Stack Graphs

Name Resolution at Scale

Douglas A. Creager
GitHub, Southborough, MA, USA

Hendrik van Antwerpen
GitHub, Amsterdam, The Netherlands

Abstract

We present *stack graphs*, an extension of Visser et al.'s scope graphs framework. Stack graphs power Precise Code Navigation at GitHub, allowing users to navigate name binding references both within and across repositories. Like scope graphs, stack graphs encode the name binding information about a program in a graph structure, in which paths represent valid name bindings. Resolving a reference to its definition is then implemented with a simple path-finding search.

GitHub hosts millions of repositories, containing petabytes of total code, implemented in hundreds of different programming languages, and receiving thousands of pushes per minute. To support this scale, we ensure that the graph construction and path-finding judgments are *file-incremental*: for each source file, we create an isolated subgraph without any knowledge of, or visibility into, any other file in the program. This lets us eliminate the storage and compute costs of reanalyzing file versions that we have already seen. Since most commits change a small fraction of the files in a repository, this greatly amortizes the operational costs of indexing large, frequently changed repositories over time. To handle type-directed name lookups (which require "pausing" the current lookup to resolve another name), our name resolution algorithm maintains a stack of the currently paused (but still pending) lookups. Stack graphs can be constructed via a purely syntactic analysis of the program's source code, using a new declarative *graph construction language*. This means that we can extract name binding information for every repository without any per-package configuration, and without having to invoke an arbitrary, untrusted, package-specific build process.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Software libraries and repositories

Keywords and phrases Scope graphs, name binding, code navigation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.8

Supplementary Material *Software*: `https://github.com/github/stack-graphs/` [3]
archived at `swh:1:rel:06b63a06504d38c7a0e9ee03b1d21f6f3adc045a`
Software: `https://github.com/tree-sitter/tree-sitter-graph/` [16]
archived at `swh:1:rel:6877dac0c0f228f6450a2a2a6fdae178947e60ef`

1 Introduction

Code editors have long provided productivity features like *code navigation*, which let the user see and navigate the structural relationships in their code – specifically which entities the names in their code refer to. These features are equally useful on a software forge like GitHub. However, large software forges must support a completely different magnitude of scale, along a number of axes: the total volume of stored code; the number of changes received over time; the need to query historic versions of the code; the number of users making queries simultaneously; and the number of programming languages to support.

We have developed *stack graphs* to address these constraints. Stack graphs build on Visser et al.'s *scope graphs* framework [17], which use a graphical notation to encode the name binding rules for a programming language. Like scope graphs, each name binding in a program is represented by a path in the corresponding graph. Unlike scope graphs, our path-finding algorithm maintains a *stack* of pending queries (hence the name), allowing a

name binding to depend on the results of other, intermediate name bindings. Stack graphs are *file-incremental*, meaning that each file in the source program is represented by a disjoint subgraph, with no edges crossing between them. (Special *virtual edges* are created at query time to cross between files.) We can construct each file's subgraph in isolation, without inspecting any of the other files in the program. This lets us easily detect and reuse results for file versions that we have already seen and analyzed. Since most commits in a project's history change a small fraction of files in the project, this greatly reduces the computational and storage costs of providing this service.

This paper is organized as follows. In §2, we provide an overview of scope graphs and the limitations that led to the development of stack graphs. In §3 and §4, we describe the stack graphs formalism in detail. In §5, we discuss related work. In §6, we conclude.

2 Background and historical perspective

We first started investigating how best to add code navigation capabilities to GitHub in 2017. Code navigation is widely available in local editors, so an appealing potential approach would be to adopt whatever technology powers the local editor solution, porting it as necessary to run in our production web server environment.

However, there are clear differences between the code navigation experience in a local editor compared with a software forge. Unlike the local editor, where there is a single user viewing and interacting with a single project (and primarily a single current version of the project's code), a software forge must support an unbounded number of users simultaneously viewing any historic version of any of the code hosted on the forge. Moreover, we must support users who are *exploring* and *browsing* the code, in addition to those who are *authoring* and *maintaining* it. These explorers often view code months or years after it has been written. This means that we must either keep our code navigation data available, and readily queryable, indefinitely; or ensure that we can (quickly) regenerate that data on demand in response to a future user query. This leads to a clear distinction between *index time*, when a user "pushes" a new snapshot of a program or library to the forge, and *query time*, when a (likely different) user wants to view or explore that snapshot at some point in the future.

We have ambitious but conflicting latency goals for the index and query phases. Our most important goal is to minimize the delay between a user performing a code navigation query and us displaying the results of that query in their browser [9]. To achieve this, we must spend some time precalculating information at index time. That said, we do not want to perform *too much* precalculation work, so that code navigation is available quickly after each push,[1] and so we do not waste time precalculating code navigation data for a repository or commit that will never be viewed.

