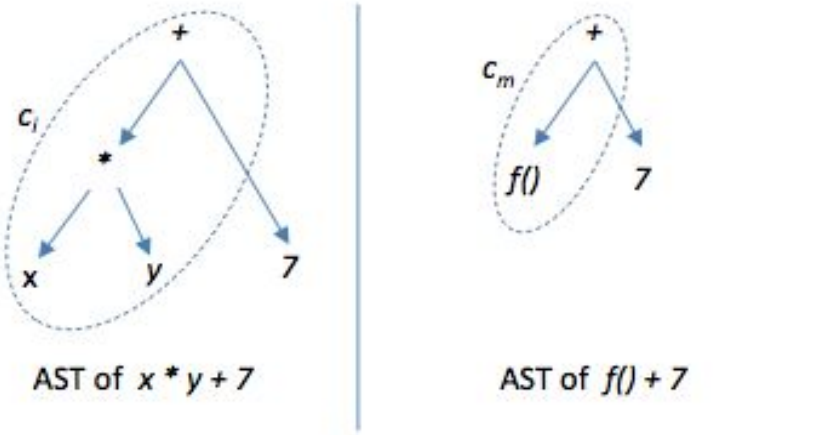
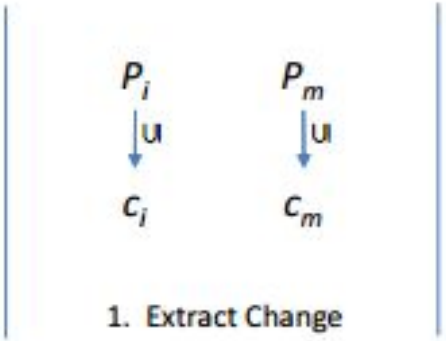



**Figure 3.** Two ASTs and the change $(c_i, c_m)$ between them. The change is captured with dotted lines.

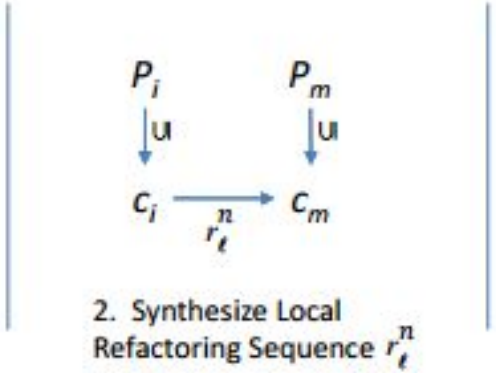
# Capture Program Change

- Create an AST for the original code
- Generate an AST for the changed code
- Capture the difference between the two AST's and get rid of the rest of the AST as repeated sections of the AST is redundant in this case



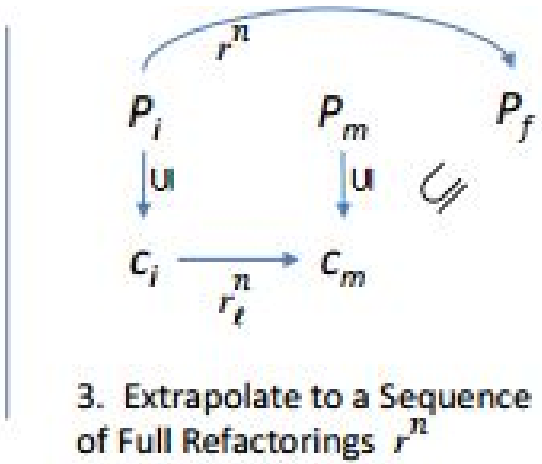
# Synthesize Local Refactoring Sequence

- Generate search space of AST's using different refactorings
- A* search through the refactorings to find the best way from the original AST to the user edited AST
- Use edit distance or expression difference as a heuristic for the A* search



# Extrapolate Full Sequence of Refactorings

- Perform extraction sequence on local refactorings
- Input local refactoring pattern into full sequence of patterns that can use same refactoring as laid out in example by user



