Pluggable Type Inference for Free

Martin Kellogg Daniel Daskiewicz Loi Ngo Duc Nguyen Muyeed Ahmed
New Jersey Institute of Technology
Newark, NJ, USA
{martin.kellogg,dd482,ln3,ma234}@njit.edu

Michael D. Ernst
University of Washington
Seattle, WA, USA
mernst@cs.washington.edu

***Abstract*—A pluggable type system extends a host programming language with type qualifiers. It lets programmers write types like `unsigned int`, `secret string`, and `nonnull object`. Typechecking with pluggable types detects and prevents more errors than the host type system. However, programmers must write type qualifiers; this is the biggest obstacle to use of pluggable types in practice. Type inference can solve this problem. Traditional approaches to type inference are type-system-specific: for each new pluggable type system, the type inference algorithm must be extended to build and then solve a system of constraints corresponding to the rules of the underlying type system.**

We propose a novel type inference algorithm that can infer type qualifiers for *any* pluggable type system with little to no new type-system-specific code—that is, "for free". The key insight is that extant practical pluggable type systems are flow-sensitive and therefore already implement local type inference. Using this insight, we can derive a global inference algorithm by re-using existing implementations of local inference. Our algorithm runs iteratively in rounds. Each round uses the results of local type inference to produce summaries (specifications) for procedures and fields. These summaries enable improved inference throughout the program in subsequent rounds. The algorithm terminates when the inferred summaries reach a fixed point.

In practice, many pluggable type systems are built on frameworks. By implementing our algorithm *once*, at the framework level, it can be reused by *any* typechecker built using that framework. Using that insight, we have implemented our algorithm for the open-source Checker Framework project, which is widely used in industry and on which dozens of specialized pluggable typecheckers have been built. In experiments with 11 distinct pluggable type systems and 12 projects, our algorithm reduced, by 45% on average, the number of warnings that developers must resolve by writing annotations.

***Index Terms*—Pluggable type systems, type qualifiers, typechecking, type inference, static analysis**

I. Introduction

A pluggable type system [1] augments a host type system with *type qualifiers* that refine it. A qualified type is more fine-grained than an unqualified one and therefore gives more precise information about what values are possible at run time. It lets programmers write types like `unsigned int`, `secret string`, and `nonnull object`. Pluggable type systems can prevent null-pointer dereferences [2, 3, 4, 5, 6, 7], out-of-bounds array accesses [8, 9], violations of locking discipline [10, 11, 12, 13, 14], mutations of immutable data [15, 16, 17, 18, 19, 20], units of measurement errors [21, 22], and more. A successful typechecking run proves that these undesirable behaviors will never occur at run time. Pluggable type systems are a standard practice in industry; for example, they are used at Amazon [23, 24, 25], Google [26], Meta [27], and Uber [5].

Pluggable types are an attractive verification and bug-finding strategy because programmers are familiar with type systems and are used to writing types. Another benefit is that the type qualifiers serve as concise, machine-checked documentation. However, writing type qualifiers in a legacy codebase is intimidating and time-consuming for developers. This has hindered more widespread adoption of pluggable types. An alternative is type inference, which computes a set of type qualifiers that are consistent with the program.

Here is a typical usage scenario to verify a legacy codebase using a type inference tool. Before starting, the user (a programmer) runs the inference tool on the whole program (a slow process), which saves inference results to a side file and outputs typechecking warnings. The number of warnings is usually smaller than the user would see if they typechecked the program without inference. For each warning, the user fixes a bug in the code; writes annotations in the source code to override undesirable inference results; or suppresses the warning if it is a false positive that cannot be resolved by writing annotations. The programmer re-runs the inference tool on only the changed code, code that depends on it, and code that depends on any changed inferences. Eventually, the codebase contains a few human-written annotations and has been verified to be free of certain errors. The annotations in the side file may be optionally inserted into the codebase. Further development uses the typechecker (which is fast) on each compilation to keep the codebase verified.

The traditional approach to type inference is constraint-based. First, generate a set of constraints induced by the source code via a syntax-directed analysis, similarly to how typechecking rules apply to code. Unlike typechecking, which is a modular analysis, this approach to type inference is inherently whole-program: constraints might be generated from type uses that are far from the corresponding declarations. The step after generating the constraints is to solve them, which requires iterating over them. For example, languages like ML and Haskell use Hindley-Milner type inference based on algorithm W [28] (but it is a good practice to write explicit types as documentation [29, 30, 31, 32]). The constraint-based approach is challenging for object-oriented programming languages such as Java and Python, because subtyping significantly complicates unification of types; combining subtyping and ML-style type

This research was supported in part by the National Science Foundation under grant CCF-2312262.

inference remains an open research problem [33, 34].

The traditional approach requires writing a type constraint generator for every (pluggable) type system—essentially, to re-implement the typechecker—which is a heavy burden. One complication is that non-trivial typecheckers contain procedural code that may be difficult to translate into a declarative form. We desire an inference algorithm that is *generic* over the pluggable type system to which it is applied. The pluggable type system designer should not need to modify their pluggable typechecker's implementation in order to access the benefits of type inference: that is, type inference should be available to any pluggable type system "for free".

We propose *iterated local type inference*, a general type inference algorithm that is applicable to any flow-sensitive pluggable type system. Our key insight is that frameworks for building pluggable type systems already provide local type inference in the form of flow-sensitivity within the body of methods. This local type inference is effectively an intra-procedural dataflow analysis; iterated local type inference lifts the local dataflow analysis that each checker already possesses to the whole program, similar to a global dataflow analysis [35, 36]. Our approach requires modifying the framework (once, ever) so that it records inferred method, class, and field summaries based on the results of flow-sensitive local typechecking. Iterated local type inference works by iteratively typechecking the program, using and improving the summaries, until reaching a fixed point. That fixed point is a candidate set of type qualifiers that are consistent with the program. This can be viewed as an adaptation of interprocedural dataflow analysis to pluggable type-checking. This idea, though simple to explain, is complex to formalize (section III) and many complications arise in applying it to real type systems (section IV).

We implemented iterated local type inference for the Checker Framework [6], an open-source pluggable type system framework for Java. In our system, which we refer to as Whole Program Inference (WPI), the user (a programmer) can decide whether to insert the inferred type qualifiers in the source code or to store them in a side file. We used WPI to run 11 different pluggable type systems on 88,680 lines of code in 12 projects. In these experiments, our inference approach exactly matched 39% of the ground-truth type qualifiers previously written by programmers and reduced by 45% the number of remaining warnings that a human would need to resolve by writing an annotation.

Our contributions are:

- iterated local type inference, a novel type inference algorithm for flow-sensitive pluggable typecheckers inspired by global dataflow analyses (section III);
- enhancements to the algorithm that make it practical (section IV);
- an implementation of our new type inference algorithm within the Checker Framework, a framework for building pluggable typecheckers (section V); and
- an evaluation of our implementation on 12 projects totaling 88,680 lines of non-comment, non-blank Java code, across 11 different pluggable type systems (section VI).

II. Background

A type is a set of run-time values. A *type qualifier* [1] is a restriction on a type that limits which run-time values the qualified type can represent. For example, `positive int` is a qualified type: `positive` is the type qualifier, and `int` is the base type. A pluggable type system defines a hierarchy of type qualifiers. Each pluggable typechecker is effectively an abstract interpretation [37], with the abstract interpretation's lattice being equivalent to the type qualifier hierarchy.

Practical pluggable type systems are intra-procedurally flow-sensitive. That is, an expression can have different types on different lines of the program, subject to its declared type as an upper bound. For example, after an assignment `x.f = somePositive` or a test `x.f > 0`, the type of `x.f` changes from `int` to `positive int` until a possible side effect or a control flow join. This is called flow-sensitive type refinement [7] or local type inference. This nearly eliminates the need for programmer-written annotations (and, by enabling reassignment, certain temporary variables) within method bodies. As a result, programmers typically do not need to write type qualifiers within method bodies, only on APIs: class, method, and field declarations.

The need to write annotations only on APIs is a welcome labor savings in pluggable type systems. Recent developments echo it in mainstream languages like Java (which has the `var` keyword) and Kotlin (where local variable types are optional). However, our goal is to lift even this burden.

A pluggable typechecker permits programmers to leave base types unqualified. On APIs, the typechecker uses defaulting rules to assign a qualifier to each unqualified base type. For example, although a programmer might write the type `object`, the type-checker interprets it as `nonnull object` or `nullable object`, depending on the defaulting rules. Within a code block, the typechecker infers type qualifiers.

