Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers

Hendrik van Antwerpen ✉ 🆔
Delft University of Technology, The Netherlands

Eelco Visser ✉ 🆔
Delft University of Technology, The Netherlands

── Abstract ──

Compilers that can type check compilation units in parallel can make more efficient use of multi-core architectures, which are nowadays widespread. Developing parallel type checker implementations is complicated by the need to handle concurrency and synchronization of parallel compilation units. Dependencies between compilation units are induced by name resolution, and a parallel type checker needs to ensure that units have defined all relevant names before other units do a lookup. Mutually recursive references and implicitly discovered dependencies between compilation units preclude determining a static compilation order for many programming languages.

In this paper, we present a new framework for implementing hierarchical type checkers that provides implicit parallel execution in the presence of dynamic and mutual dependencies between compilation units. The resulting type checkers can be written without explicit handling of communication or synchronization between different compilation units. We achieve this by providing type checkers with an API for name resolution based on scope graphs, a language-independent formalism that supports a wide range of binding patterns. We introduce the notion of *scope state* to ensure safe name resolution. Scope state tracks the completeness of a scope, and is used to decide whether a scope graph query between compilation units must be delayed. Our framework is implemented in Java using the actor paradigm. We evaluated our approach by parallelizing the solver for Statix, a meta-language for type checkers based on scope graphs, using our framework. This parallelizes every Statix-based type checker, provided its specification follows a split declaration-type style. Benchmarks show that the approach results in speedups for the parallel Statix solver of up to 5.0x on 8 cores for real-world code bases.

1 Introduction

Despite the general availability of multi-core architectures, many compilers do not take advantage of these for type checking. Parallelizing a compiler remains a challenging task, which requires dealing with explicit synchronization and communication between compilation units. For example, the authors of GCC made the following remark about their efforts to parallelize parts of the compiler [6]:

> *"One of the most tedious parts of the job was [. . .] making several global variables threadsafe, and they were the cause of most crashes in this project."*

They continue to say that even with the help of specialized tools it remained difficult to do the parallelization correctly. Compilers for many major languages do not support parallel front ends, or only experimentally. Some build tools allow the compiler to be run in parallel, but many require a static compilation order, and because they have no internal knowledge of the language being compiled, cannot generically handle cyclic dependencies between compilation units. A generic, reusable solution to the problem of how to implement type checkers that can process compilation units in parallel, which correctly deals with (potentially cyclic) dependencies between units, is missing[1].

Dependencies between compilation units are the results of name lookup from one unit into another. A correct concurrent type checker must ensure that when a lookup is done, all relevant units have progressed enough to provide a complete answer. For languages that support true separate compilation (e.g., [23]), there are no lookups into other units, and processing them in parallel is trivial. It may also be possible to run type checkers in parallel using a compilation order based on static or dynamic dependencies, which ensures units are compiled after their dependencies. But many programming languages have features, such as mutually recursive modules, that result in mutual dependencies between compilation units. When compilation units are mutually dependent, neither unit can be completed before the other is at least partially checked. A more fine-grained approach than processing compilation units in a fixed order is required.

This paper presents a new framework for the implementation of type checkers that provides *implicit parallel execution*. Type checkers are organized as a hierarchy of compilation units, which allows modeling simple scenarios such as flat files in a project, as well as package hierarchies. The framework supports *dynamic dependencies* and *mutual dependencies* between compilation units. The type checkers can be written without the need to explicitly handle communication or synchronization between units.

This is achieved by providing type checkers with an API for name resolution based on scope graphs. Scope graphs are a language-independent formalism for name binding and name resolution, which has been shown to support a wide range of binding patterns, and has successfully been applied to implement type checkers [16, 31, 20]. The key to our approach is twofold:

- delay lookups when other units have not progressed enough to give a *safe*, that is, complete answer, and
- release delayed queries as soon as possible, even if other parts of the graph are still incomplete.

Recent work by Rouvoet et al. [20] identifies the absence of *weakly critical edges* as a sufficient condition to guarantee safe name resolution in a partial scope graph. We develop the notion of *scope state* to allow fine-grained tracking of the presence or absence of weakly critical edges. Through these scope states, which are managed by the type checkers via the name resolution API, the framework ensures safety of name resolution. The provided API is asynchronous, which works with type checkers that follow a synchronous pattern, where every name resolution query is awaited, as well as with type checkers that use dynamic scheduling techniques, such as worklists and continuations.

We claim the following contributions:

- We propose the notion of *scope state* to explicitly track the presence of weakly-critical edges (Section 3.3).

---

[1] The type checkers we envision are both concurrent (i.e., units make (interleaved) progress during the same period) and parallel (i.e., units run at the same time), and we use the terms interchangeably.

```
package p;                    package p;                package p;

class A {                     class B extends A {       class C {
    p.C f;                        C g;                      A h;
    static class C {}         }                         }
}
```

- We introduce a model of hierarchical compilation units with scope sharing (Section 4.1). We extend scope state with a notion of sharing, which allows us to track weakly-critical edges in the hierarchy of compilation units (Section 4.2).
- We present a scope graph-based name resolution API for use by type checker implementations (Section 4.3).
- We present an actor-based algorithm that implements the hierarchical compilation unit model and the name resolution API, and provides implicit parallel execution of the compilation units (Section 5).
- We present a fine-grained deadlock handling approach to ensure termination that is well-suited for interactive applications of the type checkers (Section 5).
- We show that our framework captures the scheduling behavior of Rouvoet et al. [20] by porting the Statix solver to our framework. We discuss local inference and the need for a specification style that models declarations and their types as separate scopes (Section 6). We parallelize all Statix type checkers, provided they follow this specification style.
- We benchmark the parallel Statix solver using a specification for a subset of Java on a few real world projects, showing speedups up to 5.0x on 8 cores for larger projects.

All the source code and benchmark results are available in the accompanying artifact.

2 Motivation and Scope

Our goal is to develop a framework that provides implicit parallelization of type checkers. In this section we discuss the features we want to support, the difficulties these features pose to parallelization, and an overview of our solution.

We use an example of a Java program consisting of three compilation units, shown in Figure 1, to illustrate the requirements on parallel type checkers. This example shows to two dependency patterns that are challenging for parallelization. The first is *mutual dependencies*. Class `A` refers to class `p.C` (qualified to distinguish it from the nested class it defines), while class `C` refers to class `A`. The second is *dynamic dependencies*. These are dependencies that are discovered during type checking, and that are not obvious without at least partially checking the program. The reference from `B` to `C` is an example of this. The name `C` could refer to the top-level class in the package, or to the nested class in `A`. To decide that it refers to the nested class in `A`, we need to resolve the reference to the super class `A` and its interface.

Typical compiler design (e.g., [2]) divides compilation into phases, including parsing, type checking, and code generation. We focus on the type checking phase, which is often difficult to parallelize because of the context dependence of type checking and name resolution. The type checking phase of the compiler for our example may consist of several steps: (a) build a symbol table containing information on defined classes and type inheritance, (b) build the interface for each class by processing field and method declarations, and (c) check the field

and method bodies of each class. Each step depends on the information collected in the previous phases.

What does it take to parallelize this type checker? Compilation units are checked in parallel, but inevitably need information provided by other compilation units. Mutual dependencies between compilation units prevents linear ordering of compilation units, while dynamic dependencies mean part of the work must be done before all dependencies are even known. This immediately rules out simple parallelization schemes based on a topological ordering of compilation units, where units are checked after their dependencies have finished.

The main challenge introduced by parallelization is therefore how to deal with partial information during type checking (which has been called the Doesn't Know Yet Problem [22]). For example, the compilation unit for class B does a lookup of nested classes in its super class A, while the compilation unit for A has not constructed its interface yet. Solving this problem may require designing locking schemes for threads with shared data, or messaging protocols for communicating processes, as well as keeping track of the completeness of (part of) the symbol table or interface. Designing concurrent software is notoriously hard, and bugs can result in deadlocks or invalid results because of the use of incomplete information.

Our solution to this problem is a framework that allows compiler engineers to write their type checker without concern for parallelization. All the work of coordinating the parallel units and keeping track of completeness of interfaces is handled by the framework. The key idea is that all dependencies between units are the result of either a hierarchy between units (e.g., compilation units in packages) or name[2] lookups. The framework provides the type checker with a name handling API based on the expressive name binding model of scope graphs. Scope graphs are language-independent, and have been successfully used to model a wide variety of binding patterns, including mutually recursive modules, type-dependent names, and generics. Type checkers use this API to declare the name binding and scoping structure, and resolve names by queries on the resulting scope graph. The scope graphs grows monotonically with the type checker marking the parts of the graph that are completed. In return, the framework completely hides the communication between different compilation units and ensures only complete information is used to answer name resolution queries.

In terms of our earlier example, this means that the type checker of each compilation unit follows the steps of the original, non-parallel, design. When the type checker for class B queries the not-yet-constructed interface of class A, the query is simply suspended until the unit of A has progressed enough to be able to answer the query. The type checker of unit of A, on the other hand, is unaware of the query as the delay and answer mechanism is handled transparently based on the monotone completion of A's scope graph.

