

Pluggable Type Inference for Free

Martin Kellogg¹, Daniel Daskiewicz², Loi Ngo Duc Nguyen¹,
Muyeed Ahmed², Michael D. Ernst²

¹New Jersey Institute of Technology
²University of Washington

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
 - “testing shows the presence of bugs, not their absence”

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
 - “testing shows the presence of bugs, not their absence”
- To scale to real programs, verifiers must be **modular**
 - Downside: humans must **write specifications**
 - Hard for legacy code

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
 - “testing shows the presence of bugs, not their absence”
- To scale to real programs, verifiers must be **modular**
 - Downside: humans must **write specifications**
 - Hard for legacy code
- **Pluggable typecheckers** extend a host type system

High-level Problem: Specifying Legacy Code

- Verification is the only way to **guarantee correctness**
 - “testing shows the presence of bugs, not their absence”
- To scale to real programs, verifiers must be **modular**
 - Downside: humans must **write specifications**
 - Hard for legacy code
- **Pluggable typecheckers** extend a host type system
- Our contribution: a **new approach for type inference** specialized to pluggable typecheckers

Background: Pluggable Types

`int` x

Background: Pluggable Types

`@Positive int x`

Background: Pluggable Types

@Negative int x

Background: Pluggable Types

`@NonConstant int x`

Background: Pluggable Types

- **widely adopted**
 - Uber, Meta, AWS, Google, Oracle, etc.

Background: Pluggable Types

- **widely adopted**
 - Uber, Meta, AWS, Google, Oracle, etc.
- **attractive to developers**
 - familiar, high precision, sound, fast checking, modular, ...

Background: Pluggable Types

- **widely adopted**
 - Uber, Meta, AWS, Google, Oracle, etc.
- **attractive to developers**
 - familiar, high precision, sound, fast checking, modular, ...
- downside: **manual annotation** of legacy codebases

Traditional Solution: Type Inference

- Traditional type inference: **constraint solving**

Traditional Solution: Type Inference

- Traditional type inference: **constraint solving**
 - **problem**: need a new constraint system **for each type system**

Traditional Solution: Type Inference

- Traditional type inference: **constraint solving**
 - **problem**: need a new constraint system **for each type system**
 - we desire a system that is **type-system-agnostic**

Traditional Solution: Type Inference

- Traditional type inference: **constraint solving**
 - **problem**: need a new constraint system **for each type system**
 - we desire a system that is **type-system-agnostic**

Are there other things in typecheckers that are type-system-agnostic?

Observation: **Local** Type Inference

- Pluggable typecheckers implement **local type inference** within method bodies

Observation: **Local** Type Inference

- Pluggable typecheckers implement **local type inference** within method bodies
 - reduces user effort: no annotations on **local variables**

Observation: **Local** Type Inference

- Pluggable typecheckers implement **local type inference** within method bodies
 - reduces user effort: no annotations on **local variables**
 - implemented as intra-procedural **dataflow analysis**

Observation: Local Type Inference

- Pluggable typecheckers implement **local type inference** within method bodies
 - reduces user effort: no annotations on **local variables**
 - implemented as intra-procedural **dataflow analysis**

```
Fortress getFort(City city) {  
    Fortress result = null;  
    if (city != LUXEMBOURG)  
        result = fortDB.get(city);  
    return result;  
}
```

Observation: Local Type Inference

- Pluggable typecheckers implement **local type inference** within method bodies
 - reduces user effort: no annotations on **local variables**
 - implemented as intra-procedural **dataflow analysis**

**dataflow detects that
result is @Nullable
here ...**


```
Fortress getFort(City city) {  
    Fortress result = null;  
    if (city != LUXEMBOURG)  
        result = fortDB.get(city);  
    return result;  
}
```

Observation: Local Type Inference

- Pluggable typecheckers implement **local type inference** within method bodies
 - reduces user effort: no annotations on **local variables**
 - implemented as intra-procedural **dataflow analysis**

... but **@NonNull** here
(assuming `get()` cannot
return null)

```
Fortress getFort(City city) {  
    Fortress result = null;  
    if (city != LUXEMBOURG)  
        result = fortDB.get(city);  
    return result;  
}
```