For example, consider a pluggable type system designed to prevent negative array accesses. It would require that the type of any index used to access an array is non-negative. The following code does not typecheck as written:

```
// Returns the value in a at the index. For this procedure,
// the first element of the array is at index 1.
string getOneIndexed(string[] a, int index) {
  return a[index - 1];
}
```

Because `index`'s type `int` is unqualified (i.e., a base type), a pluggable typechecker would default it, most likely to a worst-case assumption that `index` could be any integer (the "top qualifier" or $\top$). With this type, every call to `getOneIndexed` would typecheck, but its body would not.

To make the code typecheck (equivalently, to verify that its array accesses are not at negative indices) a programmer would write the formal parameter as `positive int index`.

Like its host type system, a pluggable type system is modular: it can be run incrementally on a procedure or a file at a time. It uses only the specifications (i.e., the types) of called procedures: it never has to reason about the implementations of other procedures, only their specifications or summaries.

Within the procedure body, the typechecker *relies* on the fact that `index` has type `positive int`. At call sites, the typechecker *guarantees* that only positive integers are passed as arguments. This rely–guarantee approach is globally sound so long as every procedure is checked.

The annotation burden scales linearly with the size of the code base, which may be large for legacy code. Furthermore, many pluggable type systems require significant numbers of annotations. For example, a pluggable type system for preventing out-of-bounds array accesses required one type qualifier for every 32 lines of non-comment, non-blank code [9].

Our goal in this work is to avoid the burden of writing these type qualifiers. Our approach is to automatically convert a flow-sensitive pluggable typechecker (like the ones that exist in practice) into one that performs inter-procedural type inference.

III. Iterated Local Type Inference

This section presents *iterated local type inference*, our type inference algorithm. Instead of designing a specialized type inference algorithm for each type system, our approach is to modify the underlying *framework* on which the pluggable typecheckers are built. Iterated local type inference is independent of the underlying pluggable typechecker: adapting an existing pluggable typechecker is automatic.

Our key observation is that practical pluggable typecheckers already use *intra-procedural, flow-sensitive* type refinement. That is, they infer type qualifiers within method bodies (i.e., *locally*) based on dataflow facts, in a similar manner to an abstract interpretation [38]. This local dataflow analysis can be lifted into a global dataflow analysis, just as one would lift an intra-procedural dataflow analysis or abstract interpretation to an inter-procedural analysis.

The first idea behind iterated local type inference is to expose the results of local type refinement—that is, what facts about non-local locations are inferred as a by-product of local inference. The second idea is to iteratively call a typechecker, reusing these results to obtain type qualifiers for the whole program. In other words, iterated local type inference propagates the results of local, flow-sensitive refinement inter-procedurally until a global fixpoint is reached. Applying these two ideas inside a pluggable typechecking framework converts any typechecker built on that framework into an *inferring typechecker* that can perform iterated local type inference (in addition to typechecking).

More formally, a typechecker $T : P \to E$ takes a program P and outputs a (possibly empty) set of type errors. An inferring typechecker $T_I : \langle P, A\rangle \to \langle E, A'\rangle$ takes a program along with a set of additional type qualifiers A. It outputs errors E and inferences A', which is a new set of type qualifiers. The errors E are exactly those T would output, if the type qualifiers in A had been written on P by a programmer. The original program P may or may not contain programmer-written type qualifiers.

This paper describes the two parts of iterated local type inference separately. First, section III-A explains the modifications to the pluggable typechecking framework that enable iterated local type inference for a pluggable typechecker T (i.e., that convert it to an inferring typechecker T_I). The modified type rules in section III-A expose the results of local refinement. Section III-B gives the "outer-loop" fixpoint algorithm that propagates inferred types throughout the program and iteratively improves them.

A. Inferring Typecheckers: The Inner Loop

This section describes how our modified typechecking framework *automatically* converts a pluggable typechecker T into an inferring typechecker T_I.

Our key insight is to modify the *typechecking framework*'s rules once, to support inference. A pluggable typechecking framework provides the basic typechecking rules that are common to all pluggable typecheckers, support for flow-sensitive type refinement, and other conveniences. Once the pluggable typechecking framework is modified, inference is enabled for every typechecker built on it, regardless of the particular qualifiers the typechecker happens to support.

Our modifications can be conceptualized at the type-qualifier-theory level: that is, we modify the rules for typechecking used by *all* pluggable type systems so that inference is supported. The modified type rules appear in fig. 1.

In fig. 1, $m(f_0,\ \ldots,\ f_n)$ refers to a method declaration: m is a method name, and each f_i is a formal parameter declaration. The syntax $m(f_0,\ \ldots,\ f_n)\ :\ q_R\ \tau_R$ means that m's (qualified) return type is $q_R\ \tau_R$. The syntax $f_i\ :\ q_F\ \tau_F$ means that the declared type of the formal parameter f_i (of m) is $q_F\ \tau_F$.

The typing environment Γ is standard and maps expressions and declarations to qualified types. Because of defaulting, there are no unqualified types in Γ. The *inference environment* Ξ maps declarations to the results of inference, which are *possibly*-qualified types (that is, either qualified or unqualified types). Ξ only maps declarations; Γ already maps expressions to flow-sensitively refined types. Initially, Ξ maps explicitly-annotated type declarations to their qualifier, and all other declarations to "not present". Once every statement in the program has been typechecked, the current round of inference terminates. Its result is all mappings to qualified types in Ξ. Any type that remains unqualified throughout inference does not appear in the output, because no information about it was learned. Keeping types with no information available unqualified is important to prevent spurious output. No matter what qualifier was used for such types ($\bot$, $\top$, the default, ...), it could be inconsistent with, and thereby contaminate, inferred information. The "not present" abstract value is also necessary to prove theorem 1 (section III-C3), which is our termination theorem, to preserve monotonic movement through the type hierarchy between inference rounds.

The function $LUB_Q(q_1, q_2)$ is a variant of the type system's existing least upper bound function that accounts for possibly-qualified types. q_1 and q_2 are each either a type qualifier or "not present". If both arguments are qualifiers, then the result of LUB_Q is their least upper bound. If only one qualifier is present, then LUB_Q's result is that qualifier; if both qualifiers are not present, LUB_Q's result is "not present", resulting in an unqualified result. Effectively, our approach

$$\frac{\Gamma \vdash m(f_0 : q_{F_0}\ \tau_{F_0}, \ldots, f_n : q_{F_n}\ \tau_{F_n}) : q_R\ \tau_R \quad \Gamma \vdash \forall i \in 0, \ldots, n.\ e_i : q_{A_i}\ \tau_{A_i} \quad \Gamma \vdash \forall i \in 0, \ldots, n.\ q_{A_i}\ \tau_{A_i} \sqsubseteq q_{F_i}\ \tau_{F_i} \quad \Xi \vdash \forall i \in 0, \ldots, n.\ f_i : q_{I_i}\ \tau_{F_i}}{\Gamma \vdash m(e_0, \ldots, e_n) : q_R\ \tau_R \quad \Xi \vdash \forall i \in 0, \ldots, n.\ f_i : LUB_Q(q_{A_i}, q_{I_i})\ \tau_{F_i}}\ \text{INVOKE}$$

$$\frac{\Gamma \vdash \texttt{new T}(f_1 : q_{F_1}\ \tau_{F_1}, \ldots, f_n : q_{F_n}\ \tau_{F_n}) : q_R\ \tau_R \quad \Gamma \vdash \forall i \in 1, \ldots, n.\ e_i : q_{A_i}\ \tau_{A_i} \quad \Gamma \vdash \forall i \in 1, \ldots, n.\ q_{A_i}\ \tau_{A_i} \sqsubseteq q_{F_i}\ \tau_{F_i} \quad \Xi \vdash \forall i \in 1, \ldots, n.\ f_i : q_{I_i}\ \tau_{F_i}}{\Gamma \vdash \texttt{new T}(e_1, \ldots, e_n) : q_R\ \tau_R \quad \Xi \vdash \forall i \in 1, \ldots, n.\ f_i : LUB_Q(q_{A_i}, q_{I_i})\ \tau_{F_i}}\ \text{NEW}$$

$$\frac{\Gamma \vdash f : q_F\ \tau_F \quad \Gamma \vdash e : q_A\ \tau_A \quad \Gamma \vdash q_A\ \tau_A \sqsubseteq q_F\ \tau_F \quad \Xi \vdash f : q_I\ \tau_F}{\Gamma \vdash f := e \quad \Xi \vdash f : LUB_Q(q_A, q_I)\ \tau_F}\ \text{FORMAL-ASSIGN}$$

