Pluggable Type Inference for Free

Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, Michael D. Ernst

New Jersey Institute of Technology
University 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

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

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

Observation: **Local** Type Inference

- Pluggable typecheckers implement local type inference within metho
  - re es
  - implemented as intra-procedural dataflow analysis

Q: Does dataflow **already** know whether the return type is **`@NonNull`** or **`@Nullable`**?

```
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 metho
  - re es
  - implemented as intra-procedural dataflow analysis

Q: Does dataflow **already** know whether the return type is `@NonNull` or `@Nullable`? YES!

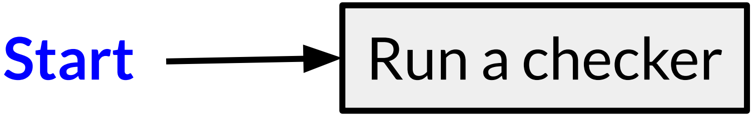
```
Fortress getFort(City city) {
  Fortress result = null;
  if (city != LUXEMBOURG)
      result = fortDB.get(city);
  return result;
}
```

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**

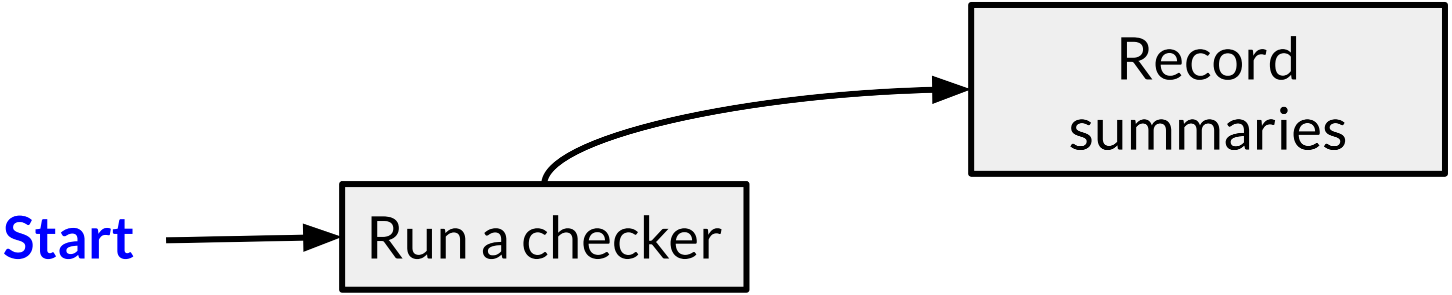


Start

Run a checker

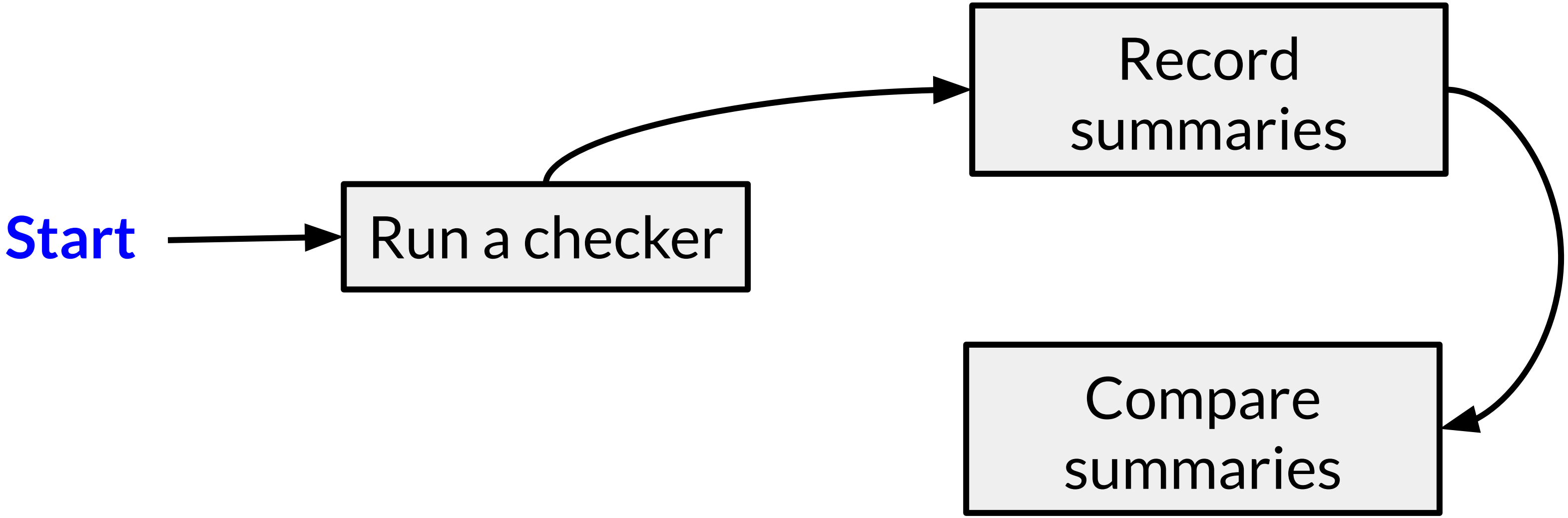
Algorithm: **Iterated** Local Type Inference

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



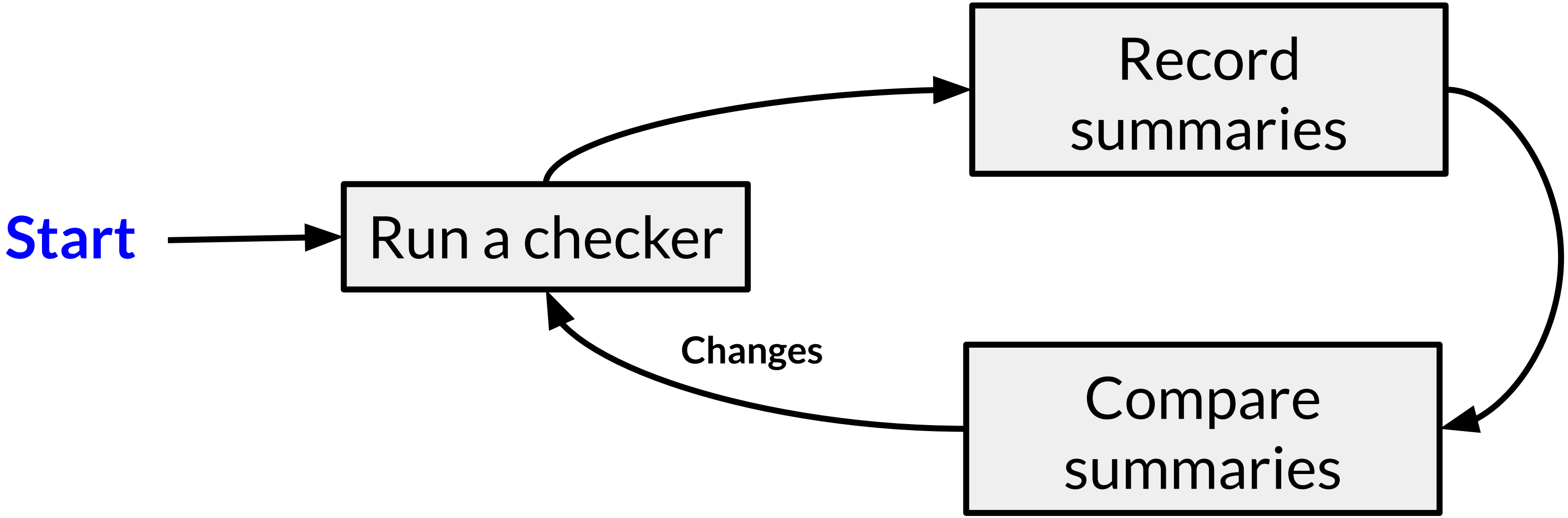
Algorithm: **Iterated** Local Type Inference

