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.

2475-1421/2022/10-ART140
https://doi.org/10.1145/3563303

itself. Incremental type-checking is an established approach to providing this [Busi et al. 2019; Pacak et al. 2020]. When a program is changed, incremental type-checkers reuse analysis results of unaffected code parts, and reanalyze only code parts affected by the change. This reduces analysis time, leading to a better editor experience.

Although incremental type-checking is highly beneficial for programmers, it is currently not a feature most type-checkers have. This is due to the fact that incremental type-checkers are challenging to implement soundly. We believe it is essential to design strategies to incrementalize type-checkers automatically. This requires principled and widely applicable solutions to fundamental challenges for incremental type-checking, which do not exist yet [Pacak et al. 2020]. Concrete implementations are often designed for a specific language [Chaudhuri et al. 2017; Eclipse 2021; Meertens 1983] or type-checker paradigm [Erdweg et al. 2015], or require substantial manual effort to apply [Busi et al. 2019]. Transferring these approaches between languages is non-trivial. Even the promising approach of Pacak et al. [2020] has only been applied to small toy languages. Thus, our goal is to provide a principled, *automatic* approach to incrementalize type-checkers that can be applied for *real-world* languages.

To achieve this, we present an approach that incrementalizes execution of type-checkers based on the *Statix* meta-language for type system specification [Van Antwerpen et al. 2018]. In this DSL, type systems can be expressed as declarative typing rules. The name binding structure of a program is represented by a scope graph [Néron et al. 2015] and name lookup is modeled using queries in that graph. Statix has an operational semantics [Rouvoet et al. 2020], implemented as a constraint solver, which turns a type system specification into an executable type-checker.

Van Antwerpen and Visser [2021] introduce the notion of compilation units to scope graphs. In their model, each compilation unit has its own type-checker and scope graph. Type-checkers can do lookups in other units by querying their scope graph. This approach allows executing scope-graph-based type-checkers concurrently, with compilation unit granularity.

The key insight of the current paper is that *queries represent dependencies* between units. By tracing whether query results remain unchanged when a unit is edited, called *query confirmation*, we decide which results can be reused. Units without changed query answers are not affected by the edit, and hence do not need to be reanalyzed. On the other hand, when a query of a unit returns a different result, its typing might change. Thus, we reanalyze such units. While query confirmation is not necessarily much faster than re-executing the query, it only needs to be computed for incoming queries of reanalyzed units. Therefore, our approach saves computation time on queries in reused units, as well as on other type-checker tasks. Although this strategy incrementalizes type-checking with only compilation unit granularity, it is sound, and it can be applied to any Statix specification with minimal attention from the type system designer. This is a step towards the goal of principled and widely applicable approaches to incremental type-checking.

In summary, the contributions of this paper are as follows:

- We show how *query confirmation* can discover which Statix name-resolution query answers change when a source program is edited, and how it can be used to execute Statix-based type-checkers *incrementally* (Sections 3 and 4).
- We show how splitting type-checking into a *context-free* and *context-sensitive* part improves precision of a confirmation-based incremental algorithm (Section 5).
- We show how cyclic dependency detection and resolution (based on deadlock detection in the framework of Van Antwerpen and Visser [2021]) allows incrementalizing the type-checking of mutually dependent compilation units, while saving significant computation time for query confirmation in unchanged units (Section 6).
- We show how *scope graph diffing* makes query confirmation applicable for type-checkers with non-deterministic scope identities, such as Statix (Section 7).

- We integrated an incremental Statix solver based on our algorithm in the Spoofax Language Workbench [Kats and Visser 2010].
- We evaluate our framework, showing that, for real-world projects, it requires only 10% overhead on initial runs compared to the original framework, while providing up to 147x speedup on synthetic benchmarks, and up to 21x speedup on real-world projects (Section 8).

The algorithm is built up gradually throughout the sections. A complete overview is provided in Appendix A. Appendix B gives detailed but non-essential information about the benchmarks. The source code and benchmarks are available in the accompanying artifact [Zwaan et al. 2022].

2 BACKGROUND: TYPE-CHECKING MULTI-UNIT PROJECTS WITH SCOPE GRAPHS

Our work builds upon a Statix solver that uses the concurrent type-checker framework presented by Van Antwerpen and Visser [2021]. In their framework, type-checkers of each compilation unit are isolated apart from name lookup. It gives us enough control over name lookup to implement our restart strategy, and provides natural units of computation to incrementalize as well. In this section, we introduce the main concepts of their framework: scope graphs, compilation units and their APIs.

scopes	$s \in \mathcal{S}$
labels	$l \in \mathcal{L}$
edges	$e \in \mathcal{E} := s \cdot l \cdot s$
data	$d \in \mathcal{D}$
scope graph	$\mathcal{G} := \langle S \subset \mathcal{S}, E \subset \mathcal{E}, \rho \in \mathcal{S} \rightarrow \mathcal{D} \rangle$
label regex	$r \in \mathcal{R} \subset \mathcal{L}^*$
data wf	$\mathbf{D} \subset \mathcal{D}$
paths	$p \in \mathcal{P} := s \mid p \cdot l \cdot s$
queries	$q \in \mathcal{Q} := \langle p, r, \mathbf{D} \rangle$
answers	$A \in \mathcal{A} := \mathcal{P}(\mathcal{P} \times \mathcal{D})$

Fig. 1. Scope graph definitions.

2.1 Name Binding using Scope Graphs

Scope graphs [Néron et al. 2015; Van Antwerpen et al. 2018], defined in Fig. 1, are a framework for modeling name binding patterns. In this model, scopes in a program are represented by nodes s in a graph $\mathcal{G}$. Visibility relations between scopes are modeled with labeled directed edges e. For example, when s_0 is the lexically surrounding scope of s_1, there will typically be a LEX-labeled edge from s_1 to s_0. Declarations are modeled using scopes that have an associated datum d. For instance, a declaration of a variable x in s_0 could be modeled using a VAR-labeled edge from s_0 to another scope s_x, where x is the datum associated with s_x (modeled as $\rho(s_x) = \mathsf{x}$).

Information from the scope graph can be retrieved using *queries*. A query q takes the following three arguments: a path prefix p that was previously traversed, a regular expression r that describes valid paths, and a data well-formedness predicate $\mathbf{D}$. Executing these queries will return an answer A that contains all resolution paths (p, d). A path consists of the sequence of edges in the scope graph that led from a reference to a matching declaration. They are defined as alternating sequences of scopes and labels, paired with a datum, such that (1) the path is an extension of the the p argument of the query (i.e., p is a prefix of the paths in the answer), (2) each step in the path has a corresponding edge in the scope graph, (3) the sequence of labels in the path is in the language described by a regular expression r (called the path well-formedness predicate), and (4) the datum d is associated with the rightmost scope s_r, and is well-formed (i.e., $\rho(s_r) = d$ and $d \in \mathbf{D}$).

We illustrate how scope graphs work with an example Java program in a simplified Java specification. Fig. 2 shows the program and its corresponding scope graph. Scope s_p is the global scope, and scopes s_A and s_B represent the scopes of classes A and B, respectively. To make them identifiable as such, their name is associated as data (e.g., $\rho(s_A) = \mathsf{A}$). Both classes are connected to the global scope using an edge with a CLS label. This indicates that s_A and s_B model classes in s_p. Conversely, both classes point to s_p with a LEX label, indicating that the package scope is their lexical parent.

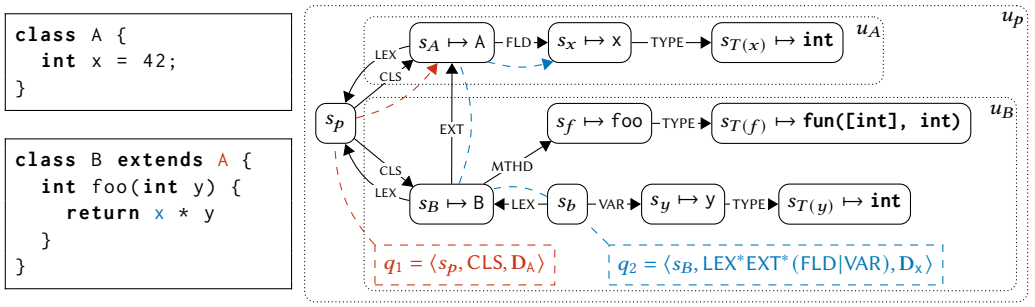

Fig. 2. Example Java program, its scope graph representation, and compilation units.

Before explaining the remainder of the graph, let us look at an example query q_1 that resolves reference A in the part of the graph we discussed so far. The first parameter of q_1 is s_p, which is the scope in which the query starts. The regular expression CLS indicates that only paths with a single CLS edge are valid for this query. (For simplicity, we ignore the possibility of imports in Java here.) Finally, the data well-formedness condition $\mathbf{D}_\text{A}$ ensures the query only returns paths of which the target scope has a datum A. As expected, the query resolves to s_A, which is shown by the red dashed line. This is the only valid path, as the path to s_B is excluded because its datum is not well-formed with respect to $\mathbf{D}_\text{A}$, and there are no other paths from s_p that match the regular expression CLS.

In this way, the type-checker of unit B can obtain a reference to s_A. This reference is used to model the **extends** A clause by creating an EXT edge from s_B to s_A. Scopes s_f and s_x model the declarations of the method foo and field x, respectively. These declarations work similar to class declarations: the name is associated with the scope, and an appropriate label (MTHD and FLD) is used to make the declaration reachable from its enclosing scope. The types of these declarations are modeled using separate scopes ($s_{T(x)}$ and $s_{T(f)}$), which are associated with the type as data. TYPE-labeled edges connect the types to the declaration scopes they correspond to.

Finally, scope s_b corresponds to the body of the method foo. The LEX edge to s_B models that s_B is the lexically enclosing scope of s_b. In s_b, method argument y is declared in a style similar to A.x. In blue, query q_2 for reference x is shown. This query starts in s_b, as the reference occurs in the body of foo. The path well-formedness condition $\text{LEX}^*\text{EXT}^*(\text{FLD}|\text{VAR})$ specifies that any number of LEX edges can be traversed, allowing lookup in enclosing scopes. After that, any number of EXT edges can be traversed to lookup names in superclasses. Eventually, the path must resolve over a FLD or a VAR edge. The data well-formedness condition $\mathbf{D}_\text{x}$ ensures that only declarations with name x are included. The resolution path for this query shows how the reference resolves to A.x.

2.2 Scope Graphs for Compilation Units

The scope graph model has been refined by Van Antwerpen and Visser [2021] in two ways. Conceptually, they make the notion of compilation units explicit in the *hierarchical compilation unit* model. In addition, they present a framework that uses this notion to type-check compilation units concurrently. As our incremental algorithm is an extension of this framework, we explain both concepts in this section and the next.