$$\frac{\Gamma \vdash x.f : q_F\ \tau_F \quad \Gamma \vdash e : q_A\ \tau_A \quad \Gamma \vdash q_A\ \tau_A \sqsubseteq q_F\ \tau_F \quad \Xi \vdash C.f : q_I\ \tau_F}{\Gamma \vdash x.f := e \quad \Xi \vdash C.f : LUB_Q(q_A, q_I)\ \tau_F}\ \text{FIELD-ASSIGN}$$

$$\frac{\texttt{return}\ e \in m \quad \Gamma \vdash m(f_0 : q_{F_0}\ \tau_{F_0}, \ldots, f_n : q_{F_0}\ \tau_{F_0}) : q_R\ \tau_R \quad \Gamma \vdash e : q_A\ \tau_A \quad \Gamma \vdash q_A\ \tau_A \sqsubseteq q_R\ \tau_R \quad \Xi \vdash m(f_0 : q_{F_0}\ \tau_{F_0}, \ldots, f_n : q_{F_n}\ \tau_{F_n}) : q_I\ \tau_R}{\Gamma \vdash \texttt{return}\ e \quad \Xi \vdash m(f_0, \ldots, f_n) : LUB_Q(q_A, q_I)\ \tau_R}\ \text{RETURN}$$

$$\frac{\begin{array}{c}\Gamma \vdash m_B(f_{0_B} : q_{B_0}\ \tau_{B_0}, \ldots, f_{n_B} : q_{B_n}\ \tau_{B_n}) : q_{R_B}\ \tau_{R_B} \\ \Gamma \vdash m_P(f_{0_P} : q_{P_0}\ \tau_{P_0}, \ldots, f_{n_P} : q_{P_n}\ \tau_{P_n}) : q_{R_P}\ \tau_{R_P} \\ \Gamma \vdash q_{R_B}\ \tau_{R_B} \sqsubseteq q_{R_P}\ \tau_{R_P} \quad \Gamma \vdash \forall i \in 0, \ldots, n_B.\ q_{B_i}\ \tau_{B_i} \sqsubseteq q_{P_i}\ \tau_{P_i} \\ \vdash n_B = n_P \quad \Xi \vdash m_B(f_{0_B} : q_{B_0}\tau_{B_0}, \ldots, f_{n_B} : q_{B_n}\tau_{B_n}) : q_{R_B\text{-}I}\ \tau_{R_B} \\ \Xi \vdash m_P(f_{0_P} : q_{P_0}\tau_{P_0}, \ldots, f_{n_P} : q_{P_n}\tau_{P_n}) : q_{R_P\text{-}I}\ \tau_{R_P} \\ \Xi \vdash \forall i \in 0, \ldots, n_B.\ f_{B_i} : q_{B_i\text{-}I}\ \tau_{B_i} \quad \Xi \vdash \forall i \in 0, \ldots, n_P.\ f_{P_i} : q_{P_i\text{-}I}\ \tau_{P_i}\end{array}}{\begin{array}{c}\Gamma \vdash m_B(f_{0_B} : q_{B_0}\ \tau_{B_0}, \ldots, f_{n_B} : q_{B_n}\ \tau_{B_n})\ \textit{is a valid override of}\ m_P(f_{0_P} : q_{P_0}\ \tau_{P_0}, \ldots, f_{n_P} : q_{P_n}\ \tau_{P_n}) \\ \Xi \vdash m_P(f_{0_P} : q_{P_0}\tau_{P_0}, \ldots, f_{n_P} : q_{P_n}\tau_{P_n}) : LUB_Q(q_{R_B\text{-}I}, q_{R_P\text{-}I})\ \tau_{R_P} \\ \Xi \vdash \forall i \in 0, \ldots, n_P.\ f_{P_i} : LUB_Q(q_{B_i\text{-}I}, q_{P_i\text{-}I})\ \tau_{P_i}\end{array}}\ \text{OVERRIDE}$$

Fig. 1. The rules of inferring typechecking, which is the inner loop of iterated local type inference. Applying these type rules once to every statement in a program constitutes the "inner loop" of the inference algorithm. Gray indicates standard type rules for an object-oriented language with Java-like syntax. Black indicates additions to support pluggable typechecking [1]. Red indicates additions to support inference, i.e., our contribution in this paper. Γ is the type environment and Ξ is the inference environment. Throughout, "R" subscripts refer to return types; "F" to formal parameters; "A" to actual arguments; and "I" to inference results. For simplicity, these rules assume that any type qualifier may be "unqualified" (though actually only types in Ξ may be unqualified, because all types in Γ are defaulted). In a (pseudo-)assignment `x=y`, `x` is the "formal" and `y` is the "actual". In the INVOKE rule, the 0th parameter is the receiver parameter, which only exists if `m` is an instance method. In the OVERRIDE rule, the subscripts "B" and "P" are mnemonics for "suBtype" and "suPertype", referring to the overriding method and the overridden method, respectively. In, "$Q_{B_i\text{-}I}$" the "-I" suffix refers to inference results. Type rules that do not require modification to support inference are elided for space.

"lifts" the type hierarchy by adding a new "bottom" type ("not present"). LUB_Q computes the least upper bound over this lifted hierarchy.

For example, consider the RETURN rule. This rule checks that the type of the expression in a `return` statement (τ_A) is a subtype ($\sqsubseteq$) of the method's declared return type (τ_R). The basic rule for a language without qualified types is presented in gray. A standard pluggable typechecker (whose rules are presented in black) extends this rule by requiring that the *qualified* type of the returned expression ($q_A\ \tau_A$) is a subtype of the qualified declared return type ($q_R\ \tau_R$). Note that the typing environment (Γ) contains qualified types, as is standard for pluggable typechecking; every entry in Γ has some qualified type, because either there is a programmer-written type or the typechecker computes a default (by applying its defaulting rules and local type inference). Inferring typechecking makes two modifications to this rule (in red): one new precedent and one new consequent. The new precedent looks up the current estimate of the inferred return type ($q_I\ \tau_R$) in the inference environment (Ξ), which may be "not present" if nothing has yet been inferred for that location. The new consequent updates the inference environment to take into account the qualified type of the return expression ($q_A\ \tau_A$): it takes the least upper bound (using the LUB_Q function described above) of the (locally-inferred) actual expression qualifier q_A and the (globally-inferred, by our algorithm) qualifier from the inference environment q_I.

B. Fixpoint Algorithm: The Outer Loop

Figure 2 gives the outer loop of the iterated local type inference algorithm. It iteratively analyzes the target program P with an inferring version of a pluggable typechecker until either there are no remaining typechecking errors ($E = \emptyset$) or the type qualifiers reach a fixed point ($prevA = A$). The helper function ENABLEINFERENCE is defined by the modifications to the type-checking framework described in section III-A.

input : program P and pluggable typechecker T
output : set of errors E and set of type qualifiers A
def `infer(`P, T`):`

$A \leftarrow \emptyset$
$T_I \leftarrow \text{ENABLEINFERENCE}(T)$
repeat
  $prevA \leftarrow A$
  $E, A \leftarrow T_I(P, prevA)$
until $E = \emptyset \vee prevA = A$
return E, A

Fig. 2. The iterated local type inference algorithm.

This algorithm is stated as a set of rules for a single round of inference and an iterative fixpoint algorithm. It produces the same result as a traditional whole-program type inference implemented with a single worklist. However, it has the major advantage of being practical. It reuses the existing typecheckers and therefore does not require re-implementation for each new type system. This is its key advantage over other approaches.

C. Properties of the Algorithm

1) Soundness: Any infer-then-check approach is sound (in the sense of never certifying an incorrect program) so long as the "check" step is sound. Even if the inference algorithm were to produce incorrect type qualifiers, the checker would reject them: T_I performs the requisite checking.

In the context of type inference, "sound" is usually used to mean "never infers an annotation that cannot be verified." In this sense, our approach is sound if the inferring typechecker is run on the whole program: on every call site, on every assignment, etc. (The proof is standard: by induction on the rules in fig. 1.) In practice, however, this is rarely the case due to separate compilation of libraries and client code; see section IV-H.

2) Completeness: This inference system is *not* complete: it does not and cannot infer all possibly-true type qualifiers for a given type system. To see why not, recall that type qualifiers on formal parameters are inferred from the actual types of the arguments at call sites. If there are no call sites for a method in a given program, then the INVOKE rule will never be fired. No information will be inferred for those formal parameters, and they will remain unqualified (assuming FORMAL-ASSIGN is never used, either, because they are not re-assigned).

