Incremental Type Checking for Free

Using Scope Graphs to Derive Incremental Type Checkers

Aron Zwaan   Hendrik van Antwerpen   Eelco Visser[†]

Dec 8, 2022

Delft University of Technology

Overview

- Writing type checkers: Hard
- Generate using Statix DSL

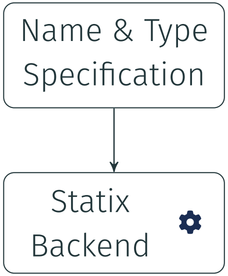
Overview

- Writing type checkers: Hard
- Generate using Statix DSL

Name & Type Specification

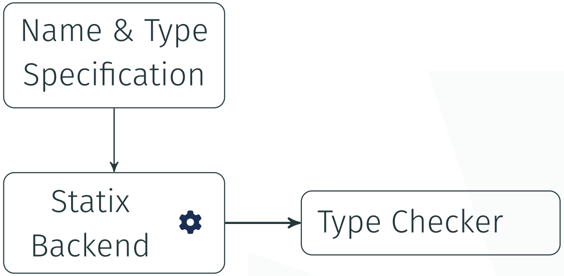
Overview

- Writing type checkers: Hard
- Generate using Statix DSL



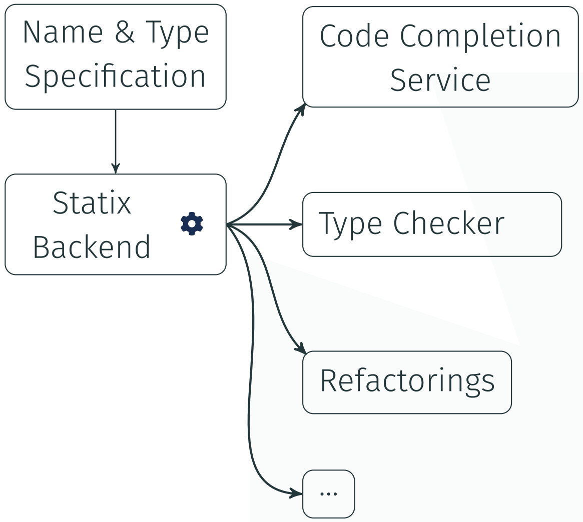
Overview

- Writing type checkers: Hard
- Generate using Statix DSL



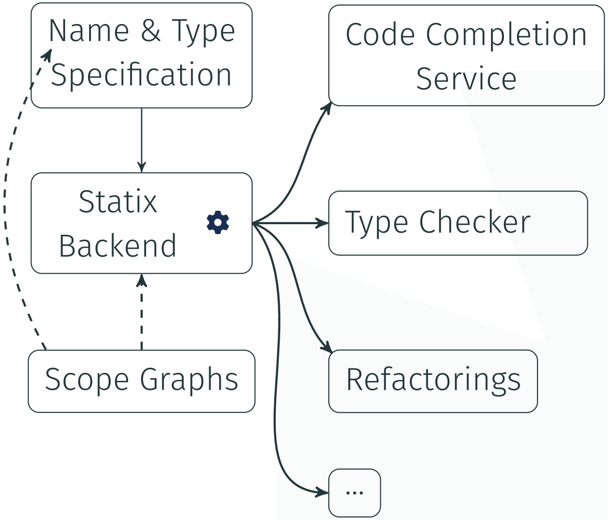
Overview

- Writing type checkers: Hard