Observation: Local Type Inference

- Pluggable type checker implement **local type inference** within method
Q: Does dataflow **already** know whether the return type is **@NonNull** or **@Nullable**?
 - required **annotations**
 - implemented as intra-procedural **dataflow analysis**

```
Fortress getFort(City city) {  
    Fortress result = null;  
    if (city != LUXEMBOURG)  
        result = fortDB.get(city);  
    return result;  
}
```

Observation: Local Type Inference

- Pluggable type checker implement **local type inference** within method
○ return type is `@NonNull` or `@Nullable`? **YES!**
○ implemented as intra-procedural **dataflow analysis**

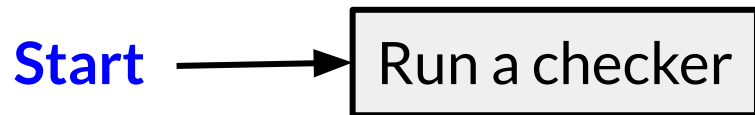
```
Fortress getFort(City city) {  
    Fortress result = null;  
    if (city != LUXEMBOURG)  
        result = fortDB.get(city);  
    return result;  
}
```


Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**

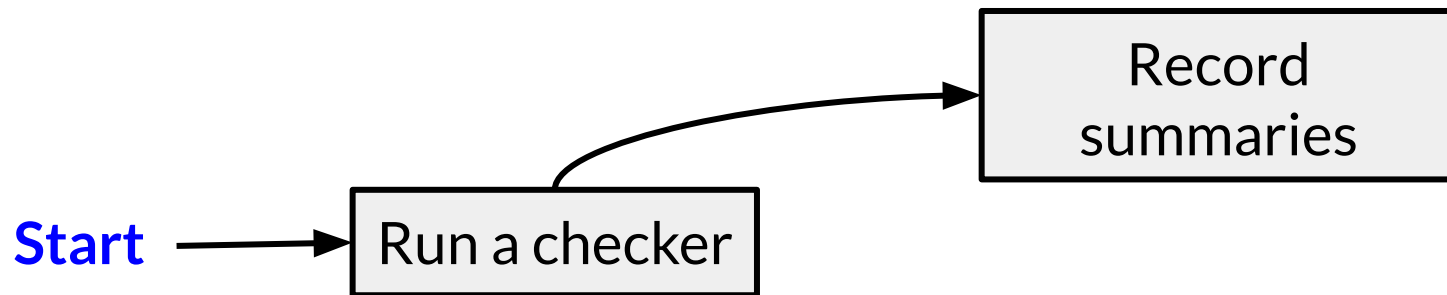
Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



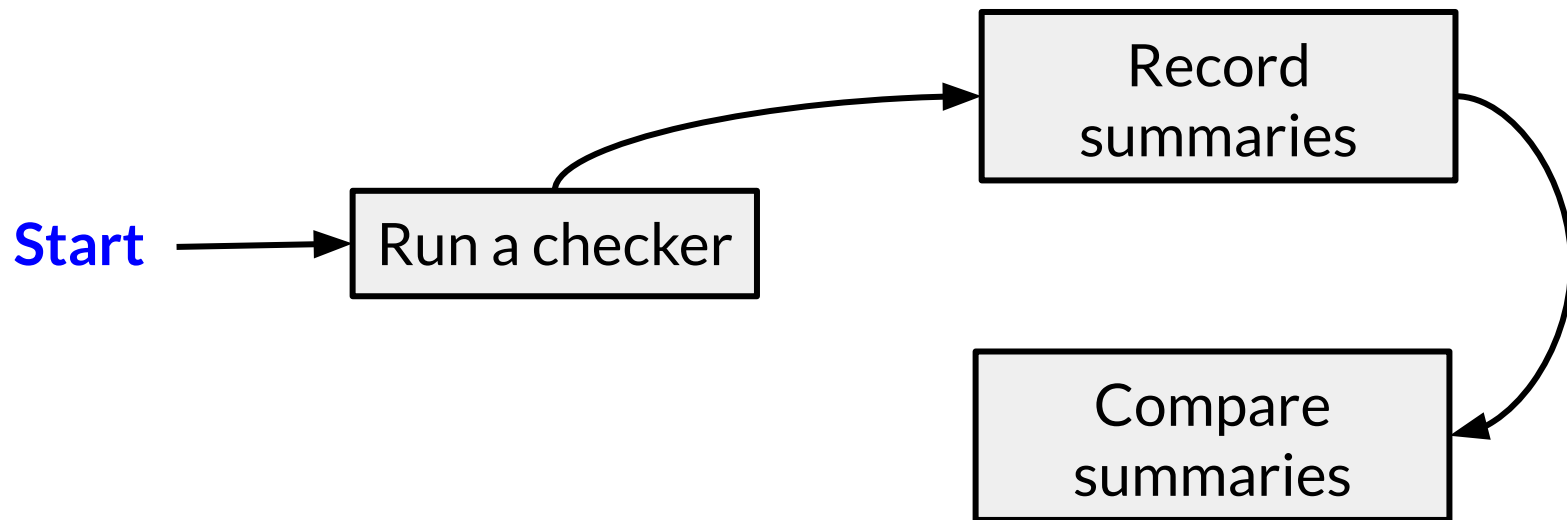
Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



