Cascade: A Universal Programmer-assisted Type Qualifier Inference Tool

Mohsen Vakilian*, Amarin Phaosawasdi*, Michael D. Ernst§, Ralph E. Johnson*
* University of Illinois at Urbana-Champaign, Urbana, IL, USA
{mvakili2, phaosaw2, rjohnson}@illinois.edu
§ University of Washington, Seattle, WA, USA
mernst@cs.washington.edu

***Abstract*—Type qualifier inference tools usually operate in batch mode and assume that the program must not be changed except to add the type qualifiers. In practice, programs must be changed to make them type-correct, and programmers must understand them. CASCADE is an interactive type qualifier inference tool that is easy to implement and universal (i.e., it can work for any type qualifier system for which a checker is implemented). It shows that qualifier inference can achieve better results by involving programmers rather than relying solely on automation.**

I. INTRODUCTION

A *type qualifier system*, or just qualifier system, augments an existing type system with *type qualifiers* to check more properties. A type qualifier, or just qualifier, is an annotation that the programmer adds to declarations and uses of types.

Researchers have developed many qualifier systems to check software properties such as concurrency [17], ownership [20], [39], immutability [21], [40], safety against null dereferences [8], [33], and security [8], [16], [33], [35], [44]. Java 8 supports the syntactic features required by qualifier systems [2].

The benefits of a qualifier system, such as safety guarantees and machine-checkable documentation, come at the cost of adding the qualifiers to code. The burden of annotating code with qualifiers is a major obstacle to the adoption of qualifier systems. Realizing the burden of adding qualifiers, researchers have developed myriad qualifier inference tools [5], [9], [11]–[13], [18], [20], [21], [21]–[23], [26], [30], [34], [37], [38], [40]. These tools employ static analysis, dynamic analysis, or a combination of the two.

Existing qualifier inference tools operate in *batch* mode. That is, they take the source code as input, analyze it, and insert all the qualifiers at once. While batch qualifier inference tools can add qualifiers to large pieces of code without involving the programmer, these tools have several weaknesses. First, they are *imprecise*, because inferring precise qualifiers requires knowledge of programmer intent and of the run-time behavior of the program, e.g., application invariants, aliasing, and interprocedural control and data flow, for all possible inputs. Second, they are *rigid*. That is, they assume that the code does not have to be changed, while usually programmers have to refactor the code to a form that the qualifier system can express [41, ch. 6][1]. Third, they are *unpredictable*, because they add many qualifiers to the code, and it is difficult for the programmer to tell why a particular code change was applied. Fourth, they are *specific* to a single qualifier checker. While frameworks such as the Checker Framework [33] reduce the cost of developing a checker, developing a qualifier inference tool is still a nontrivial engineering task.

This paper introduces CASCADE, a *universal* and easy-to-use tool for inferring type qualifiers. CASCADE takes a type-checker for a qualifier system as input and uses it to assist the programmer in inferring the qualifiers.

Type inference and refactoring (changing code without changing the program's behavior [32]) are usually thought of as unrelated concepts. However, we consider qualifier inference a refactoring, because inserting qualifiers in a program does not alter the behavior of the program. In addition, qualifier inference often goes beyond inserting qualifiers and requires code refactorings. CASCADE was inspired by our work on *compositional refactoring* [42]. In the compositional paradigm, a refactoring tool designer decomposes a refactoring into a set of primitive changes and automates them. Following the compositional paradigm, CASCADE decomposes qualifier inference into primitive changes. Each primitive change resolves one or more of the errors reported by the qualifier checker.

Another inspiration for CASCADE is speculative analysis [6], [28], [29], which is a technique that assists programmers in decision-making by presenting the consequences of their actions ahead of time. Speculative analysis improves programmers' productivity by showing potential future versions of the program and helping programmers avoid undesirable versions and unnecessary backtracking.

CASCADE is an *interactive* tool that assists programmers in adding qualifiers by guiding them through a tree of changes and error messages. It repeatedly runs the qualifier checker on variants of the source code to compute a tree of error and change nodes. The error nodes correspond to the error messages that the qualifier checker reports, and the change nodes correspond to potential code changes that resolve the

[1] "Type qualifier inference" usually refers to only inserting type qualifiers in a piece of code. However, programmers often need to make other nontrivial code changes to use the type qualifiers appropriately. Therefore, we take a broader point of view. We consider type qualifier inference to encompass any code change that is required to take full advantage of a type qualifier system.

error messages. A programmer can expand the tree to explore the effects of the changes on the errors before applying the changes. Expanding an error node shows its children, which are the changes that resolve the error. Expanding a change node shows its children, which are the new errors that the change introduces. Figure 1 shows a CASCADE tree.

The compositional aspect of CASCADE is computing the change to fix each error message. The speculative aspect of CASCADE is computing (1) the effect of each change on the error messages reported by the qualifier checker and (2) the required follow-up changes. An advantage of CASCADE's design is that it is easy to implement. This makes it easy to port CASCADE to different programming environments.

We conducted a lab study with 12 programmers to compare CASCADE with JULIA. JULIA [37], [38] is the state-of-the-art static analysis tool for inferring nullness qualifiers. The study participants finished the task faster with CASCADE and inserted better qualifiers. The participants also felt more in control with CASCADE and said that they are more likely to use CASCADE in the future.

This paper makes the following research contributions:

- It introduces CASCADE, which is the first *universal*, usable, and easy-to-implement tool that assists programmers in inferring type qualifiers. CASCADE is open source and publicly available [1].
- It compares CASCADE, a *programmer-assisted* qualifier inference tool, with JULIA, a state-of-the-art *batch* qualifier inference tool, through a user study.

II. TYPE QUALIFIERS

Type qualifiers are light-weight specifications that augment an existing type system to enable static verification of desired properties of software.

A qualifier system provides a set of qualifiers and a set of rules to check the use of the qualifiers. For example, one can add the qualifier `@NonNull` to the type of a `String` variable `s`—written in Java as `@NonNull String s`—to indicate that the variable `s` cannot be `null`. A nullness qualifier checker would ensure that the annotations are accurate and that only `@NonNull` references are dereferenced.

Researchers have proposed frameworks for building qualifier systems, e.g., CQual [16], Clarity [9], JQual [18], JavaCOP [4], and the Checker Framework [33]. Qualifier systems have been developed in these frameworks for checking software properties such as null safety [8], [33], tainting [8], [33], format strings [35], [44], internationalization [33], regular expressions [36], UI threading [17], ownership [20], and immutability [21], [33], [40], [46].