Our scale also presents unique challenges. The most obvious is the volume of code: GitHub holds petabytes of code history, and receives thousands of new code snapshots each minute. It is imperative that our framework be *incremental*, skipping the costs of reanalyzing code that we have already seen. Luckily, much of each project's history is redundant, with most commits changing only a small fraction of files. As such, we particularly prefer *file-incremental* analyses, where we can analyze each source file at index time in isolation, without inspecting (or having access to) any of the other files in the project. The Merkle tree [7] data

[1] For querying, our goal is to show results in under 100 ms. For indexing, our less precise goal is that code navigation should be available before the user can Alt-Tab from their git client over to the browser.

a.py$_1$

```
class A2:
    x3 = 0
```

b.py$_4$

```
from a5 import *

class B6(A7):
    pass

print(B8.x9)
print(B10().x11)
```

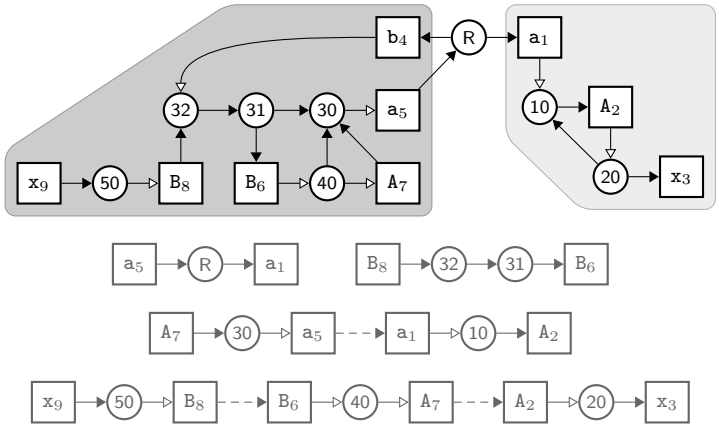


Figure 1 A sample program and its Néron scope graph.

model used by git provides a unique *blob identifier* for each file version, which depends only on the file's content. Because a file's identifier is available before analysis starts, we can skip the storage *and computation* costs of redundant file-incremental work.

Lastly, GitHub hosts code written in a staggering number of programming languages.[2] We want code navigation to work for all of them. It will never be cost-effective for GitHub engineers to implement support for the long tail of less popular languages. But there must be a path for *someone* – whether a GitHub engineer or a member of an external language community – to add support for every programming language that exists and is in use.

As we elaborate in §5, existing local editor code navigation solutions do not satisfy these constraints. While researching whether existing academic work could help tackle this problem, we discovered the *scope graphs* framework [17], which introduced a novel and intuitive approach for encoding the name binding semantics of a programming language in a graph structure.[3] This seemed likely to satisfy our language support goals: all language-specific logic would be isolated to the graph construction process, and all querying would happen via a single language-independent algorithm that operated on that graph structure. However, we quickly discovered that scope graphs, as originally described, would not meet our scale requirements. In the rest of this section, we will explore why, using a simple Python program as a running example.

2.1 Néron scope graphs

Figure 1 shows our example Python program, which consists of two files. The first defines a class $\texttt{A}_2$ containing a single class field $\texttt{x}_3$.[4] The second file imports all of the names from $\texttt{a}_5$, defines a subclass $\texttt{B}_6$, and then prints its inherited field twice: by accessing it first as a class member and then as an instance member.

[2] As of this writing, Linguist [4], our open-source language detection library, includes over 500 distinct languages and sublanguages in its ruleset.

[3] Note that our initial discovery and investigation of the scope graphs framework predates van Antwerpen joining GitHub.

[4] Following the scope graph literature, we add a unique numeric subscript to each identifier so that we can easily distinguish different occurrences of the same identifier in the program source. These subscripts are not part of the actual identifiers seen by the Python interpreter.

We also show the scope graph that we would construct for this program according to the judgments defined by Néron et al. [10]. Circular nodes represent *scopes*, which are "minimal program regions that behave uniformly with respect to name resolution."[5] Rectangular nodes represent definitions and references, with edges connecting a scope to each of the symbols defined in that scope, and connecting each reference to the scope in which that symbol should be resolved. Scope Ⓡ is a *root node*, representing the global namespace that all Python modules belong to, while $\texttt{a}_1$ and $\texttt{b}_4$ are the definitions of the modules defined in the two files.[6] Scope ⑳ contains the class members of class `A`, while the edge from ⑳ to $\texttt{x}_3$ indicates that `x` is one of those class members. Scope ㊿ represents the names that are available via the member access operator in the first `print` statement, while the edge from $\texttt{x}_9$ to ㊿ indicates that the reference must be resolved relative to those names.

Edges between scopes represent nesting. For instance, the top-level definitions in a Python module are evaluated sequentially, and each name is only visible after its definition has been evaluated. This is modeled by the edges connecting scopes ㉚, ㉛, and ㉜, which represent the names visible immediately after the `import`, `class`, and first `print` statements, respectively. Scope ㊵ contains the class members of class `B`; the edge between it and ㉚ models how references within the class body can refer to definitions in the containing lexical scope. Open-arrow edges represent "imports," which make the contents of a named scope available elsewhere. For instance, the edge between ㊵ and $\texttt{A}_7$ specifies that `B` inherits all of the class members of `A`, while the edge between $\texttt{A}_2$ and ⑳ specifies that scope ⑳ contains the class members of class `A`.