3   Type Checking with Scope Graphs

Scope graphs [16] are a language-independent formalism to specify name binding and name resolution, and a key ingredient of our approach. In this section we explain scope graphs, describe the problem of safe name resolution and its solution using critical edges [20], and we introduce the notion of scope state to track the presence of critical edges.

---

[2] We use name in a broad sense, as it can be complex data and does not necessarily have to appear literally in the original program.

3.1   Scope Graphs

Scope graphs describe the name binding structure of programs as a directed graph of scopes with associated data, connected by labeled edges. Scopes correspond to regions in the program that behave uniformly with respect to name binding. Name resolution corresponds to queries in the graph that find paths from references to scopes with matching associated data (e.g., an identifier name).

The program in Figure 2 will be our running example. It consists of two files, one containing a class A in a package p, and a class B in a package q. Class B extends class A and refers to a field f in A from the method m. The scope graph corresponding to our example is shown in Figure 3.[3] The node labeled s_R represents the root scope of the program. The scopes s_p, s_q, s_A, s_B, s_f, and s_m correspond to declarations in the program, and each has the simple class name as associated data, depicted as $s \rightarrow x$. Edges from the containing scopes to these declarations are labeled to indicate the kind of declaration: PKG, CLS, FLD and MTHD for package, class, field, and method declarations respectively. The scopes $s_{T(f)}$ and $s_T(m)$ represent the types of the declarations f and m, and each is connected to their declaration with a TYPE-labeled edge. The label LEX is used for connecting a scope to its lexical parent.[4] Finally, class extension is modeled by an EXT-labeled edge between the two class scopes, which makes the declarations from the super-class reachable from the subclass.

Name resolution is expressed by means of queries over the scope graph, finding a path from a reference's scope to a matching declaration. Query parameters control *reachability* and *disambiguation*. What data in the graph is reachable is specified with a regular expression describing valid paths and a predicate describing matching data. For example, to resolve a reference f in the lexical context or in a super-class, one would use the regular expression LEX*EXT*FLD, and a predicate matching the name f. Disambiguation determines which data is visible if multiple reachable results are found, and is specified with an order on edge labels and a predicate describing equivalent data. For example, if we prefer local definitions of field references over definitions that traverse more edges, and definitions found in super-classes over definitions found in the lexical context, we would use a label order $\$ <$ EXT $<$ LEX. The special label $\$$, assumed different from all user-provided labels, is used for end-of-path.

We use the following notation: A scope graph $\mathcal{G}$ is a triple $\langle S, E, \rho \rangle$ of scope identifiers S, edges E, and a partial function ρ from scopes to associated data. The function $scopes(\mathcal{G})$, $edges(\mathcal{G})$, and $data(\mathcal{G})$ project the three components of the triple. Query parameters are a path well-formedness regular expression re, a data matching predicate $\mathbf{D}$, and a strict partial order on labels $\mathbf{L}$. We use $\mathbf{D_x}$ for the predicate matching the name x. The result of a query is an answer set A of tuples (p, d) of a path p and a datum term d. A path p is either a single scope s, or a labeled step $p \cdot l \cdot s$, and $target(p)$ projects the last scope of path.

---

[3] This scope graph is a simplification of the scope graph that would be necessary to support all Java's name resolution features, and does not account for the possibility of named and wildcard imports, implicit package visibility, nested classes, etc.

[4] We say binding is *lexical* if binders are only visible in sub-terms of the term where they are bound. Examples are lambda and let expressions. If the scope of the binder is wider, we say the binding is *non-lexical*. Examples are identifiers imported from modules, or references to members on expressions of a class/record type.

```
package p;                       package q;
public class A {                 public class B extends p.A {
    int f;                           public int m() {
}                                        return f;
                                     }
                                 }
```
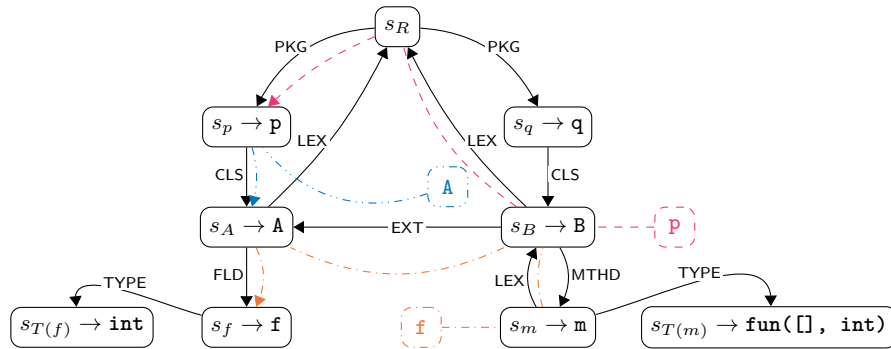
◼ **Figure 2** Example Java program with two compilation units.

◼ **Figure 3** Scope graph corresponding to the Java program in Figure 2.
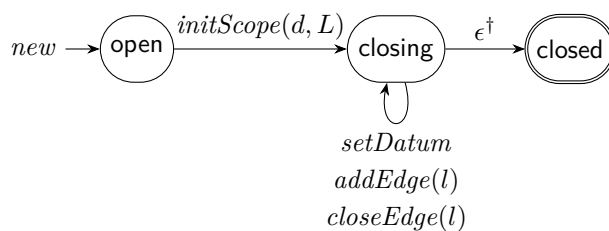
3.2   Critical Edges for Safe Name Resolution

Type checkers use the scope graph to resolve names, but must also construct the scope graph. When type checking starts, all we have is an empty scope graph, which is gradually built up as type checking progresses. Rouvoet et al. [20] observe that it is not always possible to construct the full scope graph without already querying it. This can be seen in our example in Figure 3 as well. The construction of the extension edge from s_B to s_A requires resolving the reference to A in a partial graph. This raises the question when it is safe to do so. After all, if that query was executed before the declaration of A is added to the graph, it results in an undeserved error. A query is considered *safe* to resolve when its result in the current partial scope graph is the same as its answer in the final scope graph. Rouvoet et al. [20] identify the absence of *critical edges* as the condition to guarantee safety. A critical edge is the first edge that is missing in the partial graph that will be part of the query result in the final graph. For example, if the graph in Figure 3 was complete except for the EXT edge between s_B and s_A, this edge is critical for the resolution of the reference f. But the same missing edge is not critical for the resolution of p.

Since determining the critical edges in an incomplete scope graph amounts to solving the whole name resolution problem, Rouvoet et al. [20] propose *weakly critical edges* as a conservative approximation of critical edges. Weakly critical edges are missing edges that *may* lead to a result for the query. In our earlier example, the missing EXT edge is weakly critical for the resolution of f even if s_A does not eventually contain a declaration for f.

3.3   Scope States

The final step to guarantee safe resolution is then to determine the weakly critical edges. The solution of Rouvoet et al. [20] is a predicate over a constraint set, which is specific to the Statix meta-language. The crucial property of their safety predicate is that the set of weakly critical edges only decreases as type checking progresses. They prove that this ensures

Figure 4 Scope states. Transition diagram.

$$
\begin{array}{rcl}
\{\top\} & new & \{O = \emptyset\} \\
\{\top\} & initScope(d, L) & \{O = L \uplus \{\$ \mid d = \top\}\} \\
\{\$ \in O\} & setDatum & \{\$ \notin O\} \\
\{l \in O\} & addEdge(l) & \{\top\} \\
\{l \in O\} & closeEdge(l) & \{l \notin O\} \\
\{O = \emptyset\} & \epsilon^{\dagger} & \{\top\}
\end{array}
$$

Figure 5 Scope states. Transition conditions and effects.

that once a query is executed, there can not be additions to the scope graph that lead to new results for that query. Our purpose is to develop a language-independent framework for parallel type checkers that correctly handles the dependencies between compilation units. Dependencies are the result of name resolution, thus handling the dependencies between units means ensuring name resolution between units is correct. If we can capture the presence of weakly critical edges independently of the particular type checker and object language, we can provide a general mechanism to delay queries until they are safe to execute.

To that purpose, we introduce the notion of *scope state*, which consists of a *state* (open, closing, and closed) and a set of *open labels* O, consisting of edge labels l or the special data label \$. Then, weakly critical edges are characterized by scopes that are open and/or have open edges. When a scope is closed, it is always safe to query, since its associated data and outgoing edges are final. When the scope is closing, queries over labels that are not weakly critical, i.e. that are not in O, are still safe to execute. The idea is that once the scope is closing, the set of open labels only decreases, and only labels in O are weakly critical.

Figure 4 shows the state transition diagram for scope states, and Figure 5 shows the pre- and post-conditions for the transitions. Initially, scopes are in the open state. In this state, the set of open labels has not been initialized, so all labels are considered weakly critical. Initialization with $initScope(d, L)$ changes the state to closing. The flag d specifies if the scope will have associated data. The set of edge labels L determines which labels outgoing edges from this scope may have. In the state closing, the set of open labels, and therefore the set of weakly critical edges, only decreases. The preconditions on $setDatum$ and $addEdge(l)$ guarantee that the shape of the scope with respect to a label l only changes when the label is in the set of open labels O. The associated data of a scope can only be set once, therefore $setDatum$ always removes the data label \$ from the set of open labels. Edge labels are closed with $closeEdge(l)$, which removes that label from the set of open labels, after which no new edges with that label are allowed, and the label is not weakly critical anymore. After all labels have been closed, the scope is complete and in the state closed.