Despite the advances in formalizing, implementing, and verifying custom qualifiers, *annotation burden* remains a major barrier to their adoption. Annotation burden is the cost of adding type annotations to an existing piece of code so that it satisfies the rules of a given qualifier system. To mitigate the annotation burden, researchers have proposed tools for inferring the qualifiers of a few qualifier systems. Table I lists some of the proposed qualifier inference tools, all of which are fully automatic and operate in batch mode. Each of the existing qualifier inference tools is imprecise, rigid, unpredictable, and specific to a single qualifier checker. Based on our prior studies [42] on compositional and wizard-based paradigms of refactoring, we hypothesize that the underlying problem is once again *too much automation*.

TABLE I: Existing qualifier inference tools.

Type Qualifier System	Prog. Languages	Type Qualifier Inference Tools
Nullness	Java	JastAddJ Nullness [12] Nit [22], [23] SALSA Nullness [26] Xylem [30] Julia [37], [38]
	C	Clarity Nullness [9]
	Java, C, Perl, C#, C++	Daikon Nullness [13]
Immutability	Java	JQual Immutability [18] Javarifier [34] for Javari [40] Pidasa [5] ReImInfer [21] for ReIm [21]
Ownership	Java	Inferring Universe Types [11] Inferring Ownership [20]

III. THE DESIGN OF CASCADE

We use a combination of two concepts, *compositional refactoring* and *speculative analysis*, to build the first universal programmer-assisted qualifier inference tool and mitigate the weaknesses of existing batch tools.

We followed an iterative user-centered process when designing CASCADE, our programmer-assisted inference tool. We started by creating four different low-fidelity paper prototypes. Our goal was to explore a wide design space and get early feedback on them. Based on our design goals and the feedback that we received, we selected two of the low-fidelity prototypes and turned them into high-fidelity prototypes. We implemented the high-fidelity prototypes as functional and interactive Eclipse plug-ins. Both of these prototypes incorporated the ideas of compositional refactoring and speculative analysis. The differences were in the user interaction models. Nine programmers gave feedback on the prototypes.

The compositional paradigm decomposes a complex refactoring into small steps and allows the programmer to compose them. CASCADE decomposes qualifier inference into small steps in which each step resolves an error reported by the qualifier checker. Each step may create new errors, and a step may resolve multiple errors. The most common step in qualifier inference is to propagate the qualifier of one expression to another to resolve an incompatibility between qualifiers, e.g., the two sides of an assignment. CASCADE automates only the qualifier propagation steps. The more difficult steps, such as refactoring code, are left for the programmer to perform manually. CASCADE leaves the programmer in control of the entire inference process, so it is easy for the programmer to

insert manual edits. CASCADE uses the qualifier checker to find places where there are qualifier incompatibility errors and develops a plan for fixing each error. However, it does not guarantee that these plans will work, and the programmer does not expect them all to work. The programmer will accept a plan when it seems reasonable, and make manual edits when it does not.

CASCADE forces the programmer to gradually look at parts of the code whose qualifiers are being inferred. It might seem that it would take longer to infer qualifiers this way than with a batch qualifier inference tool. However, the batch qualifier inference tools can rarely infer qualifiers perfectly, and it can take a programmer a long time to figure out why they failed. When a program needs some changes before it will be type-correct, CASCADE directs the programmer to the parts of the code that may need to change, helping the programmer to discover and make the changes.

CASCADE uses speculative analysis to show the consequences of applying each change and propose a plan for composing the changes. Speculative analysis can help programmers in making decisions by showing the consequences of the decisions in advance. CASCADE makes the programmer aware of the new error messages that the qualifier checker will report because of applying a change. In addition, CASCADE suggests changes for the new errors that may arise, effectively suggesting a plan for composing multiple changes.

A. *The* CASCADE *user interface*

CASCADE uses a tree to show errors and potential future states of the source code. The CASCADE tree (Figure 1) has two types of nodes: *error message nodes* and *change nodes*. An error node shows an error message that the qualifier checker reports. A change node offers the programmer an automated change to apply. The root of the tree is the set of error messages that the qualifier checker reports.

CASCADE provides an affordance to browse the tree. The programmer can click on a triangle icon next to a tree node to expand or collapse the tree (Figure 1), and CASCADE shows child nodes indented beneath their parent. Expanding an error message shows any change that CASCADE proposes to resolve that error message. For example, if a programmer expands the first error message shown in Figure 1, the tree will expand to propose a change described as `Change field left to @Nullable TreeNode`. Similarly, expanding a change shows the new error messages that will appear if the change is applied.

Expanding the CASCADE tree does not change the source code. Rather, it enables the programmer to explore the future states of the source code, and see how a sequence of proposed changes will affect the source code.

A programmer can expand the CASCADE tree deeply. For instance, consider the error message node highlighted in the CASCADE tree shown in Figure 1. To resolve this error message, CASCADE proposes the change `Change parameter r to @Nullable TreeNode`. Besides, CASCADE allows the programmers to expand the proposed change node. Expanding a change node shows the *new* error messages that will appear

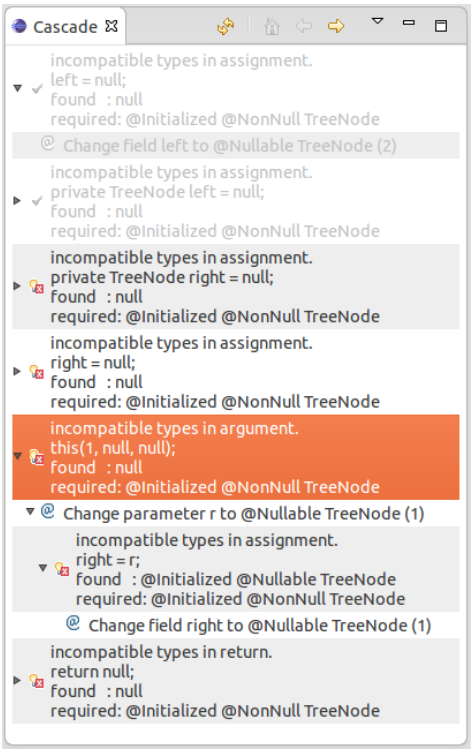

Fig. 1: A screenshot of the CASCADE tree for the JOlden TreeAdd project. The tree shows each error (top-level items), a change that fixes it (preceded by "@" and visible in the errors that have been expanded), and the errors that each change will introduce (nested under the fix). The top two errors have been resolved, so their text is gray.

if the change is applied. In this case, the error message node that CASCADE shows as a child of the expanded change involves the expression `right = r` (Figure 1). With further expansion of the above error message, CASCADE proposes the change `Change field right to @Nullable TreeNode` to fix the error message.