The lack of completeness is by design. Our goal is not a set of type qualifiers that perfectly captures every fact that is true about the program we are presented with, but rather a set of type qualifiers that is useful *in practice* for typechecking the program (say, for guaranteeing that it never suffers a NullPointerException). Returning to the example of a method with no call sites, a complete algorithm must infer the bottom type for that method's parameters. When applying type inference to real programs (which may be libraries that are intended to be linked with clients later, due to separate compilation), it is desirable to trust that the defaulting scheme of the typechecker is sensible rather than infer an overly restrictive type qualifier based on no actual information.

```
Integer getN() {
  if (*) return getNull();
  else return getZero();
}

Integer getNull() { return null; }

Integer getZero() { return 0; }
```

Fig. 3. A program that induces both upward and downward movement in a typechecker's lattice, when analyzed by WPI. "*" is non-deterministic choice.

Further, most true type qualifiers do not actually prevent a typechecking error, and are therefore unnecessary. And, each additional qualifier added to the program adds a maintenance burden: that qualifier must be read every time the code is read, etc. A complete type inference algorithm adds to this burden. Another benefit of completeness being a non-goal is that our inference system is permitted to heuristically not infer a type qualifier when doing so might lead to sub-optimal results, which simplifies the implementation when handling complex language features (e.g., reflection).

3) Termination: A termination argument is typically based on some monotonically-changing finite abstraction; for example, abstract interpretations terminate because their lattices have finite height, and the abstract values at each program point monotonically move up the lattice. In our proof of theorem 1 (below), the "not present" value is key. Ignoring the "not present" value, the main loop in fig. 2 might seem to produce a stronger *or* weaker estimate for a particular declaration's (qualified) type in subsequent iterations, due to its interaction with defaulting.

For example, consider the program in fig. 3 and two pluggable typecheckers: one for nullness, with `nonnull` and `nullable` type qualifiers; and one for integers, with $\top$ for "any integer" and `nonnegative` for non-negative integers [9]. Our algorithm's estimates for the return type of `getN()` for these two typecheckers seem to move in opposite directions:

- For nullness, in the first round, the estimate for `getN()`'s return type is the default: `nonnull`, the bottom type. WPI discovers that `getNull()` is `nullable` in the first round. In the second round the inferred type of `getN()` becomes `nullable`. `getN()`'s return type seemingly moves *upward*.
- For integers, in the first round, the estimate for `getN()`'s return type is the default ($\top$), because nothing is (yet) known about the methods it calls. In the first round, both method's types are found to be `nonnegative` (since `null` literals usually have the bottom type in non-nullness type systems, and 0 is non-negative). In the second round, `getN()`'s return type is inferred to be `nonnegative`: it seems to move *downward* from the first to the second round.

This example demonstrates the importance of the "not present" abstract value in the theory presented in section III-A: in the first round, in both of these type systems, no information is inferred about the return type of `getN()`. Without the "not present" abstract value, the defaults that the type system would

```
public int f(int x) {
  if (x >= 0 && x < 100) {
    return g(x + 1);
  } else {
    return 0;
  }
}

public int g(int y) {
  if (y >= 0 && y < 100) {
    return f(y + 1);
  } else {
    return 0;
  }
}
```

Fig. 4. An example of a program that WPI analyzes in a finite but large (and program-dependent) number of rounds.

assign create the appearance of non-monotonicity.

With the "not present" abstract value, however, the reason that our algorithm terminates becomes clear: at each step, each location to which inference applies is either "not present" or has an abstract value from the typecheckers' type hierarchy. From "not present", it can take on any value in the type hierarchy, but once it leaves "not present", it always is monotonically refined upwards. This allows us to state and prove our termination theorem:

Theorem 1: Figure 2 terminates on all programs.

Proof sketch. The proof is a standard analysis termination proof based on monotonically moving up the type hierarchy, with a special case for the "not present" abstract value. □

While theorem 1 guarantees that inference will terminate after a finite number of rounds, that number need not be small. For example, consider the program in fig. 4 and a pluggable typechecker implementing a range analysis, whose type qualifiers constrain an integer to a particular range; e.g., `range(from=0, to=10)` means an integer between 0 and 10, inclusive. In this program, functions `f` and `g` are mutually recursive. Over 100 rounds, WPI will infer larger and larger ranges for their parameters `x` and `y`: in the first round, `range(from=0, to=0)`; in the second round, `range(from=0, to=1)`; etc., all the way up to the 100th round, when the analysis settles on `range(from=0, to=100)`, which is correct.[1] In cases like this, the time the analysis takes is controlled by the program: for this example, we chose the constant `100` arbitrarily, but if `100` were replaced by any other positive integer k, then WPI would require k rounds to terminate. We have not yet observed this behavior in practice, but it suggests the need for some notion of widening in the outer loop, mirroring what is provided by the typecheckers themselves in the inner loop.

IV. Practical Considerations

This section describes some non-obvious challenges in building a system that applies to every pluggable typechecker and is effective at inferring type qualifiers for real programs.

[1]If the range checks on `x` and `y` were removed, WPI would terminate after 1 round, concluding that the type of both is $\top$.

A. Programmer-written Types

WPI never refines nor generalizes programmer-written type qualifiers. It assumes that they reflect the intended specification. Users can use this where the inference output is not as desired.

Inference reports facts (type qualifiers) that are true about a program's implementation. However, these facts may be either stronger or weaker than the programmer's intended specification, depending on the current implementation of a procedure and the observed calls to that procedure. In other words, type inference can cause leakage of implementation details, permitting clients to depend on them. This is undesirable because it constrains future refactoring.

Permitting programmer-written types is a solution to this problem. Even when types are not required such as in some functional programming languages, writing types is a best practice for its documentation benefits [29, 30, 31, 32].

Within a programmer-written `@SuppressWarnings` annotation, WPI does no inference.

B. Storing Intermediate Results

WPI needs to treat type qualifiers supplied by the programmer and by external libraries differently than type qualifiers that were produced during earlier rounds of inference: the latter may be further refined, but the former must not be.

This constraint disqualifies two ways to store intermediate candidate qualifier sets that were already supported by the Checker Framework: (1) insert the inferred type qualifiers in the program, and (2) store the inferred type qualifiers in a side file, like specifications for unannotated libraries.

Strategy 1 makes it impossible to distinguish between qualifiers written by the programmer and qualifiers inferred in previous fixpoint rounds. Strategy 2 does not permit differentiating between library qualifiers and inferred qualifiers. We tried placing all library qualifiers in source code, but this made inference not work properly on libraries for which checker-distributed library qualifier files existed, such as the JDK, Guava, etc.

Instead, we devised a new side file mechanism specifically for storing inferred annotations. Their syntax is the same as library side files, but they have different semantics and are processed separately. Library side files are never modified during typechecking or inference, but inference side files change on each iteration of the outer loop in fig. 2.

C. Generated Code and Analysis Termination

During prototyping, we encountered a program on which WPI did not terminate. WPI implements the $prevA = A$ test of fig. 2 by running Unix's `diff` on side files (section IV-B) that store inferred annotations. These side files also contain the program text, which we had assumed would remain constant. For this one program, however, the *program text itself* was not constant: the build system inserted the current date and time (as a string constant) into the program, to enable a feature that printed the time and date at which the program was compiled. The Java file contained no other code except these ever-changing "constants".

This example highlights an assumption that is easy to make ("the program text is constant") but which can cause a complex system like inference to fail to terminate. We solved the problem on this particular program by not type-checking the offending files; that is, we excluded the offending files from the input to WPI.

D. Writing Output and Cross-file Dependencies

WPI has no way of knowing when the whole program has been processed. (The Checker Framework, and therefore WPI, runs each typechecker as a javac plug-in that is given files to typecheck sequentially.) Therefore, WPI must write inference results for each Java file at the end of processing that file. However, those results might later be modified by calls that are discovered in other files. Therefore, WPI records every modification to inferred types, and at the end of processing each Java file, WPI writes every inferred annotation that has changed since the last time files were written. An annotation side file may be written many times during one round of inferring typechecking.

E. Unqualified Annotations and Defaults

Consistency among overridden methods is different between WPI's and the typechecker's points of view. WPI's unqualified types mean inference has not yet learned any information, and section III-A's LUB_Q ignores unqualified types. However, the typechecker uses a default when there is no annotation. When WPI would write no annotation at some location in a method signature, it checks that the default annotation is consistent with supertypes and subtypes (i.e., it checks behavioral subtyping for methods). If not, it writes consistent, conservative annotations. This process does not change WPI's inference results, only what is written to the file. Changing WPI's inference results would cause WPI to lose the distinction between unqualified types and inference of some type, which would harm further inference in the same round of inferring typechecking.

F. Preconditions and Postconditions

The Checker Framework's specification language contains not only type annotations but also pre- and post-conditions: e.g., a method can specify that it may only be called when field `f` of its first formal parameter is non-null, or can specify that after the method returns, field `g` of its receiver is non-null. These contracts are essential to verifying real-world code.