4   Hierarchical Compilation Units

In this section we introduce a model of hierarchical compilation units. We extend scope states with a notion of sharing that is required by this model. Finally, we present the API of our framework, and code samples for the type checkers that may check our running example.

4.1   The Compilation Unit Model

We propose a model of hierarchical compilation units. The goal of this model is to be flexible enough to handle many different project structures. Examples of typical project structures that we support are:
- a flat set of compilation units for the source files in the project, each of which introduces global declarations that are accessible from other source files,
- a tree of compilation units, where intermediate nodes represent packages or modules, and the individual source files are the leaves, or
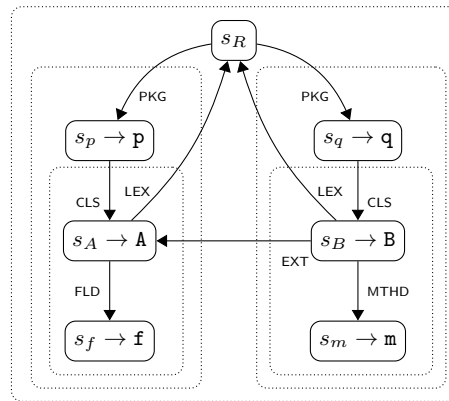- a project which depends on a library that is otherwise independent of the project.

Each compilation unit in our model has an associated, user defined, type checker. Compilation units can spawn sub-units, with their own associated type checkers. Each compilation unit builds a local scope graph by creating scopes, setting data, and adding edges. The compilation units are connected via scopes that are shared between units and their sub-units. Shared scopes allow sub-units to provide declarations that are reachable for other units, and to resolve to names in other units. The examples above all fit into this model. A project with a flat structure consists of one project unit, which creates a global scope that is shared with all file units, which add globally reachable declarations to the global scope. A project with hierarchical packages has compilation units for each package level, each with their own package scope, which is declared in the scope of the parent package. In a project consisting of a library and a program that depends on it, the program and the library have their own root scopes, and the dependency is reflected by an edge from the program root scope to the library root scope.

The compilation units for our Java example of Figure 2 follow the package hierarchy. Figure 6 shows how the scope graph of our example is distributed over compilation units. Our example program has five compilation units, which are depicted by the dashed boxes. At the top level, surrounding the whole scope graph, is the unit that represents the whole program. The packages p and q are sub units, each containing the units for the class in that package. The *owner* of a scope or edge in the graph is the unit that created the scope or edge. For example, the root scope s_R is owned by the root unit, while the class scope s_B is owned by the innermost unit for the class B. Visually, a scope is owned by the innermost unit that contains it, while an edge is owned by the innermost unit that contains the edge label. The MTHD edge is therefore owned by the unit of B, as is the LEX edge to s_R.
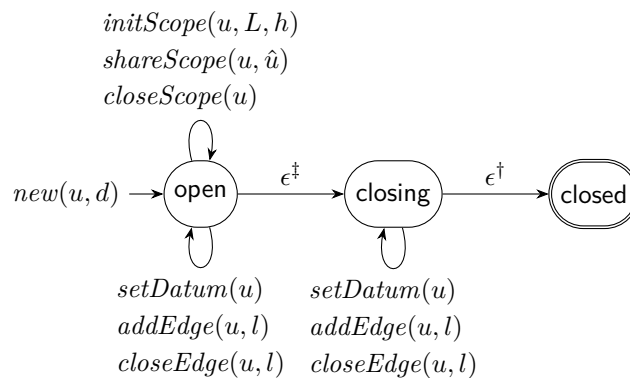
4.2   Safe Name Resolution with Sharing

The connection between units and sub-units is established through scopes that are shared from a unit to its sub-units. As a result, multiple units may contribute to a scope, something that the unit owning the scope must take into account when handling queries in that scope. Therefore, we extend scope state with sharing, to account for the fact that scope state is determined by the owner as well as by sub-units with which the scope was shared.

Sharing can result in outgoing edges having a different owner than their source scope, as units can contribute outgoing edges to either their own scopes, or scopes owned by one of

Figure 6 Compilation units for the Java program of Figure 2. The dashed boxes indicate the boundaries of the compilation units. A scope is owned by the innermost unit in which it appears. Edges are owned by the innermost unit that contains their label.

Figure 7 Scope states with sharing. Transition diagram.

their enclosing units that are *shared* with the unit. In our example, the CLS edge to s_B has source scope s_q, which is owned by the unit of package q, not by the unit of B. The data associated with scopes can only be provided by the owner.

The compilation unit that owns a scope is responsible for executing queries on that scope. Every unit maintains an aggregate view of each scope it owns, consisting of all the edges contributed by itself or by sub-units that the scope is shared with. To ensure safe name resolution in this model, we must account for sharing of scopes between units. When a unit initializes one of its scopes, it does not necessarily know what edges any of the sub-units may contribute. The sub-units must therefore initialize shared scopes as well, so that the scope owner has a complete picture of the state of the scope. In the previous section, a scope moved immediately to the closing state when it was initialized. When the scope is, or can be, shared, this is not correct. When a scope is not in state open, we expect that the set of open labels only decreases. The initialization of the scope by a sub-unit could increase the set of open labels. On top of that, if the scope is shared with a new sub-unit, this sub-unit must now also initialize the scope, potentially adding open labels. We can be sure that the set of open labels will only decrease, when all units that the scope is shared with have initialized it, and none of those units will share the scope with new sub-units.

To handle sharing correctly, we extend scope states with an explicit notion of sharing.

$$\{\top\} \qquad new(u,d) \qquad \begin{cases} I = \{u\}, \\ O = \{\$ \mid d = \top\}, \\ H = \emptyset \end{cases}$$

$$\begin{cases} u \in I, \\ O = O', \\ u \notin H \end{cases} initScope(u,L,h) \begin{cases} u \notin I, \\ O = O' \uplus \{(u,l) \mid l \in L\}, \\ u \in H \end{cases}$$

$$\begin{cases} u \in H, \\ \hat{u} \notin I \end{cases} shareScope(u,\hat{u}) \ \{\hat{u} \in I\}$$

$$\{u \in H\} \quad closeScope(u) \quad \{u \notin H\}$$

$$\{(u,\$) \in O\} \quad setDatum(u) \quad \{(u,\$) \notin O\}$$

$$\{(u,l) \in O\} \quad addEdge(u,l) \quad \{\top\}$$

$$\{(u,l) \in O\} \quad closeEdge(u,l) \quad \{(u,l) \notin O\}$$

$$\{I = \emptyset, H = \emptyset\} \qquad \epsilon^{\ddagger} \qquad \{\top\}$$

$$\{O = \emptyset\} \qquad \epsilon^{\dagger} \qquad \{\top\}$$

■ **Figure 8** Scope states with sharing. Transition conditions and effects.

The state diagram for scope states with sharing is shown in Figure 7, and the pre- and postconditions for the transitions in Figure 8. The extended scope state consists of a set of open labels per unit O, a set I of units that must initialize the scope, and a set H of units that may share the scope with new sub-units. All transitions take a parameter u that indicates which unit is responsible for the event. Creation of a scope is indicated by $new(u,d)$, where u is the owner and the flag d indicates whether the scope has associated data. The flag d is not part of *initScope* anymore, because only the owner can set data, but the scope is initialized by all units that the scope is shared with. When a scope is shared with a unit $\hat{u}$ with $shareScope(u,\hat{u})$, that unit is added to the set I. Every unit that the scope is shared with must initialize it with $initScope(u,L,h)$, after which it is removed from I. A unit initializes the scope with the set of open labels L, a well as the flag h to indicate that the unit may share the scope with sub-units. The scope moves to the state closing when the set of uninitialized units I and the set of sharing units H are both empty. When the state is not open anymore, the set of open labels will only decrease. The events $setDatum(u)$ indicates associated data is set, while $addEdge(u,s,l,s')$, and $closeEdge(u,s,l)$ indicate adding an edge and closing an edge label. The preconditions require that these events are only coming from units that have already initialized the scope. Note that these events are allowed in the states open *and* closing. This ensures that, if a scope is shared between multiple units, each unit can extend that scope without having to wait for all other units to initialize the scope first.

4.3   Name Resolution API

The key to support parallel execution of type checkers is to correctly handle the dependencies between compilation units, which result from name resolution. Queries into a unit that has not constructed the relevant part of its scope graph must be delayed, and executed whenever the scope graph is complete enough. Our framework hides this scheduling from type checkers, and thus provides implicit parallel execution. Type checkers are programmed

```
1  interface TypeChecker
2  │   function run(S)
3  end
4  interface CompilationUnit
5  │   function freshScope(d) : s
6  │   function addSubUnit(tc, S)
7  │   function initScope(s, L, h)
8  │   function closeScope(s)
9  │   function setDatum(s, d)
10 │   function addEdge(s, l, s')
11 │   function closeEdge(s, l)
12 │   async function query(s, re, D, L) : A
13 end
```

Algorithm 1 Type Checker and Name Resolution API.

against a name resolution API, shown in Algorithm 1, which contains methods to specify name binding by building a unit's scope graph, resolve names by querying the scope graph, and start sub-units.