CASCADE maintains the link between the source code and the nodes on the tree. If the user single-clicks on an error message or change, CASCADE will open and highlight the piece of code related to the selected node.

If the user double-clicks on a change tree node, CASCADE will apply the change on the code and open the affected code in the editor. In addition, CASCADE will show the applied change and the error messages that it resolves as disabled nodes. CASCADE provides an undo feature. If the user undoes a change, CASCADE will show the change and the error messages that the change resolves as enabled nodes.

The programmer is free to apply the proposed changes in any order and intersperse them with manual edits. However, applying manual or out-of-order changes may make the tree inconsistent with the source code, in which case the programmer can press the refresh icon to recompute the tree.

IV. Universality Assumptions

We define a *universal* qualifier inference tool as a tool that can infer qualifiers according to the rules of any qualifier system. Our compositional approach to qualifier inference is universal under two assumptions:

- A1. A qualifier checker for the qualifiers is available.
- A2. It is possible to automatically compute the code changes that would fix some of the problems reported by the qualifier checker.

The implementation of CASCADE satisfies the above assumptions by requiring the following.

- A1.1. The qualifiers are compatible with Java 8.
- A1.2. A qualifier checker for the given set of qualifiers is implemented in the Checker Framework.
- A1.3. The Checker Framework Eclipse plug-in is configured to run the qualifier checker.
- A2.1 The qualifier checker reports qualifier incompatibilities, which are caused by mismatches in the actual and expected qualifiers of expressions.
- A2.2 For each qualifier incompatibility, the Checker Framework reports the location (enclosing file, offset, and length) of the code snippet that causes the problem as well as the actual and expected qualifiers.

The next section explains how CASCADE achieves universality under the above assumptions.

V. Implementation

Although we demonstrated CASCADE for inferring the nullness qualifiers (Figure 1), CASCADE is a universal qualifier inference tool and supports the inference of all qualifiers that come with a qualifier checker developed on top of the Checker Framework. Reusing the qualifier checker makes CASCADE universal and easier to implement than a typical batch qualifier inference tool.

We implemented CASCADE as an Eclipse plug-in that depends on the Checker Framework Eclipse plug-in, which in turn depends on the Checker Framework. The simplicity of CASCADE makes it easy to port it to other programming environments.

A. Getting the Checker Error Messages

CASCADE relies on the error messages that the qualifier checker reports on the given source code and the variants of the source code. To get the error message, CASCADE invokes the qualifier checker through the Checker Framework Eclipse plug-in. The Checker Framework runs the qualifier checker and populates the Eclipse problems view with *markers* that capture information about the error messages reported by the qualifier checker. Each marker contains information such as the error message, the offset and length of the piece of code that caused the error message, and the expected and actual qualifiers.

B. Proposing Changes to Resolve the Error Messages

CASCADE proposes changes to resolve the qualifier incompatibilities reported by the qualifier checker. If the qualifier of the right-hand side of an assignment statement is not a subtype of that of the left-side hand side, the qualifier checker will report a qualifier incompatibility error. Similarly, the qualifiers of method parameters and corresponding method arguments must be compatible. As another example, the qualifiers of method return expressions and the corresponding method return types must be compatible. These programming language constructs that require the qualifiers of two program elements be compatible are referred to as *pseudo-assignments*. CASCADE resolves the qualifier incompatibilities by propagating the qualifier of the right-hand side of the pseudo-assignment to that of the left-hand side.

C. Proposing a Change Composition Plan

CASCADE proposes a plan for composing the changes in the form of a tree. It computes the plan through a speculative analysis.

Usually, changing a qualifier comes with a *cascade effect*. A cascade effect refers to all the qualifier changes required by a given qualifier change. For instance, if a variable changes to `@Nullable`, any method parameter that the variable is passed to has to change to `@Nullable`, too. To support these cascade effects, CASCADE applies the change on a copy of the code and reruns the qualifier checker in the background to see if the change causes new qualifier incompatibilities. If new incompatibilities occur, CASCADE will propose changes for fixing them and recursively continue to compute the consequences of those changes. Figure 2 illustrates the pseudocode of the speculative analysis.

CASCADE uses the binding information computed by the Eclipse Java Development Tools (JDT) to reliably apply changes on variants of the code. Eclipse JDT generates a *binding key* for declarations such as method and variable declarations. The binding key is a string that encodes the path to the declaration through the AST. By storing the change category, necessary binding keys, and qualifier change, CASCADE can reliably apply the change on variants of the code for which the binding keys are valid. If the code changes drastically, the binding keys may no longer identify the desired declaration. In practice, because the changes that CASCADE makes to the code during its speculative analysis are only qualifier changes, they preserve the binding keys.

CASCADE neither represents the changes as textual changes nor AST changes. If it represented the changes as textual changes, a change computed during the speculative analysis against a variant of the source code would have been unlikely to be applicable to the original source code. Similarly, if CASCADE stored references to the AST nodes that it modified in a copy of the code, the changes would have not been applicable to the original source code, because the AST node objects of the original code and its copy have different identities.

```
   input  : C, a piece of code
            R, the root node of the tree to compute
   output : a tree of changes and errors for inferring the type qualifiers of
            C rooted at R
1  function computeTree(C, R)
   // Let P be the set of problems that the type
      qualifier checker reports for C.
2  P ← check(C)
3  foreach p ∈ P do
4      pn ← createTreeNode(p)      // Create a new tree
       node for problem p.
5      makeNodeChildOf(pn, R)  // Make pn a child of R.
6      F ← suggestedFixes(p)   // Let F be the set of
       code changes that fix p.
7      foreach f ∈ F do
8          fn ← createTreeNode(f)  // Create a new tree
           node for change f.
9          makeNodeChildOf(fn, pn)     // Make fn a child
           of pn.
10         C′ ← changedCode(C, f)    // Let C′ be a copy
           of C with code change f.
11         computeTree(C′, fn)
```

Fig. 2: The speculative analysis that computes the tree of changes is a recursive computation. The main call makes C be the original version of the code and R be a tree node that will be the only invisible node of the tree. The result will be a tree of changes and errors rooted at R.