On entry to and exit from each method definition, WPI converts annotations in the type store into their pre- and post-condition equivalents. For example, if the exit store contains `@NonNull T` as the type for expression `a.b`, then it creates an `@EnsuresNonNull("a.b")` annotation for the method. `@EnsuresNonNull` is a declaration annotation rather than a type qualifier. The implementation is generic across all type systems, not hard-coded for a specific type system such as nullness.

G. Non-type-system Properties

The Checker Framework's specification language contains information about whether a method is deterministic (returns the same output each time it is given the same input), side-effect-free (does not produce any effects observable from outside the method other than its output), or pure (both deterministic and side-effect-free). These are not type properties, but method specifications. These specifications aid flow-sensitive typechecking. The framework conservatively assumes every method is impure, and whenever a (possibly) side-effecting method is called, local inference must discard most type refinements, because the method might re-assign fields.

Because purity is not a type system, the Checker Framework verifies it directly. Our modifications enhance that by inferring these annotations whenever they would be verifiable. We defined a lattice of purity annotations: pure is the top element, deterministic and side-effect-free are sibling elements below it, and we defined a new "impure" abstract value (and annotation, which is invisible to users) to complete the lattice. To determine the purity annotation to infer for a given method, WPI applies this least upper bound to the purity of that method's implementation and the implementations of all of its overriding methods, using the logic of the LUB_Q function (section III-A).

H. Separate Compilation

An inference approach outputs annotations that pass the typechecker only if the whole program is analyzed at once. In practice, Java programs are usually compiled separately. A simple example is that when compiling a library, its set of possible callers is not known. We have also observed this problem in Maven and Gradle sub-projects: a naive application of our approach would treat these sub-projects separately, but this could lead to inferences that are not sufficiently general. Suppose `m` is defined in sub-project P_1 and is only passed non-null values in P_1. Inferring that `m`'s formal parameter is non-null might lead to a spurious typechecking error in sub-project P_2 that passes null to `m`.

Practically, this means that soundness in the traditional sense is not a useful goal for practical inference tools for languages that support separate compilation. A user can mitigate this issue by including test suites in the "whole program" provided to inference. Another mitigation is for users to write specifications for module entry points. However, we still view this as an open problem for practical type inference.

I. Type-system-specific Inference

We added three type-system-specific rules to improve WPI's real-world performance. Together, these modifications comprise 61 lines of code in 2 type systems. We did not observe the need for type-system-specific rules in any of the other 9 type systems. We stress that these modifications are optional: WPI is functional even without them. Section VI-E2 discusses the effect of these changes on our experiments.

1) Nullness: (22 lines of code) We disabled inference of qualified types for receiver parameters, which are always `@NonNull`. We also added a rule to infer the `@MonotonicNonNull` annotation, which means that a field may be null, but once it is assigned a non-null value, it remains non-null forever. Only fields can be `@MonotonicNonNull`, so local inference has no rules

to infer it. We added such a rule: if the only null assignments to a field are in the enclosing class's constructor or the field's initializer, WPI infers it to be `@MonotonicNonNull`, not `@Nullable`.

2) Formatter: (39 lines of code) This checker supports the `@FormatMethod` declaration annotation provided by Google's Error Prone [39], which is redundant with the checker's own `@Format` annotations. Our modifications remove inferred `@Format` annotations from methods that already have `@FormatMethod` annotations, to avoid code clutter and duplication of effort.

V. Implementation

We implemented WPI by modifying the Checker Framework [6]. By implementing our algorithms in the framework itself, our approach is applicable to any typechecker that builds on top of the framework without any modification. WPI provides two implementations of fig. 2:

- A template outer-loop shell script that repeatedly invokes the target project's build system. This requires the user to copy the shell script and edit four variables at the top of the script to set them to project specific values, and to modify the build script itself so that WPI is on the project's annotation processor path (which typically requires <10 lines of modified build script code).
- A tool that automatically instruments a target build script (Ant, Maven, or Gradle) to run a given typechecker with inference enabled, based on the `do-like-javac` tool [40].

Our implementation of iterated local type inference (and, therefore, of whole-program type inference) has been publicly available and documented in the Checker Framework distribution (https://checkerframework.org/) since version 2.0.0 (May 2, 2016). This paper is the first formalization and evaluation of the approach, though some prior work used an earlier version of WPI to ease its evaluation [23, §4.5].

VI. Evaluation

We have deployed WPI at a large software company. We cannot discuss that experience for reasons of confidentiality, so we ran experiments on open-source programs.

Our experiments aim to answer two research questions:

- **RQ1:** Is iterated local type inference *easy to apply* to an existing typechecker? How much per-checker work is required?
- **RQ2:** Is iterated local type inference *effective*? How many of the annotations that humans have written can it reconstruct? After it runs, are there fewer warnings for humans to address?

A. Methodology

We collected projects that have already been annotated for pluggable typechecking (section VI-B). For each subject project, we performed the following steps:

1) remove the existing, ground-truth annotations that express specifications;
2) manually modify the project's build system and WPI's template outer-loop script (usually by adding and/or removing a few command-line arguments to the invocations of typecheckers);
3) run the outer-loop script. Its output is the inferred type annotations.

TABLE I
Projects annotated by a human so that they typecheck with one or more pluggable typecheckers. "LoC" is non-comment, non-blank lines of Java code.

Benchmark	LoC	Checkers
cache2k/cache2k-api [41]	2,615	1: Null
RxNorm-explorer [42]	993	1: Null
Nameless-Java-API [43]	2,831	4: Formatter, Null, Optional, Regex
icalavailable [44]	385	9 (← without Resource Leak)
lookup [45]	274	10 Formatter, Index, Interning,
multi-version-control [46]	1,252	10 Lock, Null, Regex,
reflection-util [47]	1,163	10 Resource Leak, Signature,
require-javadoc [48]	973	10 Signedness, Initialized Fields
randoop [49]	66,200	2: Resource Leak, Signature
dmn-check [50]	5,338	3: Formatter, Null, Regex
table-wrapper-api [51]	4,803	1: Null
table-wrapper-csv-impl [52]	1,853	1: Null
Total	88,680	11 distinct checkers

Finally, we compared the final set of annotations generated by our approach to the original, human-written ground-truth annotations that we removed in step (1). We measure the following quantities:

- The number of inferred annotations. This number is always much larger than the number of human-written annotations: not all inferred annotations are interesting to humans, nor are they necessary to verify the code. However, this number gives a sense of how much information our approach recovers about the program.
- The percentage of the human-written ground truth annotations that were inferred.
- The number of typechecking errors before (on the unannotated program) and after inference completes.

The latter two metrics are proxies for how much work would remain for a human to typecheck the project after running inference. Note that writing a single annotation may enable inference of other annotations and/or resolve multiple warnings, so neither proxy is perfect (cf. section VII-B).

Our scripts and data, including detailed instructions on the manual process we used to adapt existing, annotated projects to use our approach, are available at https://github.com/kelloggm/wpi-experiments. The WPI implementation is open-source as part of the Checker Framework (https://checkerframework.org).

B. Subject Programs

We performed a type reconstruction experiment in which the ground truth is programmer-written type annotations that typecheck using a pluggable typechecker built on the Checker Framework. We emphasize that our inference approach is intended for programs without complete human-written type annotations: we only use human-written annotations here as ground truth to compare to the output of inference.

We constructed our dataset from open-source projects that use the Checker Framework. To find such projects, we searched

GitHub (using the SourceGraph tool [53]) for use of the Checker Framework's Gradle plugin, and similarly for other common integrations with build systems, which we obtained from the Checker Framework manual. (We had initially tried searching for projects that use type annotations, such as `import org.checkerframework...`. This search yielded hundreds of thousands of hits, because the Checker Framework's annotations, such as `@Nullable`, are the *de facto* standard and are used by other tools and for documentation.)

This search yielded 59 projects. Of these, 30 build successfully, and 14 typecheck with at least one pluggable typechecker using version 3.25.2 of the Checker Framework. (We also permitted subprojects; for example, in the cache2k project, only the cache2k-api subproject satisfied these constraints.) We excluded two that cause WPI to crash. The resulting dataset (table I) contains 12 projects, 803 human-written annotations that we use as ground truth, and 88,680 lines of non-comment, non-blank Java source code. The version of the software that we analyzed (that is, an exact commit SHA) appears in our artifact at https://github.com/kelloggm/wpi-experiments.