In the hierarchical compilation unit model, each project is divided into a set of units $u \in \mathcal{U}$. Each unit has its own local scope graph $\mathcal{G}_u$, instead of a single project-level one. Given a scope or an edge, its owning unit can be retrieved using the $\text{owner}(s) = u$ projection. To model the structure of real-world projects, units are organized hierarchically. That is, each unit except the root unit has a single parent, denoted as $\text{parent}(u) = u'$.

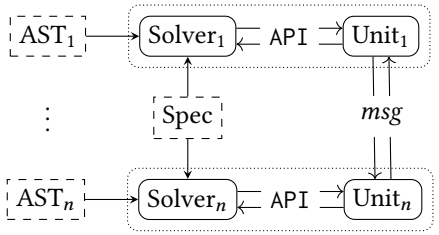

Fig. 3. Framework architecture.

interface *TypeChecker*
  async fun `Run`($\hat{S}$) : $\mathcal{T}$
interface *CompilationUnit*
  fun `FreshScope`(d) : $\mathcal{S}$
  fun `AddEdge`(s, l, s')
  fun `CloseEdge`(s, l)
  async fun `Query`(s, r, **D**) : $\mathcal{A}$

$$\begin{array}{rcl} msg & := & \mathit{AddEdge} : \mathcal{S} \times \mathcal{L} \times \mathcal{S} \\ & | & \mathit{CloseEdge} : \mathcal{S} \times \mathcal{L} \\ & | & \mathit{Query} : \mathcal{Q} \rightarrow \mathcal{A} \end{array}$$

Fig. 4. Concurrent protocol.

To model real-world projects, scope graphs of units cannot be fully disjoint. After all, a module in a particular compilation unit may be visible in the package (i.e., its parent unit) in which it resides. This is modeled using *shared scopes*, which are scopes that a unit shares with a subunit. The subunit can use such scopes to query its context, and can add edges to these shared scopes. This allows the subunit to create declarations in its parent unit, exposing itself in its immediate context.

Fig. 2 illustrates this model. In this example, there are three compilation units, indicated by dotted boxes, with their names in the top right corners. The outer box represents the project compilation unit, while the inner boxes represent its subunits, the classes. Each scope is owned by the innermost unit that contains it. For example, s_p is owned by the project unit u_p, while s_A is owned by unit u_A. The edges are owned by the innermost unit that contains its label. For example, the $s_p \cdot \mathtt{CLS} \cdot s_B$ and $s_B \cdot \mathtt{EXT} \cdot s_A$ edges are both owned by unit u_B. In this example, u_p shared s_p with u_A and u_B. This can be recognized from the fact that both CLS edges are owned by one of the subunits of u_p. In addition, q_1 illustrates how shared scopes allow lookup into other units.

2.3 Executing Type-Checkers Concurrently

The hierarchical compilation unit model is used to implement a type-checker framework that allows type-checking units concurrently. The architecture of this framework is shown in Fig. 3. Each compilation unit is split into two components: a unit and a type-checker. In our case, the type-checker is a constraint solver using a Statix specification. The unit maintains the scope graph of the compilation unit, and executes queries in it. The solver performs all other type-checking related tasks, such as solving constraints, maintaining a unifier and emitting error messages. The protocol that the unit and the type-checker use to coordinate is captured in the interfaces shown in the top half of Fig. 4. When type-checking starts, the unit invokes the `Run` method of the type-checker. The shared scopes $\hat{S}$ are passed as argument, and a typing $T \in \mathcal{T}$ is returned asynchronously. Conversely, the type-checker can manipulate the scope graph using the methods in the *CompilationUnit* interface. The `FreshScope` method creates a new scope s with associated datum d (i.e., $s \in S_u$, $\mathsf{owner}(s) = u$ and $\rho_u(s) = d$). Similarly, `AddEdge` allows the type-checker to add edges to the local scope graph. Last, the `Query` method computes a query answer[1].

Additionally, units communicate with each other by message passing, following the actor paradigm [Agha 1990]. The messages that units exchange are shown in the bottom half of Fig. 4. The *Query* message is used to resolve queries in other compilation units. For example, when the

[1]In fact, `Query` has a few additional arguments related to shadowing and local inference. Because we do not take shadowing into account in this paper, for reasons explained in Section 4.2, we omit all shadowing-related arguments here. The local inference arguments are omitted because only non-local queries are relevant for our algorithm.

type-checker for class B in Fig. 2 issues q_1, unit u_B discovers that it cannot resolve it locally, because it does not own s_p. Therefore, it forwards the query to u_p, as u_p is the owner of s_p. However, to resolve the query correctly, u_p also needs to know that u_A added a class declaration to s_p. For that reason, compilation units also forward the addition of edges in shared scopes to their parent unit using the *AddEdge* message. For example, when the type-checker of class A adds edge $s_p \cdot \texttt{CLS} \cdot s_A$ by invoking `AddEdge(`s_p`, CLS, `s_A`)`, u_A will send an $\mathit{AddEdge}(s_p, \texttt{CLS}, s_A)$ message to u_p. This allows correct query resolution in shared scopes.

Finally, the framework supports *stable query resolution* in *incomplete scope graphs*. That means that it can answer queries during type-checking, while guaranteeing that resolving the query in the complete graph would return the same result. A sound solution to this problem for Statix was presented by Rouvoet et al. [2020], and generalized by Van Antwerpen and Visser [2021]. Slightly simplified[2], it works as follows. Each unit maintains a multiset of scope-label pairs O. When $(s, l) \in O$, then a type-checker might still add edges with label l to s. Initially, O contains (s, l) $n + 1$ times for each scope and label, given that s is shared with n subunits. Once the type-checker of unit u added all edges for a label l to s, it will invoke `CloseEdge(`s`, `l`)`. If u owns s, it will remove (s, l) from O once. Otherwise, it will send a $\mathit{CloseEdge}(s, l)$ message to its parent, which will then remove (s, l) from O once. Eventually, when all edges are closed, O is empty. When query resolution needs to traverse edges with label l in scope s, it waits until $(s, l) \notin O$.

The algorithm that resolves a query is shown in Fig. 5. If the target s_t of the current path prefix p is not owned by the current unit, the query is forwarded to its owner using a $\mathit{Query}(q)$ message. Otherwise, the current path is included in the query answer A if it would be accepted by the original regular expression and its datum is well-formed. In addition, all outgoing edges of s_t with valid labels are traversed. A label is valid if the language of its Brzozowski derivative [Brzozowski 1964] $\partial_l r$ is non-empty. For all valid labels, the target scopes of the outgoing edges are retrieved using the getEdges function. This function waits for l to be closed in s_t (i.e., $(s_t, l) \notin O$), and then returns all target scopes of the outgoing l-labeled edges of s_t. For each target, a *residual query* is executed. This query uses the old path prefix p extended with the label and target scope as new path argument. Moreover, $\partial_l r$ is used as the new path-wellformedness condition, because a step with label l was added to the path. This ensures that the labels of fully resolved paths match the initial regex. Van Antwerpen and Visser [2021] show that this resolution algorithm returns stable query answers.

```
async fun Resolve(q = ⟨p, r, D⟩) : A
  A := ∅, s_t := target(p), u := owner(s_t)
  if u ≠ self then
    return send Query(q) to u
  if ε ∈ lang(r) ∧ ρ(s_t) ∈ D then
    A' += (p, ρ(s_t))
  foreach l ∈ L such that ∂_l r ≠ ∅ do
    S' := await getEdges(s_t, l, G)
    A' := { Resolve(p · l · s, ∂_l r, D) | s ∈ S' }
    A += awaitAll A'
  return A
```

Fig. 5. Query resolution algorithm.

3 ALGORITHM OUTLINE

In this section, we introduce some notational conventions, and give an high-level overview of our incremental algorithm, presented using the actor framework [Agha 1990]. This overview will be refined in Sections 4 to 7.

Notation. We use the following notational conventions:

- The **proc** keyword is used to indicate procedures that modify the *actor state*, whereas the **fun** keyword is used for pure functions.

[2]For the full version of the protocol, which additionally makes scope sharing explicit, gives more precise control over which labels O is initialized with and allows multi-level sharing, we refer to Van Antwerpen and Visser [2021].

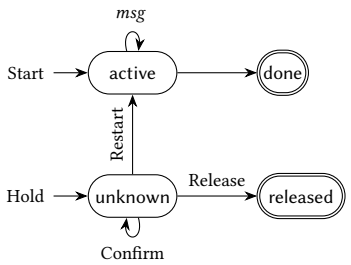

Fig. 6. Unit state diagram.

```
actor Unit(TC : TypeChecker) : CompilationUnit
    proc Start(Ŝ)
        T_n := Run_TC(Ŝ)
    proc Hold(Ŝ)
        C := { send Confirm(q) to u | (u, q) ∈ Q_{n−1} }
        if ⋀ awaitAll C then Release(Ŝ)
        else Restart(Ŝ)
    proc Release(Ŝ)
        G_n := G_{n−1}, Q_n := Q_{n−1}, T_n := T_{n−1}
        foreach (s, l) ∈ Ŝ × ℒ do CloseEdge(s, l)
    proc Restart(Ŝ)
        T_n := Run_TC(Ŝ)
```

Fig. 7. Algorithm outline.

- The **self** and **parent** keywords indicate the current actor and its parent, respectively.
- Actors can send messages to other actors using the **send** *msg* **to** *u* syntax. Incoming message handlers are indicated by the **on receive** *msg* construct. In a message receiving context, the sender can be retrieved using the **sender** keyword. Return statements in message handlers deliver the returned value as an answer to the original sender.
- Units can wait for a message answer with the **await** keyword. An **await** pauses the function until the message is answered. A list of expected replies can be awaited using **awaitAll**. Waits are non-blocking: the actor can handle other messages and replies while waiting. The **async** keyword indicates that a function might internally wait for a message reply. Finally, **await**s can be used to wait until a unit is in a particular state as well.
- Where relevant, functions are annotated with their return type. A type can be a set, a tuple of types or a powerset of another type. In addition, we use $T^?$ for *optional* types, where $\perp$ represents the *none* value. We distinguish between $\perp$ and 'real' return values in **if**-statements.
- Finally, an n subscript denotes that the subscripted value corresponds to the *current* type-checker iteration, whereas $n-1$ indicates the value is an output of the previous analysis.
- Such subscripted values are generally part of the actor state, which is accessible everywhere. Current actor state values can be modified in procedures.