We also show the name binding path that correctly resolves the reference $\texttt{x}_9$ to its definition $\texttt{x}_3$. This requires resolving several intermediate references via import edges: resolving $\texttt{B}_8$ to the definition of class `B`; $\texttt{A}_7$ to its superclass; and $\texttt{a}_5$ to the module defining that superclass.

Importantly, the Néron scope graph can be divided into disjoint subgraphs for each file, indicated by shading. Apart from the root node (which is a singleton node shared across all files), every node belongs to some file's subgraph. Every edge connects two nodes that belong to the same file, or connects a node to the shared root node. At query time, we must consider the scope graph as a whole, since name binding paths can cross from one subgraph to another. However, we can *construct* each file's subgraph at index time without having to consider the content of any other file in the program. Néron scope graph construction is therefore file-incremental in exactly the way that we need.

2.2 Van Antwerpen scope graphs

Unfortunately, the Néron scope graph does not allow us to resolve $\texttt{x}_{11}$, since it depends on *type-dependent name resolution*. To resolve $\texttt{x}_{11}$, we must know the return type that we get from invoking $\texttt{B}_{10}$, so that we can resolve the reference in the return type's scope. The return type will depend on what kind of entity $\texttt{B}_{10}$ resolves to, which we do not know *a priori*. In this particular example, it resolves to a class definition, and so the return type is an instance of that class. But it could resolve to a function (or any other "callable"), with an arbitrary return type defined by the function body.

---

[5] We have been careful to label scope nodes consistently across all of our examples: scopes with the same number in each figure represent the same regions of the example source program. In the grand tradition of a BASIC programmer choosing line numbers, we have left "gaps" in the sequence of assigned scope numbers so that related scopes can have numbers close to each other, while leaving room to add additional scopes in later examples.

[6] Note that in Python, the name of a module does not appear explicitly in the source code, and is instead inferred from the name of the file defining it.

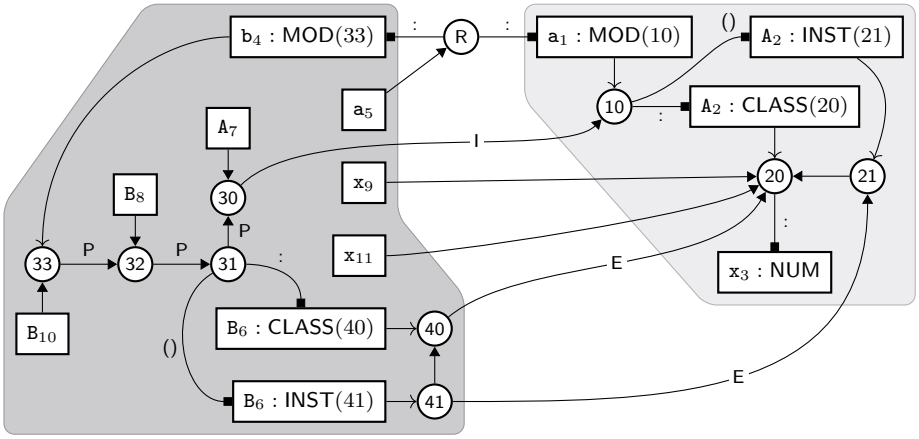


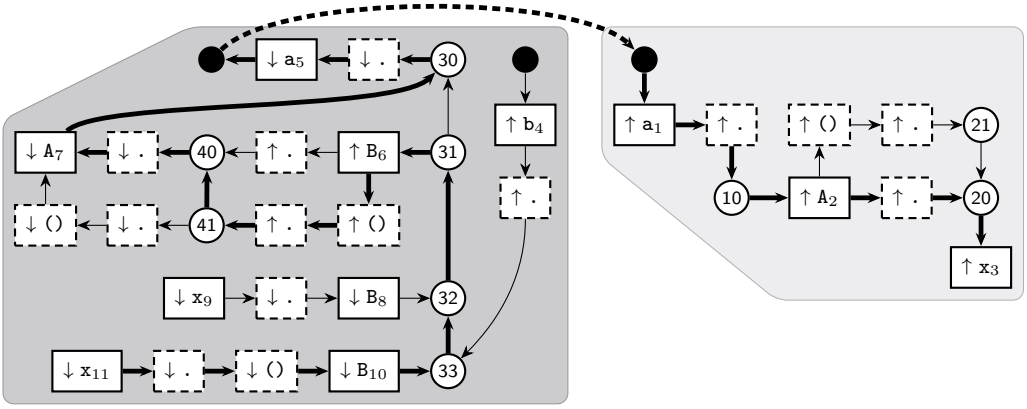
Figure 2 The sample program's van Antwerpen scope graph.