Our technique and tool are applicable to projects of arbitrary size, and its modular (method-at-a-time) approach makes it scalable. The projects in our dataset are small to medium in size. We suspect that this is in part because of the difficulty of reverse-engineering a legacy program to annotate it with pluggable types. Our work makes such a task easier.

C. Handling Warning Suppressions

When removing human-written annotations, we did not remove `@SuppressWarnings` nor annotations within its bounds. Warning suppressions indicate code that does not pass the typechecker, and users sometimes write a few annotations within the bounds of a warning suppression to indicate facts that they believe are true, but which the type system is unable to prove. A warning suppression indicates there is no verifiable set of type qualifiers, so inference there would produce unverifiable type qualifiers. Our experiments focus on verifiable annotations.

D. RQ1: Generality

We can answer RQ1 in the affirmative: WPI is general over pluggable type systems. WPI is applicable to many typecheckers without modification: 9 of the 11 we considered were completely unmodified. The changes that we made to two typecheckers (section IV-I) totaled only 61 lines of code, which is 0.2% of the implementation of the 11 checkers. Further, these modifications were not strictly *necessary* to get these typecheckers working with WPI: they simply improved performance in some obvious cases that we noticed.

E. RQ2: Effectiveness

Table II shows that for most projects, inference reduces the developer's burden, by inferring a reasonable percentage of annotations and reducing the overall number of warnings that developers must triage. Inference is expected to be slow, but on our server (one node in an HPC cluster, with 32GB of RAM and 8 Intel Xeon Gold 6354 CPUs at 3GHz), the longest-running experiment (randoop) took only 23 minutes; 8 of the 11 projects converge in under one minute.

TABLE II
WPI performance on human-annotated benchmarks without the annotations.

	Annotations				Warnings		
Benchmark	Human	WPI	Human∩WPI		Original	WPI	Reduction
cache2k/cache2k-api	149	1,365	47	32%	62	20	42%
RxNormExplorer	53	415	15	28%	17	9	47%
Nameless-Java-API	251	2,304	176	70%	45	4	91%
icalavailable	8	668	6	75%	5	1	80%
lookup	7	36	1	14%	2	0	100%
multi-version-control	26	848	12	46%	28	4	86%
reflection-util	107	1,130	10	9%	29	20	31%
require-javadoc	2	821	2	100%	5	2	60%
randoop	92	7,899	16	17%	136	100	26%
dmn-check	14	1,163	9	64%	14	14	0%
table-wrapper-api	81	1,103	20	25%	13	18	-38%
table-wrapper-csv-impl	13	188	2	15%	5	5	0%
Total	803	17,940	316	39%	361	197	45%

TABLE III
The number of annotations that WPI cannot infer, by cause (section VI-E1).

Cause	**#**	**%**
Methods with no callers	91	21%
Generics	85	19%
Inferring a stronger annotation	68	15%
Other (no single cause >10%)	198	45%

For one project (table-wrapper-api), WPI's inferred annotations increase the number of warnings that developers need to triage. The primary reason is that six excess warnings are caused by a single annotation that WPI infers as `@MonotonicNonNull` but that ought to be `@NonNull` (WPI's inferred annotations eliminate one other warning, so the total warning count in table II only increases by five, not six). The reason that WPI infers `@MonotonicNonNull` is that the field is non-final and lacks an initializer (either at the field declaration or in the constructor, which is automatically generated by Lombok), so at object creation its value is actually `null`. Though the setter for this field appears to be called before any of the other methods (and its parameter is non-null), WPI is technically correct that an incorrect usage of this class can cause null pointer dereferences. This inference causes six warnings about unguarded dereferences of the field. This sort of overly-pessimistic inference is a downside of WPI's design: it aims to only infer annotations that the original typechecker can verify, which can lead to many warnings about a single problem when the typechecker is unable to prove some linchpin fact.

1) Why Inference Fails: Table III summarizes the reasons that WPI fails to infer annotations that humans wrote. We collected this data by examining each failed inference manually.

a) Methods With No Callers: The inference rules in fig. 1 update the inference environment for method parameters only at call sites of the methods. If a method has no callers in a given project, WPI cannot infer a type qualifier for any of that method's parameters, and therefore leaves them unqualified. The same is true of fields with no assignments. Such methods often appear in projects that are intended to be used as libraries.

Ideally, such projects would come with tests or examples that would enable inference, but they do not always do so.

b) Generics: The type rules in fig. 1 do not include any rules for adding a type to a generic type use or to its bounds (i.e., supporting Java generics); our prototype currently also does not support these locations. WPI could learn that e.g., a `List<int>` that only has `positive int`s added to it actually has the type `List<positive int>`, but this may be overly specific and, due to invariant subtyping for generic types, is incomparable to `List<int>`. Full support for generics—in particular, inferring the qualified bounds of type variables—is much harder [54, 55, 56] due to the triplicate complexity of Java generics, which are Turing-complete and therefore undecidable [57]; pluggable types; and Java's complex generic type inference rules [58, pp. 765–798].

c) Inferring a Stronger Annotation: WPI infers the strongest annotation that is compatible with the code that it analyzes. Sometimes, this annotation is more specific than the human intended. For example, consider the `@EnsuresNonNull` annotation of the Nullness Checker. This postcondition annotation (section IV-F) indicates that its argument(s) are non-null after the annotated method completes. WPI sometimes infers an `@EnsuresNonNull` annotation with *more* information than what the human wrote: for example, it might infer that two fields `f` and `g` are non-null, but the human-written code only accounts for `f`. The human-written annotation in this case is correct, but incomplete: WPI discovers more information. Our experiments conservatively assume that such extra information is not desirable: if WPI does not infer *exactly* the same annotation as the one that the human wrote, we consider it not to have found the correct annotation. If we include stronger annotations when computing the percentage of human-written annotations inferred by WPI, WPI infers an annotation that is at least as strong as 48% of the human-written annotations, compared to 39% as reported in table II.

2) Effect of Checker Modifications: Section IV-I noted that we made three small changes to checker implementations to improve inference. This section discusses how those three changes impact the results presented above. Disabling inference of `@NonNull` on receiver parameters has no effect on correctness (only performance): they would always be inferred to be non-null, the default. Adding a rule for inferring `@MonotonicNonNull` has a non-trivial impact on the results. While there is only a single human-written `@MonotonicNonNull` in our benchmarks (in the reflection-util project), this rule actually is responsible for many of the "Stronger Annotation" cases described in section VI-E1, paragraph *c*: humans seem to commonly write `@Nullable` on fields that are actually `@MonotonicNonNull` (we observed at least 12 cases of this pattern). So, in our experiments, this rule actually makes WPI appear to do worse, because it infers the stronger (and more useful) `@MonotonicNonNull` annotation where humans write `@Nullable`. In our experience actually using WPI, `@MonotonicNonNull` is usually the more useful annotation. The changes to the formatter checker remove a single redundant `@Format` annotation in Randoop, so they have a negligible impact on the results presented here.

VII. Limitations and Threats to Validity

A. Limitations and Future Work

The most important limitation to our approach is that its results depend on the uses in the program or test suite. If the uses from which it learns are unrepresentative, it may infer a qualified type that is different than what the programmer intended. If there are no uses of a declaration in a program at all, WPI cannot infer anything; in practice, this is a serious limitation when analyzing libraries (cf. section VI-E1a). Currently, WPI uses forward reasoning for inference. We plan to extend WPI to use backward reasoning; for example, if a formal parameter is dereferenced before being checked, then WPI could assume that the programmer meant the parameter type to be `nonnull`.

WPI's output also depends on the target program's implementation. If the program contains bugs, then no specification enables verification. A research challenge is to produce specifications such that the verifier warnings are well localized: they indicate the part of the program that needs to be fixed. We plan to investigate whether previous work [59, 60, 61, 62] generalizes to the domain of pluggable type checking. We also plan to investigate prioritizing error messages [63, 64, 65, 66, 67].

The high volume of output from WPI currently makes it implausible to directly present WPI's output to humans. Programmers often write annotations in order to prove a property (such as no null pointer exceptions) rather than to exhaustively document the program's specification (such as the nullness of every reference at every program point). We plan to filter its output so that only facts necessary for a correctness proof are output, which will also reduce the number of warnings that a programmer must address.

WPI lacks support for some language and framework features, such as Java generics and qualifier polymorphism. Section VI-E1 suggests avenues for addressing this limitation.

The current implementation of WPI is too slow for a user to run it repeatedly on the whole program during development. However, it could be incrementalized. On each iteration of the outer loop, the inner loop only needs to analyze code that depends on specifications that changed in the previous iteration. For example, if the inferred types for a class change, then all uses of the class, and also all subtypes of the class, must be re-analyzed. Implementing this optimization is future work.

