HyperAST: Enabling Efficient Analysis of Software Histories at Scale

Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, Jean-Marc Jézéquel

HyperAST: Enabling Efficient Analysis of Software Histories at Scale

Quentin Le Dilavrec
Univ Rennes, IRISA, Inria
Rennes, France
quentin.le-dilavrec@irisa.fr

Djamel Eddine Khelladi
CNRS, Univ Rennes, IRISA, Inria
Rennes, France
djamel-eddine.khelladi@irisa.fr

Arnaud Blouin
INSA Rennes, Univ Rennes, IRISA, Inria
Rennes, France
arnaud.blouin@irisa.fr

Jean-Marc Jézéquel
Univ Rennes, IRISA, Inria
Rennes, France
jean-marc.jezequel@irisa.fr

ABSTRACT

Abstract Syntax Trees (ASTs) are widely used beyond compilers in many tools that measure and improve code quality, such as code analysis, bug detection, mining code metrics, refactoring. With the advent of fast software evolution and multistage releases, the temporal analysis of an AST history is becoming useful to understand and maintain code.

However, jointly analyzing thousands versions of ASTs independently faces scalability issues, mostly combinatorial, both in terms of memory and CPU usage. In this paper, we propose a novel type of AST, called *HyperAST*, that enables efficient temporal code analysis on a given software history by: 1/ leveraging code redundancy through space (between code elements) and time (between versions); 2/ reusing intermediate computation results. We show how the *HyperAST* can be built incrementally on a set of commits to capture all multiple ASTs at once in an optimized way. We evaluated the *HyperAST* on a curated list of large software projects. Compared to Spoon, a state-of-the-art technique, we observed that the *HyperAST* outperforms it with an order-of-magnitude difference from ×6 up to ×8076 in CPU construction time and from ×12 up to ×1159 in memory footprint. While the *HyperAST* requires up to 2 h 22 min and 7.2 GB for the biggest project, Spoon requires up to 93 h and 31 min and 2.2 TB. The gains in construction time varied from 83.4 % to 99.99 % and the gains in memory footprint varied from 91.8 % to 99.9 %. We further compared the task of finding references of declarations with the *HyperAST* and Spoon. We observed on average 90 % precision and 97 % recall without a significant difference in search time.

1 INTRODUCTION

The emergence of distributed version control systems (VCS), such as GitHub or GitLab, has permitted accessing a massive quantity of software and their histories. This offers golden opportunities for both researchers and engineers to perform code analysis of software at large.

Researchers have been analyzing software code histories from different angles, such as recovery of traceability links [2, 3], refactorings [46], edit scripts (aka. Diffs) [14], duplicate code [29], bad smells and their origins [47], or mining of fixes for program repair [23]. Engineers in the software industry developed code analysis tools at scale, such as Copilot[1] for code completion, LGTM[2] for bug detection, or CodeQL[3] for querying the code.

To do so, relying on structured code representation, namely the Abstract Syntax Trees (AST), has become a foundation for many of the above-mentioned tools and software engineering activities.

One of the intrinsic property of software is its continuous evolution [30] and its growing complexity as it evolves. This new dimension asks for temporal code analysis of software histories. In particular, to consistently analyze code elements through their evolutions (*i.e.,* different commits and releases) and also linking their analysis. Such a temporal code analysis can focus, for instance, on origin of code smells [48], bug prediction [36], class stability [41] throughout commits, co-evolution [25, 26] by linking impacting changes and their resolutions.

However, a temporal code analysis requires to simultaneously handle multiple ASTs, corresponding to the different versions of the corresponding software across its history. Unfortunately, doing so on large set of commits for large sized software faces major scalability issues both in terms of memory and CPU usage. With state of the art analysis tools, the whole computation is indeed typically redone from scratch for each version, *i.e.,* commit, even if the commit under analysis is almost identical to previously analyzed ones [25, 26, 48]. Boldi et al. [6] proposed to compress the file structure of a software history without considering the content and for storage purposes of the archive. Alexandru et al. [1] rely on a vertex compression algorithm to share AST nodes among revisions. However, without sharing identical code subtrees independently of their position.