D. Presenting the Composition Plan

CASCADE presents its composition plan using the standard Eclipse tree view to achieve a tight integration with Eclipse. The programmers can use the tree to locate the pieces of code that correspond to changes and error messages. The location of the piece of code corresponding to an error message is often different in the original code and a variant of the code. Because CASCADE does not change the original source code, the tree is expected to be consistent with the original source code that programmers have access to. CASCADE uses a heuristic to locate the same piece of code in the original version of the code. It first locates the piece of code that caused the error message in the copy of the source code. Then, it expands that piece of code to include a larger part of the code. Finally, it searches the same piece of code in the original code and prefers the match whose offset is closest to the offset of the piece of code in the copy of the source code.

VI. EVALUATION

We conducted a user study to to provide qualitative and quantitative insight about the strengths and weaknesses of two paradigms of inferring type qualifiers: *batch* and *compositional*. In the batch paradigm, the qualifier inference tool takes a piece of code as input and inserts all the remaining qualifiers in the code. In the compositional paradigm, the programmer manually composes multiple refactorings each of which inserts qualifiers to a narrow piece of code.

In this study, we used two tools: JULIA [37], [38] and CASCADE, which support qualifier inference in the batch and compositional paradigms, respectively. JULIA provides the state-of-the-art static analysis qualifier inference tool for nullness in the batch paradigm. Its goal is to infer nullness annotations that the nullness checker of the Checker Framework accepts. CASCADE is the qualifier inference tool that we developed based on the concepts of compositional refactoring and speculative analysis.

A. Research Questions

The goal of the study was to answer the following research questions for automated qualifier inference tools.

- RQ1 How do JULIA and CASCADE compare along the following dimensions?
  - RQ1a task completion time
  - RQ1b quality of results
  - RQ1c learnability
  - RQ1d control
  - RQ1e willingness to use
  - RQ1f predictability
- RQ2 How useful is the speculative analysis of CASCADE?
- RQ3 What strategies do programmers employ in inferring the qualifiers using JULIA and CASCADE?

B. Methodology

To evaluate CASCADE and answer the aforementioned research questions, we conducted a comparative lab study. This study sheds light on the advantages and disadvantages of the batch and compositional paradigms.

1) Recruitment: We recruited 12 participants from the computer science department of the University of Illinois at Urbana-Champaign, and we offered a $25 gift card to each participant.

2) Lab Setup: We set up the lab to study one participant at a time. We set up a PC with the qualifier inference tools under study. We stayed outside the lab during the study and instructed the participants to reach out to us with any questions.

3) Training: We prepared written and multimedia tutorials to introduce the participants to the nullness qualifier checker, JULIA, and CASCADE. We have made these artifacts publicly available [3].

a) Nullness Checker: Given that Java with qualifier support was released shortly before the study, few programmers knew about this feature during the time frame of our study. Thus, we prepared a tutorial based on the manual of the Checker Framework to familiarize the participants with the nullness checker. Since it is easy to misuse the annotations and suppress warnings unnecessarily, the tutorial distinguished justified and unjustified annotations and encouraged the reader to use justified annotations. We asked the participants to study our tutorial about the nullness checker before the study. The tutorial had a few exercises to make sure that the participants grasp the key concepts of the nullness checker. The participants sent us their solutions to the exercises. We reviewed the solutions and asked the participants to correct their solutions if necessary.

b) JULIA *and* CASCADE*:* During the lab study, the participants watched video tutorials of JULIA and CASCADE and had access to written versions of the tutorials. The maintainers of JULIA made a version of it with a command-line interface available to us. This version of JULIA can infer the nullness annotations for a piece of code and output them to a separate annotation file. We wrote an Ant script to enable the participants run the nullness checker and JULIA from within Eclipse. The Ant script can run the nullness checker and report the problems. It can also run JULIA and automatically insert the inferred annotations in the code. We informed the participants that the subject programs were stored in a Git repository. The participants were allowed to use Git or the history features of Eclipse to view or revert the changes made by JULIA and CASCADE.

4) Prequestionnaire: Before starting the task, we asked the participants to fill out a prequestionnaire to collect their demographic information.

5) Task Design: We designed the study as a *within-subject*, *counterbalanced* one. In a within-subject study, each participant evaluates both tools under study. An advantage of a within-subject design is that it enables the participants to qualitatively compare the two tools. Another advantage is that it mitigates the variance in the results due to the difference in the expertise levels of the participants. A standard disadvantage of the within-subject design is the *carryover effect* from the first task to the second one. Two common carryover effects are the *learning* and *fatigue* effects. Learning refers to the experience that the participant gains in qualifiers and qualifier inference during the first task. Fatigue refers to participants getting tired after finishing the first task. We employed two strategies to mitigate the carryover effect. First, we used two programs to avoid the learning effect of annotating the same program twice. Another benefit of using two programs is that it avoids the limitation of the results to a single program. Second, we used a *counterbalanced* design to balance the order in which the participants annotated the two programs using the two tools (JULIA and CASCADE). To achieve a counterbalanced design, we randomly divided our participants into four groups of the same size. Then, we had each group use the two qualifier inference tools on the two programs in a unique order.

We used two programs, Barnes-Hut (BH) and Minimum Spanning Tree (MST), from the JOlden benchmark suite for our user study. We had two criteria for the programs under study. First, it had to be possible to annotate the selected piece of code in 30–40 minutes. Second, the program should be representative of real code.

During our pilot studies, we found that that the participants could not annotate the selected programs within the allotted time. So, we removed parts of the code and simplified other parts of it to make it possible to annotate it within the allotted time. We have made our simplified versions of the BH and MST programs, which we used during the study, publicly available as part of the artifacts of the study [3].

We asked the participants to insert the nullness annotations in each program (BH and MST) using the designated qualifier inference tool (JULIA and CASCADE). We told the participants that they are allowed to refactor the code but not change the behavior of the program. We required the participants to make sure that their uses of the annotations are well justified. For example, we asked them to avoid using `@SuppressWarnings` annotations and `assert` statements where they believe they are inappropriate or not needed. We also asked the participants to avoid `@Nullable` annotations where `@NonNull` is appropriate and vice versa. Similarly, we asked them to avoid `@NonNull` annotations where no annotation is needed. We provided a test suite along the program and required that the test suite passes before and after the task. After the participants finished the task, we asked them to run the tests and verify their annotations.