Names are resolved with the $query(s, re, \mathbf{D}, \mathbf{L})$ function, which is defined as *async* to reflect the fact that queries cannot always be answered directly by other compilation units. It is up to the type checker to decide if the result should be immediately awaited, or if other work can be done until the answer is available. The framework ensures correct query answers by keeping track of scope states and scheduling queries based on these scope states. Type checkers are responsible for providing the framework with the necessary information to maintain the scope state. The type checker must therefore initialize the set of (locally) open labels and announce whether it may share the scope with sub-units, and it must close edge labels once all edges with that label are added. All the interaction with other units, such as forwarding queries to the right unit, delaying queries, and maintain scope state on sharing is completely hidden from the type checker. For example, the function $addSubUnit(tc, S)$, which starts a sub-unit with the given type checker tc and initial scopes S, takes care of recording the sharing of scopes, and starts the type checker to run in parallel. Type checkers specify what they do locally, the framework implicitly takes care of their parallel execution.

The pseudo code in Algorithm 2 shows how the API could be used to implement a type checker that checks the Java running example.[5] Each type checker is an actor that extends the *CompilationUnit* actor that provides the API, which is explained in detail in Section 5. At the top level is *JavaRootTC*, which takes no scope arguments, and creates the root scope of the project. Initialization specifies no open edge labels, but does allow sharing. For each package a new sub unit is started with a package type checker that takes the root scope as argument. The first is the root scope, which is passed down to the class scopes. After creating the sub units, the scope is closed, to indicate it will not be shared anymore. The package type checkers start by initializing the shared root scope, indicating the scope may be shared with sub units, and marking PKG as open to allow adding the package declaration. A new package scope s_p is created, with the package name as associated data. The root

---

[5] We show all API calls directly, to show how the API can be used. We imagine that in an actual type checker implementation, common patterns of usage would be abstracted away for nicer code.

```
 1  actor JavaRootTC(P)  extends TypeChecker
 2  │   function Run({})
 3  │   │   s_R := freshScope(⊥)
 4  │   │   initScope(s_R, ∅, ⊤)
 5  │   │   for each (p, C) ∈ P do
 6  │   │   │   addSubUnit(JavaPkgTC(p, C), {s_R})
 7  │   │   closeScope(s_R)
 8  │   end
 9  end
10  actor JavaPkgTC(⟦package x;⟧, C)  extends TypeChecker
11  │   function Run({s_R})
12  │   │   initScope(s_R, {PKG}, ⊤)
13  │   │   s_p := freshScope(⊤)
14  │   │   initScope(s_p, ∅, ⊤)
15  │   │   setDatum(s_p, x)
16  │   │   for each c ∈ C do
17  │   │   │   addSubUnit(JavaClassTC(c), {s_R, s_p})
18  │   │   closeScope(s_R)
19  │   │   closeScope(s_p)
20  │   │   addEdge(s_R, PKG, s_p)
21  │   │   closeEdge(s_R, PKG)
22  │   end
23  end
24  actor JavaClassTC(⟦class x extends y { ... }⟧)  extends TypeChecker
25  │   function Run({s_R, s_p})
26  │   │   initScope(s_R, ∅, ⊥)
27  │   │   initScope(s_p, {CLS}, ⊥)
28  │   │   s_c := freshScope(⊤)
29  │   │   initScope(s_c, {LEX, EXT, FLD, MTHD}, ⊥)
30  │   │   setDatum(s_c, x)
31  │   │   addEdge(s_c, LEX, s_R)
32  │   │   closeEdge(s_c, LEX)
33  │   │   addEdge(s_p, CLS, s_c)
34  │   │   closeEdge(s_p, CLS)
35  │   │   {(p, z)} := await query(s_c, LEX*CLS, D_x, ...)
36  │   │   s'_c := target(p)
37  │   │   addEdge(s_c, EXT, s'_c)
38  │   │   closeEdge(s_c, EXT)
39  │   │   // ... etc ...
40  │   end
41  end
```

■ **Algorithm 2** Sketch of a simplified type checker implementation for Java packages and classes. The type checker is defined as compilation units *JavaRootTC* for the project root, *JavaPkgTC* for packages, and *JavaClassTC* for classes. The presented code shows the construction and querying of package and class definitions.

$$
\begin{aligned}
msg \quad := \quad & Start(S) \mid InitScope(s, L, h) \mid ShareScope(s) \mid CloseScope(s) \\
\mid \quad & AddEdge(s, l, s') \mid CloseLabel(s, l) \mid Query(s, re, \mathbf{D}, \mathbf{L}) \\
\mid \quad & DeadlockQuery(u, m) \mid DeadlockReply(u, m, U) \mid Deadlocked(U) \\
token \quad := \quad & \mathsf{initScope}(s) \mid \mathsf{closeScope}(s) \mid \mathsf{closeLabel}(s, l) \mid \mathsf{answer}(f)
\end{aligned}
$$

Figure 9 Compilation Unit. Messages and wait-for tokens.

scope and package scope are shared with the sub units for the classes in the package, after which both scopes are closed. Finally, the package declaration is added to the root scope and the PKG label is closed.

Although not immediately evident in this small example, the fact that the API is fine-grained (e.g., separating closing a scope for sharing from closing an open edge label) allows greater flexibility in how the type checker is implemented than when a type checker would be responsible for aggregating all these events until one final event can be constructed.

The pseudo code for $JavaClassTC$ shows a pattern in which scope graph construction and querying are interleaved. The query for the super class is executed and the type checker waits for the result to be able to construct the EXT edge between the class scopes. It is important to realize that, because the framework ensures safe name resolution, this also introduces the possibility of *deadlock*. If, for example, the $JavaClassTC$ type checker would postpone $closeEdge(s_c, \mathsf{LEX})$ until after awaiting the query result, the query would get stuck on the still open edge label. It is therefore important to realize that type checker developers are still responsible for scheduling concerns that are part of any compiler implementation (concurrent or not), such as ensuring declarations are introduced before they are queried. The framework cannot solve these issues, as they are dependent on the specifics of the object language, but it ensures the local behavior is preserved when run in parallel. The Statix meta-language [20] provides implicit maintenance of scope state and flexible scheduling as part of the meta-language semantics, so that these concerns can be left implicit in a Statix type system specification. The case study in Section 6 shows that it is possible to implement a Statix solver on top of our framework, which gives the best of both worlds: implicit parallelism *and* implicit handling of scope state and scheduling.

5 Parallel Actor-based Algorithm

In this section we present an algorithm that implements the compilation unit model and API that were introduced in the previous section. First we introduce the actor model that forms the basis of our algorithm, then we discuss the three main aspects of the algorithm:

- maintaining the scope graph and scope states for owned and shared scopes,
- safely resolve queries on own scopes and delegate queries on other scopes, and
- detect deadlock between compilation units to ensure termination.

5.1 Compilation Unit Actors

The algorithm is written following the actor paradigm [1]. Actors are a concurrency model based on message passing. An actor has only local state, and communicates with other actors through messages. Actors are not internally concurrent, and they do not share state. This makes reasoning about concurrency easier with actors than with approaches based on shared state and explicit synchronization.

A compilation unit corresponds to a *CompilationUnit* actor, which definition and local state is shown in Algorithm 3. The members of a *CompilationUnit*, which are introduced in

1 **actor** *CompilationUnit()*
2 | **var:** scope graph $\mathcal{G}$
3 | **var:** counting wait-for graph *WFG*
4 | **var:** delays $\mathcal{Z} := \emptyset$
5 | **abstract function** *run(S)*

■ **Algorithm 3** Compilation Unit. Local actor state.

the following sections, are shown in Algorithms 4–6 and 8. The local state of compilation units consists of a scope graph $\mathcal{G}$, a counting wait-for graph *WFG*, and a set of delayed queries $\mathcal{Z}$. The messages that form the protocol between compilation units are listed in Figure 9. Type checkers are implemented by extending the actor and implementing the abstract *run* method.

Since there are many variations of the actor model, we give a quick overview of the features that we assume in the model:

- Actors are started using **start**, and form a hierarchy. The keywords **self** and **parent** refer to the current actor or its parent actor, respectively. Actor references can be sent to other actors.
- Actors implement **receive** members for all messages that they accept. Inside a message handler, the **sender** keyword refers to the sender of the current message. Messages are sent using **send** *actor, msg*. Messages that require a response are sent with **request** *actor, msg* and the response is sent from the message handler with **reply** *msg*. Messages from one actor to another are delivered in order, but delivery of messages from different actors is arbitrarily interleaved.
- Actors may implement auxiliary **function** members, which can only be invoked locally.

Some algorithms are presented in an asynchronous style, using futures. They use the following primitives:

- A **future** f represents a value that may be provided later. The value of a future is set by applying it, written as $f(v)$.
- Functions can be marked as **async** to indicate that they return a future. Inside asynchronous functions, the **await** keyword is used to await the results of futures.
- Awaited futures do not block the actor, but suspend the currently handled message and allow other messages to be processed by the same actor. A resumed computation (as a result of a reply or an applied future) always runs in the context of the actor that started it, and never concurrently with message handling or other resumed computations.