Algorithm Outline. Our idea relies on the observation that a type-checker result is determined completely by the AST of the compilation unit, and the results of external name lookups. Thus, a type-checker result can be reused if its AST and external name lookups do not change. The input of the algorithm contains which units are edited, while it provides a way to decide which query answers changed. This is used to reanalyze the appropriate units.

The algorithm implements the state diagram in Fig. 6. The labels on the edges indicate the procedures that implement the transitions, which are explained in the following sections. Units that are edited start in the active state, which means that they are currently reanalyzed. When the type-checker is finished, the unit transitions to the done state. This means that there is a new type-checker result for this unit. A unit that is not edited starts in the unknown state. In this state it is not yet decided whether the unit must be reanalyzed, as one of its dependencies might be changed in a way that invalidates the current result. If an invalidating change is detected, the unit transitions to the active state, in which it is reanalyzed. Otherwise, the unit transitions to the released state, which means the previous result can be reused.

An outline of the algorithm is shown in Fig. 7. The `Start` procedure, called for edited units, simply executes the type-checker with the shared scopes $\hat{S}$. Unchanged units execute the `Hold` procedure, which checks for external changes that invalidate the previous result as follows. For each query q executed in the previous iteration, a *Confirm* : $Q_{n-1} \rightarrow$ Bool message is sent to the original receiver of the query (u). If $\bot$ is returned in reply, the answer to the query changed, and the unit must be reanalyzed. In such cases, the type-checker is re-executed. Otherwise, if all replies contain $\top$, none of the inputs to its type-checker changed, and thus the unit is released. Releasing means that the previous scope graph, set of recorded queries, and type-checker result are reused. In addition, all labels are closed in the shared scopes, enabling query resolution to proceed.

4 DETECTING CHANGED QUERIES USING ENVIRONMENT DIFFING

The algorithm outline in the previous section left two questions unanswered. First, we assumed a variable Q_{n-1} contained all queries of the previous type-checker iteration, but did not show how those were obtained. Second, we did not explain how the answer to an incoming *Confirm* message is computed. In this section, we define non-local queries, and show how those are recorded. In addition, we present our query confirmation algorithm, and show how it is used to answer *Confirm* messages. The section concludes with an example illustrating both concepts.

4.1 Which Queries Require Confirmation?

A key ingredient for our algorithm is confirming the queries that led to a previous result. However, we only need to confirm queries that other units contribute to, as those queries represent dependencies on other units. External influence occurs when query resolution traverses a scope that another unit can add edges to. We call such scopes *non-local* with respect to unit u, defined as:

$$\text{non-local}_u(s) \triangleq \text{owner}(s) \neq u \vee (\exists u'.\, \text{parent}(u') = u \wedge s \in \hat{S}_{u'})$$

This captures the fact that a scope s can be extended by a unit other than u when s is not owned by u or u has shared s with a subunit u'. A *query* is non-local when its resolution invokes `Resolve` with a path of which the target is non-local. We need to confirm precisely these queries.

To make these queries available for future type-checker invocations, we adapt the query resolution algorithm to maintain a set Q of non-local residual queries. These queries are collected by adding the following line to the query resolution algorithm in Fig. 5:

$$\textbf{if } \text{non-local}_{\textbf{self}}(s_t) \textbf{ then } Q \mathrel{+}= (u, q)$$

In addition, the resolution algorithm accumulates the recorded queries of sub-queries, and returns the resulting set Q alongside the query answer. When a query is fully resolved (i.e., an invocation of `Resolve` by `Query` returns), Q is inserted in the set of recorded queries Q_n.

Note that this approach separately collects transitive queries. For example, suppose unit u_1 sends $Query(q_1)$ to u_2 and during resolution of q_1, u_2 sends $Query(q_2)$ to u_3. Then both q_1 and q_2 are collected in Q_n, as they are non-local in u_1 and u_2, respectively. This is required to prevent unsound interaction with other parts of the algorithm, which will be explained in more detail in Section 6.2.

4.2 Using Environment Diffs to Release Units

To confirm a query, we compute whether there are any paths that are added or removed when the query would have been executed in the new scope graph. Any added path can be split into two parts $p_1 \cdot l \cdot p_2$ where $\text{target}(p_1) \cdot l \cdot \text{source}(p_2)$ is an edge that is added in $\mathcal{G}_n$. Similarly, when a path is removed from an environment, at least one of its edges is removed from the scope graph. To identify such paths, we emulate the query resolution algorithm, but collect added and removed edges instead of resolution paths.

The `EnvDiff` function, defined in Fig. 8, implements this step. It takes a scope and a path well-formedness condition as arguments, and returns an *environment diff*. An environment diff consists of two sets: δE^+ and δE^-, which contain the added and removed subenvironments, respectively. Both sets contain (s, r) pairs, which indicate that an edge with target s is added or removed. The argument r is the path-wellformedness condition that a residual query from s would have used. An environment diff is computed as follows. For each label that can be followed by the regular expression, the *scope diff* is calculated by the `Diff` function. This function computes the targets of the l-labeled edges for the current and previous scope graph, and divides them in three subsets. The first subset contains the *matched* edge targets, which are the targets of edges that occur in both graphs. The other sets contain the added and removed edge targets, respectively. For each added and removed scope, an appropriate marker is inserted in the correct set. For all matched edges, `EnvDiff` recursively computes the environment diffs. In this way, the `EnvDiff` returns a collection of scopes that are targets of added or removed edges along possible resolution paths.

Now, for each marker pair in δE^+ we know that a prefix $p_1 \cdot l \cdot s_t$ exists, because it was traversed by `EnvDiff`. However, we do not yet know if s_t actually leads to a valid declaration for the query (i.e., if a p_2 segment exists). To validate that, `Confirm` executes residual queries from s_t, using the *Query* message. When the residual query returns a non-empty result, a valid residual path from s_t exists, and hence a new valid path in the answer to the original query. Therefore, we return $\bot$ to indicate that the query could not be confirmed. Otherwise, we add the non-local queries of the

```
async fun Diff(s, l) : 𝒫(𝒮) × 𝒫(𝒮) × 𝒫(𝒮)
  S_n := await getEdges(s, l, 𝒢_n)
  S_{n-1} := getEdges(s, l, 𝒢_{n-1})
  return (S_n ∩ S_{n-1}, S_n \ S_{n-1}, S_{n-1} \ S_n)
async fun EnvDiff(s, r) : δE⁺ × δE⁻
  if owner(s) ≠ self then return (∅, ∅)
  δE⁺ := ∅, δE⁻ := ∅
  foreach l ∈ ℒ such that ∂_l r ≠ ∅ do
    (S_t^~, S_t^+, S_t^-) := await Diff(s, l)
    δE⁺ += { (s_t, ∂_l r) | s_t ∈ S_t^+ }
    δE⁻ += { (s_t, ∂_l r) | s_t ∈ S_t^- }
    foreach s_t ∈ S_t^~ do
      (δE⁺', δE⁻') := await EnvDiff(s_t, ∂_l r)
      δE⁺ += δE⁺'
      δE⁻ += δE⁻'
  return (δE⁺, δE⁻)
async fun Confirm(s, r, D) : (𝒫(Q_{n-1}) × 𝒫(Q_n))?
  Q⁻ := ∅, Q⁺ := ∅
  (δE⁺, δE⁻) := await EnvDiff(s, r)
  foreach (s_t, r') ∈ δE⁺ do
    (Q', A) := await send Query(s_t r', D) to owner(s_t)
    if A ≠ ∅ then return ⊥
    else Q⁺ += Q'
  foreach (s_t, r') ∈ δE⁻ do
    (Q', A) := await send PQuery(s_t, r', D) to owner(s_t)
    if A ≠ ∅ then return ⊥
    else Q⁻ += Q'
  return (Q⁻, Q⁺)
proc Hold(Ŝ)
  Q⁻ := ∅, Q⁺ := ∅
  foreach (u, q) ∈ Q_{n-1} do
    if (Q⁻', Q⁺') := await send Confirm(q) to u then
      Q⁻ += Q⁻'
      Q⁺ += Q⁺'
    else
      Restart(Ŝ)
      return
  Release(Ŝ, Q_{n-1} \ Q⁻ ∪ Q⁺)
proc Release(Ŝ, Q)
  𝒢_n := 𝒢_{n-1}, Q_n := Q, T_n := T_{n-1}
  foreach (s, l) ∈ Ŝ × ℒ do CloseEdge(s, l)
```

Fig. 8. Confirm query using environment diffs.

residual query to a set of *additional recorded queries* Q^+. This is necessary to ensure future iterations execute correctly. Although an added edge might not immediately lead to new declarations, additional edges added in future edits might do so. Thus, we need to confirm the new non-local queries after new edits. We will illustrate this in Section 4.3.

For removed edges (in δE^-), we apply the reverse reasoning. We query the scope graph of the *previous version* of the project using the newly introduced *PQuery* : $\mathcal{Q} \to \mathcal{A}$ message. If that returns results, valid paths in the original query answer existed, but are now removed. This invalidates the query, and therefore $\bot$ is returned again. Otherwise, we add Q' to the set Q^- of queries that can be removed. These queries do not need to be confirmed in future iterations anymore, because their sub-environments became unreachable by the edge removal.

When no added or removed edge leads to a change in the environment, the query is confirmed and Q^- and Q^+ are returned. The adapted `Hold` procedure shows how these sets are handled. First, they are accumulated for all confirmed queries. Finally, when the unit is released, the previous set of recorded queries is updated by removing Q^- and adding Q^+. In this way, the set of recorded queries is kept in sync with the recorded queries the original type-checker would have created in a non-incremental run.

This algorithm slightly overapproximates the environment diffs, as it does not take shadowing in the original query into account. When an added or removed path was shadowed in the original result, it in fact does not change the query answer. However, to check that, we would need to record the answer of a query. We expect that recording all answers would require a significant amount of memory, and that verifying whether diffs are shadowed would be computationally expensive. Instead, we accept this small overapproximation, which was never observed in practice.

4.3 Confirmation Example

We illustrate this algorithm with the example in Fig. 9. This example shows two consecutive edits in a Java program. The order of the edits is indicated using the circled superscripts. In the first edit, an **`extends`** `A` clause is added to class `B`. Second, the field `x` in class `A` is renamed to `y`. Together, both edits make the reference to `y` in class `C` resolve. In the scope graph on the right (in which the `TYPE` edges are omitted for brevity), the changes in the scope graph that correspond to these edits are shown.