- Generate using Statix DSL



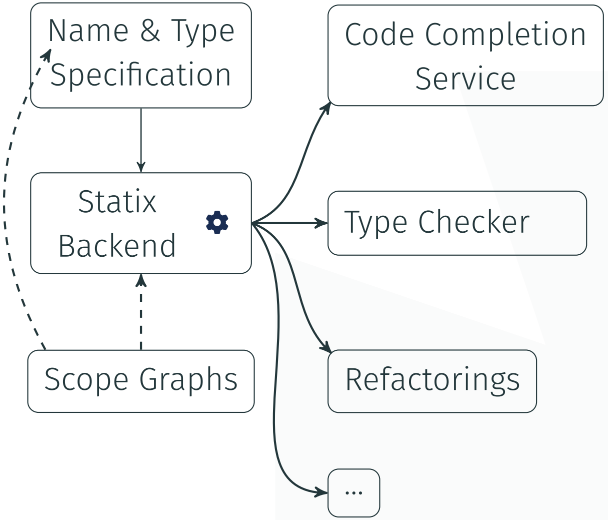
Overview

- Writing type checkers: Hard
- Generate using Statix DSL



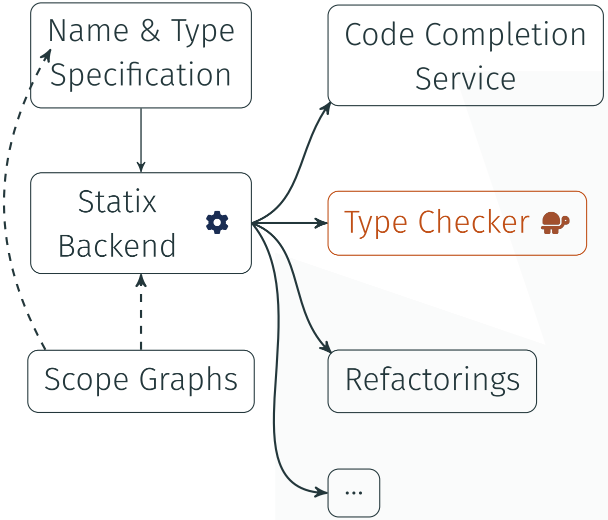
Overview

- Writing type checkers: Hard
- Generate using Statix DSL
  1. Easy
  2. Consistent
  3. Allows reasoning



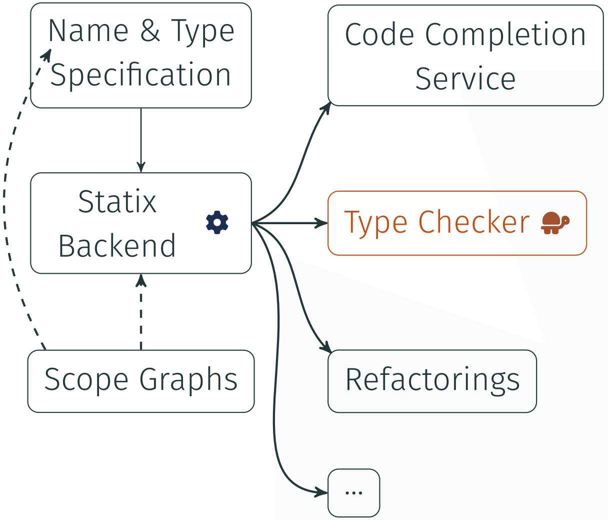
Overview

- Writing type checkers: Hard
- Generate using Statix DSL
  1. Easy
  2. Consistent
  3. Allows reasoning
- Problem: Performance



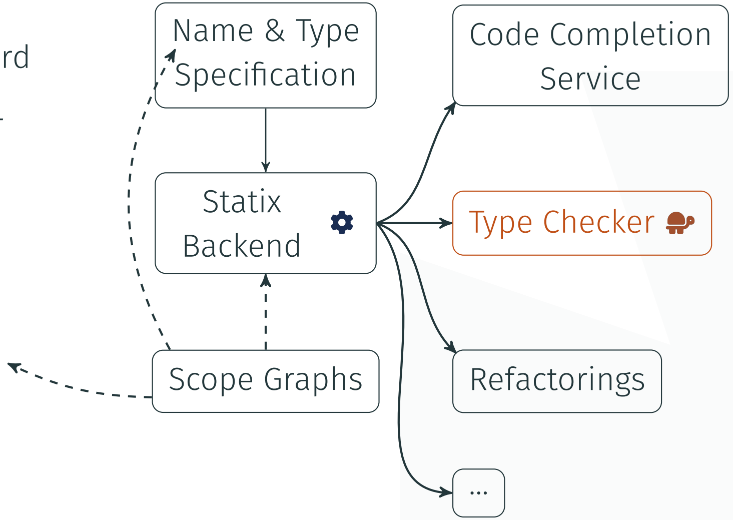
Overview

- Writing type checkers: Hard
- Generate using Statix DSL
  1. Easy
  2. Consistent
  3. Allows reasoning
- Problem: Performance
- Solution: Incrementalize