The message handlers and functions of the type checker API are implemented in a straightforward way. The handler for the message $AddEdge(s, l, s')$ calls $addEdge(\textbf{sender}, s, l, s')$, and the API function $addEdge(s, l, s')$ calls $addEdge(\textbf{self}, s, l, s')$.

5.2 Maintaining Scope Graph and Scope States

A compilation unit locally maintains its scope graph $\mathcal{G}$ and the states of the scopes it owns. This is done by the group of functions shown in Algorithm 4. These functions are called to handle API calls from the local type checker, or to handle messages received from other units. In the former case, the argument u equals **self**, in the latter u equals **sender**.
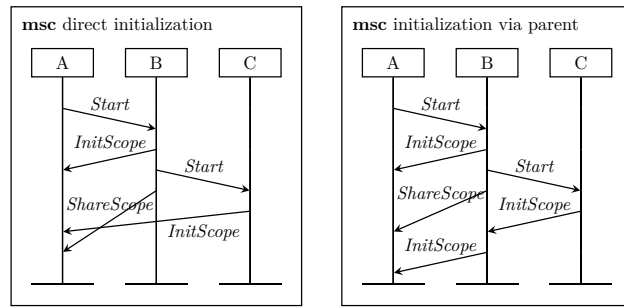
Scope state is maintained in a wait-for graph *WFG*, which consists of edges between units, labeled by a token indicating an expected action from the target unit. The tokens that may appear in the wait-for graph are listed in Figure 9. The state of the sets I, H,

```
 1  function start(S)
 2  │   G := G ⊎ ⟨S, ∅, ∅⟩
 3  │   foreach s ∈ S do  waitFor(self, initScope(s))
 4  │   run(S)
 5  end
 6  function freshScope(u, d)
 7  │   pick s fresh in scopes(G)
 8  │   G := G ⊎ ⟨{s}, ∅, ∅⟩
 9  │   waitFor(u, initScope(s))
10  │   if d = ⊤ then  waitFor(u, closeLabel(s, $))
11  │   return s
12  end
13  function initScope(u, s, L, h)
14  │   granted(u, initScope(s))
15  │   foreach l ∈ L do  waitFor(u, closeLabel(s, l))
16  │   foreach i ∈ 0 . . . h do  waitFor(u, closeScope(s))
17  │   if owner(s) = self then  tryReleaseScopeDelays(s)
18  │   else  send parent, InitScope(s, L, h)
19  end
20  function addSubUnit(u, û, S)
21  │   foreach s ∈ S do  shareScope(û, s)
22  │   start û
23  │   send û, Start(S)
24  end
25  function shareScope(u, s)
26  │   waitFor(u, initScope(s))
27  │   if owner(s) ≠ self then  send parent, ShareScope(s)
28  end
29  function closeScope(u, s)
30  │   granted(u, closeScope(s))
31  │   if owner(s) = self then  tryReleaseScopeDelays(s)
32  │   else  send parent, CloseScope(s)
33  end
34  function setDatum(u, s, d)
35  │   G := G ⊎ ⟨∅, ∅, {(s, d)}⟩
36  │   closeLabel(u, s, $)
37  end
38  function addEdge(u, s, l, s′)
39  │   G := G ⊎ ⟨∅, (s, l, s′), ∅⟩
40  │   if owner(s) ≠ self then  send parent, AddEdge(s, l, s′)
41  end
42  function closeLabel(u, s, l)
43  │   granted(u, closeLabel(s, l))
44  │   if owner(s) = self then  tryReleaseLabelDelays(s, l)
45  │   else if l ≠ $ then  send parent, CloseLabel(s, l)
46  end
```

■ **Algorithm 4** Compilation Unit. Scope graph.

Figure 10 Different initialization scenarios.

and O of the scope state is determined by the tokens in the wait-for graph. An initScope(s) edge to u implies $u \in I_s$. A closeScope(s) edge to u implies $u \in H_s$. A closeLabel(s, l) edge to u implies $(u, l) \in O_s$. The state of a scope s can be determined from the tokens in the wait-for graph. If the graph contains initScope(s) or closeScope(s) tokens, the scope is open. If the graph only contains closeLabel(s, l) tokens, the scope is closing. If there are no tokens concerning s, the scope is closed.

The functions in Algorithm 4 update the wait-for graph in correspondence with the postconditions of the scope state transitions. When an element is added to one of the sets of the scope state, an edge is added with *waitFor*(u, *token*). When an element is removed from one of the sets of the scope state, an edge is removed with *granted*(u, *token*). For example, when a fresh scope is created with *freshScope*, the function adds an initScope(s) token, corresponding to the postcondition $u \in I$ of *new*. When the scope is initialized with *initScope*(s, L, h), the initScope(s) is removed, corresponding to the postcondition $u \notin I$ of *initScope*.

The removal of tokens from the wait-for graph may result in changes to the weakly critical edges of a scope. Therefore, the functions *initScope*, *closeScope*, and *closeLabel* call one of the *tryRelease*∗ functions to trigger the release of queries that can now be executed safely.

Maintaining the scope graph and state locally is not enough for a shared scope s that is not owned by the current unit. In such cases, when owner(s) ≠ **self**, the event is propagated to the parent. Because scopes can only be shared with sub units, this means that the message eventually reaches the owner of that scope. The benefit of propagating the message via the parent instead of sending it to the owner of the scope directly has to do with message ordering. Messages coming from two different units are not meaningfully ordered. This can lead to messages arriving in unexpected order, as illustrated by the two scenarios in Figure 10. Without message ordering, a scenario where a top-level unit A, shares a scope with a sub-unit B, which in turn shares that scope with a sub-unit C, could result in A receiving the initialization of C before the message from B that the scope was shared. If the initialization goes via the parent B, then unit A always gets the *ShareScope* message before the corresponding initialization.

Receiving the messages in order simplifies maintenance of the wait-for graph and makes it easier to enforce correct usage of the API in the implementation. This is a simple solution to achieve that without the need for more complex message ordering mechanisms such as vector clocks. Sending messages about shared scopes via the parent is also the reason that the wait-for graph is a counting graph, that is, tokens may appear multiple times in the graph. To the unit A it looks as if the unit B has to initialize the shared scope twice, as it does not know about the unit C. All messages about sharing and initialization appear to

come from B.

5.3 Resolving Queries

```
 1 async function query(p, re, D, L)
 2     u := owner(target(p))
 3     if u = self then  return await getEnv(p, re, D, L)
 4     else
 5         f := request u, Query(p, re, D, L)
 6         waitFor(u, answer(f))
 7         A := await f
 8         granted(u, answer(f))
 9         return A
10 end
11 async function getEnv(p, re, D, L)
12     L := {l | L(∂_l re) ≠ ∅} ∪ {$ | ε ∈ L(re)}
13     return await getEnvForLabels(L, p, re, D, L)
14 end
15 async function getEnvForLabels(L, p, re, D, L)
16     K⃗ := ∅
17     L_max := {l | l ∈ L, ∄l′ ∈ L. L(l, l′)}
18     for each l ∈ L_max do
19         L′ := {l′ | l′ ∈ L. L(l′, l)}
20         K⃗ := K⃗ ∪ {getShadowedEnv(L′, l, p, re, D, L)}
21     A⃗ := awaitAll K⃗
22     return ⋃_{A ∈ A⃗} A
23 end
24 async function getShadowedEnv(L, l, p, re, D, L)
25     k_L := getEnvForLabels(L, p, re, D, L)
26     k_l := getEnvForLabel(l, p, re, D, L)
27     [A_L, A_l] := awaitAll [k_L, k_l]
28     return shadow(A_L, A_l)
29 end
30 async function getEnvForLabel(l, p, re, D, L)
31     if l = $ then
32         d := await getDatum(target(p))
33         return {a | a = (p, d), D(d)}
34     else
35         S := await getEdges(target(p), l)
36         P⃗ := {p′ | s′ ∈ S, p′ = p · l · s′, ∄p′. (p′ · l · s prefix of p)}
37         K⃗ := {query(p′, ∂_l re, D, L) | p′ ∈ P}
38         A⃗ := awaitAll K⃗
39         return ⋃_{A ∈ A⃗} A
40 end
41 function shadow(A_1, A_2)
42     return A_1 ∪ {(p_2, d_2) | (p_2, d_2) ∈ A_2, ∄p_1, d_1. ((p_1, d_1) ∈ A_1, d_1 ≈ d_2)}
43 end
```

Algorithm 5 Compilation Unit. Query resolution.