Edit 1. We first show how the incremental analysis behaves after the first edit. At that stage, unit u_A is released immediately, because it has no (relevant) non-local queries. Unit u_B is analyzed, because it was edited. Since edge $s_C \cdot \mathsf{EXT} \cdot s_B$ already existed in the initial version, a query $q_1 = \langle s_B, \mathsf{EXT}^*\mathsf{FLD}, \mathbf{D}_\mathsf{y} \rangle$ was recorded for u_C. (Here $\mathsf{EXT}^*\mathsf{FLD}$ originates from $\partial_\mathsf{EXT}\mathsf{LEX}^*\mathsf{EXT}^*\mathsf{FLD}$.) u_C requests confirmation of this query by sending $\mathit{Confirm}(q_1)$ to u_B. Confirming this query in s_B yields an added edge $(s_A, \mathsf{EXT}^*\mathsf{FLD}) \in \delta E^+$. However, the residual query in this scope, executed by u_B, yields an empty answer because no declaration with name `y` exists yet. Moreover, it returns an additional recorded query $q_2 = \langle s_A, \mathsf{EXT}^*\mathsf{FLD}, \mathbf{D}_\mathsf{y} \rangle$ because s_A is non-local to u_B. Since the answer was empty, u_B confirms q_1, and u_C is released with q_2 added to the set of recorded queries Q_1. This outcome corresponds to the intuition that the new extends clause does not bring another declaration of `y` in scope, and therefore u_C does not need to be reanalyzed.

Edit 2. Second, the field `x` in class `A` is renamed to `y`. Intuitively, we would expect that u_C should be reanalyzed, because its reference to `y` can now resolve. Our algorithm ensures that happens as follows. Confirming q_1 in s_B gives a single matched edge $s_B \cdot \mathsf{EXT} \cdot s_A$. But as s_A is external to u_B, no added and removed edges in u_B are calculated. Thus, u_B confirms q_1. However, the confirmation of q_2 by u_A finds two changed edges: $(s_x, \varepsilon) \in \delta E^-$ and $(s_y, \varepsilon) \in \delta E^+$. Here ε, which is the regular expression matching the empty word, arises because $\partial_\mathsf{FLD}\mathsf{EXT}^*\mathsf{FLD} = \varepsilon$. Executing the residual

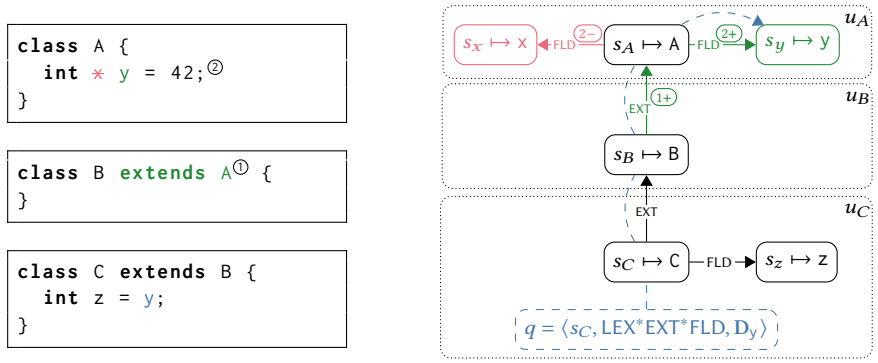

Fig. 9. Confirmation example. The left part shows a Java program, in which a field x is renamed to y. At the right side, the corresponding scope graph is shown, with the parts corresponding to the changes colored and numbered accordingly. In the program as well as in the scope graph, the color red denotes removals, while green denotes additions. In the scope graph, additions are also indicated with a plus symbol, and removals with a minus. Finally, q shows the query for y in C, which eventually resolves to A.y.

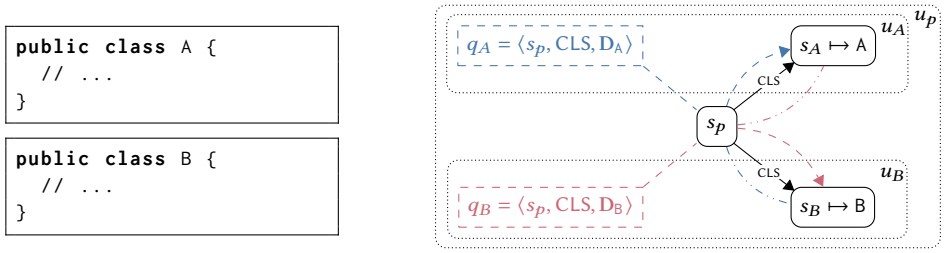

Fig. 10. Mutual dependency example. The queries represent duplicate name checks. The paths with alternating dashes and dots represent paths traversed by the query resolution algorithm that yielded no results but nonetheless gave rise to dependencies due to the fact the resolution step resulted in a new recorded query.

query for the removed edge gives no results since $\text{x} \notin \text{D}_\text{y}$, but the query in the added edge returns a path (s_y, y). Therefore, $\bot$ is returned to u_C, leading to a restart of the unit. Note that q_2, which led to the invalidation of this result, was added when confirming the first edit. This demonstrates the need to update the set of recorded queries for added and removed edges.

5 CONTEXT-FREE SNAPSHOTS

Essential to the viability of an incremental type-checker is its ability to deal with mutual dependencies. In this section, we explain how we deal with mutual dependencies around shared scopes. The key idea, inspired by Cardelli [1997] and Shao and Appel [1993], is to split the type-checking task into a part that can be decided locally and a part that depends on the context (i.e., the other compilation units). In Section 6, we show how to handle other occurrences of cyclic dependencies.

To illustrate the problem with mutual dependencies around shared scopes, we explain how our algorithm behaves for the example in Fig. 10. In this example, there are two classes (with their bodies omitted for brevity). Both classes execute a duplicate name check by doing a query for a class with their own name. When their name is not used by another class, this query should resolve to a single result. As shown in the graph, this generates residual queries in all classes in s_p. Thus, there is a mutual dependency between classes A and B.

```
interface TypeChecker
    async fun RunLocal(Ŝ) : 𝒴
    async fun RunInContext(Y) : 𝒯
proc Start(Ŝ)
    Y_n := await RunLocal_TC(Ŝ)
    𝒢*_n := 𝒢_n
    O*_n := O
    state_self := active
    T_n := await RunInContext_TC(Y_n)
proc Restart()
    T_n := RunInContext_TC(Y_{n−1})
proc Hold(Ŝ)
    𝒢_n := 𝒢*_{n−1}
    foreach (s, l) ∈ O \ O*_{n−1} do
        CloseEdge(s, l)
    // confirm queries as earlier
async fun Query(s, r, D)
    (A, Q) := Resolve(s, r, D)
    if Q ≠ ∅ then
        await state_self = active
        Q_n += Q
    return A
```

Fig. 11. Algorithm with adaptions to capture and restore context-free snapshots of type-checker.

Now suppose the body of class A is edited. That class will then re-execute its type-checker, re-creating the class declaration $s_p \cdot \texttt{CLS} \cdot s_A$ and issuing q_A. However, the query will get stuck in s_p because CLS is not closed, as it must still be closed by u_B. At the same time, u_B tries to confirm q_B. But the confirmation of q_B is stuck on CLS in s_p as well. In this situation, our algorithm cannot make progress anymore because both u_A and u_B are waiting for u_B to close CLS in s_p.

However, when we look into more detail in this example, it is obvious that this stuckness is not necessary. The declarations of the classes can be determined from just their source, without external information. Hence, it is not necessary for unit u_B to wait for the confirmation of its queries before declaring itself in s_p. Regardless of whether its queries are confirmed or not, $s_p \cdot \texttt{CLS} \cdot s_B$ will exist.

5.1 Capturing and Reusing Context-Free Snapshots

Generalizing this observation, we solve this problem as follows. We split the type-checking task into two subsequent subtasks: a *context-free* part and a *context-dependent* part. The context-free part will type-check the program partially, only using local information. Hence, in this phase, the unit will not deliver answers to non-local queries to the type-checker. Instead, these are delayed until the second subtasks starts. Thus, the result of the first step, dubbed the *context-free snapshot*, does not depend on external information. Hence, it can be reused when other units are edited, even when that leads to a restart. This behavior is shown in Fig. 12. A changed unit starts in state local, in which it executes the first part of the type-checking task. When that is finished, it transitions to the active state, where it executes the context-dependent checks. A restarted unit reuses the context-free snapshot of the previous iteration, and thus immediately starts in state active.

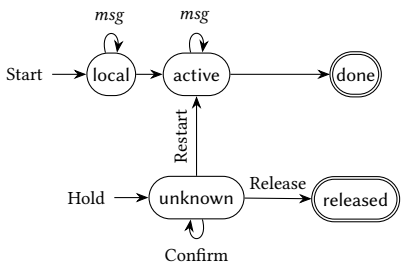

Fig. 12. Extended unit state diagram.

Fig. 11 shows the adaptations to the algorithm that implements this state diagram. First, the *TypeChecker* interface is split into two functions. The RunLocal function runs the context-free part of the algorithm, and returns a snapshot $Y \in \mathcal{Y}$. RunInContext takes such a snapshot, and returns a type-checker result $T \in \mathcal{T}$. Start executes these functions in state local and state active, respectively. Before transitioning to active, the current unit state is captured in $\mathcal{G}^*_n$ (called the *context-free scope graph*) and O^*_n. When initializing an unchanged unit (in Hold), the context-free scope graph of the previous iteration is restored, and all edges that were closed in the local phase are

closed again. If such a unit is restarted, `RunInContext` is used to re-execute the context-dependent part of the type-checker. Finally, the `Query` function (which is implemented by the unit, and can be invoked by the type-checker), is adapted. If a query was non-local, which can be deduced from its recorded residual queries, the unit waits until it is in state active before delivering the answer.

For Statix, a constraint-based type-checker, the new interface is implemented as follows. `RunLocal` solves constraints until all remaining constraints are blocked. At that point, these constraints and the current solver state (unifier etc.) are returned as Y. `RunInContext` then simply continues solving the constraints that are unblocked by the delivery of the answers to the non-local queries.

6 RESOLVING MUTUAL DEPENDENCIES

Essential to the soundness of the algorithm is handling mutual dependencies correctly. In the previous section, we showed how mutual dependencies around shared scopes are resolved by restoring the context-free part of the type-checker result at unit initialization time. In this section, we present a mechanism that detects and resolves all remaining mutual dependencies. This mechanism is additionally used to improve efficiency by saving redundant confirmations. Finally, we explain why only *transitive query recording* is sound with this resolution protocol.