6) Postquestionnaire: After the participants finished the task, we asked them to fill out a postquestionnaire that captured their relative preferences towards the two tools along various dimensions. The postquestionnaire asked the participants to rank the two tools along multiple dimensions including ease of use, transparency, control, and willingness to use. In addition, it asked the participants to elaborate on the strengths and weaknesses of each tool.

C. Interview

After filling out the postquestionnaires, we conducted a brief semi-structured interview with each participant. During the interview session, we asked questions including the ones listed below.

- How did each of JULIA and CASCADE affect your strategies for adding qualifiers?
- How intuitive and useful was the tree of changes and errors presented by CASCADE? How useful did you find CASCADE's suggestions of future errors and changes?
- What are your suggestions for improving JULIA and CASCADE?

D. Results

1) Participant Demographics: The participants came from nine different research labs at the computer science department of the University of Illinois at Urbana-Champaign. One participant was a post-doc, one was a master's student, and the rest were PhD students. The participants worked in a variety of areas such as high performance computing, natural language processing, security, mobile computing, compilers, and computer architecture. The prequestionnaire asked the participants to rate their familiarities with Java and Eclipse along a 5-point Likert scale. All participants considered themselves at least familiar with Java, and 11 considered themselves at least familiar with Eclipse.

2) Task Completion Time (RQ1a): To compare the efficiency of programmers with each qualifier inference tool, we measured the task completion times with each tool. With a Welch's t test ($t(11) = 2.89$, $p = 0.01$, Cohen's $d = 1.13$), we found that the participants were significantly faster using CASCADE (mean = 28 minutes) than JULIA (mean = 39 minutes).

TABLE II: The distribution of the different annotations that the participants added to the subject programs (BH and MST). Rows `@SuppressWarnings`, `assert`, `@Nullable`, `@NonNull` report the total number of each annotation for each program and TQI tool. "Suppressed Statements" is the number of statements suppressed by the `@SuppressWarnings` annotations, which we calculated by counting the number of semicolons in the suppressed piece of code. "Unresolved Errors" is the number of problems that the qualifier checker reported after the participants finished annotating the programs.

	BH		MST	
	JULIA	CASCADE	JULIA	CASCADE
`@Nullable`	71	61	80	55
`@NonNull`	2	1	4	9
`@Suppress-Warnings`	1	1	6	4
Suppressed Statements	1	1	6	12
`assert`	1	7	7	4
Unresolved Errors	0	0	2	0

3) Quality of Results (RQ1b): We computed the distribution of the annotations that the participants inserted using each of CASCADE and JULIA as a proxy for the overall quality of the annotations. Table II shows the total number of each annotation for each program and qualifier inference tool. Overall, the participants inserted fewer `@Nullable` annotations using CASCADE than JULIA. This indicates that the participants were able to avoid unnecessary `@Nullable` annotations by placing a combination of `@SuppressWarnings` and `@NonNull` annotations and `assert` statements at appropriate places.

The `@SuppressWarnings` annotation is used to suppress a false positive reported by the qualifier checker. The participants inserted fewer `@SuppressWarnings` annotations with CASCADE than JULIA. However, the `@SuppressWarnings` annotations that the participants inserted with CASCADE suppressed more statements. The reason was that one participant chose to annotate a whole method as `@SuppressWarnings` while all other `@SuppressWarnings` annotations suppressed the checker for a single variable declaration.

Programmers can write `assert` statements to make the qualifier checker aware of certain properties. Use of `assert` is justified for those properties that hold at run time but the qualifier checker cannot verify statically. The participants inserted slightly fewer `assert` statements with CASCADE than JULIA.

One participant ran out of time while annotating MST with JULIA and left two errors of the qualifier checker unresolved.

4) Ease of Learning (RQ1c): Despite the fact that JULIA is a single push-button tool and CASCADE offers several interactive features, the participants rated the two tools equally easy to learn (Table III). For example, P_{12} said:

> I just watched the tutorial once and found them easy to understand and learn.

5) Control (RQ1d): The results of the postquestionnaire (Table III) indicate that the participants felt more in control with CASCADE than JULIA. P_5 mentioned the following on the postquestionnaire:

> Even though I could normally go back and change things in JULIA, I had no control over the process (since it was a batch script)
> With CASCADE I could choose whether I wanted to make each of the changes it suggested.

TABLE III: Number of participants of the lab study who preferred each qualifier inference tool (the first two columns) with respect to each quality (rows). The last column lists the number of participants with no preference.

T = CASCADE or JULIA	CASCADE	JULIA	no preference
I found T easy to learn.	3	3	6
I know why T inserted each annotation.	8	0	4
Using T, I have control over the annotation process.	9	3	0
I am willing to use T in future.	11	1	0

6) Willingness to Use (RQ1e): According to the postquestionnaire results (Table III), the participants are more willing to use CASCADE than JULIA, assuming that robust and efficient implementations of both tools are available.

P_9 mentioned that she would be willing to use JULIA for legacy software.

> I would use JULIA if I wanted to annotate a project that is not starting now, so there is a lot of code that needs to be annotated immediately. The feature of automatically inserting annotations would be very useful in this case, provided that the amount of annotations I do not understand is not excessive.

7) Predictability (RQ1f): A predictable program transformation tool is one that makes it easy for the programmers to tell what code changes it made and why. Table III indicates that the participants knew better with CASCADE than JULIA why the tool inserted each annotation. Participants reported that CASCADE made it easier to find the reason ($N = 7$) and location ($N = 1$) of inserted qualifiers. On the other hand, participants found it difficult to find the reason ($N = 6$) and location ($N = 1$) of the qualifiers that JULIA inserts. Participants said that JULIA adds many annotations ($N = 2$) including unnecessary annotations ($N = 2$) and does not explain why it inserts each annotation ($N = 3$).

For example, P_9 said:

> For CASCADE, since there were no annotations in the program, I had to start adding annotations, which was good for me because I could see why I needed each annotation and I had control over this procedure. With JULIA, a lot of annotations were already added, so, I had to change them. For some reason, this was harder for me because changing something that is already there is harder. You need to consider why it was placed at this point without having the whole program in your head.