Van Antwerpen et al. [15, 13] extend the scope graph judgments to support these kinds of lookups. Figure 2 shows the scope graph for our example program using these modified judgments. We use square-cap edges to connect each scope with the definitions in that scope. These edges are labeled to indicate the kind of definition: a `:` edge specifies the type of a symbol defined in the scope, while a `()` edge specifies the return type of a callable symbol defined in the scope. We need edges with both labels for Python classes, since they can both be referred to by name (yielding the class itself) and called (yielding an instance of the class). Similarly, we have two associated scopes for each class: one (⑳ and ㊵) for the class members, and one (㉑ and ㊶) for the instance members. Each pair of scopes is connected by an edge, modeling how class members are also available as instance members.

There are paths in the van Antwerpen scope graph that resolve both $\texttt{x}_9$ and $\texttt{x}_{11}$ to the definition $\texttt{x}_3$. However, the paths encoding these name bindings ($\texttt{x}_9 \rightarrow ⑳ \rightarrow \texttt{x}_3$ and $\texttt{x}_{11} \rightarrow ⑳ \rightarrow \texttt{x}_3$) are shorter than we might expect, since they do not directly encode all of the intermediate lookups that are needed.

The intermediate lookups do happen, but at a different time. In the Néron scope graph, there are import edges connecting ㉚ to $\texttt{a}_5$ and $\texttt{a}_1$ to ⑩, which we follow lazily at query time. In the van Antwerpen scope graph, we perform these intermediate lookups eagerly at graph construction time. When we encounter the `import` statement in the source, we immediately resolve the $\texttt{a}_5$ reference. Doing so takes us to $\texttt{a}_1$ and its connected scope ⑩, as in the Néron scope graph. However, we then persist this lookup result into the graph structure, as an `I` edge directly connecting ㉚ to ⑩. We perform (and persist) similar construction-time lookups of `B`'s superclass reference $\texttt{A}_7$, and of the type references $\texttt{B}_8$ and $\texttt{B}_{10}$.

While we have gained the ability to perform type-dependent lookups, we have lost file incrementality. The eager intermediate lookups produce edges that cross file boundaries, violating the disjointedness property of Néron scope graphs. This violation is not superficial; it highlights that each intermediate lookup could depend on *any other* file in the program. We cannot analyze each file purely in isolation, since we cannot know in advance which other files we might have to inspect. Worse, future updates to other files might invalidate the subgraph that we create. In the pathological case, a change to one file might require us to regenerate the scope graph for the *entire program*.



↓ x11	⇝ ↓ x11	⟨x⟩	⇝ ↑ .	⟨x⟩		⇝ ↑ a1	⟨.A.x⟩	
	⇝ ↓ .	⟨.x⟩	⇝ (41)	⟨x⟩	(2)	⇝ ↑ .	⟨A.x⟩	
	⇝ ↓ ()	⟨().x⟩	⇝ (40)	⟨x⟩		⇝ (10)	⟨A.x⟩	(5)
	⇝ ↓ B10	⟨B().x⟩	⇝ ↓ .	⟨.x⟩		⇝ ↑ A2	⟨.x⟩	
	⇝ (33)	⟨B().x⟩ (1)	⇝ ↓ A7	⟨A.x⟩		⇝ ↑ .	⟨x⟩	
	⇝ (32)	⟨B().x⟩	⇝ (30)	⟨A.x⟩	(3)	⇝ (20)	⟨x⟩	(6)
	⇝ (31)	⟨B().x⟩	⇝ ↓ .	⟨.A.x⟩		⇝ ↑ x3	◇	
	⇝ ↑ B6	⟨().x⟩	⇝ ↓ a5	⟨a.A.x⟩				
	⇝ ↑ ()	⟨.x⟩	⇝ ●	⟨a.A.x⟩	(4)			

Figure 3 The sample program's stack graph.

3 Stack graphs

Stack graphs support the advanced type-dependent lookups of van Antwerpen scope graphs. They also retain the file incrementality of Néron scope graphs, by performing intermediate lookups lazily at query time. Our key insight is to maintain an explicit *stack* of the currently pending intermediate lookups during the name resolution algorithm.

Figure 3 shows the stack graph for our example program. There is no longer a shared singleton root node; the graph can contain multiple root nodes, which are indicated by filled circles. Definition and reference nodes have a solid border. Nodes with a dashed border are *push* and *pop* nodes; we will see below how they enable type-dependent lookups. We add arrows to clearly distinguish definition and pop nodes (↑) from reference and push nodes (↓).

