Kayak: Safe Semantic Refactoring to Java Streams

Cristina David
University of Oxford
cristina.david@cs.ox.ac.uk

Pascal Kesseli
University of Oxford
pascal.kesseli@cs.ox.ac.uk

Daniel Kroening
University of Oxford
kroening@cs.ox.ac.uk

ABSTRACT

Refactorings are structured changes to existing software that leave its externally observable behaviour unchanged. Their intent is to improve readability, performance or other non-behavioural properties. State-of-the-art automatic refactoring tools are *syntax*-driven and, therefore, overly conservative. In this paper we explore *semantics*-driven refactoring, which enables much more sophisticated refactoring schemata. As an exemplar of this broader idea, we present Kayak, an automatic refactoring tool that transforms Java with external iteration over collections into code that uses Streams, a new abstraction introduced by Java 8. Our refactoring procedure performs semantic reasoning and search in the space of possible refactorings using automated program synthesis. Our experimental results support the conjecture that semantics-driven refactorings are more precise and are able to rewrite more complex code scenarios when compared to syntax-driven refactorings.

KEYWORDS

program refactoring, program synthesis, program verification

ACM Reference format:
Cristina David, Pascal Kesseli, and Daniel Kroening. 2017. Kayak: Safe Semantic Refactoring to Java Streams. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September, 2017 (ESEC/FSE 2017),* 13 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Refactorings are structured changes to existing software which leave its externally observable behaviour unchanged. They improve non-functional properties of the program code, such as testability, maintainability and extensibility while retaining the semantics of the program. Ultimately, refactorings can improve the design of code, help finding bugs as well as increase development speed and are therefore seen as an integral part of agile software engineering processes [12, 25].

However, manual refactorings are a costly, time-intensive, and not least error-prone process. This has motivated work on automating specific refactorings, which promises safe application to large code bases at low cost. We differentiate in this context between *syntax-driven* and *semantics-driven* refactorings. While the former address structural changes to the program requiring only limited information about a program's semantics, the latter require detailed understanding of the program semantics in order to be applied soundly. An example of a refactoring that requires a semantics-driven approach is *Substitute Algorithm*, where an algorithm is replaced by a clearer, but equivalent version [12]. A syntax-driven approach is insufficient to perform such substantial transformations. Figure 1 illustrates this using an example: Both loops in the code implement the same behaviour. In order to recognise this and apply *Substitute Algorithm*, pattern-based approaches need explicit patterns for vastly different syntaxes implementing the same semantics, which is infeasible for practical applications.

```
List<Integer> org = getData();
List<Integer> copy = new ArrayList<>();
for (int i=0; i < org.size(); ++i)
  if (org.get(i) > 0) copy.add(2 * org.get(i));

Iterator<Integer> it=org.iterator();
while (it.hasNext()) {
  int tmp = it.next() * 2;
  if (tmp <= 0) continue;
  copy.add(tmp);
}
```

Figure 1: Limitations of pattern-based refactorings.

Notably, the limitations of syntax-driven refactorings have been observed in several works, resulting in an emerging trend to incorporate more *semantic* information into refactoring decisions, such as Abstract Syntax Tree (AST) type information, further preventing compilation errors and behaviour changes [36–38].

In this paper, we take a step further in this direction by proposing a fully semantic refactoring approach. There is a very broad space of methods that are able to reason about program semantics. The desire to perform refactorings safely suggests the use of techniques that overapproximate program behaviours. As one possible embodiment of semantics-driven refactoring, we leverage software verification technologies with the goal of reliably automating refactoring decisions based on program semantics, as in the case of the *Substitute Algorithm* refactoring. Our research hypothesis is that semantics-driven refactorings are more precise and can handle more complex code scenarios in comparison with syntax-driven refactorings.

Demonstrator: Refactoring Iteration over Collections. We use a particular refactoring as demonstrator for our idea. Nearly every modern Java application constructs and processes collections. A key algorithmic pattern when using collections is iteration over the

ESEC/FSE 2017, Paderborn, Germany

DOI: 10.1145/nnnnnnn.nnnnnnn

```
Integer result = null;
List<Integer> data = getData();
for (int el : data)
  if (el % 2 == 0) {
    result = el;
    break;
  }
```

Figure 2: Find element in list using external iteration.

contents of the collection. We distinguish *external* from *internal* iteration.

To enable external iteration, a Collection provides the means to enumerate its elements by implementing Iterable. Clients that use an external iterator must advance the traversal and request the next element explicitly from the iterator. External iteration has a few shortcomings:

- Is inherently sequential, and must process the elements in the order specified by the collection. This bars the code from using concurrency to increase performance.
- Does not describe the intended functionality, only that each element is visited. Readers must deduce the actual semantics, such as finding an element or transforming each item, from the loop body.

The alternative to external iteration is internal iteration, where instead of controlling the iteration, the client passes an operation to perform to an internal iteration procedure, which applies that operation to the elements in the collection based on the algorithm it implements. Examples of internal iteration patterns include finding an element by a user-provided predicate or transforming each element in a list using a provided transformer. In order to enable internal iteration, Java SE 8 introduces a new abstraction called *Stream* that lets users process data in a declarative way. The *Stream* package provides implementations of common internal iteration algorithms such as *foreach*, *find* and *sort* using optimised iteration orders and even concurrency where applicable. Users can thus leverage multicore architectures transparently without having to write multithreaded code. Internal iterations using *Stream* also explicitly declare the intended functionality through domain-specific algorithms. A call to Java 8 *find* using a predicate immediately conveys the code's intent, whereas an externally iterating **for** loop implementing the same semantics is more difficult to understand. Figures 2 and 3 illustrate this difference for the same *find* semantics. Finally, external iteration using a **for** loop violates Thomas' *DRY* principle (*"Don't repeat yourself"* [19]) if the intended functionality is available as a *Stream* template. Internal iteration through *Stream* thus eliminates code duplication.

For illustration, consider the example in Fig. 4 (a). This example uses external iteration to create a new list by multiplying all the positive values in the list `list` by 2. In this variant of the code, we use a **while** loop to sequentially process the elements in the list.

In Fig. 4 (b), we have re-written the code using streams. This variant of the code does not use a loop statement to iterate through the list. Instead, the iteration is done internally by the stream. Essentially, we create a stream of Integer objects via `Collection.stream()`,

```
List<Integer> newList = getData();
Optional<Integer> result = list.stream()
                           .filter(el -> el % 2)
                           .findFirst();
```

Figure 3: Find element in list using Java 8 Streams.

filter it to produce a stream containing only positive values, and then transform it into a stream representing the doubled values of the filtered list.

Goal of the paper. In this paper, we are interested in refactoring Java code handing collections through external iteration to use streams. Our refactoring procedure is based on the program semantics and makes use of program synthesis.

Contributions:

- We present a program synthesis based refactoring procedure for Java code that handles collections through external iteration.
- We present an abstraction for the Java Collection and Java Stream interfaces. This abstraction is tailored for refactorings.
- We have implemented our refactoring method in the tool Kayak. Our experimental results support our conjecture that semantics-driven refactorings are more precise and can handle more complex code scenarios than syntax-driven refactorings.

2 PRELIMINARIES

General refactorings. As we want to preserve generality, we are interested in refactorings that are correct independent of their context. To motivate our decision, let's look at the example in Fig. 5. We define a method `removeNeg` that removes the negative values in the list received as argument, which we later call for the list `data`. However, given that `data` contains only positive values, applying `removeNeg` does not have any effect.

Thus, for this particular calling context, we could refactor the body of `removeNeg` to a NO-OP. While this refactoring is correct for the code given in Fig. 5, it may cause problems during future evolution of the code as someone might use it for its original intended functionality (that of removing negative values). As we envision that our refactoring procedure will be used during the development process, we choose to not perform such strict refactorings.