As another example, P_{12} said:

> Since I'm not very familiar with the algorithm JULIA uses, it feels a bit like a black box to me.

8) Speculative Analysis (RQ2): During the interviews, we asked the participants about the usefulness of the tree that CASCADE computes through a speculative analysis. Eight

participants indicated that the speculative analysis of CASCADE is useful. For instance, P_8 said:

> I really like that. I could see how the warnings propagated through things more easily as opposed to just running the checker, which is kinda like oh this doesn't work, well, what if I do this, oh there is a new one.

The participants said that CASCADE (i) makes them more careful ($N = 3$), (ii) helps them understand and think about the code structure ($N = 2$), and (iii) helps them focus on one problem at a time ($N = 1$) by showing the consequences of applying each change.

For example, P_4 said:

> It [The CASCADE tree] was very good. [...] To some extent, I got a feel about how deep the effects are, where actually the source of the error is, where you could follow multiple paths, either you could fix the source or fix the whole tree. [...] It made you think more about is the solution it gave you the right solution or you could change something at the source of the tree so that it would give you a different possible fix.

As another example, P_3 mentioned:

> I looked at the suggestions by CASCADE but also it made me actually think about the code structure.

9) Strategies (RQ3): We investigated how the participants used each qualifier inference tool to identify the common strategies that they employed. By encouraging the good strategies and discouraging the bad ones, qualifier inference tools can become more effective.

a) Exploration Order: CASCADE does not impose any restrictions on the order in which the programmers expand the tree and apply changes. Four participants first applied the changes on the shorter paths of the tree. This observation can guide the automatic ordering of the nodes of the tree.

Two participants did not expand any changes. They only expanded errors, applied the suggested changes, and recompute the tree. Effectively, these participants explored the tree in a breadth-first order. Since these participants did not use the speculative feature of CASCADE much, they had to frequently recompute the tree, which was slow. As a result, these two participants were less satisfied with CASCADE.

b) Change Application Order: Although CASCADE computes the changes from the root to the leaves of the tree, it does not impose an order for applying the changes. Two participants applied the changes on a path from the leaf to the root of the tree, and one participant applied the changes of a path out of order. The rest applied the changes of a path from the root to the leaf.

c) Long Paths: Longs paths in the CASCADE tree indicate deep consequences of a qualifier change. A good strategy that the participants employed when they encountered long paths was to examine the path, read the code, and refactor the code, if possible, to cut the path short. On the other hand, some participants ran into problems in handling long paths. Two participants applied the changes on the paths as they expanded the paths. Rather than first examining the whole path, they applied the changes prematurely. When these participants reached the leaves of the tree and noticed the unresolved errors, this strategy made the participants backtrack some of the changes they had applied. Similarly, one participant applied the last few changes on a path without examining them. Programmers should be more careful in handling the paths that leave unresolved errors. One way to improve CASCADE is to make it discourage careless change applications by warning the programmers about those paths of the tree that leave unresolved errors and require closer attention.

10) Performance: Participants said that the tree computation of CASCADE was slow ($N = 5$) and JULIA was faster ($N = 3$).

For instance, P_7 said:

> With CASCADE, one thing that annoyed me was that it's kinda slow, because I think it is meant to be interactive. [...] With JULIA, yes, I'm going to do it in the old "write code, compile, look at error" cycle. So, it was familiar. I didn't expect any better of it. So, it didn't matter that it was on the slower side. If CASCADE was faster, I'd probably be happier using it.

Four said that the manual work required to use CASCADE may make it unsuitable for large projects. On the other hand, they complained that the overhead of understanding and fixing the annotations inserted by JULIA is high ($N = 7$).

Although CASCADE's tree computation was slow, the participants finished the tasks more quickly with it than JULIA (Section VI-D2).

11) Suggestions for Improvements: Two participants suggested that CASCADE lets the programmer apply all the changes inside a subtree at once. For instance, P_5 said:

> [Had the code been more complicated,] I would have liked the strategy to have been right-click, fix this tree for me rather than me visiting the entire tree. And, if there is some point in the leaf this sort of error, then tell me and I'll deal with it manually. Basically, kinda what JULIA is doing but localized on the tree basis. So, I would've said run Julia on that tree kinda thing. [...] This is why I feel like a combination of the two would have been better.

Two participants expected that applying all the changes that CASCADE suggests would resolve all the problems. For instance P_6 said:

> With CASCADE, when you apply a change, new errors will come and it was deceiving that you would apply a change and you'll get new errors. [...] After I applied all changes on the tree, I still had errors.

Similarly, P_7 said:

> With CASCADE, I got a little cocky because it seemed CASCADE would be very clever. And, so, I just kept accepting its annotations and suggestions and ended up annotating myself into a corner, because I ended up in a case where you had a `@Nullable` annotation and it was being dereferenced and wasn't obvious how to fix it. So, I had to go back and again look at the code, understand it myself, and fix things. I think CASCADE was a false sense of security because of the way it was working.

CASCADE's change composition plan is meant to be suggestive as opposed to definitive. While applying all changes on some paths of the tree leaves no problems behind, some other paths eventually leave some problems. If the leaf node of a path is an error message node, applying all the changes on that path will result in the error message of the leaf node. One

way to discourage the programmer from taking CASCADE's suggestions as definitive is to warn the programmer about the paths in the tree that leave some problems behind.

One participant found the overlaps between the subtrees of CASCADE confusing. Two participants suggested that CASCADE organizes the changes according to data structures instead of the call hierarchy.

Participants suggested that JULIA be made more interactive ($N = 3$) and explain what ($N = 1$) and why ($N = 3$) it changes.

One suggested that JULIA avoids making changes that cause the checker to report errors.

> With JULIA, I'll say that the program should run automatically but not implement all the changes. For example, we ran JULIA and there was error given by the checker. Let's say it does not apply changes to that set of tree where after compilation it gives you an error. You just ask the programmer I cannot figure out for this tree. A mix of CASCADE and JULIA, that's what I'm talking about.

Qualifier inference tools are bound to be inaccurate. The results of the study suggest that the qualifier inference tools should avoid propagating inaccurate qualifiers across the code base. Such a propagation often causes additional overhead as the programmers have to understand and revise the propagated qualifiers. One possible way to improve the existing batch qualifier inference tools is to report possible sources of inaccuracies and qualifier checker errors to the programmers and let the programmers decide how to handle such difficult cases. This will essentially bring some of the strengths of CASCADE into batch qualifier inference tools and make them more iterative.