Overview

- Writing type checkers: Hard
- Generate using Statix DSL
  1. Easy
  2. Consistent
  3. Allows reasoning
- Problem: Performance
- Solution: Incrementalize



Statix Rules

$$\frac{}{\vdash n : \textbf{int}}$$

$$\frac{\vdash c : \textbf{bool} \quad \vdash e_1 : T \quad \vdash e_2 : T}{\vdash \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 : T}$$

```
typeOfExpr: Expr -> Type

typeOfExpr(Int(n)) = INT().

typeOfExpr(If(c, e1, e2)) = T :-
  typeOfExpr(c) == BOOL(),
  typeOfExpr(e1) == T,
  typeOfExpr(e2) == T.
```

Scope Graphs

```
class A {

}
```

```
class B extends A {

}
```

Scope Graphs

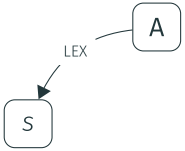
```
class A {

}
```

```
class B extends A {

}
```

Scope Graphs

```
class A {

}
```

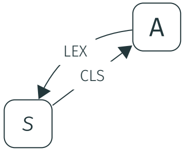
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

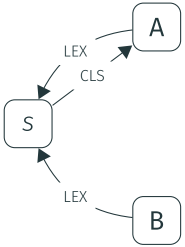
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

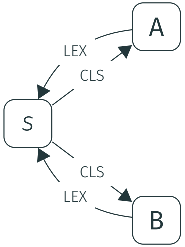
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

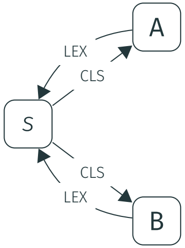
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

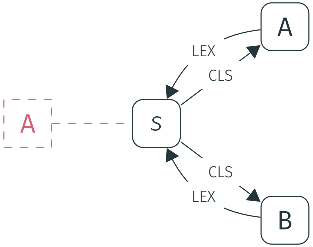
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

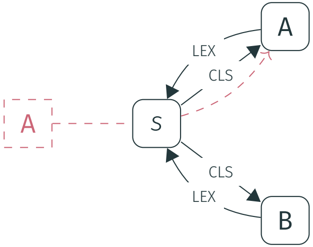
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

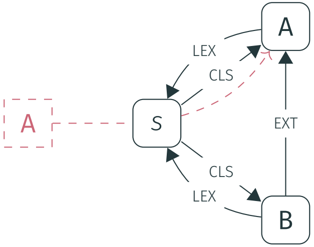
```
class B extends A {

}
```



Scope Graphs

```
class A {

}
```

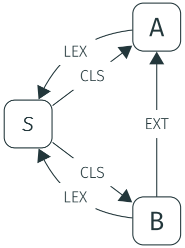
```
class B extends A {

}
```



Scope Graphs

```
class A {
  int x = 42;
}
```

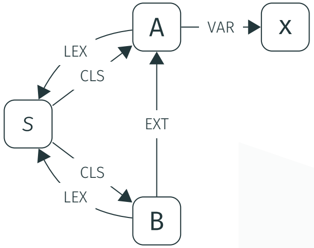
```
class B extends A {
  int y = x;
}
```


$$\begin{tikzcd}
 & & A \arrow[dll, bend right, "\text{LEX}"'] \\
S \arrow[urr, "\text{CLS}"] \arrow[drr, "\text{CLS}"'] & & \\
 & & B \arrow[ull, bend left, "\text{LEX}"] \arrow[uu, "\text{EXT}"']
\end{tikzcd}$$