Safety invariants. We assume a generic loop with a pre- and postcondition, guard G and transition relation T: $\{Precondition\}$ **while**(G) T $\{Postcondition\}$. For such a loop, we can prove partial correctness, i.e., any terminating execution starting in a state satisfying *Precondition* reaches a state satisfying *Postcondition*, by finding a *safety invariant*, *Inv*, with the following properties:

$$\exists Inv.\forall x, x'.Precondition \rightarrow Inv(x) \wedge \quad (1)$$

$$Inv(x) \wedge G(x) \wedge T(x, x') \rightarrow Inv(x') \wedge \quad (2)$$

$$Inv(x) \wedge \neg G(x) \rightarrow Postcondition \quad (3)$$

```
Iterator<Integer> it=list.iterator();
List<Integer> newList=new ArrayList<Integer>();
while (it.hasNext()) {
  int el=it.next().intValue();
  if (el > 0)
      newList.add(2 * el);
}
```
(a)

```
List<Integer> newList=new ArrayList<Integer>();
newList=list.stream()
          .filter(el -> el>0)
          .map(el -> new Integer(2 * el));
          .collect(toList());

return newList;
```
(b)

Figure 4: Filtering and mapping example with external (a) vs. internal (b) iteration.

```
void removeNeg(ArrayList<Integer> l) {
  Iterator<Integer> it = l.iterator();
  while (it.hasNext())
    if (it.next() < 0) it.remove();
}
List<Integer> data = new ArrayList<>();
Collections.addAll(data, 1, 2, 3);
removeNeg(data);
```

Figure 5: Filter example.

In this formula, (1) ensures that the safety invariant holds in the initial state, (2) checks that the invariant is inductive with respect to the transition relation, i.e. the transition relation maintains the invariant, and (3) ensures that the invariant establishes the post-condition on exit from the loop. This can be generalised to multiple, potentially nested, loops.

Symbolic execution. Symbolic execution examines the semantics of a given program statically by computing a symbolic model of the program's possible states [28]. Model checkers like CBMC [9] employ symbolic execution to map a program's semantics to a SAT formula which is satisfiable iff a certain property about the program semantics holds. This technique is used to find bugs in programs, such as an assertion violation or a null pointer dereference. CBMC's symbolic execution by default limits the number of iterations for loops in the program in order to produce a finite SAT formula in the context of possibly unbounded loops. By introducing loop invariants CBMC's analysis can be generalised to programs with unbounded loops. A common property to check using symbolic execution is the functional equivalence of two programs or functions under any possible input [15], which permits to prove the soundness of refactorings and other code mutations.

3 OVERVIEW OF OUR APPROACH

Given an *original code* Origin, we want to infer the *refactored code* Stream such that, for any initial program state S_i, Origin and Stream produce the same final state, i.e., they are observationally equivalent. We consider a *program state* to consist of assignments to all the scalar variables plus a heap representation mapping all the Java reference variables to their corresponding heap addresses. Then:

$$\forall S_i.S_f{=}\text{Origin}(S_i) \wedge S'_f{=}\text{Stream}(S_i) \Rightarrow S_f{=}S'_f \quad (4)$$

This equivalence check can be reduced to checking the partial correctness of the triple

$$\{S_i{=}*\}\ \text{Origin}\ \{\text{Stream}(S_i){=}S_f\}$$

where $S_i{=}*$ means that we non-deterministically pick the initial state (i.e., we non-deterministically assign all the variables and the contents of collections). Essentially, this says that, starting with a nondeterministic state S_i, every terminating trace ends up in a state where $\text{Stream}(S_i){=}S_f$ holds (we discuss non-terminating behaviours in the last paragraph of Sec. 6).

Note that this overapproximates the context of the initial code in the sense that it may require us to consider more initial states than those reachable at the start of Origin in the user code. As a consequence, we obtain general refactorings (see Sec. 2). In the rest of the paper we use the notation *equivState* to refer to the equivalence postcondition $\text{Stream}(S_i){=}S_f$. Next, we explain the main steps of our refactoring procedure:

(i) Given the original code and a nondeterministic initial state as inputs, we generate the constraint $Post(S_i, \text{Origin}) \Rightarrow equivState$ encoding the observational equivalence check. Here, *Post* computes the postcondition of Origin starting from the initial program state S_i. We compute *Post* by symbolically executing the code [9]. For programs with loops we assume the existence of safety invariants and generate constraints as shown in Sec 2. These safety invariants, *Inv*, are synthesised together with *equivState* in the next step.

(ii) We provide the equivalence constraints to our program synthesiser (see Sec. 6), which generates an equivalence proof in the form of *equivState* and the necessary safety invariants. As *equivState* captures the semantics of the refactored code, Stream can be generated directly from it. Moreover, given that *equivState* is a postcondition of the original code, the refactored code is *guaranteed* to be equivalent to the original one by construction.

Logical encoding. In order to generate the constraints at point (i), we must identify a logical encoding for our analysis, which we use to express *Inv* and *equivState*. As *equivState* must capture the semantics of the stream refactoring as well as equivalence between program states, our logic must have the ability to express: (1) operations supported by the Java Collection interface, (2) operations supported by the Java Stream interface, as well as (3) equality between collections (for lists this implies that we must be able to reason about both content of lists and the order of elements).

For this purpose, we define the Java Stream Theory (JST). At this point, we informally present JST in Fig. 6 (the formal description will be given in Sec. 5). For brevity, we omit some of the operations that have the same semantics as their direct counterpart provided

by the Java Collection or Stream API. Additionally, we use the notion of incomplete collection/list represented by a *list segment* $x \rightarrow^* y$, i.e., the list starting at the node pointed by x and ending at the node pointed by y.

Throughout the paper we take the liberty of referring to collections as *lists* (we will explain in Sec. 5 why, for the purpose of our analysis, lists can be used as a representation for other types of collections as well, e.g. sets). Also note that we capture side-effects by explicitly naming the current heap – heap variables h, h' etc. are being introduced (as a front-end transformation), denoting the heap in which each function is to be interpreted. The mutation operators (e.g. *get*, *add*, *set*, *remove*) then become pure functions mapping heaps to heaps.

We illustrate JST through the graphical representation given in Figure 7, where the circles denote the nodes in the list with their associated values. We use the dashed arrows to represent the list references. Note that heap H_2 returned by the *filter* method contains both the list received as argument and the result list l.

3.1 Discussion on aliasing

In the overview of our approach (Sec. 3), when expressing equivalence between program states, we only consider the variables (and collections) that are accessed by Origin (as opposed to all the live program variables). Thus, one might wonder if there aren't any side-effects due to aliasing that we are not considering. The answer is no, our approach is safe for reference variables as well as the only two potential aliasing scenarios involving a reference variable p that is not directly used by Origin, which are the following:

1. p points to a collection that is modified by Origin. As the Stream refactoring is going to perform an equivalent transformation in-place, the refactoring will be transparent to p.
2. p is an iterator over a collection accessed by Origin. Then, if Origin modifies the collection, so will Stream, which will result in p being invalidated in both scenarios. Contrary, if Origin does not modify the collection, neither will Stream, and p will not be affected in either one of the cases.

Next, we illustrate scenario 1 by considering again method removeNeg in Fig. 5 with the following calling context, where we assume p points to some list and we create an alias p' of p:

```
ArrayList<Integer> p' = p;
removeNeg(p);
```

At a first glance, a potential refactoring for removeNeg is:

```
l = l.stream().filter(el -> el>=0)
      .collect(toList());
```

However, this is incorrect when using the refactored function in the calling context mentioned above: While the list p points to is correct, the list pointed by p' is not updated. Thus, after the call to removeNeg, p will correctly point to the filtered list, whereas p' will continue pointing to the old unfiltered list. To avoid such situations, we perform refactorings of code that mutates collections in-place. Thus, a correct refactoring for method *removeNeg* is:

```
ArrayList<Integer> copy = new ArrayList<>(l);
l.clear();
copy.stream().filter(el -> el>=0)
             .forEachOrdered(l::add);
```

Here, we first create a copy *copy* of l. After performing the filtering on *copy*, we use *forEachOrdered*, provided by the Stream API, to add each element of the temporary stream back to the list pointed to by l (in the order encountered in the stream). Thus, we are not creating a new list with a new reference, but using the original one, which makes the refactoring transparent to the rest of the program, regardless of potential aliases.