On the other hand, CASCADE requires the programmers to confirm the insertion of each qualifier. While these confirmations put the programmers in control, they can also make the inference process tedious. One way to make the inference process of CASCADE more efficient is to incorporate some of the analysis that batch qualifier inference tools perform in CASCADE. Such an analysis can be used to make CASCADE automatically insert some of the qualifiers that are accurate and do not cause the qualifier checker to introduce new errors.

VII. LIMITATIONS

The generalizability of the findings is limited by the extent to which JULIA represents batch qualifier inference tools. Among the qualifier inference tools that we evaluated, we believe that JULIA is currently the most mature and accurate qualifier inference tool for nullness in the batch paradigm. Similarly, CASCADE is only one of the possible incarnations of compositional refactoring and speculative analysis.

The time limits on lab studies constrain the choice of subject programs. Factors that may affect the performance of a qualifier inference tool include the size of the subject programs, preexisting qualifiers in the programs, and dependence of the subject programs on libraries. While one can speculate about the affect of these factors on the performance of a batch and compositional qualifier inference tool, more research is required to evaluate such speculations.

Because of the limited duration of the lab study, the study compared only two tools. Future research can study other configurations such as inserting qualifiers without a special qualifier inference tool and relying on only the qualifier checker or using a combination of a batch and compositional qualifier inference tool.

During a lab study, the participants do not commit to long-term maintenance of the programs. The desired maintainability of the qualifiers is another factor that may affect the desirability of a qualifier inference tool. To avoid too much variability in the subject programs and mitigate unexpected bugs of the qualifier inference tools in unknown code, we asked the participants to annotate two programs that we selected from a suite of benchmark programs.

Qualifiers were a new feature of Java by the time of our study. Thus, we trained the participants about the qualifiers. Nonetheless, the level of familiarity with qualifiers may affect the results of the study. As more programmers learn about qualifiers, future studies can experiment with those who are more experienced with qualifiers.

VIII. RELATED WORK

A. Compositional Refactorings

Compositional refactoring [42] was inspired by our studies to find the common reasons of underusing the existing automated refactorings [43]. Compositional refactorings mimic the steps that a programmer takes while performing large refactorings. Each refactoring is small and its changes are predictable and understandable. The changes are presented directly to the user in the code editor. Our field study [43], analysis of refactoring usage data [42], and survey and lab studies [42] all suggest that overall programmers prefer the compositional paradigm over the existing wizard-based paradigm.

B. Speculative Analysis

Speculative analysis assists programmers in making decisions by precomputing the consequences of the decisions. Although speculative analysis, also known as speculative execution, is an old optimization technique used in domains such as computer architecture and database systems, it has been applied to software engineering only recently. The goal of the existing applications of speculative analysis to software engineering tasks is to improve the productivity of programmers not the performance of the system. Quick Fix Scout [29] employs speculative analysis to make better suggestions for resolving compilation problems. Solstice [28] is a general framework that uses speculative analysis to turn an offline analysis into a continuous analysis. Crystal [6] informs programmers ahead of time about the conflicts, build errors, and test failures that upcoming Version Control System (VCS) operations will introduce.

CASCADE differs from existing applications of speculative analysis in two major ways. First, it brings speculative analysis to a new domain, namely, qualifier inference. Second, it uses *deep* as opposed to *shallow* speculative analysis. A shallow speculative analysis reasons about the state of the system only

one step away. However, a deep speculative analysis reasons about the state of the system several steps away.

C. Type Qualifier Inference

Researchers have developed many qualifier inference tools (Table I). We discussed the differences between CASCADE and the existing batch qualifier inference tools through our comparative lab study (Section VI). In the following, we discuss two tools that, similar to CASCADE, rely on an existing checker for inferring annotations. Houdini [14] is a tool for inferring the annotations required by ESC/Java [15], a static program checker, and CANAPA [10] is a tool for inferring nullness annotations for ESC/Java2, a static checker for the Java Modeling Language [7]. Houdini, CANAPA, and CASCADE are similar in that they use an existing checker for inferring annotations. This technique makes these three tools general. Unlike CASCADE, Houdini and CANAPA are not compositional, because they insert all the annotations that they infer at once, a change that tends to be large and unpredictable. The annotations inferred by Houdini and CANAPA may cause the checkers to report errors. Houdini generates an HTML report to help the programmer in finding the cause of each error. CASCADE assists the programmer to find the root causes of errors by computing an interactive tree of related changes and errors. Houdini and CANAPA are not speculative either, because they do not present the consequences of inserting the annotations in advance.

D. Type-Error Messages

CASCADE infers individual automated changes from the error messages that the type-checker reports. Techniques that improve type-error messages [19], [24], [25], [45] can enable CASCADE to infer better changes for more type-errors.

IX. Future Work

A. Appropriate Levels of Automation for Software Engineering Tasks

Researchers and practitioners invest in automating many software engineering tasks. Some of these efforts have led to technologies that programmers have adopted, e.g., IDEs, Version Control Systems, Continuous Integration Systems. However, some other automation efforts, such as automated refactorings, have not been widely adopted [27], [31], [43]. Our prior work shows that a major cause of the low adoption of refactoring tools is their over-automation [42], [43]. Over-automation leads to inappropriate feedback and interaction, which discourages programmers from using the automation. Do the automation technologies for other software engineering tasks suffer from over-automation? Asking this question is an important first step in finding an appropriate level of automation and interaction model.

This paper advocates lower levels of automation for qualifier inference. The participants appreciated the compositional and speculative aspects of CASCADE. Thus, we expect CASCADE to be more effective than only a type-checker for inferring qualifiers. Nonetheless, future studies can evaluate even lower levels of automation for qualifier inference.

B. Type Qualifier Inference

Our participants pointed out that CASCADE's tree computation is slow. Given that CASCADE is an interactive tool, it is important for it to be fast. The goal of the user study was to assess the user interaction model and not the performance of CASCADE. After the user study, we made the tree computation of CASCADE asynchronous so that the programmer can explore the tree while it is being computed. There are several ways to optimize the performance of CASCADE in future. First, disjoint subtrees of CASCADE can be computed in parallel. Second, the tree can be computed lazily as the programmer expands the tree. Finally, it is possible to use the copy-on-write strategy to avoid taking many copies of the whole code for speculative analysis. Such optimizations may make it possible to automatically refresh the CASCADE tree to keep it consistent with the code.