We also highlight a name binding path that resolves $\texttt{x}_{11}$ to $\texttt{x}_3$, and trace the discovery of that path. We start with an "empty" path from $\texttt{x}_{11}$ to itself. With each step, we append an edge to the path, and show the path's current frontier and the stack of currently pending lookups. This stack contains both program identifiers (e.g. `x`) indicating names that we need to resolve, and operators (e.g. `.`) indicating how the resolved definitions will be used. Whenever we encounter a reference or push node, we prepend the node's symbol to the stack. Whenever we encounter a definition or pop node, we verify that the stack starts with the node's symbol, and remove it from the stack. The final name binding path ends at a definition node and has an empty stack, indicating that there are no more pending lookups, and the name binding path is complete.

Several steps of interest are highlighted. At step (1) we have seeded the stack with a representation of the full expression being resolved. The top of the stack specifies that the first intermediate lookup that we must perform is to resolve $\texttt{B}_{10}$. The path's frontier indicates that this initial lookup is to be performed in the context of scope (33), which represents the

stack graph	G	*node*	$\mathcal{N}_G^i := \bullet \mid \bigcirc \mid \downarrow x \mid \uparrow x$
source program	P	*edge*	$\mathcal{E}_G \ni i \to i'$
symbol	x	*symbol stack*	$\hat{x} := \diamond \mid x \cdot \hat{x}$
node identifier	i	*path*	$p := i \rightsquigarrow i' \ \{\hat{x}\}$
source file	$\mathcal{F}_G^i$		

$$i \to i' \in \mathcal{E}_G \Rightarrow \mathcal{F}_G^i = \mathcal{F}_G^{i'}$$

Figure 4 Stack graph structure and paths.

At step (2) we have resolved $\texttt{B}_{10}$, and have just popped off the `()` and `.` operator symbols to determine what occurs when we invoke its definition and perform member access on the result. The stack still contains `x`, which will be resolved next. The path's frontier is (41), indicating that $\texttt{x}_{11}$ will be resolved as an instance member of `B`. The very next edge takes us to (40), which allows us to attempt to resolve $\texttt{x}_{11}$ as a class member, remembering that the instance members of a class include all of its class members.

At step (3) we have followed the edges modeling the class inheritance. Doing so pushes additional symbols onto the stack, indicating that we might find `x` as an inherited class member of `A`. Our next step is to resolve $\texttt{A}_7$ in the context of the path's frontier – scope (30), which represents the names that are visible immediately before `B`'s class definition.

At step (4) we have followed the edges representing the `import` statement. The stack now describes how *any* symbol that we are currently looking for might be imported from module `a`. The path's frontier is a root node. Root nodes are the only way that a name binding path can cross from one file to another: when we encounter a root node, we can add a *virtual edge* (the dashed edge in Figure 3) to any other root node, in any file.

At step (5) we have resolved $\texttt{a}_5$ to scope (10), which represents the definition of the imported module. At step (6), we have resolved $\texttt{A}_7$ to scope (20), which contains the class members of class `A`. From there, all that remains is to resolve $\texttt{x}_{11}$ to its definition, $\texttt{x}_3$.

The formal definition of a stack graph is shown in Figure 4. A stack graph G is a representation of a *source program* P, which consists of a set of *source files*. Each source file can be parsed into a set of *syntax nodes*. A subset of those syntax nodes represent *definitions*, and a different (not necessarily disjoint) subset of syntax nodes represent *references*. A *symbol* x is an identifier from the source language, representing (part of) the name of an entity in the program, or an operator symbol like `.` or `()`.

Each node in a stack graph has a unique *node identifier*, and must be one of the following: a *root* node $\bullet$, a *scope* node $\bigcirc$, a *push symbol* node $\downarrow x$, or a *pop symbol* node $\uparrow x$. $\mathcal{N}_G^i$ denotes the node in G with node identifier i. Each node *belongs to* exactly one source file, denoted $\mathcal{F}_G^i$. Some nodes belong to a specific syntax node within that file. A pop symbol node is a *definition node* if it belongs to a syntax node that is a definition. A push symbol node is a *reference* if it belongs to a syntax node that is a reference. $\mathcal{E}_G$ denotes the set of edges in stack graph G. Each edge e in a stack graph is directed, connecting a *source node* i to a *sink node* i'. Edges can only connect nodes that belong to the same file.

The judgment $G \vdash p$ states that p is a valid path in stack graph G. A path consists of a *start node* i, an *end node* i', and a *symbol stack* $\hat{x}$. A path is *complete* if its start node is a reference node, its end node is a definition node, and its symbol stack is empty. Every name binding in a source program P is represented by a complete path in the corresponding stack graph G.