The contribution of this paper is to add a time dimension to an AST, that is to turn it into a *HyperAST* capturing all of it history. The goal is to ease temporal code analysis at large scale on a large timeline of a software history. To do so, the *HyperAST* is built incrementally on a set of commits. It integrates the ASTs of the different versions in one place by leveraging on the code redundancy through space (between code elements) and through time (between versions). This stems from the observation that for a set of commits in a software history, most of the ASTs' elements are similar since most often only small code changes are applied in each commit compared to the rest of the code base. In its core, the *HyperAST* is a

[1] https://copilot.github.com/
[2] https://lgtm.com/
[3] https://codeql.github.com/

Direct Acyclic Graph (DAG) where nodes are unique, allowing for an efficient reuse of nodes in a single version and across versions if unchanged. Moreover, intermediate computation results on top of the *HyperAST*, such as hashes and references, are calculated and stored as metadata along the nodes of the *HyperAST*. Thus, allowing for an efficient reuse of the metadata across versions. The goal of metadata is also to serve as basis for a further in-depth temporal code analysis. To the best of our knowledge, the *HyperAST* is the first attempt to offer an optimized AST covering multiple versions at once in contrast to maintaining multiple ASTs for each version.

We evaluate the *HyperAST* on three levels: (1) its feasibility, (2) its scalability, (3) its usefulness on a practical scenario of usage, namely finding references of declarations.

In particular, we evaluate the *HyperAST* on a data set of 18 real-word and representative Java projects taken from GitHub, by comparing it on a sample of thousand of commits per project to a state-of-the-art tool as a baseline, namely Spoon [38]. Our results show the *HyperAST* correctly represents a software history of a set of commits. It always was able to serialize the code back to its files in 99.98 % and 0.02 % were due to erroneous code in its original state. Moreover, we were able to scale on large repositories with an order-of-magnitude difference between the *HyperAST* and Spoon, from 6 up to 8076 in CPU construction time and from 12 up to 1159 in memory footprint. The minimum and a maximum of construction time in CPU for the *HyperAST* ranged from 1 min and 2 h 22 min, while it ranged from 1 h 14 min and 93 h 31 min for Spoon. Besides, The minimum and a maximum of memory footprint for the *HyperAST* ranged from 63 MB and 7.2 GB, while it ranged from 16 GB and 2.2 TB for Spoon. Thus, the gains in construction time varied from 83.4 % to 99.9 % and the gains in memory footprint varied from 91.8 % to 99.9 %. Finally, we also compared the scenario of finding references of declarations with Spoon and with our implementation of the same scenario on top of *HyperAST*. We observed on average 90 % precision and 97 % recall, respectively, ranging from 68.2 % to 98.2 % and from 84.4 % to 99.7 %. Results also show better search time with the *HyperAST* up to medium-sized projects and better search time with Spoon for large-sized projects. Overall, results are promising towards opening new perspectives for scalable temporal code analysis on large software histories at once.

The main contributions of this paper are:

(1) a novel kind of AST, namely the *HyperAST*, that aims to enable efficient large scale temporal code analyses on software histories;
(2) an open-source implementation of the *HyperAST*;
(3) an evaluation that demonstrates the feasibility, the scalability and the relevance of the *HyperAST*;
(4) a replication package for open-science.

The rest of the paper is structured as follows. Section 2 motivates the problem using an example, classifies temporal code analyses, and introduces mandatory background concepts for the paper. Section 3 presents the *HyperAST*. Section 4 details the evaluation. Section 5 discusses the related work. Section 6 concludes the paper with research directions.

```
1 public class A {
2   public void a1() {
3     B b = new B();
4     this.a2(b);
5   }
6   public void a2(B b) {
7     b.b1();
8   }
9 }
```

(a) Version v_x: Method $a1$ creates a B object and calls method $a2$ with it.

```
1 public class A {
2  public void a1(int value){
3   B b = new B(value);
4   this.a2(b);
5  }
6  public void a2(B b) {
7   b.b1();
8  }
9 }
```

(b) Version v_{x+1}: Method $a1$ now has a parameter used to create b.

```
1  public class A {
2   public void a1(int value){
3    B b = null;
4    if(value > 0) {
5       b = new B();
6    }
7    this.a2(b);
8   }
9   public void a2(B b) {
10   b.b1();
11  }
12 }
```

(c) Version v_{x+2}, the null pointer issue: Method $a1$ checks that $value$ is positive for creating b. The method invocation line 10 may now lead to a null pointer exception.

```
1  public class A {
2   public void a1(int value){
3    B b = null;
4    if(value >= 0) {
5       b = new B();
6    }
7    this.a2(b);
8   }
9   public void a2(B b) {
10   if(b != null) {
11      b.b1();
12   }
13  }
14 }
```

(d) Version v_{x+3}, the fix: Method $a2$ now has a condition for checking that b is not null before calling its method $b1$.