4 MOTIVATING EXAMPLES

In this section, we illustrate our refactoring procedure on three examples.

First example. We start with the one in Fig. 4, where we create a new list by multiplying by 2 each positive value in the list list. As aforementioned, we must first introduce heap variables that capture the side-effects. For this purpose, we will use the following naming convention: the heap before executing the code (i.e., the initial heap for both the original and the refactored code) is called h_i. All the other heaps manipulated by the original program have subscript *o*, and those manipulated by the stream in *equivState* have subscript *s*.

```
Iterator<Integer> it = iterator(h_i,list);
List<Integer> newList;
h_o = new ArrayList<Integer>(h_i, newList);
while (hasNext(h_o, it)) {
  int (el, h_o) = next(h_o, it).intValue();
  if (el > 0) {
    h_o = add_last(h_o, newlist, 2 * el);}}
```

We check the program state equivalence captured by *equivState* by verifying that:

- the heap states reached by executing the original code and the refactored code, respectively, are equivalent (denoted as $h_0{=}h'_s$ below). We will formally define heap equivalence as graph isomorphism in the next section (Def. 5.2). Informally, it means that all the lists reachable from reference variables used by the original code (except for local variables not visible outside the original code), must be equal.
- all the scalar program variables accessed by the original code must have equal values after the execution of the original code and the refactored one, respectively. Again, we do not consider local variables that are not visible outside it (e.g. el in the original code in Fig.4).

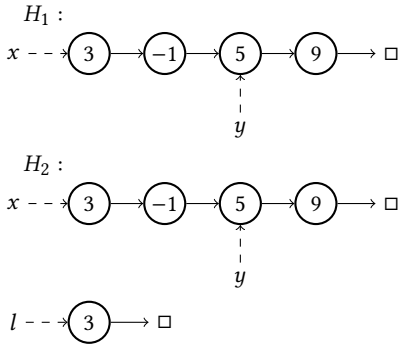
For this example, as there are no scalar variables handled by the code and visible outside, we only check heap equivalence. Thus, we find the following *equivState*:

$$\begin{aligned} &\mathit{equivState}(h_i, h_o, \mathit{list}) = (h_o{=}h'_s \wedge \\ &\quad h_s{=}\mathit{filter}(h_i, \mathit{list}, \mathbf{null}, \lambda v.v > 0, \mathit{list}') \wedge \\ &\quad h'_s{=}\mathit{map}(h_s, \mathit{list}', \mathbf{null}, \lambda v.2{\times}v, \mathit{list}'')) \end{aligned}$$

The above says that the heap h_o generated by the original code is equivalent to the heap h'_s generated by applying *filter* and *map* directly. Then, the safety invariant required to prove *equivState* is identical with *equivState* with the exception that it considers that the list pointed by *list* has only been partially processed (up to the

$alias(h, x, y)$:	do x and y point to the same node in heap h?
$size(h, x, y)$:	what is the length of the list segment from x to y in h?
$get(h, x, i)$:	what is the value stored in the i-th node of the list pointed by x in heap h?
$h' = add(h, x, i, v)$:	obtain h' from h by inserting value v at position i in the list pointed by x.
$h' = add\_last(h, x, v)$:	equivalent to $add(h, x, size(h, x, \textbf{null}), v)$
$h' = set(h, x, i, v)$:	obtain h' from h by setting the value of the i-th element in the list pointed by x to v.
$h' = remove(h, x)$:	obtain h' from h by removing the node pointed by x. In h', x and all its aliases will point to the successor of the removed node.
$h' = removeVal(h, x, y, v)$:	obtain h' from h by removing the node with value v from the list segment $x \rightarrow^* y$.
$exists(h, x, y, \lambda v.P(v))$:	is there any value v in the list segment $x \rightarrow^* y$ such that $P(v)$ holds?
$forall(h, x, y, \lambda v.P(v))$:	is it the case that for all values $v_1 \dots v_n$ in the list segment $x \rightarrow^* y$, $P(v_1) \dots P(v_n)$ hold?
$h' = sorted(h, x, y, ret)$:	obtain h' from h by sorting the elements stored in the list segment $x \rightarrow^* y$ in the list ret (h' will contain both the list segment $x \rightarrow^* y$ and the list ret).
$h' = filter(h, x, y, \lambda v.P(v), ret)$:	obtain h' from h by creating a new list ret containing all the elements in the list segment $x \rightarrow^* y$ that match the predicate P.
$max(h, x, y)$:	what is the maximum value stored in the list segment $x \rightarrow^* y$?
$min(h, x, y)$:	what is the minimum value stored in the list segment $x \rightarrow^* y$?
$h' = map(h, x, y, \lambda v.f(v), ret)$:	obtain h' from h by applying the mapping function f to each value in the list segment $x \rightarrow^* y$ and storing the result in the list pointed by ret.
$h' = skip(h, x, y, done, n, ret)$:	obtain h' by creating a new list ret containing the remaining elements of the list segment $x \rightarrow^* y$ after discarding the first n elements ($done$ denotes the number of elements that were already skipped).
$h' = limit(h, x, y, done, n, ret)$:	obtain h' by creating a new list ret containing the elements of the list segment $x \rightarrow^* y$, after its length was truncated to n ($done$ denotes the number of elements that were dropped).
$reduce(h, x, y, v, \lambda a\ b.f(a, b))$:	performs a reduction on the elements of the list segment $x \rightarrow^* y$, using the identity value v and the accumulation function f, and returns the reduced value.
$h' = concat(h, x, y, a, b, ret)$:	obtain h' from h by creating a new list ret containing all the elements in the list segment $x \rightarrow^* y$ followed by all the elements in the list segment $a \rightarrow^* b$.
$h' = copy(h, x, y, ret)$:	obtain h' by creating a new list ret that contains the elements of the list segment $x \rightarrow^* y$.
$h' = new(h, x)$	obtain h' from h by assigning x to point to **null**.
$equalLists(h, x, y, h', a, b)$	is list segment $x \rightarrow^* y$ in heap h equal to list segment $a \rightarrow^* b$ in heap h' (i.e., do they contain the same elements in the same order)?
$h' = getIterator(h, x, i, it)$	obtain heap h' by creating a new iterator it that points to the i-th element in the list pointed-to by x.

Figure 6: Informal Description of JST.



$alias(H_1, x, y)$ = false
$forall(H_1, x, y, \lambda v.v \leq 4)$ = false
$exists(H_1, x, \textbf{null}, \lambda v.v = -1)$ = true
$size(H_1, x, \textbf{null})$ = 4
$min(H_1, x, y) = -1$
$max(H_1, x, y) = 9$
$H_2 = filter(H_1, x, y, \lambda v.v > 0, l)$
$forall(H_2, l, \textbf{null}, \lambda v.v > 0)$ = true

Figure 7: JST Example.

iterator it):

$Inv(h_i, h_o, list, it) = (h_o = h'_s \wedge$ $h_s = filter(h_i, list, it, \lambda v.v > 0, list') \wedge$

$$h'_s{=}map(h_s, list', \textbf{null}, \lambda v.2{\times}v, newList))$$

The invariant states that, given the iterator *it* over the list *list*, after processing the list up until *it*, both the original code and the stream postcondition generate the same heaps.

As JST directly models the Java Streams semantics, from a *equivState* postcondition we generate stream code (see Fig 4 (b)).