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



Algorithm: **Iterated** Local Type Inference

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



Start

Run a checker

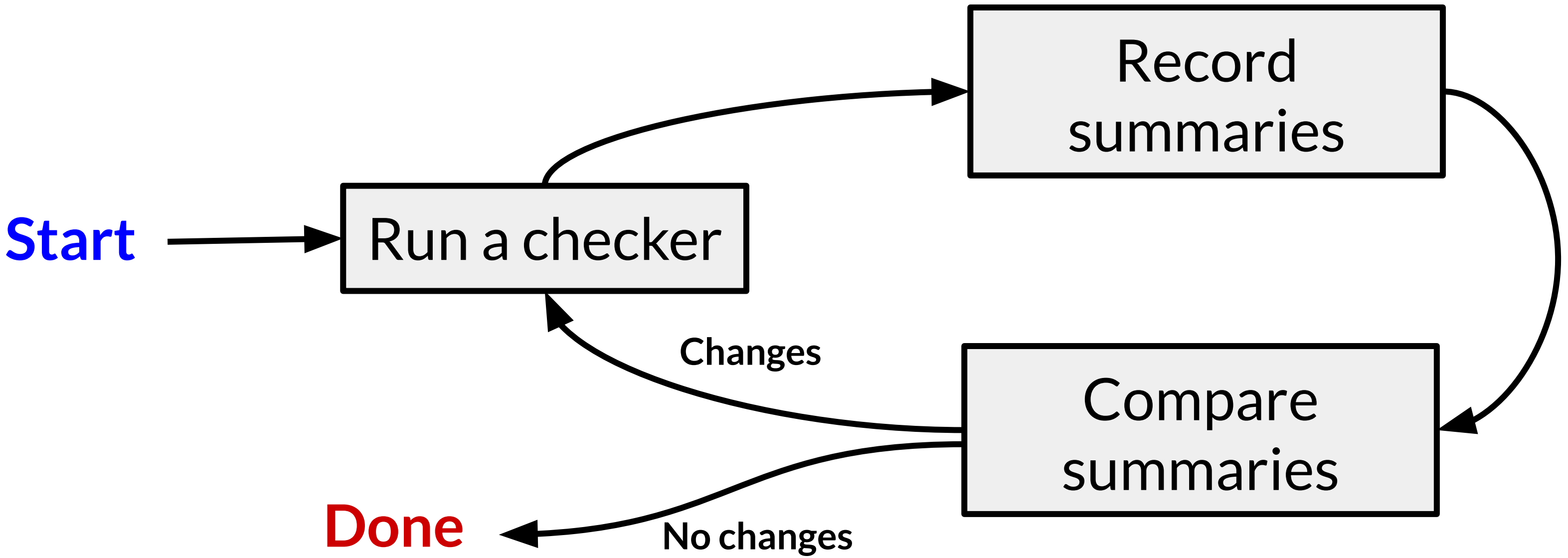
Record summaries

Compare summaries

Changes

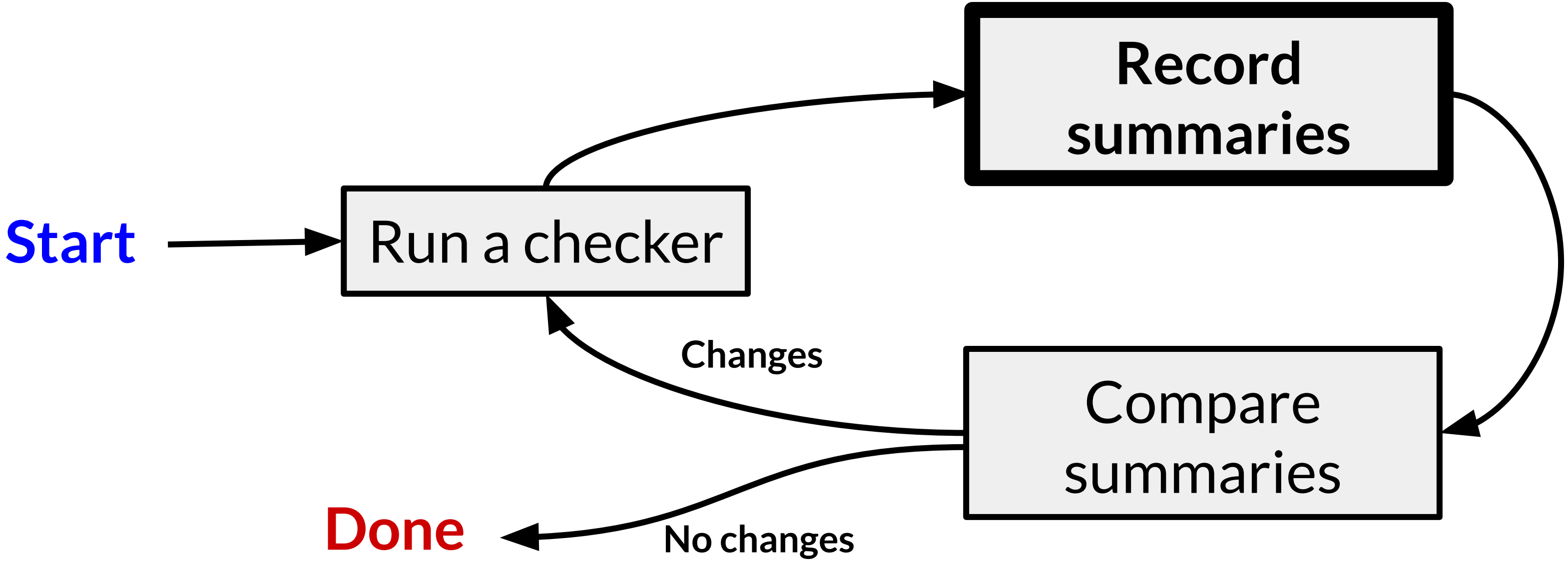
Algorithm: **Iterated** Local Type Inference

- wrap existing local inference algorithm in a fixpoint loop

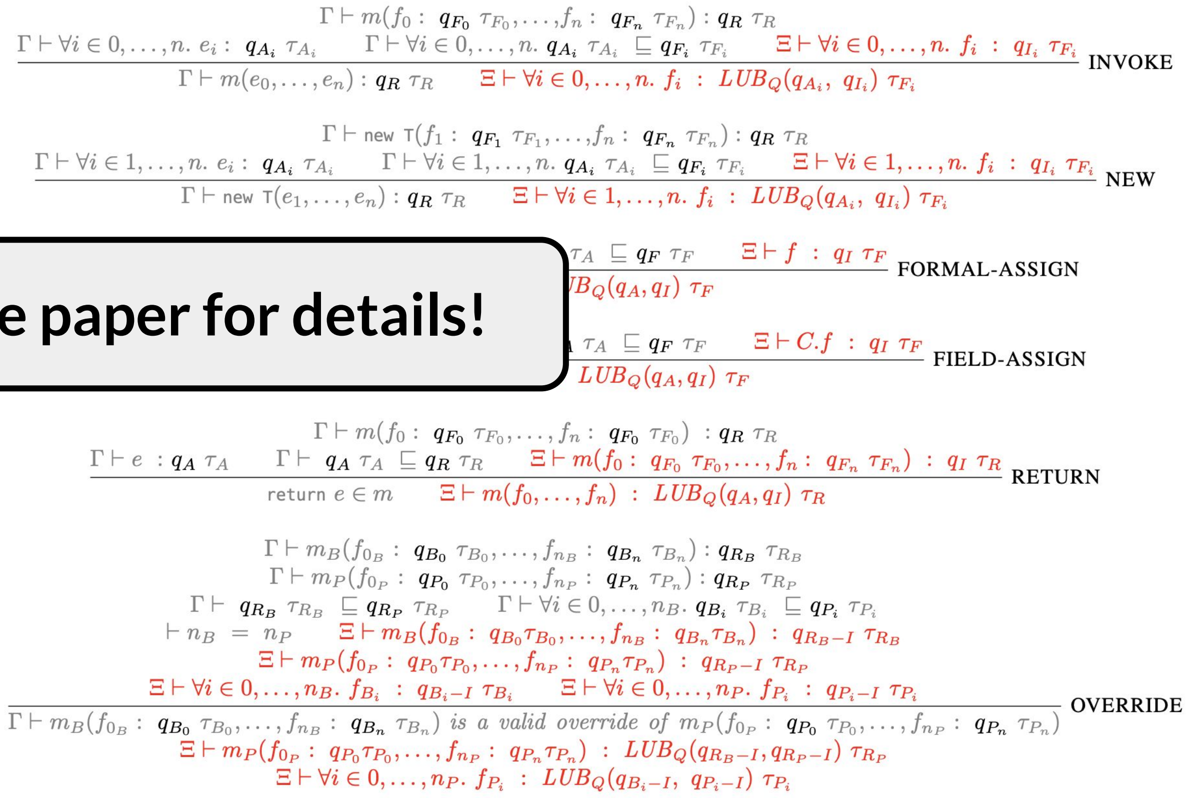


Algorithm: **Iterated** Local Type Inference

- wrap existing local inference algorithm in a fixpoint loop



More complicated than it sounds...



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

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

Read the paper for details!

$$\frac{\tau_A \sqsubseteq q_F\ \tau_F \qquad \Xi \vdash f\ :\ q_I\ \tau_F}{B_Q(q_A, q_I)\ \tau_F}\ \text{FORMAL-ASSIGN}$$

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

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

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

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-conditions, and other 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-writt
  - with annotations removed, the checkers issue [illegible] 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

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**

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