Paths are constructed according to the rules in Figure 5. An *empty* path is one with no edges. It is created by *lifting* a stack graph node. Push symbol nodes "seed" the path's

$$\textsc{LiftPush}\quad \frac{\mathcal{N}^i_G = \downarrow x}{G \vdash i \rightsquigarrow i\ \{x\}}$$

$$\textsc{LiftNoop}\quad \frac{\mathcal{N}^i_G \in \{\bullet, \bigcirc\}}{G \vdash i \rightsquigarrow i\ \{\diamond\}}$$

$$\textsc{Noop}\quad \frac{G \vdash i_0 \rightsquigarrow i_1\ \{\hat{x}\} \qquad i_1 \to i_2 \in \mathcal{E}_G \qquad \mathcal{N}^{i_2}_G \in \{\bullet, \bigcirc\}}{G \vdash i_0 \rightsquigarrow i_2\ \{\hat{x}\}}$$

$$\textsc{Push}\quad \frac{G \vdash i_0 \rightsquigarrow i_1\ \{\hat{x}\} \qquad i_1 \to i_2 \in \mathcal{E}_G \qquad \mathcal{N}^{i_2}_G = \downarrow x}{G \vdash i_0 \rightsquigarrow i_2\ \{x \cdot \hat{x}\}}$$

$$\textsc{Pop}\quad \frac{G \vdash i_0 \rightsquigarrow i_1\ \{x \cdot \hat{x}\} \qquad i_1 \to i_2 \in \mathcal{E}_G \qquad \mathcal{N}^{i_2}_G = \uparrow x}{G \vdash i_0 \rightsquigarrow i_2\ \{\hat{x}\}}$$

$$\textsc{Root}\quad \frac{G \vdash i_0 \rightsquigarrow i_1\ \{\hat{x}\} \qquad \mathcal{N}^{i_1}_G = \bullet \qquad \mathcal{N}^{i_2}_G = \bullet \qquad i_1 \neq i_2}{G \vdash i_0 \rightsquigarrow i_2\ \{\hat{x}\}}$$

Figure 5 Constructing paths by lifting nodes and appending edges.

empty path with an empty symbol stack. We can extend a path by *appending* any edge whose start node is the same as the path's end node. The path's symbol stack might change, depending on the edge's end node. Root nodes and scope nodes are "noops," which leave the symbol stack unchanged. Push symbol nodes prepend an element onto the symbol stack. Pop symbol nodes act as a "guard," requiring that the top of the symbol stack match the node's symbol. This symbol is removed from the symbol stack. (This explains why pop symbol nodes cannot be lifted into empty paths, since there is no symbol stack yet to satisfy this constraint.) If a path ends at a root node, we can immediately extend it to any other root node in the stack graph via a *virtual edge*. (This is the only way that paths cross from one file to another.)

Given these path construction judgments, we can implement a *jump to definition* algorithm: given a source program, and a reference in the program, we want to find all of the definitions that the reference resolves to.[7] First, we construct the stack graph for the program. We then perform a breadth-first search of the graph, maintaining a set of *pending* paths, starting with the empty path we get from lifting the reference node. We use a fixed-point loop to find new pending paths by appending (possibly virtual) edges to existing ones. When appending edges, we must satisfy the constraints of the path construction judgments in Figure 5, and be careful to detect cycles. Whenever we encounter a path that is complete, its end node identifies one of the definitions that the reference resolves to.

This algorithm can be divided such that we construct the stack graph at index time, and save a representation of the graph to persistent storage. At query time, we load in the stack graph and perform the path-finding search. This allows us to amortize the cost of constructing the stack graph across multiple queries. Moveover, stack graph construction is file-incremental, since each node and (non-virtual) edge belongs to exactly one file. By structuring the persistent representation of the stack graph so that each file's subgraph can be identified and stored independently, we can track which file versions we have already created subgraphs for. When we receive a new commit for a repository at index time, we only have to parse and generate new subgraphs for the file versions not seen in previously indexed commits.

[7] When compiling or executing a program, the source language's semantics will typically require that each reference resolve to *exactly one* definition. For an exploratory feature like code navigation, we loosen this restriction to allow ambiguous and missing bindings. This lets us present useful information even in the presence of incorrect programs or an incomplete model of the language's semantics.

$$
\begin{array}{rl}
\textit{symbol stack variable} & \psi \\
\textit{partial symbol stack} & \Psi := \hat{x} \mid \hat{x} \cdot \psi \\
\textit{partial path} & \tilde{p} := \{\Psi\}\ i \rightsquigarrow i'\ \{\Psi'\}
\end{array}
$$

$$
[\![i \rightsquigarrow i'\ \{\hat{x}\}]\!] \triangleq \{\diamond\}\ i \rightsquigarrow i'\ \{\hat{x}\}
$$

$$
\begin{array}{rcl}
\{\diamond\} \downarrow \texttt{x}_{11} & \rightsquigarrow & \textcircled{33}\ \{\langle \texttt{B().x} \rangle\} \\
\{\langle \texttt{B().} \rangle \cdot \psi\} & \textcircled{33} \rightsquigarrow \textcircled{41} & \{\psi\} \\
\{\psi\} & \textcircled{41} \rightsquigarrow \textcircled{30} & \{\langle \texttt{A.} \rangle \cdot \psi\} \\
\{\psi\} & \textcircled{30} \rightsquigarrow \bullet & \{\langle \texttt{a.} \rangle \cdot \psi\} \\
\{\langle \texttt{a.} \rangle \cdot \psi\} & \bullet \rightsquigarrow \textcircled{10} & \{\psi\} \\
\{\langle \texttt{A.} \rangle \cdot \psi\} & \textcircled{10} \rightsquigarrow \textcircled{20} & \{\psi\} \\
\{\langle \texttt{x} \rangle \cdot \psi\} & \textcircled{20} \rightsquigarrow \uparrow \texttt{x}_3 & \{\psi\}
\end{array}
$$

Figure 6 Partial paths.