Second example. Next, we provide a more involved example where the original code has nested loops. For this purpose we use the code for selection sort in Fig. 8 (a). First, we introduce the heap variable as shown in Fig. 8 (b). If Inv_{out} and Inv_{in} are the safety invariants for the outer and inner loops, respectively, then the constraints for the outer loop are (we omit the inner loop as it follows directly from the equations (1), (2), (3) in Sec. 2):

$$\forall h_i, h_o, l, j.\exists min.Inv_{out}(h_i, h_o, l, 0) \wedge \quad (5)$$
$$(Inv_{out}(h_i, h_o, l, j) \wedge j{<}(size(h_o, l){-}1) \wedge \quad (6)$$
$$Inv_{in}(h_i, h_o, l, size(h_o, l), j, min) \wedge \quad (7)$$
$$temp{=}get(h_o, l, j) \wedge h'_o{=}set(h_o, l, j, get(h_o, l, min)) \wedge \quad (8)$$
$$h_o{=}set(h'_o, l, min, temp)) \Rightarrow Inv_{out}(h_i, h_o, l, j{+}1) \wedge \quad (9)$$
$$Inv_{out}(h_i, h_o, l, j) \wedge j{\geq}(size(h_o, l){-}1) \Rightarrow equivState(h_i, h_o, l) \quad (10)$$

Constraint (5) says that the outer loop's invariant must hold in the initial state, constraints (6), (7), (8) and (9) check that Inv_{out} is re-established by the outer loop's body (by making use of Inv_{in}), whereas (10) asserts that the *equivState* postcondition must hold on exit from the outer loop. For this example, we find the following solution:

$$Inv_{out}(h_i, h_o, l, j) = equalLists(h'_o, l, it_{lj}, h'_s, l_s, it_{lsj}) \wedge$$
$$h_s = sorted(h_i, l, \textbf{null}, l_s) \wedge$$
$$h'_o = getIterator(h_o, l, j, it_{lj}) \wedge$$
$$h'_s = getIterator(h_s, l_s, j, it_{lsj}) \wedge$$
$$max(h'_o, l, it_{lj}){\leq}min(h'_o, it_{lj}, \textbf{null})$$
$$Inv_{in}(h_i, h\_l, i, j, min) = (min(h''_o, it_{lj}, it_{li}){=}\min \wedge$$
$$h'_o{=}getIterator(h_o, l, j, it_{lj}) \wedge h''_o{=}getIterator(h_o, l, i, it_{li}))$$
$$equivState(h_i, h_o, l) = (h_o{=}h_s \wedge$$
$$h_s{=}sorted(h_i, l, \textbf{null}, l))$$

The invariant of the outer loop expresses the fact that the lists sorted through external iteration and via the stream operation, respectively, are equal up until element *j*. As our theory JST supports iterator-based equality between lists (rather than index-based), we need to create iterators it_{ls} and it_{lsj} to the *j*-th element in the lists *l* and l_s, respectively. Additionally, the invariant of the outer loop captures the fact that the maximum element in the already sorted portion of list *l* is at most equal to the minimum element from the portion still to be sorted.

The inner loop's invariant captures the fact that the minimum element in the list segment between the *j*-th and the *i*-th element is min (program variable). The postcondition *equivState* captures the equality between the final heap in the original program, h_o, and the final heap in the refactored code, h_s.

From postcondition *equivState* we generate the following refactored code, where we modify *l* in-place by using a local copy.

```
List<Integer> sorting(List<Integer> l){
  List<Integer> copy = new List<>(l);
  l.clear();
  copy.stream().sorted()
              .forEachOrdered(l::add);}
```

Third example. In this example, we illustrate an aggregate refactoring, as well as the importance of checking equivalence between heap states. For this purpose, we use the code below, where we compute the sum of all the elements in the list pointed-to by *l*, while at the same time removing from the list pointed-to by *p* a number of elements equal to the size of *l*.

```
Iterator<Integer> it = p.iterator();
int sum = 0;
for(i = 0; i<l.size(); i++) {
  sum += l.get(i);
  if(it.hasNext()) {
    it.next();
    it.remove();
}}
```

If we were to only verify that the scalar variables after executing the original and the refactored code, respectively, are equal, and omit checking heap equivalence, then the following refactoring would be considered correct:

```
sum = l.stream().reduce(0, (a b)->a+b);
```

This refactoring ignores the modifications performed to list *p* and only computes the sum of elements in the list pointed-to by *l*. In our case, we correctly find this refactoring to be unsound as the heap state reached after executing the original code (where *p* points to a modified list) is not equivalent to the one reached after executing this refactoring (where *p* points to the unmodified list). Instead, we find the following refactoring, where we correctly capture the mutation of *p*:

```
sum = l.stream().reduce(0, (a b)->a+b);
ArrayList<Integer> copy = new ArrayList<>(p);
p.clear();
copy.stream().skip(l.size())
              .forEachOrdered(p::add);
```

5 JAVA STREAM THEORY

We designed JST such that it meets several criteria:

1. Express operations allowed by the Java Collection interface, operations allowed by the Java Stream interface as well as equality between collections (for lists this implies that we must be able to reason about both content of lists and the order of elements).
2. JST must be able to reason about the content and size of partially constructed lists (i.e., list segments), which are required when expressing safety invariants. For illustration, in Fig. 4, the safety invariant captures the fact that h_s is obtained from h_i by filtering the list segment $list{\rightarrow}^*it$.
3. JST must enable concise *equivState* postconditions and invariants as we use program synthesis to infer these. Thus, the smaller they are, the easier to synthesise.

```
void sorting(List<Integer> l) {                    (a)
  int min, temp;
  for (int j = 0; j < l.size()-1; j++) {
    min = j;
    for (int i = j+1; i < l.size(); i++)
      if (l.get(i)<l.get(min)) min = i;


    temp = l.get(j);
    l.set(j, l.get(min));
    l.set(min, temp);}}
```

```
void sorting(List<Integer> l) {                    (b)
  int min, temp;
  h_o = copyHeap(h_i);
  for (int j = 0; j < size(h_o,l)-1; j++) {
    min = j;
    for (int i = j+1; i < size(h_o,l); i++)
      if (get(h_i,l,i)<l.get(h_o,l,min)) min = i;

    temp = get(h_o,l,j);
    h_o' = set(h_o, l, j, get(h_o, l, min));
    h_o = set(h_o', l, min, temp);}}
```

Figure 8: Selection sort: (a) original code (b) with explicit heap variables.

To the best of our knowledge, there is no existing logic that meets all the criteria above. The majority of recently developed decidable heap logics are not expressive enough (fail points 1 and 2) [5, 6, 11, 21, 29, 32], whereas very expressive logics such as FOL with transitive closure are not concise and easily translatable to stream code (fail point 3).

While our theory is undecidable, we found it works well for our particular use case.

Semantics. We first define the model used to interpret JST formulae. The set of reference variables is denoted by PV. Note that, as already mentioned in the paper, these reference variables are those accessed in the code to be refactored (as opposed to all the reference variables in the program).

Definition 5.1 (Heap). A heap over reference variables PV is a tuple $H = \langle G, L_P, L_D \rangle$. G is a graph with vertices $V(G)$ and edges $E(G)$, $L_P : PV \rightarrow V(G)$ is a labelling function mapping each reference variable to a vertex of G and $L_D : V(G) \rightarrow D$ is a labelling function associating each vertex to its data value (where D is the domain of the data values).

Given that we are interested in heaps managed by Java Collections, we restrict the class of models to those where each vertex has outdegree 0 or 1 (i.e. we cannot have multiple edges coming out of a node). We assume that the reference variables include a special name **null**.