Currently, the speculative analysis of CASCADE is *unidirectional*. That is, it propagates qualifiers in one direction: right to left. For instance, if the qualifier checker reports a type mismatch in an assignment, CASCADE proposes a change to propagate the qualifier from the right-hand side of the assignment to its left-hand side. However, the programmer may sometimes have to propagate qualifiers from right to left, e.g., when the code is partially annotated or it depends on a library. Currently, CASCADE requires the programmer to do the left-to-right propagation manually and then refresh CASCADE to see the consequences of the propagation. Future research can extend CASCADE's speculative analysis to be *bidirectional*. Bidirectional speculative analysis shows the programmer a larger part of the solution space. The challenge is to present this larger solution space to the programmer concisely and intuitively.

X. Conclusions

Compositional refactoring and speculative analysis worked well together to shape the design of CASCADE. It is likely that these two concepts be suitable for automating other software engineering tasks, e.g., debugging and other program transformations. CASCADE is the first *universal* type qualifier inference tool. That is, it takes a type qualifier checker as input and uses it to assist the programmer infer the qualifiers. Rather than offering more automation, CASCADE takes the opposite direction and reduces the level of automation for inferring type qualifiers. Reducing the level of automation makes CASCADE easier to implement and more usable than existing batch type qualifier inference tools. The results of our comparative lab study show that CASCADE is easy to learn, gives more control to the programmers, is faster, and programmers are more willing to use it. A lesson to learn from this work is that reducing the level of automation can lead to superior results, especially when programmers can achieve better results than the automatic approach.

XI. Acknowledgments

We thank Vikram S. Adve, Brian P. Bailey, and Roshanak Zilouchian Moghaddam for their valuable suggestions and

our participants for taking part in the study. This work is partially supported by NSF CCF 11-17960 and CCF-0963757 and DOE DE-FC02-06ER25752. This material is based on research sponsored by DARPA under agreements number FA8750-12-2-0107 and FA8750-12-C-0174. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] Cascade. https://github.com/reprogrammer/cascade.

[2] JSR 308: Annotations on Java Types. http://jcp.org/en/jsr/detail?id=308.

[3] Type Qualifier Inference Study Artifacts. https://github.com/reprogrammer/tqi-study.

[4] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A Framework for Implementing Pluggable Type Systems. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–74, 2006.

[5] S. Artzi, J. Quinonez, A. Kieżun, and M. D. Ernst. Parameter reference immutability: Formal definition, inference tool, and comparison. *Automated Software Engineering*, pages 145–192, 2009.

[6] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early Detection of Collaboration Conflicts and Risks. *IEEE Transactions on Software Engineering*, pages 1358–1375, 2013.

[7] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO)*, pages 342–363, 2006.

[8] B. Chin, S. Markstrum, and T. Millstein. Semantic Type Qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–95, 2005.

[9] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of User-Defined Type Qualifiers and Qualifier Rules. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 264–278, 2006.

[10] M. Cielecki, J. Fulara, K. Jakubczyk, and Ł. Jancewicz. Propagation of JML Non-null Annotations in Java Programs. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java (PPPJ)*, pages 135–140, 2006.

[11] W. Dietl, M. D. Ernst, and P. Müller. Tunable Static Inference for Generic Universe Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 333–357, 2011.

[12] T. Ekman and G. Hedin. Pluggable checking and inferencing of nonnull types for Java. *Journal of Object Technology*, 2007.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, pages 99–123, 2001.

[14] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, pages 500–517, 2001.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[16] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 1035–1087, 2006.

[17] C. S. Gordon, W. Dietl, M. D. Ernst, and D. Grossman. JavaUI: Effects for Controlling UI Object Access. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 179–204, 2013.

[18] D. Greenfieldboyce and J. S. Foster. Type Qualifier Inference for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 321–336, 2007.

[19] T. J. Halloran. *Analysis-Based Verification: A Programmer-Oriented Approach to the Assurance of Mechanical Program Properties*. PhD thesis, 2010.

[20] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and Checking of Object Ownership. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 181–206, 2012.

[21] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 879–896, 2012.

[22] L. Hubert. A Non-Null Annotation Inferencer for Java Bytecode. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 36–42, 2008.

[23] L. Hubert, T. Jensen, and D. Pichardie. Semantic Foundations and Inference of Non-null Annotations. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 132–149, 2008.

[24] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. Path Projection for User-Centered Static Analysis Tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 57–63, 2008.

[25] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for Type-Error Messages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 425–434, 2007.

[26] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying Dereference Safety via Expanding-Scope Analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–223, 2008.

[27] E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, pages 5–18, 2011.

[28] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin. Making Offline Analyses Continuous. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 323–333, 2013.

[29] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative Analysis of Integrated Development Environment Recommendations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 669–682, 2012.

[30] M. G. Nanda and S. Sinha. Accurate Interprocedural Null-Dereference Analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 133–143, 2009.

[31] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 552–576, 2013.

[32] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[33] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 201–212, 2008.

[34] J. Quinonez, M. S. Tschantz, and M. D. Ernst. Inference of Reference Immutability. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 616–641, 2008.

[35] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium (SSYM)*, 2001.

[36] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *FTfJP 2012: 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26, June 12, 2012.

[37] F. Spoto. The Nullness Analyser of Julia. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 405–424, 2010.

[38] F. Spoto and M. D. Ernst. Inference of Field Initialization. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 231–240, 2011.

[39] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 104–128, 2008.

[40] M. S. Tschantz and M. D. Ernst. Javari: Adding Reference Immutability to Java. In *Proceedings of the ACM SIGPLAN Conference on Object-*

Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 211–230, 2005.

[41] M. Vakilian. *Less Is Sometimes More in the Automation of Software Evolution Tasks*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

[42] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A Compositional Paradigm of Automating Refactorings. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 527–551, 2013.

[43] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, Disuse, and Misuse of Automated Refactorings. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 233–243, 2012.

[44] K. Weitz, G. Kim, S. Srisakaokul, and M. D. Ernst. A Type System for Format Strings. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 127–137, 2014.

[45] D. Zhang and A. C. Myers. Toward General Diagnosis of Static Errors. In *Proceedings of the 41st Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 569–581, 2014.

[46] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and immutability in generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 598–617, Oct. 19–21, 2010.