Scope Graphs

```
class A {
  int x = 42;
}
```

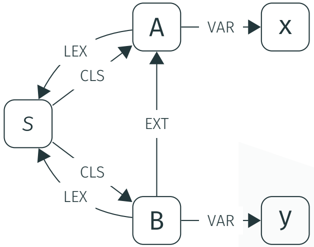
```
class B extends A {
  int y = x;
}
```


$$\begin{tikzcd}
 & A \arrow[ld, bend right, "LEX"'] \arrow[r, "VAR"] & x \\
S \arrow[ru, "CLS"] \arrow[rd, "CLS"'] & & \\
 & B \arrow[lu, bend left, "LEX"] \arrow[uu, "EXT"'] &
\end{tikzcd}$$

Scope Graphs

```
class A {
  int x = 42;
}
```

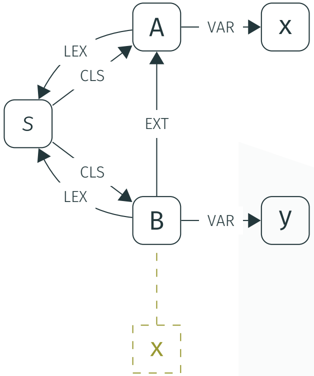
```
class B extends A {
  int y = x;
}
```



Scope Graphs

```
class A {
  int x = 42;
}
```

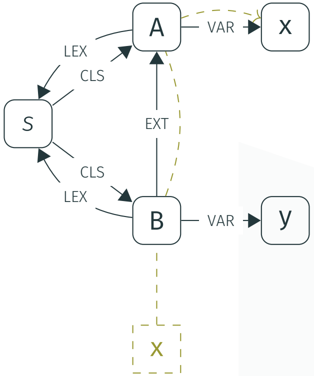
```
class B extends A {
  int y = x;
}
```



Scope Graphs

```
class A {
  int x = 42;
}
```

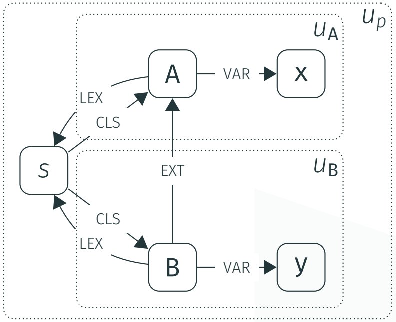
```
class B extends A {
  int y = x;
}
```



Scope Graphs

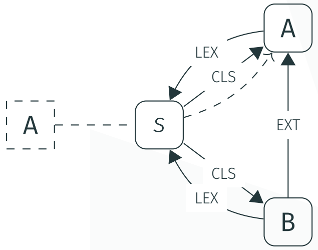
```
class A {
  int x = 42;
}
```

```
class B extends A {
  int y = x;
}
```



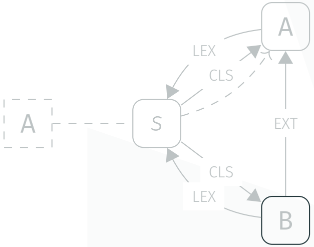
Scope Graphs in Statix

```
declOk: scope * Decl

declOk(s, Class(name, prnt, body)) :-
  {s_cls s_parent}
    new s_cls,
    s_cls -LEX-> s,
    !CLS[name, s_cls] in s,
    query CLS
      filter eq(prnt)
      in s |-> [s_parent],
    s_cls -EXT-> s_parent,
    classBodyOk(s_cls, body).
```



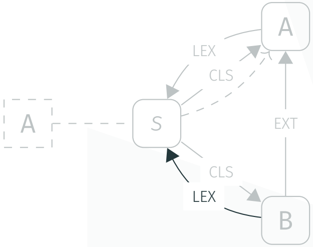
Scope Graphs in Statix

```
declOk: scope * Decl

declOk(s, Class(name, prnt, body)) :-
  {s_cls s_parent}
    new s_cls,
    s_cls -LEX-> s,
    !CLS[name, s_cls] in s,
    query CLS
      filter eq(prnt)
      in s |-> [s_parent],
    s_cls -EXT-> s_parent,
    classBodyOk(s_cls, body).
```