The name resolution algorithm, shown in Algorithm 5, implements a graph search that follows well-formed paths to matching declarations. It is a reformulation of the algorithm

presented by Van Antwerpen et al. [30]. The entry point is the function $query(p, re, \mathbf{D}, \mathbf{L})$, which returns the environment of paths starting with the prefix path p that matches the given query parameters. The search starts at the target scope of the current path. If the current scope is not owned by the current unit, the query is forwarded to the owner's compilation unit. Otherwise, the environment is computed locally by $getEnv(p, re, \mathbf{D}, \mathbf{L})$. That function determines the set of labels L that is relevant given the current path well-formedness regular expression. Edge labels l are relevant if the Brzozowski derivative [3] does not result in the empty language. If the current regular expression is accepting, that is, its language contains the empty string ϵ, the current scope may be an end-point, and the data label is also relevant. The functions $getEnvForLabels$ and $getShadowedEnv$ together ensure that the environment implements the label order specified for disambiguation, by ensuring results from more specific labels shadow results from the least specific labels. For example, if the current set of labels $L = \{\$, \mathsf{FLD}, \mathsf{LEX}, \mathsf{EXT}\}$, and the label order is $\$ < \mathsf{FLD} < \mathsf{EXT} < \mathsf{LEX}$, then the resulting environment is

$$shadow(shadow(shadow(A_\$, A_{\mathsf{FLD}}), A_{\mathsf{EXT}}), A_{\mathsf{LEX}})$$

A_l is the answer set for the label l, and $shadow$ is the function that removes answers from the right-hand set if its datum matches any answer in the left-hand set. The environment for a single label l is computed by $getEnvForLabel(l, p, re, \mathbf{D}, \mathbf{L})$. If the label is the data label $\$$, $getDatum$ is called in the current scope to construct an answer (p, d). Otherwise, $getEdges$ is used to return the target scopes of all outgoing l-labeled edges, cyclic paths are filtered out to ensure search termination, and environments are resolved for each new prefix path p' with the updated path well-formedness. The result is the union of all resulting environments. The updated set of parameters is itself a valid, residual, query, which allows us to simply call the top-level $query$ function, which takes care of delegating the query to the right compilation unit.

Compilation units must also ensure that name resolution is safe. When edges or data are requested for which the label is weakly critical, the answer is delayed. When a label's status changes, pending delays are released. The functions *isScopeOpen* and *isEdgeOpen* implement the check for weakly critical edges based on the wait-for graph, as explained in Section 5.2. The functions *getEdges* and *getDatum* decide based on the result of *isEdgeOpen* whether the scope graph can be used, or if it has to delay the answer. If the label is critical, a new future is created, which is stored, together with the scope and label, in the set of delays $\mathcal{Z}$. The functions *tryReleaseScopeDelays* and *tryReleaseLabelDelays* are called whenever the scope state changes, and return the results for any label that is not critical anymore by applying the stored future.

5.4 Handling Deadlock

The type checkers implemented with our framework can deadlock for various reasons. The type checker may contain obvious bugs, such as querying a scope before it is properly closed. But many subtle situations can occur as well, if ill-bound or ill-typed input programs cause scope graph construction to get stuck, even if no deadlocks can occur on well-typed inputs.

Whatever the reason, it is important for the user experience to ensure termination of the type checking process and the possibility of graceful handling of deadlocks by the type checker. Our goal is a fine-grained approach where deadlock is handled by failing individual queries that contribute to the deadlock, and only fail whole units as a last resort. Being fine-grained is especially important in interactive settings, when a type checker is employed as part of an IDE. Failing the type checker without returning a result because of an ill-typed

```
1  function waitFor(u, token)
2  │    WFG := WFG ∪ {(u, token)}
3  end
4  function granted(u, t)
5  │    WFG := WFG − {(u, token)}
6  end
7  function isWaitingFor(u, t)
8  │    return (self, t, u) ∈ WFG
9  end
10 function isScopeOpen(s)
11 │    return ∃u. (isWaitingFor(u, initScope(s)) ∨ isWaitingFor(u, closeScope(s)))
12 end
13 function isEdgeOpen(s, l)
14 │    return isScopeOpen(s) ∨ ∃u. isWaitingFor(u, closeLabel(s, l))
15 end
16 async function getDatum(s)
17 │    if isEdgeOpen(s, $) then
18 │    │    future k
19 │    │    Z := Z ∪ {(s, $, k)}
20 │    │    return await k
21 │    else return data(G)(s)
22 end
23 async function getEdges(s, l)
24 │    if isEdgeOpen(s, l) then
25 │    │    future k
26 │    │    Z := Z ∪ {(s, l, k)}
27 │    │    return await k
28 │    else return {e | e ∈ edges(G), ∃s′. e = (s, l, s′)}
29 end
30 function tryReleaseScopeDelays(s)
31 │    if isScopeOpen(s) then return
32 │    foreach {l | ∃k. (s, l, k) ∈ Z} do tryReleaseLabelDelays(s, l)
33 │
34 end
35 function tryReleaseLabelDelays(s, l)
36 │    if isEdgeOpen(s, l) then return
37 │    for each {k | ((s, l), k) ∈ Z} do
38 │    │    Z := Z − {(s, l, k)}
39 │    │    if l = $ then k(data(G)(s))
40 │    │    else k({e | e ∈ edges(G), ∃s′. e = (s, l, s′)})
41 end
```

🟨 **Algorithm 6** Compilation Unit. Delays and wait-for graph maintenance.

```
1  actor JavaClassTC(⟦class x extends y { ... }⟧) extends CompilationUnit
2     function Run({s_R, s_p})
3        // ...
4        s_c := freshScope(⊤)
5        addEdge(s_p, CLS, s_c)
6        A := await query(s_c, LEX*CLS, D_x, ...)
7        closeEdge(s_p, CLS)
8        // ...
9     end
10  end
```

◼ **Algorithm 7** Java Type Checker with Incorrect Internal Scheduling

```
package p;              package p;
class A {}              class B extends A {}
```

◼ **Figure 11** Example program that deadlocks with the buggy type checker from Algorithm 7.

input program completely negates the usefulness of the type checker to the programmer in helping them fix their program.

A deadlock occurs when a group of units waits on each other without any unit being able to make progress without receiving a message from one of the other units. We illustrate this using a faulty version of our Java type checker example, shown in Algorithm 7. In this implementation, the super class is resolved before closing the CLS label after the class declaration is added. The program causing deadlock, shown in Figure 11, consists of a class A and a class B that extends A, both defined in a package p. The two class definitions are checked by their own units A and B, who declare the classes in the scope s_p that is shared with them by the package unit p. Unit B tries to resolve the class A before closing the CLS edge on the shared scope s_p, and the query gets delayed on that edge by unit p. Now the units are in deadlock, since p is waiting for B to close the edge, while B is waiting for an answer from p. We can visualize the dependencies between the units by combining the wait-for graphs *WFG* of all units, as shown in Figure 12. We see that deadlock in the graph from the knot[6] of units that cannot make progress.

To understand how we can handle such deadlocks in a fine-grained way, we must understand the shapes these graphs can have. The key insight is that deadlocks involving more than one unit *always* involve a query. If we do not consider queries, the structure of the wait-for graph is always a tree. Units only wait for initScope, closeScope, and closeLabel on themselves or direct sub-units. It is waiting on answers that breaks the tree structure. Therefore, a knot between different units can only exist if at least one query is involved. Our approach handles deadlocks by failing involved queries whenever possible. These failures become exceptions in the type checker, which can be handled if desired. If a deadlock does not involve any queries, and thus involves only a single unit, the whole unit is failed and any remaining open scopes and labels are closed.

The functions for deadlock handling are shown in Algorithm 8. Deadlock detection is implemented using the distributed communication deadlock detection algorithm of Chandy et al. [4],

---

[6] In a directed graph, a knot is a set of nodes in the graph such that each node can reach all other nodes in the set. Communication deadlocks are characterized by knots, while resource deadlocks are characterized by cycles.
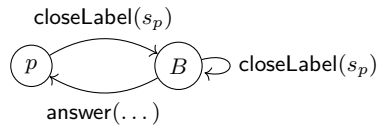
**Figure 12** Wait-for graph for the deadlocked example in Figure 11.

```
1  function deadlocked(U)
2      if |U| = 1 then
3          if failDelays(U) = false then  failAll()
4      else
5          failDelays(U)
6  end
7  function failDelays(U)
8      Z := {f | (u, answer(f)) ∈ WFG, u ∈ U}
9      foreach Z do  f(⊥)
10     return Z ≠ ∅
11 end
12 function failAll()
13     for each {t | (u, t) ∈ WFG} do
14         granted(self, t)
15         switch t do
16             case initScope(s) do
17                 if owner(s) ≠ self then  send parent, InitScope(s, ∅, false)
18                 tryReleaseScopeDelays(s)
19             case closeScope(s) do
20                 if owner(s) ≠ self then  send parent, CloseScope(s)
21                 tryReleaseScopeDelays(s)
22             case closeLabel(s, l) do
23                 if owner(s) ≠ self then  send parent, CloseLabel(s, l)
24                 tryReleaseLabelDelays(s, l)
25         end
26 end
```

Algorithm 8 Compilation Unit. Deadlock handling.

modified so that it collects all units involved in a deadlock. When a deadlock is detected, the *deadlocked* function is called on all units involved, receiving the set U of involved units an argument. In the case that the set U is a singleton, and the deadlock is local, failing queries is attempted by *failDelays*, and, if unsuccessful, the unit is failed with *failAll*. The function *failDelays* finds all unanswered queries to units in the deadlock and raises an exception locally (indicated by the application of the future with $\perp$). The function *failAll* closes any remaining open scopes and labels and informs the parent if appropriate. At this point the type checker of the failed unit is never invoked anymore, but the unit itself can still resolve queries for other units and participate in deadlock detection. In the case that the set U is not a singleton, the *failDelays* function is used to fail any queries on other units in the deadlock. We explicitly prevent falling back to *failAll* in non-singleton deadlocks because not every unit has queries it can fail. Failing such a unit in such cases would be unnecessary, as some other units in the deadlock can fail queries and resume type checking.
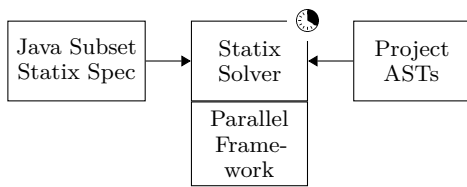
Project	#Files	LOC
commons-csv 1.7	12	1845
commons-io 2.6	118	9984
commons-lang3 3.11	210	29642
single-unit-clusters-call	100	~32K