# Example of Synthesis Algorithm

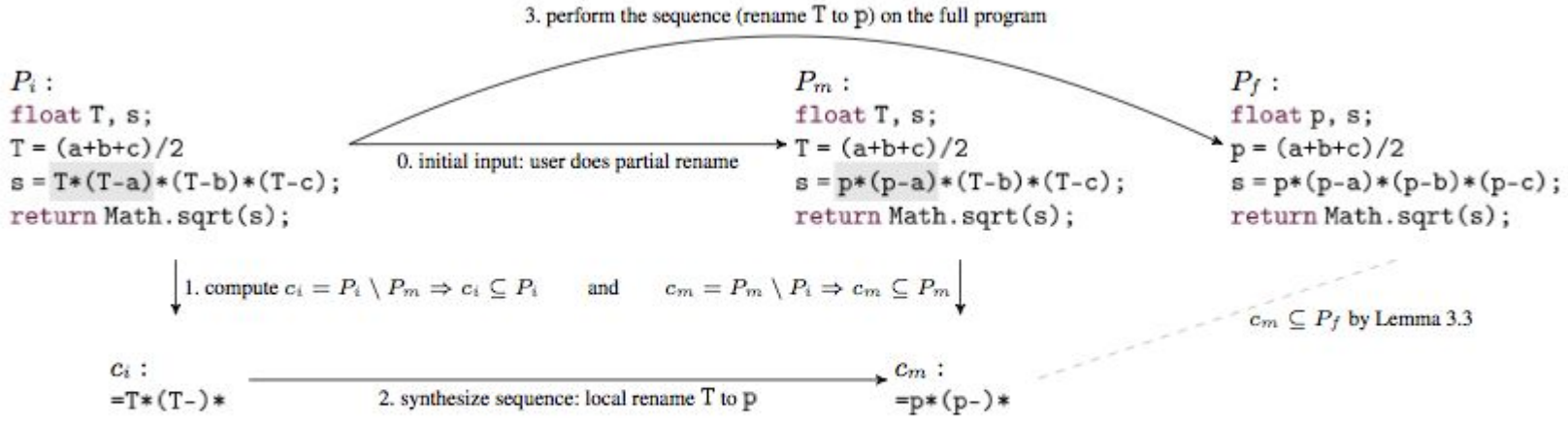



**Figure 5.** Example of synthesizing a refactoring sequence. Initially (stage 0), the user performs part of the rename (the user change is highlighted in both programs). Then $c_i$ and $c_m$ are computed (stage 1). Then, a sequence of one local rename is discovered (stage 2). Finally, the rename is applied to the full program (stage 3).

# Related Work

While BeneFactor and WitchDoctor also infer refactorings from user edits, they differ from our system in that:

(1) they do not require the user to indicate a refactoring is occurring
(2) they cannot perform transformations requiring a sequence of refactorings.

- While (1) can be an advantage for novice users, requiring the user to indicate the start of the refactoring enables the tool to discover more complex sequences, and allows for performing several independent refactorings “atomically.”

# Evaluation

- Some examples of refactoring and how ReSynth worked on them.
- While Eclipse provides an implementation of INTRODUCE PARAMETER, it fails to handle this example.

| Example | steps | Source |
|---|---|---|
| ENCAPSULATE DOWNCAST | 3 | literature [6] |
| EXTRACT METHOD (advanced) | 4 | literature [6] |
| DECOMPOSE CONDITIONAL | 6 | literature [6] |
| INTRODUCE FOREIGN METHOD | 2 | literature [6] |
| REPLACE TEMP WITH QUERY | 3 | literature [6] |
| REPLACE PARAMETER WITH METHOD | 3 | literature [6] |
| SWAP FIELDS | 3 | literature [32] |
| SWAP FIELD AND PARAMETER | 3 | literature [25] |
| INTRODUCE PARAMETER | 6 | Stack Overflow[9] |

**Table 1.** Realistic examples used to test RESYNTH.

# Stress Testing Using Benchmarks

- Used additional testing by generating java classes with a few methods and variables
- Then applied edits to these class
- Used the tool to try to preserve the edits.

| Metric | Dataset | |
|---|---|---|
| | **Real** | **Synthetic** |
| Number of tests | 9 | 100 |
| Avg. number of trees searched | 87 | 3752 |
| Avg. number of successors in a search | 1296 | 105310 |
| Avg. search time | 0.014s | 1.629s |
| Avg. Eclipse refactoring time | 2.953s | 1.654s |
| Refactoring sequence length | | |
| 1 refactoring | 0 | 2 |
| 2 refactorings | 1 | 45 |
| 3 refactorings | 5 | 7 |
| 4 refactorings | 1 | 15 |
| 5 refactorings | 0 | 3 |
| 6 refactorings | 2 | 2 |
| 7 refactorings | 0 | 9 |
| 8 refactorings | 0 | 0 |
| 9 refactorings | 0 | 1 |
| Failure to find sequence after 20000 searched trees | 0 | 16 |

**Table 2.** Results for our refactoring sequence search. The $A^*$ heuristic function weights (see Section 3.4) were $a_1 = 0.125$ and $a_2 = 0.25$.

| Metric | Search space limit (# trees) | | |
|---|---|---|---|
| | 20,000 | 100,000 | 500,000 |
| Num. failed tests | 16 | 15 | 9 |
| Avg. number of searched trees | 3752 | 15900 | 64392 |
| Avg. search time | 1.629s | 7.802s | 34.473s |