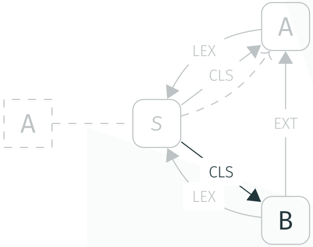
Scope Graphs in Statix

```
declOk: scope * Decl

declOk(s, Class(name, prnt, body)) :-
  {s_cls s_parent}
    new s_cls,
    s_cls -LEX-> s,
    !CLS[name, s_cls] in s,
    query CLS
      filter eq(prnt)
      in s |-> [s_parent],
    s_cls -EXT-> s_parent,
    classBodyOk(s_cls, body).
```



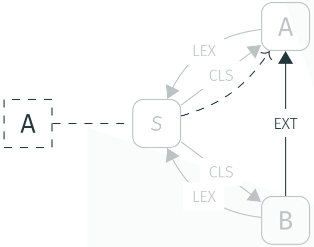
Scope Graphs in Statix

```
declOk: scope * Decl

declOk(s, Class(name, prnt, body)) :-
  {s_cls s_parent}
    new s_cls,
    s_cls -LEX-> s,
    !CLS[name, s_cls] in s,
    query CLS
      filter eq(prnt)
      in s |-> [s_parent],
    s_cls -EXT-> s_parent,
    classBodyOk(s_cls, body).
```



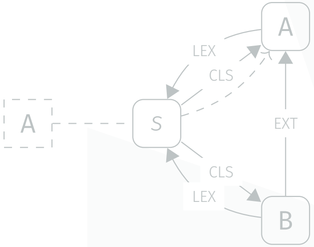
Scope Graphs in Statix

```
declOk: scope * Decl

declOk(s, Class(name, prnt, body)) :-
  {s_cls s_parent}
    new s_cls,
    s_cls -LEX-> s,
    !CLS[name, s_cls] in s,
    query CLS
      filter eq(prnt)
      in s |-> [s_parent],
    s_cls -EXT-> s_parent,
    classBodyOk(s_cls, body).
```



Scope Graphs in Statix

```
declOk: scope * Decl

declOk(s, Class(name, prnt, body)) :-
  {s_cls s_parent}
    new s_cls,
    s_cls -LEX-> s,
    !CLS[name, s_cls] in s,
    query CLS
      filter eq(prnt)
      in s |-> [s_parent],
    s_cls -EXT-> s_parent,
    classBodyOk(s_cls, body).
```



Problem & Solution Setup

- Problem: Performance
- Ideally: Generate *incremental* type checkers
  - … but no generic approach exists
- Challenge: tracking (mutual) dependencies
- Solution: using scope graph diffing

```
if AST changed then
  reanalyze
else if any query changed then
  reanalyze
else
  reuse previous result
```

Problem & Solution Setup

- Problem: Performance
- Ideally: Generate *incremental* type checkers
  - … but no generic approach exists
- Challenge: tracking (mutual) dependencies
- Solution: using scope graph diffing

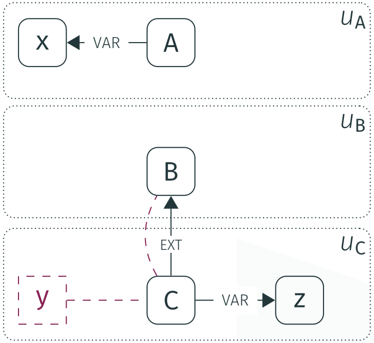
```
if AST changed then
  reanalyze
else if any query changed then
  reanalyze
else
  reuse previous result
```

Verifying Queries

```
class A {
  int x = 42;
}
```

```
class B {
}
```

```
class C extends B {
  int z = y;
}
```