Figure 13 Benchmark setup

6   Evaluation

We evaluated our approach by porting an existing scope graph-based type checker for a subset of Java to our framework, and measuring speedup resulting from using multiple cores when analyzing Java projects. The diagram in Figure 13 summarizes the setup of the benchmark. The benchmark executable, source code, and data of our experiments can be found in the artifact that accompanies this paper.
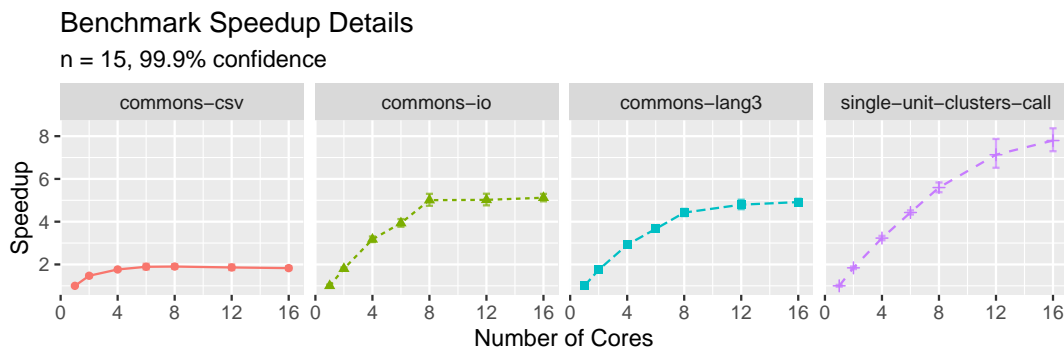
6.1   Benchmark

We implemented the type checker by porting the solver of the Statix meta-language to our framework. Adapting the Statix solver was an attractive case study, because it is a mature project that already uses scope graphs for name resolution. The Statix solver uses dynamic scheduling for constraint solving, where constraints are delayed on logical variable instantiation, and was thus a good test case to show that the API provided by the framework is flexible enough to cope with such dynamic scheduling. Therefore, we expect that type checkers in many different scheduling styles can be implemented with our framework, which we plan to explore in future work.

The type checker used a Statix specification for a subset of Java based on an existing MiniStatix specification [20]. This specification focuses on name binding aspects of Java, and implements packages, top-level and nested type definitions, type inheritance. Overloading is partly supported, while generics and lambda expressions are not supported.

We used three existing Java projects (`commons-{csv,io,lang3}`) and one generated Java project for the benchmark (`single-unit-clusters-call`). The existing Java projects are projects from the Apache Commons project that have no dependencies besides the Java standard library (JRE). The projects have different sizes, which allows us to asses the impact of project size on potential speedup. The generated project serves as a baseline for what is achievable with our parallel Statix implementation. It consists of a 100 classes, each class in its own package. The classes contain a number member methods, and each method body consists of method calls to other members of the same class. Because the classes are isolated and do not depend on other classes, the resulting compilation units only interact with the package's compilation unit and the unit for the Java standard library, and represent an ideal scenario in terms of parallelization. All projects and project sizes are listed in Figure 13.

Early experiments showed that the JRE, which is also treated as a compilation unit, often became the critical unit if it was served by a single actor. To eliminate this effect, the JRE is hosted on as many actors as the number of used cores, using round-robin scheduling to distribute queries over the actors. This is possible because the scope graph for the JRE is precomputed and statically loaded at the start of type checking.

We ran our type checker on each of these projects using an increasing number of cores. The benchmarks were executed using the JMH benchmark tool [17] in single-shot mode (the

Benchmark Speedup Details

Figure 14 Benchmark results for type checking Java projects. Each subplot shows the speedup, relative to single-core speed, versus the number of used cores for each project. The benchmark was executed with 15 sample iterations, and the error bars represent a 99.9% confidence interval.

analysis was run once per iteration) using 5 warm-up and 15 measurement iterations. The benchmarks were executed on a Linux system with 128 AMD EPYC 7502 32-Core Processors 1.5GHz and 256GB RAM.

The results, shown in Figure 14, show the speedup of the parallel type checker, relative to the single core case, for the number of used cores. The error bars indicate the 99.9% confidence interval.

First, we see that the generated baseline project scales up to 5.6x for 8 cores. The scaling slows down more cores are used, but keeps increasing to ~7.8x for 16 cores.

Second, we see that the other projects all have a cut-off point after which adding more cores does not result in much speedup. The cut-offs are approximately at 4 cores for `commons-csv`, the smallest of the three, with a speedup of 1.8x, at 8 cores for `commons-io`, with a speedup of 5.0x, and at 8 cores for `commons-lang3`, with a speedup of 4.42x.

The cut-off in scaling can be explained by looking at the run time of individual compilation units. All projects contain a few source files that are significantly larger than most others. The cut-off happens when the run time of the whole problem is dominated by the run time of the largest compilation unit. If we look at the speedups discussed before, the run time of the longest-running unit as a percentage of the total run time was 84% for `commons-csv`, 100% for `commons-io`, and 81% for `commons-lang3`. Understanding why scaling slows down for some projects before the longest-running unit *completely* dominates the run time is an interesting question for future research.

These results suggest that our approach can give significant speedups for the Statix type checker. How well the approach scales depends on the type checker implementation as well as the granularity of parallelism. Our choice to parallelize on files means that the distribution of file sizes is important for the speedup that can be achieved. A type checker that supports more fine-grained parallelization (e.g., on method bodies), could possibly scale further. Our framework does not require file granularity and supports more fine-grained parallelism. Thus, users can experiment with the granularity that works well for their target language.

Note that these results are for a single type checker and for a single programming language. Both the implementation of the type checker and the design of the language may influence the possibility for effective parallel execution. The relation between language design, the resulting dependency patterns between compilation units, and the opportunity for parallelization is an interesting topic for future research. Our framework enables such experiments with parallel type checkers, by taking the hard parts of parallelization away from the compiler writer.

6.2 Supporting Local Inference

The Statix solver uses unification, and often relies on unification variables in scope graph data to be able to do inference. This posed a challenge when porting Statix to our framework. Our framework operates under the assumption that compilation units only communicate via the scope graph. This means the unifier of one compilation unit is not accessible to other compilation units. While the owning compilation unit can interpret that data relative to the local unifier, other units can not. We have a situation where a unit requires an incomplete view of its own data, but other units should only ever get the complete data.

We added a small extension to the framework to support such local inference patterns. Type checkers can define a function that produces a representation of data that is fit for other units:

$$\textbf{async function } \textit{GetExternalRepresentation(d)}$$

The function is asynchronous so the type checker can delay returning the external representation until unification variables are instantiated. It is applied to the data of any scope whose owner is not the unit that issued the query. This solution allows units to do local type inference via the scope graph, while still presenting complete data to other units.

In the Statix literature, different patterns are used to associate declarations with types. In the first, the declaration and type are combined as a tuple $(x : T)$, and stored as the data in a single scope [31]. In the second, the declaration only carries the name as data, and the type is represented as the data of a separate scope connected to the declaration by an edge [20]. We observed that the first encoding quickly results in deadlock if name resolution queries (resolve x) are necessary to instantiate the types T: The external representation of the whole tuple gets stuck on the logical variables in the type, therefore blocking the query for the name. The representation using tuples can easily be converted into the latter, but is a necessary consequence of the isolated nature of the compilation units in our approach.

7 Related Work

In this section we discuss related work on parallel approaches to build systems, compilers, and program models used for compiler and static analysis implementation.

7.1 Parallel Compilers

Parallel compilers are certainly not a given, even for often used languages, but there are several languages for which parallel compilers (mature or research prototypes) exist. These compilers are all for specific languages, but it is interesting discuss the techniques they use or the performance results they achieve. Although it is hard to find reliable information on the parallel capabilities of compilers, online discussion in StackExchange suggest that compilers for at least Java, C/C++, and C#, all often used, are all single-threaded [24, 25, 26]. The concurrent compiler for Active Oberon [18] implements ideas that are similar to ours. Scopes (following the program nesting structure) have an associated state describing whether all symbols in the scope have been defined, and queries are delayed if scope information is incomplete. The supported scoping structure is specific for the target language and deadlock is avoided by being careful about what queries are done in what compilation phase. The implementation uses a shared data structure for the symbol table with a global lock, which is different from our approach of a distributed scope graph and units communicating by messages only. Hydra [29] is a commercial parallel compiler for Scala, which parallelizes the

Scala compiler by running the many phases of the Scala compiler in parallel. Hydra publishes benchmark results and reports speedups between 1.8–3.5x, depending on the project, on 4 cores [29]. Work has been done to parallelize the Rust compiler [21]. The approach is focused in parallelizing loops in the compiler, while maintaining most of the current structure of the compiler. However, at the time of writing the documentation mentions that "work on explicitly parallelizing the compiler has stalled. There is a lot of design and correctness work that needs to be done." The Go compiler supports parallel compilation at certain levels of the program [7]. Particularly, the compilation of functions inside a package is executed in parallel. Finally, the Swift compiler takes an interesting approach to achieving parallel build [28]. Every compilation task has a focus, the compilation unit it "really" needs to compile. In the process it also compiles other units, but only as much as necessary for the focus unit. They claim this generally works well, because the necessary work on other units is limited.