6.1 Resolving Mutual Dependencies Involving Units in State Unknown

Van Antwerpen and Visser [2021] show that mutual dependencies result in a cluster of deadlocked units, which can be detected using deadlock detection algorithms. They provide a procedure (which we call `ResolveCyclesRegular`) that resolves such mutual dependencies. Because we extended the framework with new unit states (unknown and released), we need to handle mutual dependencies involving units in these states correctly. This can be done as follows. If *all* units involved in the deadlock are in state unknown or released, we release them all. This is sound because units in state unknown only wait for *Confirm* messages. When there is no active unit in the system, the answer to a query cannot be changed, as only active or done units can have a changed scope graph. Hence, they can be released, as their previous result of unknown units is still valid. Otherwise, if there is a unit in state active or done in the cluster, the units that are in state unknown must be restarted.

Fig. 13 shows how this is implemented. When there is no unit in the unknown state, the regular deadlock handling is applied. Otherwise, when all units are either in state unknown or released, all unknown units are released using the new *Release* message. If, on the other hand, there is a unit in $\hat{U}$ that was reanalyzed, all units in $\hat{U}$ must be reanalyzed. Therefore, a *Restart* message is sent to them. This is needed, because there is a changed unit in $\hat{U}$ on which all other units depend, but there is no precise way of determining which units to restart.

```
input: Û                                          set of mutually dependent units
proc ResolveCycles(Û)
  if ∀u ∈ Û. state_u ≠ unknown then ResolveCyclesRegular(Û)
  else if ∀u ∈ Û. state_u = unknown ∨ state_u = released then
    foreach u ∈ Û such that state_u = unknown do
      send Release() to u
  else foreach u ∈ Û such that state_u = unknown do
    send Restart() to u
```

Fig. 13. Resolution of cyclic dependencies.

This mechanism can be used to use query confirmation only when necessary. A query can only be denied when there are changed edges in the new scope graph. However, the scope graph of an unknown or released unit can not be changed (yet). Therefore, we can save computation time by postponing query confirmation until a unit is activated. When that results in a cluster of unknown units waiting for each other (which is likely to happen), the dependency resolution procedure will ensure all units are released. This behavior is implemented in the message handler shown in Fig. 15.

```
class A {
  int y = C.a;
}

class B extends A {
  int b = C.a;
}

class C extends B {
  static int a = 42;
  int z = y;
}
```

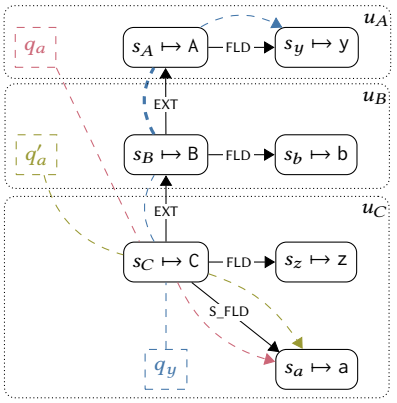

Fig. 14. Example demonstrating unsound confirmation. References are colored similar to the queries they correspond to. The resolution step with thick dashes between s_B and s_A indicates a transitive dependency.

6.2 Unsoundness of Transitive Confirmation

In Section 4.1, we mentioned that we choose recording *transitive queries* over transitive environment diffing or transitive query confirmation. We illustrate why that is needed using the example in Fig. 14. This example shows classes A and B that depend on class C, caused by queries for C.a. Similarly, C has a transitive dependency on A via B, indicated by the edge with thick dashes. Our algorithm records and confirms a separate residual query for this edge. However, we now suppose that is not the case, but EnvDiff calculates environment diffs for external scopes as well.

on receive *Confirm*$(\langle s, r, \mathbf{D}\rangle)$
  await $state_{\mathbf{self}}$ = active
  return await Confirm$(s, r, \mathbf{D})$

Fig. 15. *Confirm* message handler waiting until the unit is in state active.

Change in B. Suppose an irrelevant change in u_B is made. u_C will send *Confirm*(q_y) to u_B. Now, the (transitive) environment diff computation in u_B has to wait until unit u_A is active, before it can validate the query. However, unit A is in state unknown, and will never reply. So, u_B waits for u_A and u_C for u_B. Similarly, u_A waits for u_C for the confirmation of q_a. This is a mutual dependency between these three units, which our algorithm will resolve. However, as an active unit is involved, u_A and u_C are restarted, which is suboptimal.

Change in A. An even worse situation occurs when a change that invalidates q_y occurs in A (such as renaming y to x). In that case, confirmation of q_y is stuck in u_B, as it is inactive. Thus, no explicit wait for unit u_A is introduced. Hence, we arrive in the situation where only u_B and u_C are mutually dependent. Since both units are in state unknown, the mutual dependency resolution will release both units. However, this is unsound, as the change in u_A should have invalidated q_y.

Transitive Recording. On the other hand, explicitly recording and confirming transitive queries will immediately create dependencies from u_C to u_A and u_B, while u_B will not need to wait for u_A. In the first case, u_B can therefore answer u_C. Hence, the mutually dependent set only contains u_A and u_C, which results in a release of both. In the second case, u_C immediately sends a *Confirm* message to u_A, which u_A will deny, leading to a correct restart of u_C. In conclusion, we see that explicitly recording transitive queries turns transitive dependencies into non-transitive waits, which work well with the mutual dependency resolution.

7 SCOPE GRAPH DIFFING

Until now, we have assumed that scopes that have the same *semantic meaning* also have the same *identity* in both the current and the previous scope graph. For example, we used s_A to refer to a scope that models a class `A`, in both $\mathcal{G}_n$ and $\mathcal{G}_{n-1}$. However, the scope generation function `FreshScope` in fact creates nondeterministic identities. For example, the scope modeling a class might have identity s_0 in $\mathcal{G}_{n-1}$, but s_1 in $\mathcal{G}_n$. This section discusses two problems this poses for our incremental algorithm. First, we need to decide which scopes and edges in both scope graphs have have the same semantic meaning. We solve this problem using a *graph diffing algorithm*. Second, we need to update references to these scopes in the type-checker outputs. This is done by collecting pairs of matching scopes during environment diffing, and applying these as a substitution to the outputs of the type-checker.

7.1 Scope Graph Diffing Algorithm

Finding which scopes have the same semantic meaning in two different scope graphs essentially means solving a graph isomorphism problem. Because there are no known polynomial algorithms for graph isomorphism [Grohe and Schweitzer 2020], we aimed at a greedy approximation algorithm that works well on scope graphs in practise. This algorithm can compute partial diffs in *incomplete scope graphs*. We used the following constraints on scope graph diffs to guide the algorithm: First, we use shared scopes as initial matches. Second, matched scopes must always have the same owner and data. These constraints reduce the number of candidate matches for a scope significantly. For most name binding patterns, this results in single candidates, making the match trivial to decide. Finally, when an incorrect match is chosen, the incremental algorithm is less precise, but not unsound.

```
input: Ŝ~                                              initial matches

var: S~ ⊂ S_n × S_{n-1} := Ŝ~                          matched scopes
var: S⁻ ⊂ S_{n-1} := ∅                                 removed scopes
var: S⁺ ⊂ S_n := ∅                                     added scopes

var: E~ ⊂ E_n × E_{n-1} := ∅                           matched edges
var: E⁻ ⊂ E_{n-1} := ∅                                 removed edges
var: E⁺ ⊂ E_n := ∅                                     added edges

var: Z ⊂ E_n × E_{n-1} = ∅                             queue of matches

async fun Diff(Ŝ~) : ∂G
  foreach s_n ~ s_{n-1} ∈ Ŝ~ do
    ScheduleEdges(s_n, s_{n-1})
  while (e_n, e_{n-1}) := await dequeue(Z) do
    TryMatch(e_n, e_{n-1})
  return Finalize()
proc ScheduleEdges(s_n, s_{n-1})
  foreach l ∈ ℒ do
    E_n := await getEdges(s_n, l, G_n)
    E_{n-1} := getEdges(s_{n-1}, l, G_{n-1})
    foreach (s'_n, s'_{n-1}) ∈ E_n × E_{n-1} do
      if CanMatch(s'_n, s'_{n-1}) then
        e_n := s_n · l · s'_n, e_{n-1} := s_{n-1} · l · s'_{n-1}
        Z += (e_n, e_{n-1})
fun CanMatch(s_n, s_{n-1})
  o := owner(s_{n-1}) = owner(s_n)
  d := ρ_{n-1}(s_{n-1}) = ρ_n(s_n)
  return o ∧ d
proc TryMatch(e_n, e_{n-1})
  s'_n := target(e_n), s'_{n-1} := target(e_{n-1})
  if Consistent(s'_n, s'_{n-1}) then
    E~ += e_n ~ e_{n-1}, S~ += s'_n ~ s'_{n-1}
    ScheduleEdges(s'_n, s'_{n-1})
  if e_n ∉ E~ ∧ e_n ∉ dom(Z) then E⁺ += e_n
  if e_{n-1} ∉ E~ ∧ e_{n-1} ∉ ran(Z) then E⁻ += e_{n-1}
fun Consistent(s_n, s_{n-1}) : Bool
  free := s_n ∉ dom(S~) ∧ s_{n-1} ∉ ran(S~)
  return s_n ~ s_{n-1} ∈ S~ ∨ free
fun Finalize() : ∂G
  S⁺ := S_n \ dom(S~), S⁻ := S_{n-1} \ ran(S~)
  return (S~, S⁺, S⁻, E~, E⁺, E⁻)
```

Fig. 16. Scope graph diffing algorithm.

The input and the local state of the diffing algorithm are shown in Fig. 16. The input consists of a non-empty set of initial matches. The algorithm maintains monotonically increasing sets of scopes and edges that are matched, added or removed. Finally, there is a queue Z that contains entries for all edges that can possibly be matched, which is the case when their targets can be matched.

The diffing algorithm starts in `Diff`, shown in Fig. 16, with matching the outgoing edges of the shared scopes. Matching edges is performed using the following steps. First, the algorithm waits until all outgoing edges for the current graph are available. Second, pairs of current and previous edges that have potentially matching targets are enqueued in Z. Third, when an edge pair is dequeued, and the match is consistent, it is applied to the current state. Finally, for all newly matched scopes, this procedure is repeated.

The `ScheduleEdges` procedure implements the first two steps. For a pair of matched scopes, all outgoing current edges are collected. When these are known, all potential matches are inserted in Z. Note that these steps are asynchronous, as they need to wait for the edges. Because no global variables are updated (except inserting in Z), this will not cause data races or similar problems.

After `Diff` scheduled the initial edge matches, it will peek the queue of pending matches. This operation is asynchronous, because the queue can be empty while there are still pending calls to getEdges. In such cases, dequeue will wait until either all these returned, or a new match is enqueued. When there is a candidate match dequeued from Z, the `Consistent` function checks if it is consistent with the differ state. Consistency means that the target scopes are either already matched to each other, or both not matched at all. When that is the case, the edge match and all requirements are inserted in the differ state. When either the current or the previous edge is not matched and not scheduled anymore, it is marked as added or removed, respectively. When the queue is empty and there are no pending asynchronous method calls, the scope graph diff is finalized by marking the remaining scopes and edges as added or removed, as appropriate.