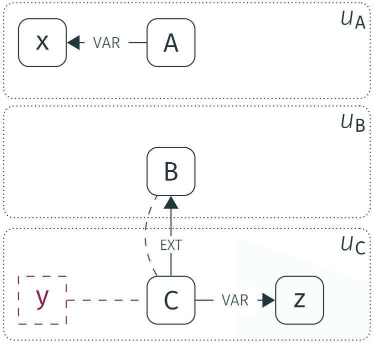


Verifying Queries

```
class A {
  int x = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```

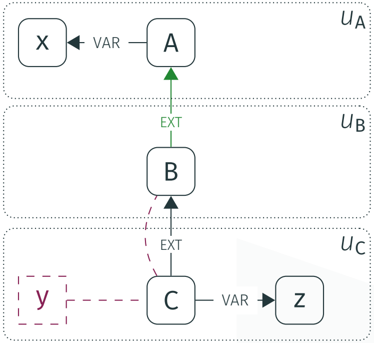


Verifying Queries

```
class A {
  int x = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```

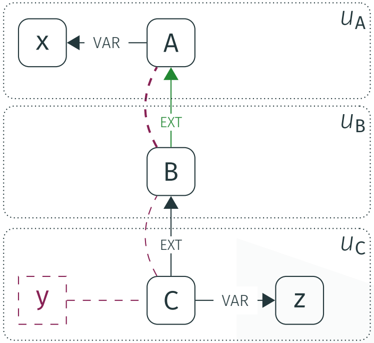


Verifying Queries

```
class A {
  int x = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```



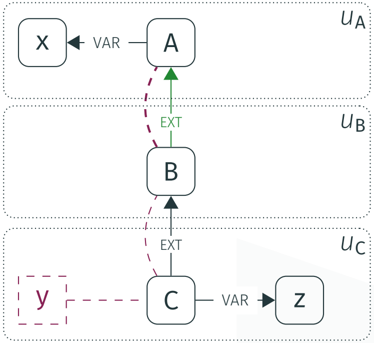
New scope A becomes reachable

Verifying Queries

```
class A {
  int x = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```



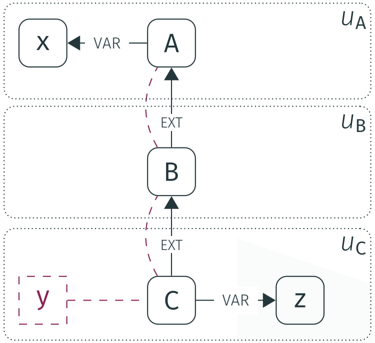
No new results in A, thus no reanalysis.

Verifying Queries

```
class A {
  int ~~x~~ y = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```

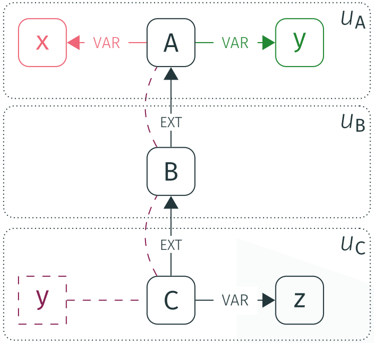


Verifying Queries

```
class A {
  int ~~x~~ y = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```

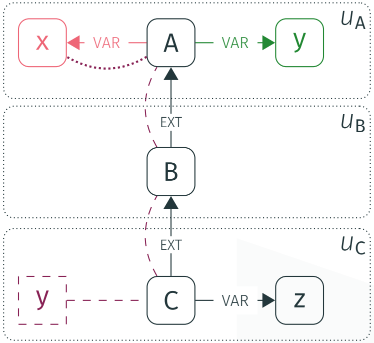


Verifying Queries

```
class A {
  int x y = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```



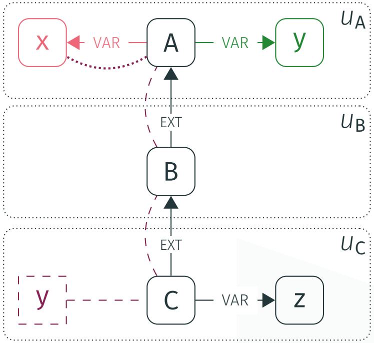
Scope x becomes unreachable

Verifying Queries