4 Optimizing queries using partial paths

The process described in §3 splits work between index time and query time, and ensures that the work we do at index time is file-incremental, with all *non*-file-incremental work happening at query time. Unfortunately, this process performs *too much* work at query time. The path-finding search is not cheap, and as users perform many queries over time, we will duplicate the work of discovering the overlapping parts of the resulting name binding paths.

We would like to shift some of this work back to index time, while ensuring that all index-time work remains file-incremental. To do this, we calculate *partial paths* for each file, which precompute portions of the path-finding search. Because stack graphs have limited places where a path can cross from one file into another, we can calculate all of the possible partial paths that remain within a single file, or which reach an "import/export" point.

Partial paths are defined in Figure 6. A partial path consists of a *start node* i, an *end node* i', and a *precondition* Ψ and *postcondition* Ψ', which are each partial symbol stacks. A *partial symbol stack* is a symbol stack with an optional *symbol stack variable*. (Note that a symbol stack variable can only appear at the *end* of a partial symbol stack.) Every path has an equivalent partial path whose precondition is empty, and whose postcondition is the path's symbol stack.

Partial paths are constructed using *lift* and *append* judgments similar to those for paths. Partial paths can be *concatenated* by unifying the left-hand side's postcondition with the right-hand side's precondition, and substituting any resulting symbol stack variable assignments into the left-hand side's precondition and right-hand side's postcondition.

Figure 6 also shows several example partial paths. Each partial path precomputes a portion of the name binding path that we traced through in Figure 3, starting and/or ending at one of the highlighted steps. These partial paths can be concatenated together, yielding the complete name binding path from $\texttt{x}_{11}$ to $\texttt{x}_3$.

Partial paths give us a better balance of work between index time and query time. At index time, we parse each previously unseen source file and produce a stack subgraph for it, as before. We then find all partial paths within the file between certain *endpoint* nodes (root nodes, definition and reference nodes, and certain important scope nodes). Instead of saving the subgraph structure to persistent storage, we save this list of partial paths. At query time, our algorithm has the same overall structure as before. Instead of tracking pending paths and appending compatible edges to them, we track pending partial paths and concatenate them with compatible *partial path extensions*. This process is guaranteed to find all name binding paths that satisfy the path construction judgments from §3.[8] Like our previous algorithm, we amortize the cost of stack graph construction across multiple queries by performing it once at index time. By precalculating partial paths at index time, the new algorithm also amortizes large parts of the path-finding search.

5 Discussion

In this section we describe how stack graphs relate to other work in this area. The most obvious comparison is with scope graphs. In §3 we described how stack graphs were inspired by scope graphs, and were developed in an attempt to support type-dependent lookups while retaining file incrementality.

Ours is not the only attempt to add incrementality to an existing program analysis. Other approaches typically focus on what we term *delta incrementality* [5, 6]. Given a full result set, one determines which files each result depends on. When a file is then changed, only a subset of results are invalidated and recomputed.

Zwaan [18] shows how to add delta incrementality to van Antwerpen scope graphs. In our running example, this approach can detect if an edit potentially causes $\texttt{x}_{11}$ to resolve to some other definition. If so, the entire file is reanalyzed. The updated scope graph still has edges that cross file boundaries, whose sink nodes might have changed due to edits in other files. This means that we must store separate copies of each file's scope subgraph for each commit that the file version appears in. The resulting storage costs would be prohibitive. In contrast, because stack graph content is file-incremental, each file's subgraph can be stored once, in isolation, and reused however many times that file version appears in the project's history. Moreover, we can detect skippable precalculation work early, using only the blob identifiers provided by git.

The problem of language support is not unique to large software forges like GitHub. Local editors have coalesced around the Language Server Protocol (LSP) standard [8], which provides an abstraction barrier between language-specific and editor-specific tooling. Instead of requiring $L \times E$ integrations (one for every combination of language and editor), there are L "server" implementations and E "client" implementations. This greatly reduces the total amount of development work needed to support code navigation "everywhere."

An LSP server typically runs alongside the local editor as an interactive "sidecar" process, answering query requests triggered by user interactions in the editor. This design makes LSP servers unsuitable for large software forges, since it would require maintaining a large enough fleet of LSP servers to handle the maximum expected number of simultaneous user queries. Moreover, we would have to maintain separate fleets of LSP servers for each supported language. The operational burden of maintaining these fleets counteracts the development time saved by reusing existing LSP implementations.