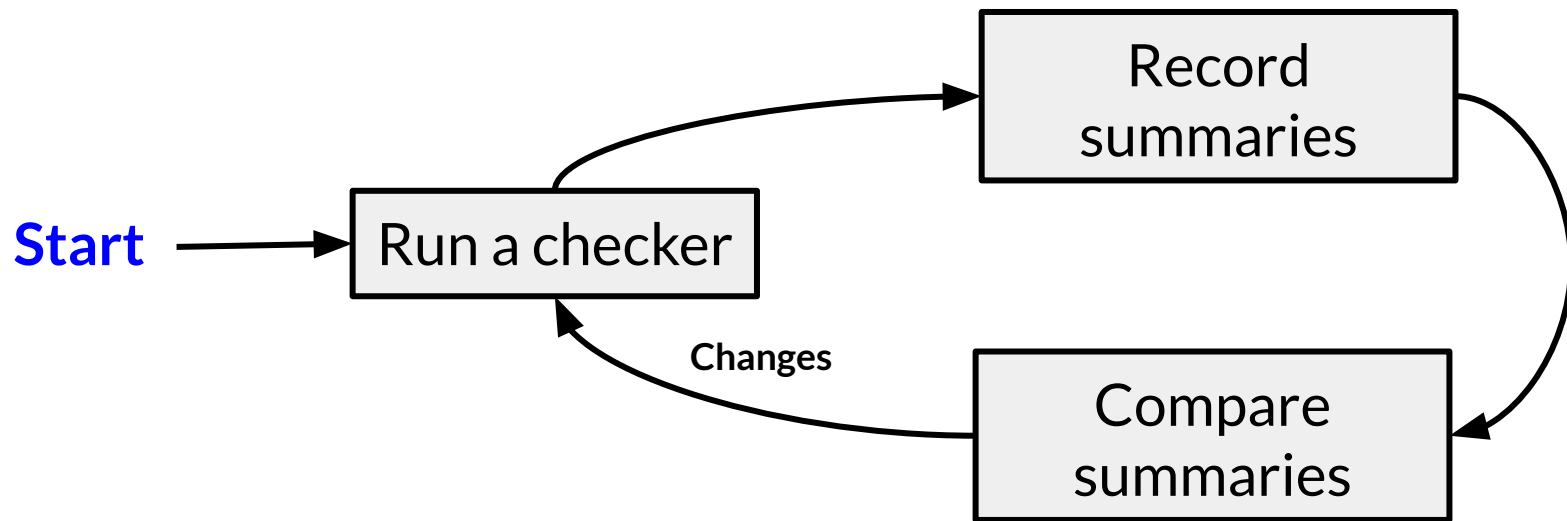
Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