Function $val(h, x)$ returns the value stored in the node pointed by x, $next(h, x)$ returns a reference to the next node after the one pointed by x and it is defined as the unique vertex such that $(x, next(x)) \in E(h)$, and $add0(h, e, x)$ returns the heap obtained by appending element e at the beginning of the list pointed by x. For the latter we provide the pointwise definition:

$$add0_V(h, e, x) \stackrel{\text{def}}{=} V(h) \cup \{q\} \quad \text{where } q \text{ is a fresh vertex}$$

$$add0_{L_D}(h, e, x) \stackrel{\text{def}}{=} L_D(h)[q \mapsto e]$$

$$add0_E(h, e, x) \stackrel{\text{def}}{=} E(h) \cup \{(q, L_P(h)(x))\}$$

The semantics of JST is defined recursively in Fig. 9. Note that functions *minimum* and *maximum* return the minimum and maximum between the values receives as arguments, respectively. While in Fig. 9 we provide the semantics for index-based operations (e.g. $set(x, y, i, v)$), we also support iterator-based ones (e.g. h'=$set(h, it, v)$ returns the heap obtained by setting the value of the node pointed by it to v in heap h).

One important check that we must be able to perform in order to prove equivalence between program states is is that of heap equivalence. In order to define this notion we first assign $PV_\cap$ to be the set of reference variables that are used by both the original code and the refactored one (this excludes local variables such as iterators that are used by only one of the codes). Then:

Definition 5.2. Heap h and h' are equivalent, written as h=h', iff the underlying graphs reachable from $PV_\cap$ are isomorphic.

Intuitively, this means that all the lists in h and h' pointed-to by the same variable from $PV_\cap$, respectively, are equal.

Set collections. For our refactoring procedure, we use lists as the internal representation for collections denoting sets, meaning that we impose an order on the elements of sets. While we may miss some refactorings, this procedure is sound: if two collections are equal with respect to some ordering, they are also equal when no order is imposed.

6 SYNTHESISING REFACTORINGS

We compute the postcondition *equivState* and safety invariants by using a program synthesis engine. Such engines are used increasingly in program verification [11, 34]. Our program synthesiser makes use of Counter-Example Guided Inductive Synthesis (CEGIS) [35] for stream refactoring. We present its general architecture followed by a description of the parts specific to refactoring.

General architecture of the program synthesiser. The design of our synthesiser is given in Fig. 10 and consists of two phases, SYNTHESISE and VERIFY. We will illustrate each of these phases by using as running example the first motivational example in Sec. 4. Our goal is to synthesise a solution $(equivState, Inv)$.

We start with a vacuous SYNTHESISE phase, where we generate a random candidate solution, which we pass to the VERIFY phase. For this example, let's assume that the random solution says that the stream manipulated heap is the same as the initial one (i.e. the stream code does not affect the heap): $equivState(h_i, h_o, list, it) = h_o{=}h'_s \wedge h'_s{=}h_s{=}h_i$.

$$alias(h,x,y) = (L_P(h)(x)==L_P(h)(y)) \quad val(h,x) = L_D(h)(x)$$

$$\frac{alias(h,x,y)}{size(h,x,y) = 0} \quad \frac{\neg alias(h,x,y)}{size(h,x,y) = 1+size(h,next(h,x),y)}$$

$$\frac{i == 0}{get(h,x,i){=}val(h,x)} \quad \frac{i > 0}{get(h,x,i){=}get(h,next(h,x),i{-}1)}$$

$$\frac{alias(h,x,y)}{exists(h,x,y,\lambda v.P(v)) = false} \quad \frac{\neg alias(h,x,y)}{exists(h,x,y,\lambda v.P(v)) = P(val(h,x)) \vee exists(h,next(h,x),y,\lambda v.P(v))}$$

$$\frac{alias(h,x,y)}{forall(h,x,y,\lambda v.P(v)) = true} \quad \frac{\neg alias(h,x,y)}{forall(h,x,y,\lambda v.P(v)) = P(val(h,x)) \wedge forall(h,next(h,x),y,\lambda v.P(v))}$$

$$\frac{alias(h,x,y)}{max(h,x,y) = -\infty} \quad \frac{\neg alias(h,x,y)}{max(h,x,y) = maximum(P(val(h,x)), max(h,next(h,x),y))}$$

$$\frac{alias(h,x,y)}{min(h,x,y) = \infty} \quad \frac{\neg alias(h,x,y)}{min(h,x,y) = minimum(P(val(h,x)), min(h,next(h,x),y))}$$

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h) \cup \{ret \mapsto \mathbf{null}\}]}{copy(h,x,y,ret){=}h'} \quad \frac{\neg alias(h,x,y) \wedge h' = copy(h,next(h,x),y,ret)}{copy(h,x,y,ret){=}add0(h',val(h,x),ret)}$$

$$\frac{h' = copy(h,x,\mathbf{null},l)}{add(h,x,0,v){=}add0(h',v,l)} \quad \frac{i > 0 \wedge h' = add(h,next(h,x),i{-}1,v)}{add(h,x,i,v){=}add0(h',val(h,x),next(h',x))}$$

$$\frac{h' = copy(h,next(h,x),\mathbf{null},l)}{set(h,x,0,v){=}add0(h',v,l)} \quad \frac{i > 0 \wedge h' = set(h,next(h,x),i{-}1,v)}{set(h,x,i,v){=}add0(h',val(h,x),next(h',x))}$$

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{sorted(h,x,y,ret) = h'}$$

$$\frac{\neg alias(h',x,y) \wedge h'{=}sorted(removeVal(h,x,y,min(h,x,y)),x,y,ret)}{sorted(h,x,y,ret) = add0(h',min(h,x,y),ret)}$$

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{filter(h,x,y,\lambda v.P(v),ret){=}h'} \quad \frac{\neg alias(h,x,y) \wedge \neg P(val(h,x))}{filter(h,x,y,\lambda v.P(v),ret){=}filter(h,next(h,x),y,\lambda v.P(v),ret)}$$

$$\frac{\neg alias(h,x,y) \wedge P(val(h,x)) \wedge h'{=}filter(h,next(h,x),y,\lambda v.P(v),ret)}{filter(h,x,y,\lambda v.P(v),ret) = add0(h',val(h,x),ret)}$$

$$\frac{alias(h,x,y) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{map(h,x,y,\lambda v.f(v),ret){=}h'} \quad \frac{\neg alias(h,x,y) \wedge h'{=}map(h,next(h,x),y,\lambda v.f(v),ret)}{map(h,x,y,\lambda v.f(v),ret) = add0(h',f(val(h,x)),ret)}$$

$$\frac{alias(h,x,y)}{reduce(h,x,y,v,\lambda a\ b.f(a,b)) = v} \quad \frac{\neg alias(h,x,y)}{reduce(h,x,y,v,\lambda a\ b.f(a,b)) = f(v,reduce(h,next(h,x),y,v,\lambda a\ b.f(a,b)))}$$

$$\frac{\neg alias(h,x,y) \wedge n{=}0 \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}L_P(h)(x)\}]}{skip(h,x,y,done,n,ret){=}h'} \quad \frac{alias(h,x,y){\wedge}h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{skip(h,x,y,done,n,ret){=}h'}$$

$$\frac{\neg alias(h,x,y) \wedge n{>}0}{skip(h,x,y,done,n,ret){=}skip(h,next(h,x),y,done{+}1,n{-}1,ret)}$$

$$\frac{(alias(h,x,y) \vee n{=}0) \wedge h'{=}h[L_p(h'){=}L_P(h){\cup}\{ret{\mapsto}\mathbf{null}\}]}{limit(h,x,y,done,n,ret){=}h'}$$

$$\frac{\neg alias(h,x,y) \wedge n{>}0 \wedge h'{=}limit(h,next(h,x),y,done{+}1,n{-}1,ret)}{limit(h,x,y,done,n,ret){=}add0(h',val(h,x),ret)}$$

$$\frac{alias(h_1,x,y) \wedge \neg alias(h_2,a,b)}{equalLists(h_1,x,y,h_2,a,b) = false} \quad \frac{\neg halias(h_1,x,y) \wedge alias(h_2,a,b)}{equalLists(h_1,x,y,h_2,a,b) = false}$$

$$\frac{halias(h_1,x,y) \wedge alias(h_2,a,b)}{equalLists(h_1,x,y,h_2,a,b) = true} \quad \frac{\neg alias(h_1,x,y) \wedge \neg alias(h_2,a,b)}{val(h_1,x) == val(h_2,a) \wedge equalLists(h_1,next(h_1,x),y,h_2,next(h_2,a),b)}$$

Figure 9: Inference rules for Java Collection Theory.