7.2   Parallel Build Systems

Our framework shares many characteristics with build systems, as they run and order compilation tasks based on a dependency graph. The well known build tool Make [27] executes build tasks based on a statically known acyclic dependency graph. When the object language allows separate compilation, it can run these tasks in parallel as well. Many other build tools follow the model of Make, and require the dependencies to be acyclic and known a-priori. Some build tools such as Pluto [5] and PIE [10] improve on this model by supporting dynamic dependency discovery. The resulting dependency graph is still required to be acyclic. What all these have in common is that the build tasks are all treated as atomic operations, producing outputs from inputs. The build system is concerned with ordering these tasks correctly. This is in contrast with our approach, which makes partial results of a unit available to other units before it is completely finished. This allows us to support not only dynamic dependencies, but also cycles in the dependency graph, something that build systems cannot handle.

7.3   Parallel Programming Models

Another approach is to write the compiler in a programming model that supports parallel execution, and the parallelization is not organized around compilation units anymore.

The JastAdd framework for reference attribute grammars supports implicitly parallel attribute evaluation [34]. The resulting concurrency is more fine-grained than in our approach, and not necessarily driven by dependencies between compilation units. If one writes a compiler using reference attribute grammars, this is a convenient way to parallelize the compiler. Compared to our approach, reference attribute grammars do not provide a ready to use model for name binding. This means it falls on the developer to come up with suitable representations and algorithms for the object language. Applying parallel attribute evaluation to the ExtendJ Java compiler resulted in speedups of 1.52–2.43x on 4 cores. Although their evaluation was done on a different set of Java projects, these results suggest that the performance of our approach is competitive.

LVish [13] proposes a parallel programming model based on monotonically growing data and freezing variables that reached a final state to achieve quasi-deterministic parallelism. It has been successfully applied to parallel type inference [15]. This model is very similar to how we handle scope states: closing scopes and edges corresponds to freezing. The difference is that LVish is only a model for monotone state, which leaves users to build parallelization

around it. The scope state model is specialized for our purpose, which allows us to make the parallelization and deadlock handling implicit for the user.

The theorem prover Isabelle/PIDE has strong support for implicit parallelization of proof checking [32, 14, 33]. The granularity is much smaller than in our approach. They do not support cyclic dependencies between parallel task, but a high degree of parallelism is achieved by exploiting proof irrelevance: most dependencies are only on the level of the theorem statements, but not their proofs. They report speedups up to 5.2–6.4x on 8 cores [33].

Several parallel programming models have been developed targeting static analyses. Because of their focus, these approaches target certain kinds of computations that are commonly used in static analyses, such as fixed points over lattices [8], parallel iteration over sets [12], or established algorithms such as IDFS [19].

7.4   Scope Graphs

Scope graphs [16] were introduced as a language-independent model of name binding with a focus on expressive, non-lexical, binding patterns, formalizing and generalizing the semantics of NaBL [11]. This model was subsequently extended and used to develop formalisms for the specification of type checkers [30, 31], resulting in the meta-language Statix. Followup work [20] defined a formal, non-parallel, operational semantics for Statix, and proved it correct. It introduced the notion of *critical edges* as a tool to reason about query answer correctness in evolving scope graphs. Critical edges were defined in terms of the constructs of the Statix language and the presented operational semantics. In this paper we introduce *scope state* as an explicit and application independent description of the state and transitions of a scope in a evolving scope graphs, which was only implicitly present in the Statix operational semantics. Porting Statix to the parallel framework of this paper required reformulating the safety conditions of the original operational semantics to explicit scope state operations. All this work has been developed and applied in the context of the Spoofax language workbench [9].

8   Conclusion

In this paper we have introduced a framework for the implementation of implicitly parallel type checkers. We have introduced the concept of scope state to make the notion of weakly critical edges in evolving scope graphs explicit. We have presented a case study and shown that the approach does result in speedups for the larger projects in our benchmark. For all real-world projects in the benchmark the scaling was limited by a few large files, which suggest that more fine-grained parallelism (e.g., checking method bodies in parallel) could improve parallelism for this Java type checker. In general, investigating the relation between the type checker implementation/Statix specification and the achievable parallelization for different target languages is interesting follow-up research. Other interesting directions for future research are (a) extending this work to *incremental* type checking of large software projects during development, (b) developing useful abstractions for managing scope state that sit between the fine-grained API of this paper, where scope state is completely explicit, and the high-level abstraction offered by the Statix meta-language, where scope state and evaluation order are completely implicit, and (c) investigating how this work can be extended to and/or integrated with other compiler tasks such as parsing, and code generation to create a fully parallelized compiler pipeline.

─────── **References** ───────

1 Gul A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

2 Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

3 Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

4 K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, 1(2):144–156, 1983. `doi:10.1145/357360.357365`.

5 Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 89–106. ACM, 2015. `doi:10.1145/2814270.2814316`.

6 GCC. The parallel gcc. URL: `https://gcc.gnu.org/wiki/ParallelGcc`.

7 Go. Go 1.9: Parallel compilation. URL: `https://golang.org/doc/go1.9#parallel-compile`.

8 Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. A programming model for semi-implicit parallelization of static analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 428–439, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3395363.3397367`.

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

10 Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Scalable incremental building with dynamic task dependencies. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 76–86. ACM, 2018. `doi:10.1145/3238147.3238196`.

11 Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. `doi:10.1007/978-3-642-36089-3_18`.

12 Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 211–222. ACM, 2007. `doi:10.1145/1250734.1250759`.

13 Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 257–270. ACM, 2014. `doi:10.1145/2535838.2535842`.

14 David C. J. Matthews and Makarius Wenzel. Efficient parallel programming in poly/ml and isabelle/ml. In Leaf Petersen and Enrico Pontelli, editors, *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*, pages 53–62. ACM, 2010. `doi:10.1145/1708046.1708058`.

15 Ryan R. Newton, Ömer S. Agacan, Peter P. Fogg, and Sam Tobin-Hochstadt. Parallel type-checking with haskell using saturating lvars and stream generators. In Rafael Asenjo 0001 and

Tim Harris, editors, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, page 6. ACM, 2016. `doi:10.1145/2851141.2851142`.

16   Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *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*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. `doi:10.1007/978-3-662-46669-8_9`.

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

18   Patrik Reali. Structuring a compiler with active objects. In Jürg Gutknecht and Wolfgang Weck, editors, *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2000, Zurich, Switzerland, September 6-8, 2000, Proceedings*, volume 1897 of *Lecture Notes in Computer Science*, pages 250–262. Springer, 2000.

19   Jonathan Rodriguez and Ondrej Lhoták. Actor-based parallel dataflow analysis. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6601 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2011. `doi:10.1007/978-3-642-19861-8_11`.

20   Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 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. `doi:10.1145/3428248`.

21   Rust. Parallel compilation. URL: `https://rustc-dev-guide.rust-lang.org/parallel-rustc.html`.

22   V. Seshadri, David B. Wortman, Michael D. Junkin, S. Weber, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In *PLDI*, pages 233–240, 1988.

23   Zhong Shao and Andrew W. Appel. Smartest recompilation. In *POPL*, pages 439–450, 1993.

24   StackExchange. Do compilers utilize multithreading for faster compile times? URL: `https://softwareengineering.stackexchange.com/questions/322494/do-compilers-utilize-multithreading-for-faster-compile-times`.

25   StackExchange. Is there something that prevents a multithreaded c# compiler implementation? URL: `https://softwareengineering.stackexchange.com/questions/330026/is-there-something-that-prevents-a-multithreaded-c-compiler-implementation`.

26   StackExchange. Why isn't javac running on multiple cores? URL: `https://stackoverflow.com/questions/46461757/why-isnt-javac-running-on-multiple-cores`.

27   Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make*. Free Software Foundation, May 2016.

28   Swift. Swift compiler performance. URL: `https://github.com/apple/swift/blob/master/docs/CompilerPerformance.md`.

29   Triplequote. Hydra: The parallel scala compiler. URL: `https://triplequote.com/hydra/compilation/`.

30   Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. `doi:10.1145/2847538.2847543`.

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

32 Makarius Wenzel. Parallel Proof Checking in Isabelle/Isar. In *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS '09)*, page 9, New York, NY, USA, 2009. Association for Computing Machinery.

33 Makarius Wenzel. Shared-memory multiprocessing for interactive theorem proving. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 418–434. Springer, 2013. `doi: 10.1007/978-3-642-39634-2_30`.

34 Jesper Öqvist and Görel Hedin. Concurrent circular reference attribute grammars. In Benoît Combemale, Marjan Mernik, and Bernhard Rumpe, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 151–162. ACM, 2017. `doi:10.1145/3136014.3136032`.