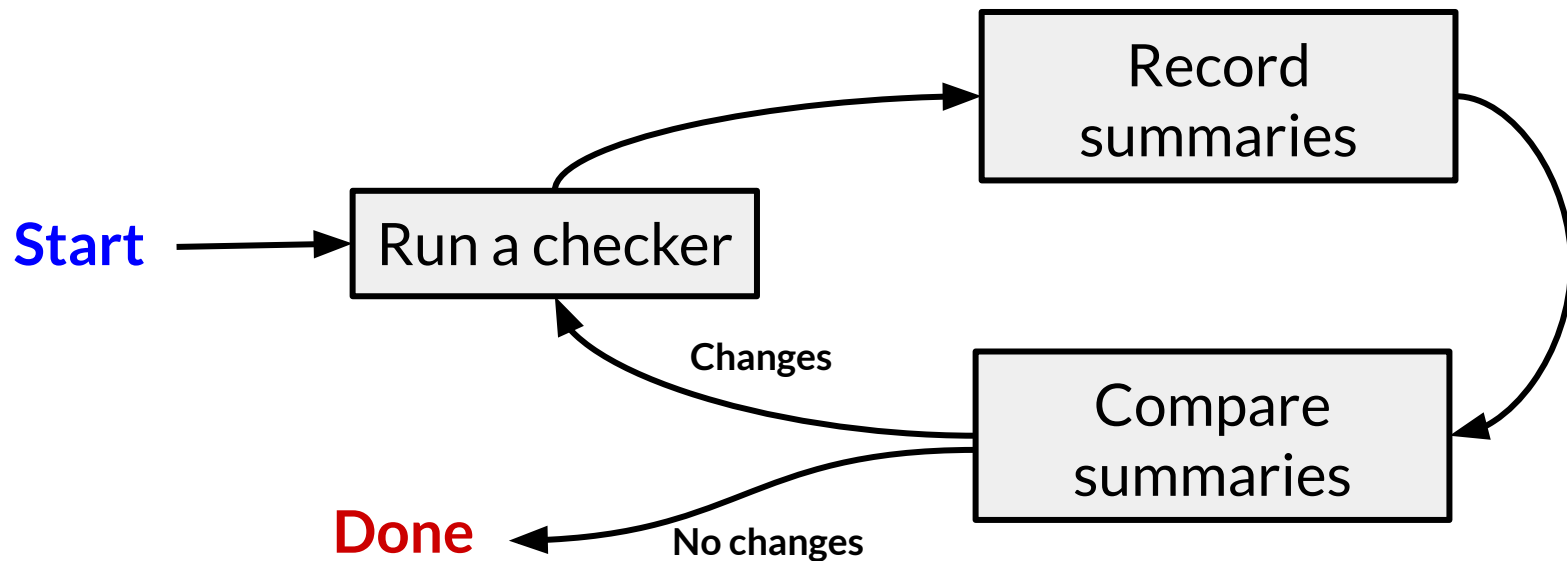
Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



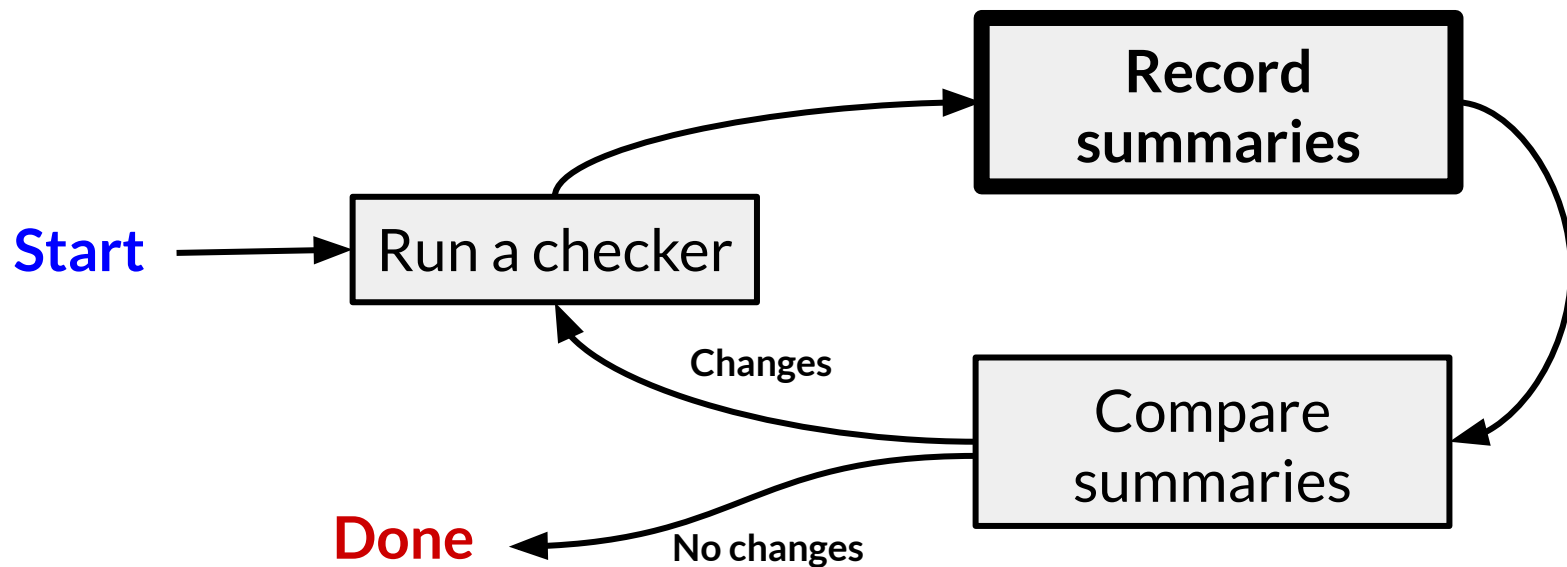
Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