In response to these difficulties, the LSP community developed the Language Server Index Format (LSIF) [2]. This allows LSP servers to run in "batch" mode, producing a list of all name binding resolutions, which can be saved into a simple database table for easy and fast querying at any point in the future. While the LSIF specification is *capable* of storing incremental results from analyzing individual files, no extant LSP *implementations* support that mode of operation. Instead, they are run against an entire project snapshot, typically in the same continuous integration (CI) pipeline used to build and test the project. When a new commit arrives, the entire project needs to be reanalyzed, even if only a small number of files have changed.[9]

[9] LSP servers typically piggy-back on existing compilers and linters, which historically have not worried about incremental operation. Updating the LSP server to be incremental would require retrofitting the existing compiler, which is a more substantial undertaking than writing an incremental LSP server from scratch [11].

LSP and LSIF are the standardization of many previous language-specific frameworks for generating code navigation data at build time. All build-time analyses suffer the same drawbacks. They require the package owner to explicitly specify how to analyze their project,[10] and by running the analysis as part of CI, require the project owner to pay for the compute resources used.

Stack graphs, on the other hand, can be produced via a purely syntactic process, since all name binding logic is encoded in the resulting graph structure. To support this, we have created a new declarative *graph construction language* called tree-sitter-graph [16]. Building on the tree-sitter parsing framework [1], the language expert defines patterns that match against the language's grammar, and which "gadgets" of stack graph nodes and edges should be created for each instance of those patterns in the concrete syntax tree of a source file. These patterns are defined *once* for each language. As a result, stack graph construction requires no configuration by the project owner, and does not need to invoke a project-specific, typically slow, build process.

6 Conclusion

In this paper we have described *stack graphs*, which build upon Visser et al.'s scope graphs framework. Stack graphs can be used to perform type-dependent name resolution, and are file-incremental, which is essential to operate efficiently and cost-effectively at GitHub's scale. Stack graphs have been running in production since November 2021, analyzing every commit to every public and private Python repository hosted on GitHub. The core name resolution algorithm is language-agnostic, and is implemented (and tested, vetted, and deployed to production) *once*. The tree-sitter [1] and tree-sitter-graph [16] libraries, the stack graph algorithms [3, 14], and our initial language-specific stack graph construction rulesets [12] are all open source. This allows other tools to incorporate stack graph code navigation features, and allows external language communities to self-serve support for their language.

References

1 Max Brunsfeld et al. Tree-sitter: An incremental parsing system for programming tools. doi:10.5281/zenodo.4619183.

2 Dirk Bäumer et al. *Language Server Index Format specification, 0.4.0*, 2019. URL: https://github.com/microsoft/language-server-protocol/blob/main/indexFormat/specification.md.

3 Douglas A. Creager and Hendrik van Antwerpen. stack-graphs library, version 0.10.2, January 2023. doi:10.5281/zenodo.7520627.

4 GitHub. Linguist: The language savant. URL: https://github.com/github/linguist/.

5 Sven Kloppenburg. *Incrementalization of Analyses for Next Generation IDEs*. PhD thesis, Technische Universität, Darmstadt, November 2009. URL: http://tuprints.ulb.tu-darmstadt.de/1960/.

6 Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. HyperAST: Enabling efficient analysis of software histories at scale. In *The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, October 2022.

7 Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. doi:10.1007/3-540-48184-2_32.

[10] Modern languages tend to provide official package managers and build tools, and often admit good heuristics for how to build a typical project. But this is not generally true for all languages.

8 Microsoft. *Language Server Protocol specification, 3.17*, May 2022. URL: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/.

9 Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. Association for Computing Machinery. doi:10.1145/1476589.1476628.

10 Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems*, pages 205–231, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46669-8_9.

11 Jeff Smits, Gabriël D. P. Konat, and Eelco Visser. Constructing hybrid incremental compilers for cross-module extensibility with an internal build system. *The Art, Science, and Engineering of Programming*, 4(3), 2020. doi:10.22152/programming-journal.org/2020/4/16.

12 Hendrik van Antwerpen. tree-sitter-stack-graphs-typescript library. URL: https://github.com/github/stack-graphs/tree/main/languages/tree-sitter-stack-graphs-typescript.

13 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.

14 Hendrik van Antwerpen and Douglas A. Creager. tree-sitter-stack-graphs library, version 0.6.0, January 2023. doi:10.5281/zenodo.7534898.

15 Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 49–60, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2847538.2847543.

16 Hendrik van Antwerpen, Rob Rix, Douglas A. Creager, et al. tree-sitter-graph 0.7.0, October 2022. doi:10.5281/zenodo.7221982.

17 Aron Zwaan and Hendrik van Antwerpen. Scope graphs: The story so far. In Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann, editors, *Eelco Visser Commemorative Symposium (EVCS 2023)*, pages 13:1–13:13. OpenAccess Series in Informatics, April 2023. doi:10.4230/OASIcs.EVCS.2023.13.

18 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: Using scope graphs to derive incremental type-checkers. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563303.