7.2 Patch Collection and Application

The fact that scope identities change when a unit is reanalyzed also implies that references to scopes in a previous result might become stale. Therefore, we need to substitute these references with the scope they are matched to. To perform this task, we collect all matched scopes during query confirmation, and apply those as a substitution to the typechecker result.

Fig. 17 shows the adapted confirmation algorithm that also tracks scope patches. In `EnvDiff`, two changes are made. First, the call to `Diff`$(s,\ l)$ is replaced by an invocation of $\mathsf{diff}(\partial\mathcal{G}, s_{n-1})$. This function probes a (possibly incomplete) scope graph diff $\partial\mathcal{G}$ for a *scope diff* of s_{n-1}. This returns the scope s_n that is matched to s_{n-1}, as well as the matched, added and removed outgoing edges. The scope diff is returned when all outgoing edges of s_{n-1} and s_n are processed. Second, a set of scope patches Π is introduced. In this set, all matches of scopes traversed during confirmation are

```
async fun EnvDiff(s_{n-1}, r, ∂G) : Π × δE⁺ × δE⁻
  if owner(s) ≠ self then return (∅, ∅, ∅)
  (s_n, E~, E⁺, E⁻) := await diff(∂G, s_{n-1})
  δE⁺ := ∅, δE⁻ := ∅, Π := { s_n ~ s_{n-1} }
  foreach s_n · l · s'_n ~ s_{n-1} · l · s'_{n-1} ∈ E~ such that ∂_l r ≠ ∅ do
    (Π', δE⁺', δE⁻') := await EnvDiff(s'_{n-1}, ∂_l r, ∂G)
    Π += Π', δE⁺ += δE⁺', δE⁻ += δE⁻'
  δE⁺ += { (s'_n, ∂_l r) | s_n · l · s'_n ∈ E⁺ }
  δE⁻ += { (s'_{n-1}, ∂_l r) | s_{n-1} · l · s'_{n-1} ∈ E⁻ }
  return (Π, δE⁺, δE⁻)
async fun Confirm(s_{n-1}, r, D, ∂G) : (P(Q_{n-1}) × P(Q_n) × Π)?
  Q⁻ := ∅, Q⁺ := ∅
  (Π, δE⁺, δE⁻) := await EnvDiff(s_{n-1}, ∅, r, ∂G)
  // Compute Residual Queries as earlier
  return (Q⁻, Q⁺, Π)
proc Release(Π)
  G_n := Π(G_{n-1}), Q_n := Π(Q_{n-1})
  G*_n := Π(G*_{n-1}), Y_n := Π(Y_{n-1})
  foreach (s, l) ∈ O do CloseEdge(s, l)
  T_n := Π(T_{n-1})
```

Fig. 17. Query confirmation algorithm with scope patching.

collected, alongside the environment diff. Eventually, `Confirm` simply returns Π to the sender of the confirmation message. The receiving unit accumulates all patches it receives (not shown in the figure) and passes them to `Release` when appropriate. This function applies the patches (written as $\Pi(\cdot)$) to the previous scope graph, recorded queries, context-free scope graph and snapshot.

To prove the correctness of this approach, recall that type-checkers can only obtain references to non-local scopes via paths in query answers. Therefore, collecting patches for all scopes in an answer to a previous query is sufficient to covering all scopes in $\mathcal{G}_{n-1}$ and T_{n-1}. As the query confirmation algorithm overapproximates the paths of a query, our algorithm overapproximates the set of required patches, and is thereby correct. Additionally, removed scopes (for which hence no path exists) are not a problem. If there is a reference to a removed scope in the type-checker result, there must be a path in a previous query answer that contained that scope. However, when that scope was removed, the edge leading to that scope was necessarily removed as well. That implies that a path from the original query answer was removed, which led to a restart of the unit. As patching is only applied to *released* units, the missing patch was irrelevant.

Scope graph diffing and result patching are only required for type-checkers that generate non-deterministic scope identities. Type-checkers for which scope identities are deterministic can just use the algorithm as presented in Fig. 8.

8 EVALUATION

To evaluate our approach, we used two Statix specifications: one for Java and one for WebDSL [Groenewegen et al. 2008], which is a domain-specific language for dynamic web applications. We chose Java because it is a language with many different features of which some are challenging to incrementalize (e.g. method overloading, transitive resolution in inheritance trees). Similarly, WebDSL has many implicit definitions, implicit imports and implicit mutual dependencies. This makes it a language that is non-trivial to incrementalize and hence a good evaluation subject.

We first determined the accuracy of the scope graph diffing algorithm (Section 8.1). In addition, we implemented synthetic benchmarks (Section 8.2), but executed the type-checker on the version history of some real-world projects (Section 8.3) as well. We executed all benchmarks with the JMH benchmark tool [OpenJDK 2021], using 5 warm-up iterations and 20 measurement iterations, and 6GB of memory. The benchmarks were executed on a Linux system with 2 AMD EPYC 7502 32-Core Processors (1.5GHz, 2 threads) and 256GB RAM. The measured time is for type-checking only, and does not include parsing and desugaring, which we regard as out of scope for this work.

8.1 Accuracy of Scope Graph Diffing

The first aspect we evaluated is the accuracy of the scope graph diffing algorithm. We computed scope graph diffs of 406 Java and 80 WebDSL files. We found that the algorithm gives minimal diffs for almost all scope graphs. Only when method overloads were supported, results were not accurate. This was caused by the fact that overloads are distinguished *by their type*, but the declaration scopes only have the identifiers as datum. Therefore, the diffing algorithm cannot distinguish between the scopes of different overloads of a method, and chooses a match arbitrarily. Incorrect declaration scope matches ensures their type nodes could not be matched, leading to unnecessarily invalidated queries. We solved this by including the syntactic type (i.e., the AST node that represents the type of a method) in the datum of the declaration scope. This guides the differ to select the correct declaration scope. When a specification includes syntactic types in its declarations, the diffing algorithm always computes minimal scope graph diffs.

We illustrate this using the example in Fig. 18. In this program, there are two overloaded methods `m`. Both declaration scopes s_{m_1} and s_{m_2} have `m` as their datum. Now suppose that two versions of this scope graph are diffed. In that case, there are two instances of each scope: s_{m_1} and s_{m_2} in $\mathcal{G}_{n-1}$,

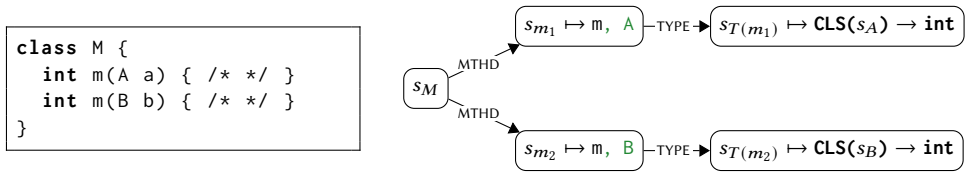

Fig. 18. Non-deterministic matching of overloaded methods. The additional information is shown in green.

and s'_{m_1} and s'_{m_2} in $\mathcal{G}_n$. Now, the scope graph differ might match s_{m_1} to s'_{m_2}, as their data are equal. Therefore, we add the syntactic representation of the arguments (shown in green) to the datum of the scopes. This allows the differ to distinguish between the different declarations of `m`, and correctly match s_{m_1} to s'_{m_1} and s_{m_2} to s'_{m_2}.

8.2 Synthetic Benchmarks

To evaluate the performance of our approach, we created the following six synthetic benchmarks in Java. These benchmarks cover typical Java project development scenarios, and additionally exercise different execution paths through our restart strategy.

(1) In the *const-change-no-refs* benchmark, the value of a static, unreferenced constant is changed.
(2) The *const-change-10-refs* benchmark is similar, but now 10 other classes reference the constant.
(3) The third benchmark (*superfield-change*) contains an inheritance chain of ten classes, starting in class `C0`, with class `C9` at the bottom of the hierarchy. `C0` contains a field, of which the name is changed. `C9` references this field, which gives rise to a query through all these classes.
(4) The *new-overload* benchmark is similar to the previous one, but instead of a field name being changed, a new method overload is added, which results in a restart for class `C9`.
(5) The *change-extends* benchmark is similar to benchmark (3), but differs in three ways: class `C9` does not extend `C8`, `C8` references a field in `C0`, and the change in the program is that `C1` changes its parent class to `C9`. This change results in a unresolved reference in `C8`.
(6) Finally, the *precedence-takeover* benchmark is similar to benchmark (3) as well, but `C1` does not inherit from `C0`, and `C9` has a field with type `C0`. Here a static inner class with name `C0` is added to `C1`. Now, the occurrence of `C0` in `C9` references this new class.

In order to benchmark behavior in different contexts, we executed these benchmarks with different 'payloads': additional unchanged classes that do not reference other classes. These payload classes each contain 10 fields, and 20 methods with 4-6 arguments, containing 4-6 method invocations. Each class resides in its own package to mitigate dependencies for duplicate name checking. We used small, medium and large payload sets, containing 5, 20 and 100 classes, respectively. These classes were added to the benchmarked project to measure the runtime characteristics of the algorithm for different project sizes.

The benchmark results are shown in Fig. 19. These graphs show that benchmarks without payload do not attain any speedup, but the bigger the payload is, the better the speedup factor becomes, up to a maximum of 147. This is expected, because for bigger payloads, more results can be reused. Next, speedups generally decrease when the number of cores is increased. This behavior has two causes. First, reused units have little work to do, so the active unit dominates the runtime. In addition, the type-checker of an active unit requires some progress before it can invalidate queries of other units. During that time, units that eventually are restarted are still idle, not utilizing the available cores. On the other hand, initial runs still benefit from a larger number of cores.