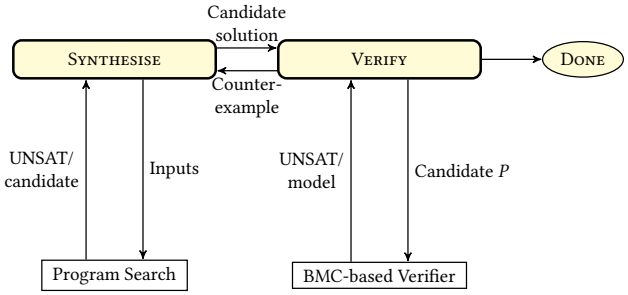


Figure 10: The refactoring refinement loop.

In the VERIFY phase, we check whether the candidate solution is indeed a true solution for our synthesis problem (then we are "DONE"), or compute a counterexample. We find such a counterexample by building a program P_{verif}, on which we run Bounded Model Checking (BMC) [4]. BMC employs symbolic execution to map program semantics to a SAT instance [9] which verifies our equivalence constraints. If we manage to prove partial correctness of P_{verif}, then we are done. Otherwise, we provide the counterexample returned by BMC to the SYNTHESISE phase. Note that it is sound to use BMC because the program P_{verif} does not contain loops as it uses loop invariants. For the running example, BMC returns a counterexample with initial heap h'_{ce} where the candidate *equivState* is not a true postcondition when *list* contains value 1 (added at position 0 through $add(h_{ce}, list, 0, 1)$): $h_{ce} = new(h_i, list) \ \wedge \ h'_{ce} = add(h_{ce}, list, 0, 1)$.

Next, in the SYNTHESISE phase, we add the counterexample from the previous phase to Inputs and search for a new candidate solution by constructing a program P_{synth} on which we run in parallel BMC and a genetic algorithm (GA) to find a new candidate solution that holds for all the Inputs. GA simulates an evolutionary process using selection, mutation and crossover operators. Its fitness function is determined by the number of passed tests. GA maintains a large population of programs which are paired using crossover operation, combining successful program features into new solutions. In order to avoid local minima, the mutation operator replaces instructions by random values at a comparatively low probability. Moreover, we use a biased crossover operation, selecting parents that solve distinct counterexample sets for reproduction. We use the result of either BMC or GA, depending on which one returns first. Again, it is sound to use BMC as the program P_{synth} does not contain loops. For the running example, BMC returns first with a candidate solution saying that the heap h'_s after the stream code is the following (for brevity, we omit the general *equivState* as, similar to before, it only denotes the fact that the original and the stream heaps are equivalent; we also omit the invariant which is very similar to *equivState*): $h_s = filter(h_i, list, \mathbf{null}, \lambda v.\text{true}, list') \wedge \quad h'_s = map(h_s, list, \mathbf{null}, \lambda v.2{\times}v, list'')$

This solution is almost correct, apart from the *filter* predicate, which does no actual filtering as the predicate is *true*. Returning to the VERIFY phase, we find one further counterexample denoting a list with value 0 (which should be filtered out but it isn't): $h_{ce}{=}new(h_i, list) \ \wedge \ h'_{ce}{=}add(h_{ce}, list, 0, 0)$.

Back in the SYNTHESISE phase, this counterexample refines the $filter$ predicate, leading to the next solution:

$h_s = filter(h_i, list, \mathbf{null}, \lambda v.v \neq 0, list') \wedge$
$h'_s = map(h_s, list, \mathbf{null}, \lambda v.2{\times}v, list'')$

Still not matching the original algorithm, the VERIFY phase provides one final counterexample (a list containing value −2 that should be filtered out, but it isn't):

$$h_{ce}{=}new(h_i, list) \ \wedge \ h'_{ce}{=}add(h_{ce}, list, 0, -2)$$

In the final SYNTHESISE phase we get the solution provided in Sec. 4.

Elements specific to stream refactoring. In order to use program synthesis for stream refactoring, we required the following:

(i) *The target instruction set is JST*, which requires both the VERIFY and SYNTHESISE phases in the program synthesiser to support the JST transformers. JST directly models Java Streams such that, once the synthesiser finds a solution *equivState*, we only require very light processing to generate valid Java Stream code. In particular, this processing involves the stream generation (see examples below).

Some examples of the generated stream code are provided below, where the LHS denotes either the stream heap h_s or some other scalar variable r captured by *equivState* (expressed in JST), and the RHS represents the corresponding stream refactoring. For illustration, in the first example, after the synthesiser finds that h_s in *equivState* is $h_s{=}filter(h_i, l, \mathbf{null}, \lambda v.P(v), l')$, we generate the stream refactoring by adding the stream generation $l.stream()$ before the stream filtering $filter(\lambda v.P(v))$.

Note that ≡ stands for reference equality. This means that, as shown in Sec. 3.1, we must generate Java code that modifies the original collection in place.

$h_s{=}filter(h_i, l, \mathbf{null}, \lambda v.P(v), l') \Rightarrow l' \equiv l.stream().filter(\lambda v.P(v))$
$h_s{=}sorted(h_i, l, \mathbf{null}, l') \Rightarrow l' \equiv l.stream().sorted()$
$h_s{=}skip(h_i, l, \mathbf{null}, k, 0, l') \Rightarrow l' \equiv l.stream().skip(k)$
$r{=}forall(h_i, l, \mathbf{null}, \lambda v.P(v)) \Rightarrow r{=}l.stream().allMatch(v \rightarrow P(v))$
$r{=}max(h, l, \mathbf{null}) \Rightarrow r{=}l.stream().max()$

(ii) *The search strategy*: we parameterise the solution language, where the main parameter is the length of the solution program, denoted by l. At each iteration we synthesise programs of length exactly l. We start with $l = 1$ and increment l whenever we determine that no program of length l can satisfy the specification. When we do successfully synthesise a program, we are *guaranteed that it is of minimal length* since we have previously established that no shorter program is correct. This is particularly useful for our setting, where we are biased towards short refactorings (see Sec. 8).

Terminating and exceptional behaviour. Next, we discuss how our refactoring interacts with non-terminating and exceptional behaviours of the original code.

If the original code throws an exception, then the same happens for our modelling, and thus we fail to find a suitable refactoring. The non-terminating behaviour can be due to either iterating over a collection with an unbounded number of elements or to a bug in the code that does not properly advance the iteration through the collection. Regarding the former, we assume that the code to

be refactored handles only collections with a bounded number of elements. With respect to the second reason for non-termination, if such a bug exists in the original code, then it will also exist in our modelling. Thus, we will fail to find a suitable refactoring.

7 EXPERIMENTS

Benchmark Selection. We provide an implementation of our refactoring decision procedure, which we have named Kayak. We employed the GitHub Code Search to find relevant Java classes that contain integer collections with refactoring opportunities to streams. Kayak currently supports refactorings from Java external iterators to Streams for integer collections only. This limitation is not conceptual, but rather due to our Java front-end based on CBMC [9], which will be extended in future work. The queries were specified conservatively as to not exceed the CBMC front-end capabilities and we manually ruled out search results which cannot be implemented using the Java 8 Stream specification. We used the following search queries on 8/8/2016:

- List<Integer>+**for**+**if**+**break**++language%3AJava&type=Code
- List<Integer>+**while**+it+remove&type=Code
- List<Integer>+**while**+add

We found 50 code snippets with loops from the results that fit these restrictions.