More complicated than it sounds...

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

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

Read the paper for details!

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

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

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

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

Both theoretical and practical problems

- **termination?**

Both theoretical and practical problems

- **termination?**
 - **proof sketch** based on a lifted type hierarchy (see paper for details)

Both theoretical and practical problems

- **termination?**
 - **proof sketch** based on a lifted type hierarchy (see paper for details)
- many **small, important details:**

Both theoretical and practical problems

- **termination?**
 - **proof sketch** based on a lifted type hierarchy (see paper for details)
- many **small, important details**:
 - separate compilation, storing intermediate results, programmer-written types, warning suppressions, interaction with defaulting, pre- and post-conditions, non-type properties like purity, side effects, etc.

Both theoretical and practical problems

- **termination?**
 - **proof sketch** based on a lifted type hierarchy (see paper for details)
- many **small, important details**:
 - separate compilation, storing intermediate results, programmer-written types, warning suppressions, interaction with defaulting, pre- and post-condition checking, etc. properties like purity, side effects, etc.

All these details (and more) in the paper!

Implementation

- Implemented as part of the Checker Framework (our tool is called “Whole Program Inference” or “WPI”) for Java
 - **automatically** works with all checkers built on the framework
- Scripts automate it for Maven and Gradle projects
- You can try it out:

<https://checkerframework.org/manual/#whole-program-inference>

Experimental Methodology

- **Collect** verified projects from GitHub
 - annotated by a human to pass a Checker Framework checker

Experimental Methodology

- **Collect** verified projects from GitHub
 - annotated by a human to pass a Checker Framework checker
- **Remove** the annotations
 - Count the checker warnings on unannotated code

Experimental Methodology

- **Collect** verified projects from GitHub
 - annotated by a human to pass a Checker Framework checker
- **Remove** the annotations
 - Count the checker warnings on unannotated code
- Use our WPI tool to **infer new annotations**

Experimental Methodology

- **Collect** verified projects from GitHub
 - annotated by a human to pass a Checker Framework checker
- **Remove** the annotations
 - Count the checker warnings on unannotated code
- Use our WPI tool to **infer new annotations**
- Two metrics:
 - **annotation %**: percentage of human-written annotations that we recover exactly

Experimental Methodology

- **Collect** verified projects from GitHub
 - annotated by a human to pass a Checker Framework checker
- **Remove** the annotations
 - Count the checker warnings on unannotated code
- Use our WPI tool to **infer new annotations**
- Two metrics:
 - **annotation %**: percentage of human-written annotations that we recover exactly
 - **warning reduction %**: percentage of warnings on unannotated code that our annotations remove

Experimental Methodology

- **Collect** verified projects from GitHub
 - annotated by a human to pass a Checker Framework checker
- **Remove** the annotations
 - Count the
- Use our WPI
- Two metrics:
 - **annotation %**: percentage of human-written annotations that we recover exactly
 - **warning reduction %**: percentage of warnings on unannotated code that our annotations remove