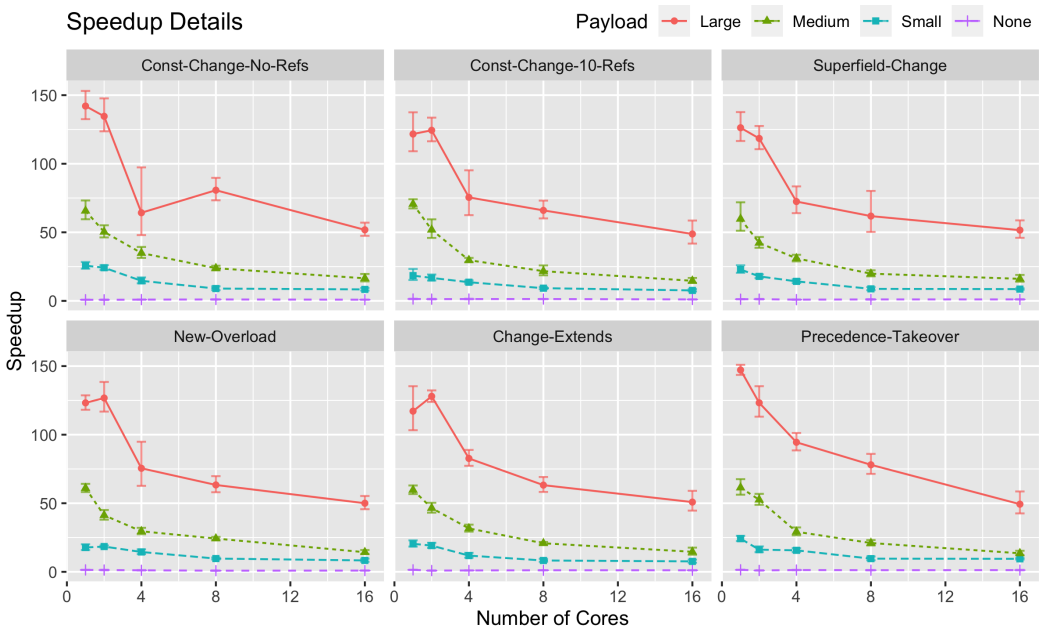

Fig. 19. Speedups of synthetic benchmarks. Speedups are decrement factors in runtime relative to non-incremental analysis at the same level of parallelism. Error bars indicate 99.9% confidence intervals.

8.3 Version Control History Benchmarks

We additionally applied the framework on three commits from each of the Apache Commons CVS, IO and Lang3 projects (Java) and the Reposearch[3] and YellowGrass[4] WebDSL applications. The number of data points is limited because benchmarking a single commit at multiple levels of parallelism already consumes significant time. The chosen commits varied in size and number of edited units and should therefore be representative for typical project evolution. Details about the commits we benchmarked can be found in Appendix B.

Table 1 gives an impression of the absolute runtimes of the single-core benchmarks[5]. The table shows that non-incremental analysis can take up to a few minutes for large projects, while incremental analysis generally takes a few seconds. Second, there is quite some variance in the incremental analysis time for different commits. In particular, the second commits of Commons-IO and YellowGrass draw attention. The relatively large runtime values for these benchmark iterations are explained by the fact that the diffs of these commits are relatively large (two and seven changed files, respectively). As our approach incrementalizes with compilation unit granularity, such commits still require significant time to reanalyze. Future research in using our approach with smaller compilation units (e.g. methods) or integrating query confirmation more directly with the underlying type-checker may improve incremental runtimes for such commits.

The speedup factors for all benchmarks are shown in Fig. 20 and Fig. 21, respectively. Similar to the synthetic benchmarks, larger projects obtain better speedups, with a maximum of 21x in the

[3] https://github.com/webdsl/reposearch
[4] https://github.com/webdsl/yellowgrass
[5] The values in the table should be interpreted with care, as the clock speed of the benchmarking machine (1.5GHz) is lower than that of modern developer machines (2.5 - 3 GHz).

Table 1. Absolute runtimes of single-core benchmarks. Values are given in seconds. *NI* columns give non-incremental (baseline) values, and *I* columns give incremental runtimes.

Project		Commons-CSV		Commons-IO		Commons-Lang		YellowGrass		Reposearch	
Mode		*NI*	*I*	*NI*	*I*	*NI*	*I*	*NI*	*I*	*NI*	*I*
Commit	1	8.1	5.4	52	4.7	124	6.7	35	2.2	28	8.5
	2	8.3	5.2	50	13.4	129	6.0	39	15.4	28	3.8
	3	7.6	4.8	59	5.1	128	8.9	36	2.8	28	6.0

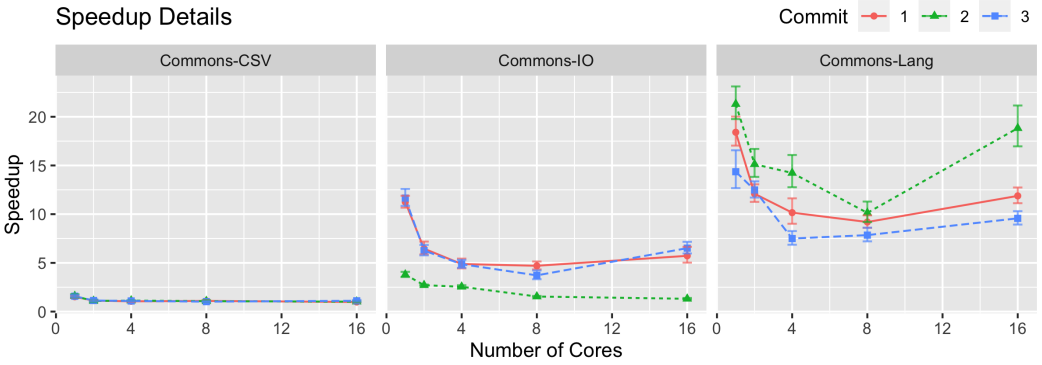

Fig. 20. Speedup factors of Apache Commons benchmarks.

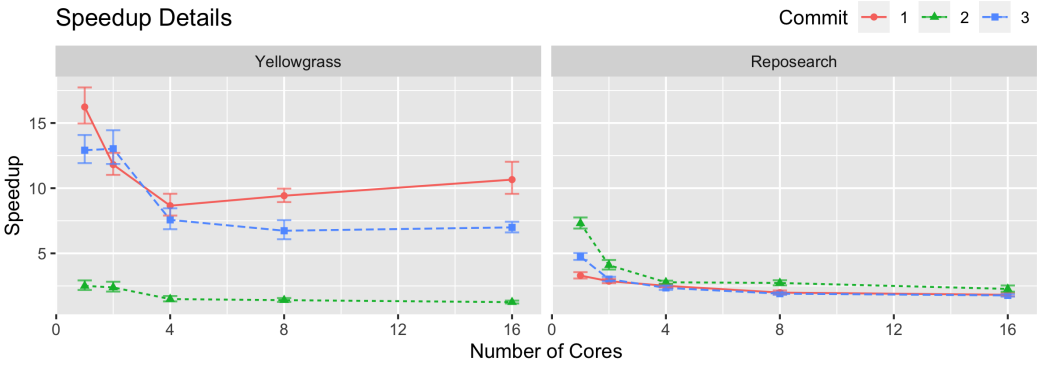

Fig. 21. Speedup factors of Reposearch and Yellowgrass WebDSL benchmarks.

Commons-Lang project, while small projects, such as Commons-CSV, barely have any speedup at all. On average, Java projects obtain more speedup than the WebDSL projects. This is likely caused by the fact that modeling the WebDSL module system required more complicated queries than Java, and the WebDSL projects are smaller than the Commons-IO and Commons-Lang projects. In addition, we found that the query recording accounted for an overhead of at most 10%. This shows that initial overhead is relatively low, and not a limiting factor of our approach.

There are interesting differences between the synthetic and the version-control-based benchmarks as well. The speedup factors of the synthetic benchmarks are relatively higher, which

suggests our algorithm behaves best on larger projects with many small compilation units. On the other hand, the VCS-based benchmarks retain their speedup when the number of cores increases. This indicates that the incremental runtime is closely correlated to the runtime of the reanalyzed units, and does not incur much overhead from message ordering.

As mentioned at the beginning of this section, we used a Statix solver as a subject type-checker in our benchmarks. This solver interprets declarative type system specifications as constraint programs. Until now, this type-checking strategy has never attained performance comparable to industrial, language-specific compilers. However, our work has reduced analysis time on large projects from minutes to seconds, bringing interactive editor response times into reach. In future work, we want to explore whether other optimization techniques can improve performance even more, making this approach competitive with respect to performance.

9 RELATED WORK

In this section, we discuss related work on principles of incremental/separate compilation, approaches to incremental type-checking, and incremental analysis and build systems.

9.1 Principles of Incremental Analysis

Incremental type-checking is closely related to separate compilation. In his work on module systems for separate compilation, Cardelli [1997] formalizes a module system that supports true separate compilation and (non-incremental) safe linking. In his approach, imports and exports, including their types, are declared explicitly. Intra-module type-checking then does not require external lookup, but instead refers to the import declarations. Additionally, recursive dependencies are precluded. Drossopoulou et al. [1999] provide a model for linking and updating fragments of linked programs that is more tailored to real-world languages, such as Java. In later work, more calculi for separate type-checking and compilation have been developed [Ancona 1998; Machkasova and Turbak 2000]. Our approach supports both implicit and explicit imports/exports as well as mutual dependencies. Moreover, we preserve soundness and completeness of the underlying specification.

Shao and Appel [1993] propose an algorithm for languages with a Damas-Milner-style type system that infers a minimum set of imports that are required to type-check a module. Whether the required imports are present is validated at link-time. In this approach, a module only needs to be recompiled when it is changed. This yields a simpler restarting policy than our framework. However, delaying inter-module type-checking to link-time might not be desirable in many situations, such as using type information in editor services. Our algorithm is a generalization over this, where our 'context-free phase' corresponds to inferring the minimum number of imports (non-local queries), and the 'context-sensitive' part to their linking phase.

In the ML-family of languages, separate compilation and incremental type-checking have received significant attention. Among one of the earliest approaches to separate compilation for ML is the work of Leroy [1994]. Just as the formal calculi of the previous section, this work enforces the use of explicit module types. This allows to separate inter-module type-checking from intra-module checking at link-time. This line of work has since then resulted in several approaches for separate compilation in ML [Elsman 2008; Swasey et al. 2006].

9.2 Incremental Type-Checkers

Among the earliest attempts to create an incremental type-checker was the *B* language [Meertens 1983]. Its type system features lexical scoping with top-level function declarations for which polymorphic types are inferred. B programs were type-checked incrementally by upward propagation of type requirement updates. This approach was feasible because there was no type-dependent name resolution.

The standard Java compiler (`javac`) supports some form of incremental compilation. When a particular dependency is available in binary format (`.class`), its corresponding source is not required to be available. It has been argued that this does not implement true separate compilation, as the tasks of compilation and linking are intertwined [Ancona et al. 2002]. For a subset of Java, true separate compilation has been formalized [Ancona et al. 2002; Ancona and Zucca 2002]. More recent work focusses on incremental type-checking for Java in general [Kuci et al. 2015], or tailored to overload resolution [Szabó et al. 2018]. Eclipse Java Development Tools (JDT) also claims to be incremental [Eclipse 2021], but little information about the implementation could be found.