```
class A {
  int x y = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```



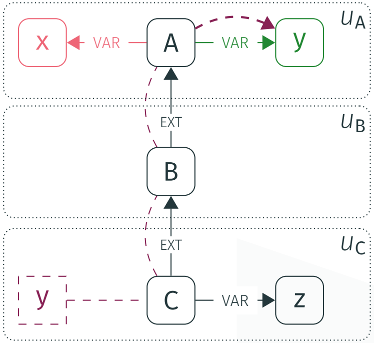
No old results in x, thus no reanalysis.

Verifying Queries

```
class A {
  int x y = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```



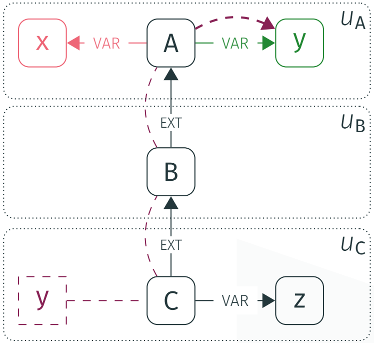
Scope y becomes reachable

Verifying Queries

```
class A {
  int ~~x~~ y = 42;
}
```

```
class B extends A {
}
```

```
class C extends B {
  int z = y;
}
```



New results in y, reanalyze unit C.

Also in the paper

- Partial reanalysis
- Mutually recursive dependencies
- Non-deterministic scope identities

140

Incremental Type-Checking for Free

Using Scope Graphs to Derive Incremental Type-Checkers

ARON ZWAAN, Delft University of Technology, Netherlands
HENDRIK VAN ANTWERPEN, Delft University of Technology, Netherlands
EELCO VISSER[†], Delft University of Technology, Netherlands

Fast analysis response times in IDEs are essential for a good editor experience. Incremental type-checking can provide that in a scalable fashion. However, existing techniques are not reusable between languages. Moreover, mutual and dynamic dependencies preclude traditional approaches to incrementality. This makes finding automatic approaches to incremental type-checking a challenging but important open question.

In this paper, we present a technique that automatically derives incremental type-checkers from type system specifications written in the Statix meta-DSL. We use name resolution queries in scope graphs (a generic model of name binding embedded in Statix) to derive dependencies between compilation units. A novel *query confirmation* algorithm finds queries for which the answer changed due to an edit in the program. Only units with such queries require reanalysis. The effectiveness of this algorithm is improved by (1) splitting the type-checking task into a *context-free* and a *context-sensitive* part, and (2) reusing a generic mechanism to resolve mutual dependencies. This automatically yields incremental type-checkers for any Statix specification.

Compared to non-incremental parallel execution, we achieve speedups up to 147x on synthetic benchmarks, and up to 21x on real-world projects, with initial overheads below 10%. This suggests that our framework can provide efficient incremental type-checking to the wide range of languages supported by Statix.