Typechecking and abstract interpretation are equivalent in expressive power [37], so our approach should generalize to arbitrary dataflow analyses. We plan to test this hypothesis and compare our approach to extant approaches for lifting dataflow analyses [35, 36]. The main requirement is that the dataflow implementation can read specifications from a side file.

B. Threats to Validity

Our choice of benchmarks is a threat to external validity. All of the subject programs already typecheck, and so may already have been modified to better suit analysis with a typechecker. WPI may perform differently on arbitrary unannotated code versus code that typechecks but has had its types erased. Future

work should run the whole-program inference on unannotated programs. Further, the benchmarks are all written in Java; our approach to inference may not generalize to other languages.

There are also a number of threats to construct validity. First, the human-written type annotations that we rely on for ground truth may contain errors: they may be correct but imprecise, or they might suppress warnings in ways that lead to the verification result being unsound. However, we lack a better source of ground truth than the annotations that developers actually write. Second, our metrics (% of human-written annotations inferred and warning reduction %) are proxies for what we are actually interested in, which is developer effort to adopt a pluggable type system. Both are of these proxies are likely weak: humans write annotations for other purposes than typechecking (e.g., for documentation); and warnings do not monotonically decrease with the amount of effort required to verify the program. Future work should include user studies. Third, we used our own judgment to categorize the reasons that WPI did not infer annotations, and we may have miscategorized some uninferred annotations. Fourth, there may be errors in our experimental scripts. To mitigate this threat, we released our scripts, data, and records of the human judgments we made at https://github.com/kelloggm/wpi-experiments.

VIII. Related Work

A. Type Inference For Pluggable Type Systems

Xiang et al. used a constraint-based approach to infer pluggable types for measurement units [22, 68]. By using a MaxSAT solver, it can output a partial typing when the constraints are unsatisfiable. Previous approaches to error explanation for type inference [62, 69, 70, 71] and gradual type migration [72] also use a constraint-based encoding and could be similarly adapted. Constraint-based approaches have a large drawback compared to our approach: they require significant work for each type system, and translating imperative code in a typechecker into constraints is nontrivial. Xiang et al.'s work [22] is specific to unit-of-measurement types, and is not directly applicable to other pluggable type systems. Our work, however, applies to any pluggable type system (including the 11 in our experiments in section VI).

Cascade [73] is an interactive type qualifier inference tool that involves programmers in the inference process. This approach is complementary to ours.

B. Type Inference Approaches

There is a rich literature on classical type inference (or type reconstruction) [74, ch. 22], which focuses on discovering if there exists a (complete) typing for an unannotated program, given a type system. Our problem is different: we nearly always target programs where no such typing exists, and we aim to find a maximal set of useful type qualifiers for such programs. Also, we focus on languages with both subtyping and generics (e.g., Java); handling both remains an open research problem for classical type inference [33, 34].

Researchers have also proposed type inference based on machine learning [75, 76, 77], often targeting partial typings. However, such approaches would lack training data for newly-created pluggable type systems or those with few open-source users. Our approach does not require any training data.

C. Dataflow Analysis

Pluggable typecheckers perform local type inference via dataflow analysis: human-written type annotations are equivalent to function summaries. With this view in mind, our work can be compared to classic work on interprocedural dataflow analysis [35, 36]: WPI is performing a 0-CFA (i.e., context-*insensitive*) analysis to try to recover the correct summaries. In the field of dataflow analysis, 0-CFA analyses are thought to be imprecise, so adding context-sensitivity to WPI is an exciting avenue for future work. Context-sensitivity might also allow WPI to infer "polymorphic" type qualifiers, which express that the annotated type takes on different qualifiers depending on the context in which it is used. In our experiments, polymorphic qualifiers were rarely written by humans (there are just two, both in the reflection-util project), so the improvement in WPI would be marginal. Our work differs from these classic results for dataflow analyses because it applies to a new domain (pluggable typecheckers), resulting in a different formalism. However, we believe that the two lines of research are closely related, and that each may be able to draw on the other.

D. Other Approaches

Like our approach, Houdini [78, 79] runs a verification tool multiple times to fixpoint. It starts from the set of all possible facts (many of which are inconsistent with one another) and on each iteration removes all those that cannot be proven. This can result in removing too many facts, when the program contains a bug or unverifiable code. The specification that Houdini outputs is provable, but it may not establish correctness (e.g., that the program has no null pointer exceptions). By contrast, WPI iteratively adds facts, starting from nothing. Its output is consistent with the program and its test cases, but is not necessarily verifiable.

IX. Conclusion

Pluggable types are a promising verification approach, but the cost of annotating existing legacy code hinders their adoption. Our type inference approach achieves global type inference by bootstrapping an existing typechecker's local inference: it converts a pluggable typechecker to an inferring pluggable typechecker for free. The inference is reasonably effective at reducing the developer's burden to annotate legacy code: in our experiments with 11 different pluggable typecheckers, WPI can infer 39% of human-written annotations and reduce the number of warnings presented to developers by 45% on average.

Acknowledgments

Suzanne Millstein, Jonathan Burke, David McArthur, and Jason Waataja contributed to WPI's implementation. Thanks to the anonymous referees and Manu Sridharan for comments that helped to improve the paper.

References

[1] J. S. Foster, M. Fähndrich, and A. Aiken, "A theory of type qualifiers," in *PLDI '99: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, USA, May 1999, pp. 192–203.

[2] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, 2002.

[3] T. Ekman and G. Hedin, "Pluggable checking and inferencing of non-null types for Java," *J. Object Tech.*, vol. 6, no. 9, pp. 455–475, Oct. 2007.

[4] C. Male and D. J. Pearce, "Non-null type inference with type aliasing for Java," Aug. 20, 2007, http://www.mcs.vuw.ac.nz/~djp/files/MP07.pdf.

[5] S. Banerjee, L. Clapp, and M. Sridharan, "NullAway: Practical type-based null safety for Java," in *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, Aug. 2019, pp. 740–750.

[6] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 2008, pp. 201–212.

[7] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller, "Building and using pluggable type-checkers," in *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*, Waikiki, Hawaii, USA, May 2011, pp. 681–690.

[8] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *PLDI '98: Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998, pp. 249–257.

[9] M. Kellogg, V. Dort, S. Millstein, and M. D. Ernst, "Lightweight verification of array indexing," in *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*, Amsterdam, Netherlands, July 2018, pp. 3–14.

[10] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002, pp. 1–12.

[11] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *POPL 2003: Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, LA, Jan. 2003, pp. 338–349.

[12] R. Agarwal, A. Sasturkar, and S. D. Stoller, "Type discovery for Parameterized Race-Free Java," Computer Science Department, SUNY at Stony Brook, Tech. Rep. DAR-04-16, Sep. 2004.

[13] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for Java," *ACM TOPLAS*, vol. 28, no. 2, pp. 207–255, Mar. 2006.

[14] M. D. Ernst, A. Lovato, D. Macedonio, F. Spoto, and J. Thaine, "Locking discipline inference and checking," in *ICSE 2016, Proceedings of the 38th International Conference on Software Engineering*, Austin, TX, USA, May 2016, pp. 1133–1144.

[15] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine, "Glacier: Transitive class immutability for Java," in *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*, Buenos Aires, Argentina, May 2017, pp. 496–506.

[16] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst, "Object and reference immutability using Java generics," in *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sep. 2007, pp. 75–84.

[17] M. S. Tschantz and M. D. Ernst, "Javari: Adding reference immutability to Java," in *OOPSLA 2005, Object-Oriented Programming Systems, Languages, and Applications*, San Diego, CA, USA, Oct. 2005, pp. 211–230.

[18] A. Birka and M. D. Ernst, "A practical type system and language for reference immutability," in *OOPSLA 2004, Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, Canada, Oct. 2004, pp. 35–49.

[19] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst, "Ownership and immutability in generic Java," in *OOPSLA 2010, Object-Oriented Programming Systems, Languages, and Applications*, Revo, NV, USA, Oct. 2010, pp. 598–617.

[20] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "ReIm & ReImInfer: Checking and inference of reference immutability and method purity," in *OOPSLA 2012, Object-Oriented Programming Systems, Languages, and Applications*, Tucson, AZ, USA, Oct. 2012, pp. 879–896.

[21] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele Jr., "Object-oriented units of measurement," in *OOPSLA 2004, Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, Canada, Oct. 2004, pp. 384–403.

[22] T. Xiang, J. Y. Luo, and W. Dietl, "Precise inference of expressive units of measurement types," in *OOPSLA 2020, Object-Oriented Programming Systems, Languages, and Applications*, Chicago, IL, USA, Nov. 2020.