9.3 Language-Parametric Incremental Type-Checking

Generic approaches for incremental type-checking have been devised as well. Demers et al. [1981] define two approaches for incremental evaluation of attributes in attribute grammars. An overview of follow-up work can be found in the survey of Ramalingam and Reps [1993, sect. 3.1.1, 4.3 and 5].

Wachsmuth et al. [2013] define an approach in which a program is type-checked in two phases. In the first phase, a collection of atomic, inter-dependent tasks is collected, which are executed in order in the second phase. When a file is changed, its corresponding tasks are recollected and re-executed when appropriate. This results in speedups of 10x on average. Their approach is tied to a predecessor of Statix, which uses a more limited graph-based name binding model, supporting only lexical scoping and imports.

Co-contextual typing rules are a promising approach to deriving incremental type-checkers [Erdweg et al. 2015]. In this approach, context requirements are propagated upward, instead of the traditional approach, where contexts are propagated downward. When a program is changed, context requirements from siblings can trivially be reused. An intelligent constraint solving scheme saves even more work. This approach gives finer-grained incrementality than our approach does, but is more limited in the type system features and name-binding patterns it supports. For example, overload resolution seems hard to incrementalize this way, due to its non-local nature [Szabó et al. 2018]. Performance-wise, their speedups in incremental runs (up to 24x) slightly exceed our results, but non-incremental runs impose slowdowns (up to 3x) as well.

Busi et al. [2019] define a procedure that derives an incremental type-checking algorithm from a non-incremental one. This procedure introduces a cache of the previous environment into the rules, which is used when the type of a term does not change under an augmented environment. Defining the transformation on typing rules has the advantage that formal analysis on the incremental type system can be performed. This 'grey-box' approach requires some domain knowledge about the type system to instantiate. Instead, our framework only requires that a specification follows a split declaration-type style with syntactic types disambiguating overloads.

Recent work derives incremental type-checkers from algorithmic typing rules expressed in Datalog [Pacak et al. 2020]. Three transformations optimize the propagation of contexts, yielding efficient incremental updates. In fact, we view the approaches as complementary for the following reasons. Pacak et al. focus on *intra-unit* incrementality (at the constraint level), while our work is aimed at incrementalizing analysis with respect to external compilation units. In future work, we aim to complement the work in this paper with incremental constraint solving within a unit. However, this is challenging because Statix has a top-down evaluation model and scope graphs are a *global* structure. On the other hand, Statix is a suitable platform for language-parametric services, such as semantic code completion [Pelsmaeker et al. 2022] and renaming [Misteli 2021], which makes it attractive as a type-checking framework. Finally, the evaluation of Pacak et al. on 200 nested lambdas in PCF shows speedups up to 2500x, but incurs 20x slowdown on initial runs. However, their approach has not yet been evaluated on real-world languages, which makes it difficult to compare absolute performances of the approaches.

9.4 Incrementality in Other Domains

Incremental analysis has recently got significant attention in the domain of static (data-flow) analysis [Arzt and Bodden 2014; Szabó et al. 2021, 2016]. This work usually builds on incremental Datalog solvers [Ryzhyk and Budiu 2019; Szabó et al. 2016]. Comparing these approaches rigorously would involve comparing properties of the analysis they perform (such as locality), which we identify as potential future research.

Another analogous research field is *Incremental Build Systems* [Greene 2015; Hammer et al. 2014; Konat et al. 2018; Sánchez et al. 2020]. Build Systems are analogous in the fact that they usually have tasks as the granularity level, which is analogous to our compilation units. Tasks are treated as black boxes, and (dynamic) dependencies are derived from task inputs and outputs. On the other hand, build systems usually rely on a topological ordering on tasks, precluding recursive dependencies. Because our system has fine-grained dependencies, changes only cascade when necessary. This is similar to early cut-off in build systems [Konat et al. 2018].

10 CONCLUSION

In this paper, we describe how scope graph queries represent fine-grained dependencies between compilation units. We present a technique which uses this insight to execute scope-graph-based type-checkers incrementally. This algorithm is implemented in a framework that automatically incrementalizes type-checkers derived from a Statix specification. The framework observes all cross-unit queries a type-checker performs, and confirms those on incremental executions by computing environment diffs. When an added/removed edge affects a query answer, its originating unit is reanalyzed. Most mutual dependencies are resolved by splitting the type-checking process into a local and a context-dependent phase. Other cyclic dependencies lead to restarts only when an active unit is involved. Benchmarks show our approach yields speedups up to 147x on synthetic projects, and speedups up to 21x on real codebases. Hence, this framework shows that scope graphs allow automatic derivation of sound and almost optimal incremental type-checkers.

Future Work. In this paper, we assume that our subject type-checker was derived from a Statix specification. However, our algorithm in fact makes only a few assumptions on the type-checker it incrementalizes. Therefore, an interesting question for future research is whether our algorithm can be applied to scope-graph-based type-checkers that do not use Statix. Another interesting future research direction is investigating whether query confirmation can be integrated more tightly with the underlying type-checker, to improve the granularity of the incrementality, while retaining implicitness and performance. In addition, integrating this framework with an incremental build system would contribute to building fully incremental compiler pipelines.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback on earlier versions of this paper. We also want to thank Casper Bach Poulsen for his assistance during the publication process, and Max de Krieger for providing us with a Statix specification for WebDSL.

REFERENCES

Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

Davide Ancona. 1998. An Algebraic Framework for Separate Type-Checking. In *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT 98, Lisbon, Portugal, April 2-4, 1998, Selected Papers (Lecture Notes in Computer Science, Vol. 1589)*, José Luiz Fiadeiro (Ed.). Springer, 1–15. https://doi.org/10.1007/3-540-48483-3_1

Davide Ancona, Giovanni Lagorio, and Elena Zucca. 2002. True separate compilation of Java classes. In *Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming, October 6-8, 2002, Pittsburgh, PA, USA (Affiliated with PLI 2002).* ACM, 189–200. https://doi.org/10.1145/571157.571177

Davide Ancona and Elena Zucca. 2002. A calculus of module systems. *Journal of Functional Programming* 12, 2 (2002), 91–132. https://doi.org/10.1017/S0956796801004257

Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 288–298. https://doi.org/10.1145/2568225.2568243

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494.

Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Using Standard Typing Algorithms Incrementally. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11460)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer, 106–122. https://doi.org/10.1007/978-3-030-20652-9_7

Luca Cardelli. 1997. Program Fragments, Linking, and Modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 266–277. https://doi.org/10.1145/263699.263735

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017). https://doi.org/10.1145/3133872

Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 105–116. https://doi.org/10.1145/567532.567544

Sophia Drossopoulou, Susan Eisenbach, and David Wragg. 1999. A Fragment Calculus – Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *Proceedings, 14th Annual IEEE Symposium on Logic in Computer Science, 2-5 July, 1999, Trento, Italy*. IEEE Computer Society, 147–156. https://doi.org/10.1109/LICS.1999.782606

Eclipse. 2021. *JDT Core Component*. Retrieved 2021-09-17 from https://www.eclipse.org/jdt/core/

Martin Elsman. 2008. *A Framework for Cut-Off Incremental Recompilation and Inter-Module Optimization*. Technical Report. IT University of Copenhagen, Copenhagen. 11 pages. https://elsman.com/pdf/sepcomp_tr.pdf

Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277

Sterling Greene. 2015. *Introducing Incremental Build Support*. Retrieved 2021-10-15 from https://blog.gradle.org/introducing-incremental-build-support

Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. 2008. WebDSL: a domain-specific language for dynamic web applications. In *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 779–780. https://doi.org/10.1145/1449814.1449858

Martin Grohe and Pascal Schweitzer. 2020. The graph isomorphism problem. *Commun. ACM* 63, 11 (10 2020), 128–134. https://doi.org/10.1145/3372123

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 18. https://doi.org/10.1145/2594291.2594324

Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable incremental building with dynamic task dependencies. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 76–86. https://doi.org/10.1145/3238147.3238196

Edlira Kuci, Sebastian Erdweg, and Mira Mezini. 2015. Toward incremental type checking for Java. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 46–47. https://doi.org/10.1145/2814189.2817272

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 109–122. https://doi.org/10.1145/174675.176926

Elena Machkasova and Franklyn A. Turbak. 2000. A Calculus for Link-Time Compilation. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 260–274. https://doi.org/10.1007/3-540-46425-5_17

Lambert G. L. T. Meertens. 1983. Incremental Polymorphic Type Checking in B. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 265–275. https://doi.org/10.1145/567067.567092

Phil Misteli. 2021. *Renaming for Everyone: Language-parametric Renaming in Spoofax*. Master's thesis. Delft University of Technology. http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5 Available at http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5..

Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

OpenJDK. 2021. Java Microbenchmark Harness (JMH). https://openjdk.java.net/projects/code-tools/jmh/

André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428195

Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages* 6, OOPSLA (2022), 1–30. https://doi.org/10.1145/3527329

Ganesan Ramalingam and Thomas W. Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 502–510. https://doi.org/10.1145/158511.158710

Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428248

Leonid Ryzhyk and Mihai Budiu. 2019. *Differential Datalog*. Retrieved 2021-10-15 from http://budiu.info/work/ddlog.pdf

Zhong Shao and Andrew W. Appel. 1993. Smartest Recompilation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 439–450. https://doi.org/10.1145/158511.158702

David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. 2006. A separate compilation extension to standard ML. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, Andrew Kennedy and François Pottier (Eds.). ACM, 32–42. https://doi.org/10.1145/1159876.1159883

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental whole-program analysis in Datalog with lattices. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1–15. https://doi.org/10.1145/3453483.3454026

Tamás Szabó, Sebastian Erdweg, and Markus Völter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

Tamás Szabó, Edlira Kuci, Matthijs Bijman, Mira Mezini, and Sebastian Erdweg. 2018. Incremental overload resolution in object-oriented programming languages. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*, Julian Dolby, William G. J. Halfond, and Ashish Mishra (Eds.). ACM, 27–33. https://doi.org/10.1145/3236454.3236485

Beatriz Sánchez, Dimitris S. Kolovos, and Richard F. Paige. 2020. To build, or not to build: ModelFlow, a build solution for MDE projects. In *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*, Eugene Syriani, Houari A. Sahraoui, Juan de Lara, and Silvia Abrahão (Eds.). ACM, 1–11. https://doi.org/10.1145/3365438.3410942

Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276484

Hendrik van Antwerpen and Eelco Visser. 2021. Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.1

Guido Wachsmuth, Gabriël Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. 2013. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 260–280. https://doi.org/10.1007/978-3-319-02654-1_15

Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. 2022. *Incremental Type-Checking for Free: Artifact*. Zenodo. https://doi.org/10.5281/zenodo.7071393