CCS Concepts: • **Software and its engineering** → **Incremental compilers**; • **Theory of computation** → **Program analysis**; Program semantics.

Additional Key Words and Phrases: type-checker, incremental type-checking, scope graphs, type systems, name binding, reference resolution, Statix

ACM Reference Format:
Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. 2022. Incremental Type-Checking for Free: Using Scope Graphs to Derive Incremental Type-Checkers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 140 (October 2022), 25 pages. https://doi.org/10.1145/3563303

1 INTRODUCTION

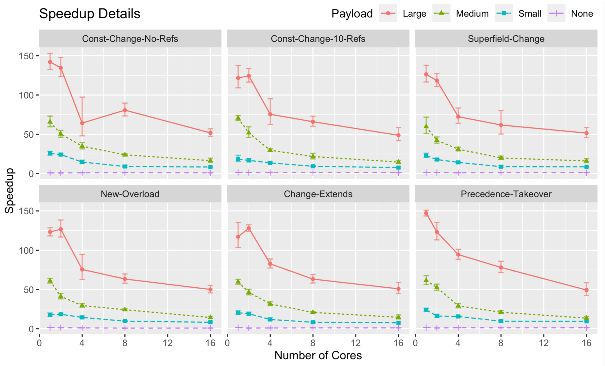
Many useful features of an IDE, such as inline error messages, code navigation and refactorings, use information from a type-checker. To provide an optimal editor experience, this type information needs to be available fast [Chaudhuri et al. 2017]. Unfortunately, as type-checking can be computationally expensive, fast editor response times are non-trivial to achieve on larger projects. To retain short feedback times for large projects, we need approaches to type-checking that have execution times proportional to the size of a change to a project, rather than to the project size

[†]Eelco worked on this paper until his untimely passing on April 5, 2022.

Authors' addresses: Aron Zwaan, Software Technology, Delft University of Technology, Delft, Netherlands, a.s.zwaan@tudelft.nl; Hendrik van Antwerpen, Software Technology, Delft University of Technology, Delft, Netherlands, h.vanantwerpen@tudelft.nl; Eelco Visser, Software Technology, Delft University of Technology, Delft, Netherlands, e.visser@tudelft.nl.

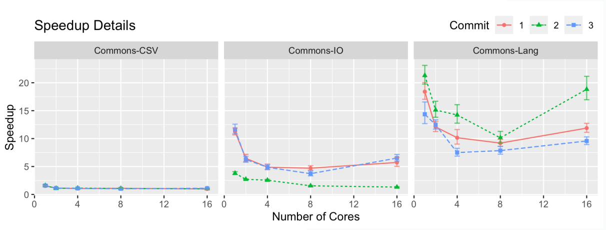
Evaluation

- Java
- Synthetic Projects
  - 1 – 100 classes
  - 20 methods
  - 5 invocations
- Synthethic Changes



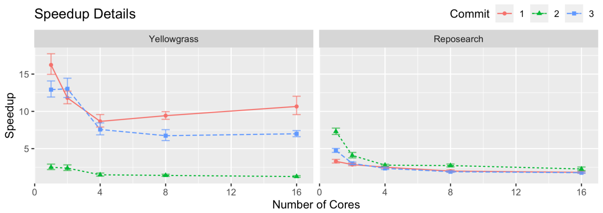
Evaluation

- Java
- Commons CSV, IO, Lang3
- Commit Sampling



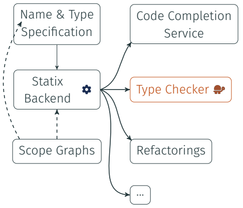
Evaluation

- WebDSL
- Internal Applications
- Commit Sampling

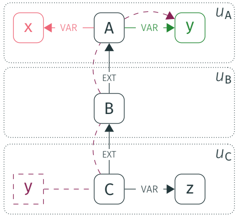


Conclusion

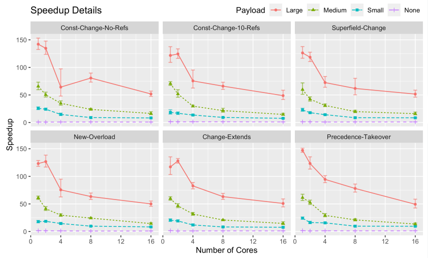
Scope graphs allow effortless type checker incrementalization.



Name & Type Specification
Code Completion Service
Statix Backend
Type Checker
Scope Graphs
Refactorings
...



x
VAR
A
VAR
y
U_A
EXT
U_B
B
EXT
U_C
y
C
VAR
z


Speedup Details
Payload
Large
Medium
Small
None
Const-Change-No-Refs
Const-Change-10-Refs
Superfield-Change
New-Overload
Change-Extends
Precedence-Takeover
Speedup
Number of Cores