Experimental Setup. In order to validate our hypothesis that semantics-driven refactorings are more precise than syntax-driven ones, we compare Kayak against the Integrated Development Environments IntelliJ IDEA 2016.1[1] and NetBeans 8.2[2], as well as against LambdaFicator by Franklin et al. [13]. These tools all provide a "Replace with collect" refactoring, which matches Java code against pre-configured external iteration patterns and transforms the code to a stream expression if they concur. We manually inspect each transformation for both tools to confirm correctness. Since Kayak's software synthesis can be a time-consuming process, we impose a time limit of 300 s for each benchmark. All experiments were run on a 12-core 2.40 GHz Intel Xeon E5-2440 with 96 GB of RAM.

Genetic Algorithm Configuration. We implemented a steady state genetic algorithm implementation in CEGIS, whose fitness function is determined by the number of passed tests. We employ a biased crossover operation, selecting parents which solve distinct counterexamples in the CEGIS counterexample set for reproduction. The intent is to have parent refactorings which work for distinct intput sets produce offspring which behave correctly for both input sets. The population size, replacement and mutation rates are configurable and were set to 2000, 15% and 1% respectively for our experimental evaluation.

Results. Our results show that Kayak outperforms IntelliJ, NetBeans 8.2 and LambdaFicator by a significant margin: Kayak finds 39 out of 50 (78%) possible refactorings, whereas IntelliJ only transforms 10 (20%) and both NetBeans 8.2 and LambdaFicator transform 11 benchmarks (22%) successfully. IntelliJ, NetBeans and LambdaFicator combined find 15 (30%) refactorings. This is due to the fact that there are many common Java paradigms, such as `ListIterator` or `Iterator::remove`, for which none of the tools contain pre-configured patterns and thus have no way of refactoring. The fact that none of the pattern-based tools provide for these situations suggests that it is impractical to try to enumerate every possible refactoring pattern in IDEs.

If the pattern-based tools find a solution, they transform the program safely and instantaneously, even in cases where Kayak fails to synthesise a refactoring within the allotted time limit. Where Kayak synthesised a valid refactoring, it did so within an average of 8.5 s. It is worth mentioning that the syntax-driven tools and Kayak complement each other very well in our experiments, which is illustrated by the fact that both approaches combined would have solved 44 out of the 50 refactorings (88%) correctly. Loops which match the expected patterns of syntax-driven tools are handled with ease by such tools, regardless of semantic complexity. Kayak on the other hand abstracts away even stark syntactical differences and recognizes equivalent semantics instead, but is limited by the computational complexity of its static analysis engine.

Kayak's maximum memory usage (heap+stack) was 125MB over all benchmarks according to valgrind massif. We found that the majority of timeouts for Kayak are due to an incomplete instruction set in the synthesis process. We plan to implement missing instructions as the program progresses out of its research prototype phase into an industrial refactoring tool set. A link to all benchmarks used in the experiment is provided in the footnote[3].

8 THREATS TO VALIDITY

Our hypothesis is that we have given exemplary evidence that semantics-based refactoring can be soundly applied, are more precise and enable more complex refactoring schemata. As we use program analysis technology, all standard threats to validity in this domain apply here as well; we summarise these only briefly.

Selection of benchmarks. Our claim relates to "usual" programs written by human programmers, and our results may be skewed by the choice of benchmarks. We address this concern by collecting our benchmarks from GitHub, which hosts a representative and exceptionally large set of open-source software packages. Commercial software may have different characteristics, was not covered by our benchmarks, and thus our claim may not extend to commercial, closed-source software. Furthermore, all our benchmarks are Java programs, and our claim may not extend to any other programming language. We focused our experimental work on the exemplar of refactoring iteration over collections, and our technique may not be more widely applicable. Finally, our Java front-end is still incomplete, only supporting lists of integers and lacking models for many Java system classes. This restricts our selection to a subset of the benchmarks in our GitHub search results, which may be biased in favour of our tool. We will address this issue by extending the front-end to accept additional Java input.

Not supported. We exclude transformers such as *peek* and *foreach*, which are included in the Stream API. The reason is that such transformers enable an equivalent transformation for virtually any loop processing a collection in iteration order. Fig. 11 illustrates such a transformation.

[1] https://www.jetbrains.com/idea/
[2] https://netbeans.org/

[3] https://drive.google.com/open?id=0ByIexo3Z5N91ZlNFZTNpdU5USjQ

```
for(int e : c) { foo(e); }                (a)
c.stream().foreach(e -> { foo(e); });     (b)
```

Figure 11: *Foreach* example with original (a) and stream (b).

Transformations as illustrated in Fig. 11 do not improve expressiveness and readability of the program and are, at best, a matter of pure *foreach* transformations. As a note, we do use them to perform in-place transformations of the collections (as shown in Sec. 3.1), but they are introduced only after the actual synthesis, when we generate code using streams.

Quality of refactorings. Refactorings need to generate code that remains understandable and maintainable. Syntax-driven refactoring has good control over the resulting code; the code generated by our semantic method arises from a complex search procedure, and may be difficult to read or maintain.

It is difficult to assess how well our technique does with respect to this subjective goal. Firstly, we conjecture that small refactorings are preferable to larger ones (measuring the number of operations). Our method guarantees that we find the shortest possible refactoring due to the way we parameterise and search the space of candidate programs (as described in Sec. 6). It is unclear whether human programmers indeed prefer the shortest possible refactoring.

Secondly, our method can exclude refactorings that do not improve readability of the program. For instance, as mentioned in Sec. 3, we exclude transformations that include only peek and foreach, which are offered by the Stream API. A refactoring that uses these transformers (Fig. 11) can be applied to virtually any loop processing a collection in iteration order but is generally undesirable.

Finally, we manually inspected the refactorings obtained with our tool and found them to represent sensible transformations.

Efficiency and scalability of the program synthesiser. We apply heavy-weight program analysis. This implies that our broader claim is threatened by scalability limits of these techniques. The scalability of our particular refactoring procedure is gated by the program synthesiser. While for the majority of our experiments the synthesiser was able to find a solution quickly, there were a few cases where it failed to find one at all. The problem was that the SYNTHESISE portion of the CEGIS loop failed to return with a candidate solution. Different instruction sets for the synthesis process can help mitigate this effect.

Better syntax-driven refactoring. Our hypothesis relates semantics-driven refactoring to syntax-driven refactoring. While we have undertaken every effort to identify and benchmark the existing syntax-driven refactoring methods, there may be means to achieve comparable or better results by improving syntax-driven refactoring.

9 RELATED WORK

Program refactoring. Cheung et al. describe a system that automatically transforms fragments of application logic into SQL queries [7]. Moreover, similar to our approach, the authors rely on synthesis technology to generate invariants and postconditions that validate their transformations (a similar approach is presented in [22]). The main difference (besides the actual goal of the work, which is different from ours) to our work is that the lists they operate on are immutable and do not support operations such as remove. Capturing the potential side effects caused by such operations is one of our work's main challenges.

In syntax-driven refactoring engines, program transformation decisions are based on observations on the program's syntax tree. Visser presents a purely syntax-driven framework [39]. The presented method is intended to be configurable for specific refactoring tasks, but cannot provide guarantees about semantics preservation. The same holds for [10] by Cordy et al., [26] by Sawin et al., [23] by Bae et al. and [8] by Christopoulou et al. In contrast to these approaches, our procedure constructs an equivalence proof before transforming the program. In [18], Gyori et al. present a similar refactoring to ours but performed in a syntax-driven manner.

Steimann et al. present Constraint-Based Refactoring in [36], [37] and [38]. Their approach generates explicit constraints over the program's abstract syntax tree to prevent compilation errors or behaviour changes by automated refactorings. This gives rise to a flexible framework of customisable refactorings, implementable through a refactoring constraint specification language (cf. [38]). The approach is limited by the information a program's AST provides and thus favours conservative implementations of syntax-focused refactorings such as *Pull Up Field*.