**Table 3.** Success rate on synthetic benchmarks with different search bounds. Tested with $a_1 = 0.125$ and $a_2 = 0.25$.

| Search Parameters | | Real Examples | | Synthetic Tests | |
|---|---|---|---|---|---|
| Edit distance weight $a_1$ | Expr. distance weight $a_2$ | Num. failed tests | Avg. num. searched trees | Num. failed tests | Avg. num. searched trees |
| 0.000 | 0.000 | 2 | 5306 | 32 | 6943 |
| 0.000 | 0.125 | 1 | 3076 | 26 | 5373 |
| 0.000 | 0.250 | **0** | 184 | 26 | 5348 |
| 0.000 | 0.500 | **0** | 119 | 24 | 4537 |
| 0.000 | 1.000 | 1 | 2350 | 16 | 3316 |
| 0.125 | 0.000 | **0** | 1248 | 25 | 5065 |
| 0.125 | 0.125 | **0** | 115 | 19 | 4642 |
| **0.125** | **0.250** | **0** | **87** | 16 | 3752 |
| 0.125 | 0.500 | **0** | 122 | 15 | 3396 |
| 0.125 | 1.000 | **0** | 1154 | **14** | **3243** |
| 0.250 | 0.000 | **0** | 291 | 21 | 4456 |
| 0.250 | 0.125 | **0** | 281 | 18 | 3885 |
| 0.250 | 0.250 | 1 | 223 | 18 | 3694 |
| 0.250 | 0.500 | 1 | 153 | 17 | 3485 |
| 0.250 | 1.000 | 1 | 623 | **14** | 3516 |
| 0.500 | 0.000 | 2 | 358 | 26 | 4481 |
| 0.500 | 0.125 | 1 | 189 | 23 | 4401 |
| 0.500 | 0.250 | 1 | 158 | 22 | 4274 |
| 0.500 | 0.500 | 1 | 158 | 20 | 4114 |
| 0.500 | 1.000 | 1 | 465 | 18 | 4092 |
| 1.000 | 0.000 | 2 | 3033 | 24 | 4704 |
| 1.000 | 0.125 | 1 | 641 | 24 | 4796 |
| 1.000 | 0.250 | 1 | 637 | 25 | 4786 |
| 1.000 | 0.500 | 1 | 477 | 26 | 4704 |
| 1.000 | 1.000 | 1 | 768 | 22 | 4490 |

**Table 4.** Search space with different parameters of the heuristic function for the A* search (When $a_1 = a_2 = 0$, the search is a breadth-first search.)

# User Study

- We recruited six participants: two undergraduate students, three graduate students, and one professional.
- Brief demonstration of RESYNTH
- We gave each participant a set of three refactoring tasks
- Asked them to complete the tasks using either RESYNTH, Eclipse’s built-in refactorings, or manual editing.

# Discussion Question

What is the scalability of this tool within the code?

# Discussion Question

What is the scalability of this tool within the code?

This depends on your definition of scalability. If you have really long code with many decently simple enough refactorings, there is no reason why this tool should not work on this. However, the more complicated the refactorings get, the harder it will be for this tool to perform, as the search space for the A* algorithm will grow to be large.

# Discussion Question

Could this approach be used to tackle other large scale code changes?

# Discussion Question

Could this approach be used to tackle other large scale code changes?

We believe yes. While the resynth tool deals directly with refactorings and probably could not be extended to other code changes, the approach as a whole of finding the difference between program AST’s and performing an A* search to derive some sort of desired property out of the code could be useful in other areas of computer science.

# Discussion Question

Would this tool help to speed up the development process?

# Discussion Question

Would this tool help to speed up the development process?

Yes, this tool can speed up the process quite a bit as it can eliminate time spent on long refactorings of code. Even just extracting out repeated lines of code into small methods can take time out of development, and this tool can find and correct all those cases that you may spend time looking for and correcting.

# Discussion Question

Is there a potential this tool could over-refactor the code and cause a nuisance?

# Discussion Question

Is there a potential this tool could over-refactor the code and cause a nuisance?

Yes, if a user is not careful he may make a pretty generic change or not see parts of code that he does not want changed and accidentally cause refactorings in the code that he did not want.

# Discussion Question

Only 6 people were used in this user study. Is that enough to judge the usefulness of the tool?

# Discussion Question

Only 6 people were used in this user study. Is that enough to judge the usefulness of the tool?

This is just a personal opinion, however we feel that no it is not enough people. 6 people is just too small a number of people trying out a tool to really see if it will go to good use. There is a huge amount of diversity in coding styles and preferences in computer science and this can’t really be represented by so few an amount of people. Also, we feel it would be good to have a few more professionals test out this tool, as it could give a more accurate representation of its usefulness.