These metrics are proxies for **human effort** to verify an unannotated codebase

Experimental Results

- Dataset of **12** projects (88,680 NCNB LoC total)
 - **11** distinct typecheckers (median 3.5 checkers/project)
 - **803** human-written annotations
 - with annotations removed, the checkers issue **361** warnings

Experimental Results

- Dataset of **12** projects (88,680 NCNB LoC total)
 - **11** distinct typecheckers (median 3.5 checkers/project)
 - **803** human-written annotations
 - with annotations removed, the checkers issue **361** warnings
- After applying our tool:
 - **39%** of human-written annotations were *exactly* inferred

Experimental Results

- Dataset of **12** projects (88,680 NCNB LoC total)
 - **11** distinct typecheckers (median 3.5 checkers/project)
 - **803** human-written annotations
 - with annotations removed, the checkers issue **361** warnings
- After applying our tool:
 - **39%** of human-written annotations were *exactly* inferred
 - **45%** of warnings are eliminated

Experimental Results

- Dataset of **12** projects (88,680 NCNB LoC total)
 - **11** distinct typecheckers (median 3.5 checkers/project)
 - **803** human-written annotations
 - with annotations removed, the checkers issue **361** warnings
- After applying our tool:
 - **39%** of human-written annotations were *exactly* inferred
 - **45%** of warnings are eliminated
 - summaries contain a total of **17,940** annotations

Experimental Results

- Dataset of **12** projects (88,680 NCNB LoC total)
 - **11** distinct typecheckers (median 3.5 checkers/project)
 - **803** human-written annotations
 - with annotations
- After applying our tool:
 - **39%** of human-written annotations were *exactly* inferred
 - **45%** of warnings are eliminated
 - summaries contain a total of **17,940** annotations

Significant reduction in human effort

Reasons WPI missed human-written annotations

- Methods with no callers (11% of human-written annotations)
 - e.g., “safe” library routine marks parameters `@Nullable`

Reasons WPI missed human-written annotations

- Methods with no callers (11% of human-written annotations)
 - e.g., “safe” library routine marks parameters `@Nullable`
- Generics (10%)
 - future work

Reasons WPI missed human-written annotations

- Methods with no callers (11% of human-written annotations)
 - e.g., “safe” library routine marks parameters `@Nullable`
- Generics (10%)
 - future work
- We inferred something stronger (9%)
 - e.g., `@Positive int` instead of `@NonNegative int`
 - Exact matching **underestimates** WPI’s effectiveness
 - If we count these, **annotation %** is **48%**

Reasons WPI missed human-written annotations

- Methods with no callers (11% of human-written annotations)
 - e.g., “safe” library routine marks parameters `@Nullable`
- Generics (10%)
 - future work
- We inferred something stronger (9%)
 - e.g., `@Positive int` instead of `@NonNegative int`
 - Exact matching **underestimates** WPI’s effectiveness
 - If we count these, **annotation %** is **48%**
- Long tail of other causes, none greater than 5%

Contributions

- *Iterated local type inference* algorithm
 - enables type inference for **any** pluggable typechecker

Contributions

- *Iterated local type inference* algorithm
 - enables type inference for **any** pluggable typechecker
 - “**for free**”: no code changes necessary

Contributions

- *Iterated local type inference* algorithm
 - enables type inference for **any** pluggable typechecker
 - “**for free**”: no code changes necessary
- **Formalization** and proof of termination

Contributions

- *Iterated local type inference* algorithm
 - enables type inference for **any** pluggable typechecker
 - “**for free**”: no code changes necessary
- **Formalization** and proof of termination
- **Implementation** for the Checker Framework
 - lots of **practical problems** solved