[23] M. Kellogg, M. Schäf, S. Tasiran, and M. D. Ernst, "Continuous compliance," in *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, Melbourne, Australia, Sep. 2020, pp. 511–523.

[24] M. Kellogg, M. Ran, M. Sridharan, M. Schäf, and M. D. Ernst, "Verifying object construction," in *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*, Seoul, Korea, May 2020, pp. 1447–1458.

[25] C. Woolf, B. Cook, and T. McAndrew, "Automate compliance verification on AWS using provable security," https://www.youtube.com/watch?v=BbXK_-b3DTk, Dec. 2019, accessed 25 August 2020.

[26] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *CACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018.

[27] A. Pianykh, I. Zorin, and D. Lyubarskiy, "Retrofitting null-safety onto Java at Meta," https://engineering.fb.com/2022/11/22/developer-tools/meta-java-nullsafe/, Nov. 2022.

[28] L. Damas and R. Milner, "Principal type-schemes for functional programs," in 9^{th} *Symp. Principles of Programming Languages*. ACM, 1982, pp. 207–212.

[29] S. L. Peyton Jones, "YACC in SASL — an exercise in functional programming," *Software: Practice and Experience*, vol. 15, no. 8, pp. 807–820, 1985.

[30] J. Harrison, *Introduction to Functional Programming*. https://dp.iit.bme.hu/mfp/mfp03s/intro2fp.pdf, 1997.

[31] P. Hudak, *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[32] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml: Functional programming for the masses*. O'Reilly, 2013.

[33] L. Parreaux and C. Y. Chau, "MLstruct: principal type inference in a boolean algebra of structural types," in *OOPSLA 2022, Object-Oriented Programming Systems, Languages, and Applications*, Auckland, New Zealand, Dec. 2022, pp. 449–478.

[34] S. Dolan and A. Mycroft, "Polymorphism, subtyping, and type inference in MLsub," in *POPL 2017: Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, Jan. 2017, pp. 60–72.

[35] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," Courant Institute of Mathematical Sciences, New York University, Tech. Rep. 002, 1978.

[36] R. Mangal, M. Naik, and H. Yang, "A correspondence between two approaches to interprocedural analysis in the presence of join," in *ESOP 2014: 22nd European Symposium on Programming*, Grenoble, France, Apr. 2014, pp. 513–533.

[37] P. Cousot, "Types as abstract interpretations," in *POPL '97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997, pp. 316–331.

[38] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, Jan. 1977, pp. 238–252.

[39] Google, "Error prone," https://errorprone.info/, Jan. 2018.

[40] do-like-javac developers, "do-like-javac," https://github.com/SRI-CSL/do-like-javac, 2023, accessed 2 May 2023.

[41] "cache2k Java caching," https://github.com/cache2k/cache2k, cache2k is an in-memory high performance Java caching library.

[42] "RxNorm-explorer," https://github.com/ITHOFAR/RxNorm-explorer.

[43] "Nameless-Java-API," https://github.com/NamelessMC/Nameless-Java-API, Java library for interacting with a NamelessMC website.

[44] "ICalAvailable," https://github.com/plume-lib/icalavailable, Given one or more calendars in iCalendar format, produces a textual summary of available times.

[45] "lookup," https://github.com/plume-lib/lookup, Lookup searches a set of files, much like grep does.

[46] "multi-version-control," https://github.com/plume-lib/multi-version-control, Lets you run a version control command, such as status or pull, on a set of CVS/Git/Hg/SVN clones/checkouts.

[47] "Reflection-util: Utilities for Java reflection," https://github.com/plume-lib/reflection-util, Utility libraries related to Java reflection.

[48] "require-javadoc," https://github.com/plume-lib/require-javadoc, This program requires that a Javadoc comment be present on every Java class, constructor, method, and field.

[49] "Randoop unit test generator for Java," https://github.com/randoop/randoop, It automatically creates unit tests for your classes, in JUnit format.

[50] "dmn-check," https://github.com/red6/dmn-check, This is a tool for the validation of Decision Model Notation (DMN) files.

[51] "table-wrapper-api," https://github.com/spacious-team/table-wrapper-api, Provides a single convenient API for accessing tabular data from files in excel, xml, csv, etc. formats.

[52] "table-wrapper-csv-impl," https://github.com/spacious-team/table-wrapper-csv-impl, Provides an implementation of the Table Wrapper API for easy access to tabular data stored in csv.

[53] Sourcegraph developers, "Sourcegraph," https://sourcegraph.com/search, 2023, accessed 1 March 2023.

[54] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring Java applications to use generic libraries," in *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 2005, pp. 71–96.

[55] A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer, "Refactoring for parameterizing Java classes," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 437–446.

[56] J. Altidor and Y. Smaragdakis, "Refactoring Java generics by inferring wildcards, in practice," in *OOPSLA 2014, Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, USA, Oct. 2014, p. 271–290.

[57] R. Grigore, "Java generics are Turing complete," in *POPL 2017: Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, Jan. 2017, pp. 73–85.

[58] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman, *The Java Language Specification*, Java SE 17 ed. Boston, MA: Addison Wesley, 2021.

[59] R. Henry, K. M. Whaley, and B. Forstall, "The University of Washington Illustrating Compiler," in *PLDI '90: Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, USA, June 1990, pp. 223–246.

[60] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *SOSP 2001, Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, Oct. 2001, pp. 57–72.

[61] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference," in *USENIX Security: 13th USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004, pp. 119–134.

[62] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *PLDI 2007: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, June 2007, pp. 425–434.

[63] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *PASTE 2005: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, Lisbon, Portugal, Sep. 2005, pp. 13–19.

[64] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *PASTE 2007: ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007)*, San Diego, CA, USA, June 2007, pp. 9–14.

[65] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sep. 2007, pp. 45–54.

[66] M. Fähndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *International Conference on Formal Verification of Object-Oriented Software*, Paris, France, June 2010, pp. 10–30.

[67] K. Tsushima, O. Chitil, and J. Sharrad, "Type debugging with counter-factual type error messages using an existing type checker," in *IFL '19: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages*, 2021.

[68] J. Li, "A general pluggable type inference framework and its use for data-flow analysis," Master's thesis, U. of Waterloo, Waterloo, Ontario, Canada, 2017.

[69] Z. Pavlinovic, T. King, and T. Wies, "Finding minimum type error sources," in *OOPSLA 2014, Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, USA, Oct. 2014, p. 525–542.

[70] C. Loncaric, S. Chandra, C. Schlesinger, and M. Sridharan, "A practical framework for type inference error explanation," in *OOPSLA 2016, Object-Oriented Programming Systems, Languages, and Applications*, Amsterdam, Nov. 2016, pp. 781–799.

[71] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones, "SHErrLoc: A static holistic error locator," *ACM TOPLAS*, vol. 39, no. 4, 2017.

[72] L. Phipps-Costin, C. J. Anderson, M. Greenberg, and A. Guha, "Solver-based gradual type migration," in *OOPSLA 2021, Object-Oriented Programming Systems, Languages, and Applications*, Chicago, IL, USA, Oct. 2021.

[73] M. Vakilian, A. Phaosawasdi, M. D. Ernst, and R. E. Johnson, "Cascade: A universal programmer-assisted type qualifier inference tool," in *ICSE 2015, Proceedings of the 37th International Conference on Software Engineering*, Florence, Italy, May 2015, pp. 234–245.

[74] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[75] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *FSE 2016: Proceedings of the ACM SIGSOFT 24th Symposium on the Foundations of Software Engineering*, Seattle, WA, USA, Nov. 2016, pp. 607–618.

[76] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "TypeWriter: neural type prediction with search-based validation," in *ESEC/FSE 2020: The ACM 28th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Sacramento, CA, USA, Nov. 2020, pp. 209–220.

[77] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. Lyu, "Static inference meets deep learning: a hybrid type inference approach for Python," in *ICSE 2022, Proceedings of the 43rd International Conference on Software Engineering*, Pittsburgh, PA, USA, May 2022, pp. 2019–2030.

[78] C. Flanagan, R. Joshi, and K. R. M. Leino, "Annotation inference for modular checkers," *Information Processing Letters*, vol. 2, no. 4, pp. 97–108, Feb. 2001.

[79] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME '01: International Symposium on Formal Methods Europe 2001: Formal Methods for Increasing Software Productivity*, Berlin, Germany, Mar. 2001, pp. 500–517.

[80] *OOPSLA 2014, Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, USA, Oct. 2014.

[81] *POPL 2017: Proceedings of the 44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, France, Jan. 2017.

[82] *OOPSLA 2004, Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, BC, Canada, Oct. 2004.

[83] *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, Sep. 2007.