Fuhrer et al. implement a type constraint system to introduce missing type parameters in uses of generic classes (cf. [14]) and to introduce generic type parameters into classes which do not provide a generic interfaces despite being used in multiple type contexts (cf. [27]).

Raychev et al. present a semi-automatic approach where users perform incomplete refactorings manually and then employ a constraint solver to find a sequence of default refactorings such as move or rename which include the users' changes. The engine is limited to syntactic matching with the users' partial changes and does not consider program semantics [33].

Weissgerber and Diehl rely on meta information to classify changes between software versions as refactorings [40]. The technique aims to identify past refactorings performed by programmers, but is not a decision procedure for automated refactorings.

O'Keffe and Cinnéide present search-based refactoring [30, 31], which is similar to syntax-driven refactoring. They rephrase refactoring as an optimisation problem, using code metrics as fitness measure. As such, the method optimises syntactical constraints and does not take program semantics into account.

Bavota et al. implement refactoring decisions in [2] using semantic information limited to identifiers and comments, which may differ from the actual semantics (e.g. due to bugs). Kataoka et al. also interpret program semantics to apply refactorings [24], but use dynamic test execution rather than formal verification, and hence their transformation lacks soundness guarantees.

Franklin et al. implement a pattern-based refactoring approach transforming statements to stream queries [17]. Their tool LambdaFicator [13] is available as a NetBeans branch. We compared Kayak against it in our experimental evaluation in Sec. 7.

Heap Logics. While many decidable heap logics have been developed recently, none are expressive enough to capture operations allowed by the Java Collection interface, operations allowed by the Java Stream interface as well as equality between collections (for lists this implies that we must be able to reason about both content of lists and the order of elements) [5, 6, 11, 21, 29, 32]. On the other hand, very expressive transitive closure logics [20] are not concise and easily translatable to stream code.

Program synthesis. An approach to program synthesis very similar to ours is Syntax Guided Synthesis (SyGuS) [1]. SyGuS synthesisers supplement the logical specification with a syntactic template that constrains the space of allowed implementations. Thus, each semantic specification is accompanied by a syntactic specification in the form of a grammar. Other second-order solvers are introduced in [3, 16]. As opposed to ours, these focus on Horn clauses.

10 CONCLUSION

We conjecture that refactorings driven by the semantics of programs have broader applicability and are able to address more complex refactoring schemata in comparison to conventional syntax-driven refactorings, thereby increasing the benefits of automated refactoring. The space of possible semantic refactoring methods is enormous; as an instance, we have presented a method for refactoring iteration over Java collection classes based on program synthesis methods. Our experiments indicate that refactoring using this specific instance is feasible, sound and sufficiently performant. Future research must broaden the evidence for our general hypothesis by considering other programming languages, further, ideally more complex refactoring schemata, and other semantics-based analysis techniques.

REFERENCES

[1] R. Alur et al. Syntax-guided synthesis. In *FMCAD*, 2013.

[2] G. Bavota, A. De Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *J. Syst. Softw.*, 84(3):397–414, Mar. 2011. ISSN 0164-1212.

[3] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, LNCS, pages 869–882. Springer, 2013.

[4] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[5] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *Automated Technology for Verification and Analysis (ATVA)*, LNCS. Springer, 2012.

[6] M. Brain, C. David, D. Kroening, and P. Schrammel. Model and proof generation for heap-manipulating programs. In *European Symposium on Programming (ESOP)*, LNCS, pages 432–452. Springer, 2014.

[7] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *Conference on Programming Language Design and Implementation, PLDI*, pages 3–14, 2013.

[8] A. Christopoulou, E. Giakoumakis, V. E. Zafeiris, and S. Vasiliki. Automated refactoring to the strategy design pattern. *Information and Software Technology*, 54(11):1202 – 1214, 2012. ISSN 0950-5849.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[10] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827 – 837, 2002. ISSN 0950-5849.

[11] C. David, D. Kroening, and M. Lewis. Using program synthesis for program analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20)*, LNCS, pages 483–498. Springer, 2015.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999. ISBN 0-201-48567-2.

[13] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. LAMBDAFICATOR: from imperative to functional programming through automated refactoring. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1287–1290, 2013. doi: 10.1109/ICSE.2013.6606699. URL http://dx.doi.org/10.1109/ICSE.2013.6606699.

[14] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently refactoring java applications to use generic libraries. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pages 71–96, 2005. doi: 10.1007/11531142_4. URL http://dx.doi.org/10.1007/11531142_4.

[15] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. In *Time for Verification, Essays in Memory of Amir Pnueli*, pages 167–184, 2010.

[16] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416, 2012.

[17] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 543–553, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491461. URL http://doi.acm.org/10.1145/2491411.2491461.

[18] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *ESEC/SIGSOFT FSE*, pages 543–553. ACM, 2013.

[19] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

[20] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004.

[21] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *Computer Aided Verification (CAV)*, LNCS, pages 756–772. Springer, 2013.

[22] M. Iu, E. Cecchet, and W. Zwaenepoel. JReq: Database queries in imperative languages. In *Compiler Construction (CC)*, pages 84–103, 2010.

[23] S.-U. Jeon, J.-S. Lee, and D.-H. Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 337–345, 2002.

[24] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold. Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01. IEEE Computer Society, 2001.

[25] J. Kerievsky. Refactoring to patterns. In *Extreme Programming and Agile Methods*, volume 3134 of *LNCS*, page 232. Springer, 2004.

[26] R. Khatchadourian, J. Sawin, and A. Rountev. Automated refactoring of legacy Java software to enumerated types. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 224–233, 2007.

[27] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing java classes. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 437–446, 2007. doi: 10.1109/ICSE.2007.70. URL http://dx.doi.org/10.1109/ICSE.2007.70.

[28] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252.

[29] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *Principles of Programming Languages (POPL)*, pages 611–622, 2011.

[30] M. O'Keeffe and M. Cinnéide. Search-based refactoring: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(5):345–364, 2008. ISSN 1532-0618.

[31] M. O'Keeffe and M. Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502 – 516, 2008. ISSN 0164-1212.

[32] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using SMT. In *Computer Aided Verification (CAV)*, LNCS, pages 773–789. Springer, 2013.

[33] V. Raychev, M. Schäfer, M. Sridharan, and M. T. Vechev. Refactoring with synthesis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 339–354, 2013. doi: 10.1145/2509136.2509544. URL http://doi.acm.org/10.1145/2509136.2509544.

[34] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *Computer Aided Verification (CAV)*, pages 88–105, 2014.

[35] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[36] F. Steimann. Constraint-based model refactoring. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems: 14th International Conference (MODELS)*, pages 440–454. Springer, 2011. ISBN 978-3-642-24485-8. doi: 10.1007/978-3-642-24485-8_32. URL http://dx.doi.org/10.1007/978-3-642-24485-8_32.

[37] F. Steimann and J. von Pilgrim. Constraint-based refactoring with Foresight. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming: 26th European Conference*, pages 535–559. Springer, 2012. ISBN 978-3-642-31057-7. doi: 10.1007/978-3-642-31057-7_24. URL http://dx.doi.org/10.1007/978-3-642-31057-7_24.

[38] F. Steimann, C. Kollee, and J. von Pilgrim. A refactoring constraint language and its application to Eiffel. In M. Mezini, editor, *ECOOP 2011 – Object-Oriented Programming: 25th European Conference*, pages 255–280. Springer, 2011. ISBN 978-3-642-22655-7. doi: 10.1007/978-3-642-22655-7_13. URL http://dx.doi.org/10.1007/978-3-642-22655-7_13.

[39] E. Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. Technical Report UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University, 2004.

[40] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Automated Software Engineering (ASE)*, pages 231–240, Sept 2006.