Contributions

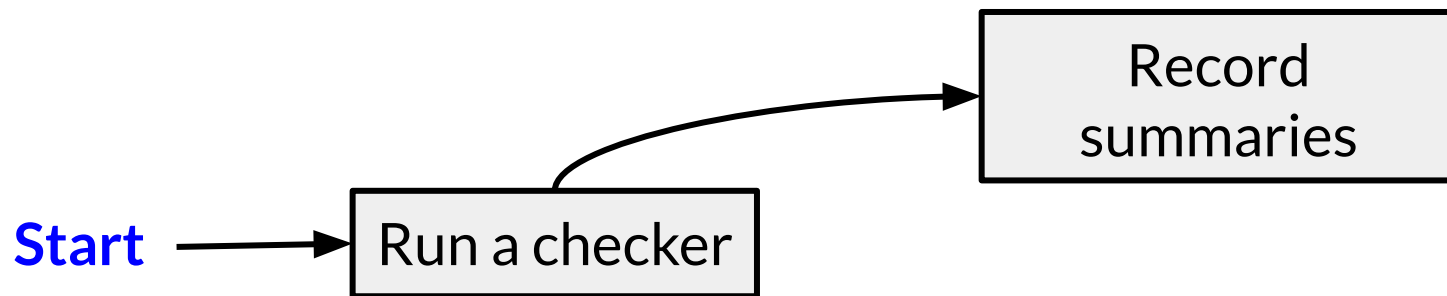
- *Iterated local type inference* algorithm
 - enables type inference for **any** pluggable typechecker
 - **“for free”**: no code changes necessary
- **Formalization** and proof of termination
- **Implementation** for the Checker Framework
 - lots of **practical problems** solved
- Experiments show that it reduces proxies for human effort:
 - annotation count **39% lower**
 - warning count **45% lower**

Contributions

- *Iterated local type inference* algorithm
 - enables type inference for **any** pluggable typechecker
 - **“for free”**: no code changes necessary
- **Formalization** and proof of termination
- **Implementation** for the Checker Framework
 - lots of **practical problems** solved
- Experiments show that it reduces proxies for human effort:
 - annotation count **39% lower**
 - warning count **45% lower**

Algorithm: Iterated Local Type Inference

- wrap existing local inference algorithm in a **fixpoint loop**



Summaries are results of local inference on externally-visible expressions!

Motivation

- pluggable typecheckers are cool (show some evidence that they're used in real life)
- but there is a problem: writing annotations
 - show an example of an annotation that's tough to write, but that WPI can find?
 - show a slide with tables from the last X Checker Framework papers, showing how many annotations were necessary just in the experiments
 - Mike doesn't think either of the above is compelling. The first makes the system seem unusable/unreadable, and the second

Key insight/approach

- briefly explain that extant frameworks already have local inference in the form of dataflow analyses within method bodies
- transition to an example. The example starts with a method, and we show how local inference works. Then, show one of the type rules from the paper (RETURN?) and show how we use the results of local dataflow to create an annotation that is global
- then, basically say “run this to fixpoint” (or show algorithm 1, which is super simple)
-
- We might not want to get really technical too quickly; that might

Theoretical properties

- soundness in the verification sense because we'll run the checker after
- termination
- completeness (i.e., all annotations we infer are verifiable) and soundness in the traditional inference sense (i.e., type all typable programs) are non-goals

Putting it into practice

- There were a surprising number of difficult, technical problems we had to overcome to get this to work in practice. Give a taste (1 or 2) and say the rest are in the paper. Here are some candidates, ordered by how well I think they're suited to presentation here:
 - generated code & termination
 - preconditions and postconditions
 - warning suppressions
 - non-type properties (purity, specifically)
 - any of the others?

Experiments

- Give a high-level summary of table 2:
 - what the experiment was, and how we collected the subject programs
 - what the resulting numbers mean
 - results
- Briefly discuss the causes for WPI missing annotations
 - generics is maybe worth discussing as future work?

Discussion

- Our results are decent (½ way there!), but not yet suitable for replacing a human annotator
 - possible combinations with other inference techniques?
- Too many annotations, making results hard for humans to interpret
- Humans often write “more conservative” annotations than WPI produces (e.g., the “defensive programming” category in table 3). This is an interesting fact on its own. What are the implications?

Contributions

- Pluggable typecheckers are awesome, but writing type annotations in legacy code is a chore
- Inference is a possible solution to this problem, which will help us convince developers to use more powerful typecheckers
- Our approach leverages the local inference that already exists inside extant typecheckers to do inference for a whole program
- We built it and it's publicly available
- It works okay! (repeat some numbers?)

Thanks to my collaborators :)