Figure 1: Illustrative example of four versions of a Java class.

2 MOTIVATING EXAMPLE AND BACKGROUND

This section discusses and illustrates with an example the problem we tackle before to introduce the necessary background.

2.1 Motivating Example

This section details the current pitfalls of conducting code analysis on the evolution dimension of a given software history through the following illustrative example (Figure 1). This example is composed of four versions (four evolutions spread in multiple commits) of a class A. At version v_x (Figure 1a), class A has two methods $a1$ and $a2$. The method $a1$ creates a B instance and calls $a2$ with it. At version v_{x+1} (Figure 1b), $a1$ now has a parameter used to create the B instance. At version v_{x+2} (Figure 1c), $a1$ now checks that the $a1$ value is positive for creating b, so that the method invocation line 10 may now lead to a null pointer exception since b might be null. Version v_{x+3} (Figure 1d) introduces a fix that checks the non-nullity of b (lines 10-12).

To understand why this null pointer issue appeared in the past commits one needs to gather, using a static or dynamic code analysis, the evolutions made on the faulty code instructions before it appears. Using a git history this implies to 1) analyze each commit prior to the faulty one to spot those evolutions and 2) keep a link between them and the different commits. This task is complex. Since it is CPU and memory intensive, the stakeholders must also, first, manually analyze the history information (*e.g.,* git commits, diffs) to put them in relation with the current ASTs. Then, to craft by hand the temporal analysis that involves multiple ASTs from

different versions. Thus, such temporal analysis requires to keep in memory elements of each AST to perform analysis. With no manual and ad-hoc optimization, this leads to a memory and CPU over-consumption preventing the analysis to scale, especially on thousands of commits. Moreover, on new commits, one must re-execute the process (*i.e.,* analyzing prior commits to spot related evolutions) to analyze the code issues that appeared in those commits and their history. Therefore, the same computations are likely to be repeated multiple times.

Current pitfalls. To summarize, performing temporal code analyses, *i.e.,* analyzing the AST of the same program at different times, faces the following issues:

- Stakeholders have to write and maintain *boilerplate glue code* to put in relation multiple ASTs, such as in [25];
- As a consequence of the previous point, analyses may *consider evolutions at the class or method level only*, such as in [34], while temporal analyses might require more fine-grained evolutions (*e.g.,* instructions);
- Analyses may face *memory and CPU over-consumption* when several ASTs have to be compared or analyzed together (*i.e.,* not a batch process), thus preventing them to scale, as detailed in [25];
- Stakeholders have to *re-analyze the program history on every new analysis*;
- Current temporal code analysis approaches (*e.g.,* [1]) support the computation of syntactical metrics in a sequential batch way as a temporal code analysis.

2.2 Background

This sub-section details the background on code histories necessary for the paper.

There are mainly two approaches of building a code history. First, the history is represented as a change set. For each version, the difference of the current state with the previous state is stored. Mercurial[4]) uses this approach. Second, each state is stored as a snapshot. To scale, this approach needs to share data between versions, hence, storing only the changed objects and files. Git uses this approach and is based on a Merkle DAG to manage the history of snapshots. Each element of the Merkle DAG is uniquely identified by the hash of its content. By definition, an element of a Merkle DAG is shared between multiple versions if it has the same content. Other representatives of the snapshot-based approach are software-heritage [12] and Piper [39]. The *HyperAST* applies the same principles on the code level, and hence, on the ASTs.

Let us focus on the second type of approach as used in Git. In Git, elements of the Merkle DAG are named *Objects*[5]. *Objects* have a unique identifier called *Oid* that is the hash of the content of the *Object*. Figure 2 depicts the three major kinds of *Objects*: a *Blob* object (square) corresponds to the content of a file; a *Tree* object (triangle) corresponds to a directory, it maps by name other *Tree* objects but also *Blob* objects; a *Commit* object (circle) is a snapshot of the codebase. It points to a *Tree* and its parent *Commit* objects. For the sake of simplicity, these objects are referenced as *Blob*, *Tree*, and *Commit* in the rest of the paper.

[4]https://www.mercurial-scm.org
[5]https://git-scm.com/docs/gitglossary

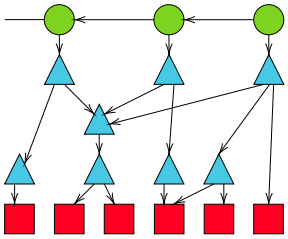

Figure 2: Example of a Git Merkle DAG.
Legend: circle: Commit, triangle: Tree, square: Blob.

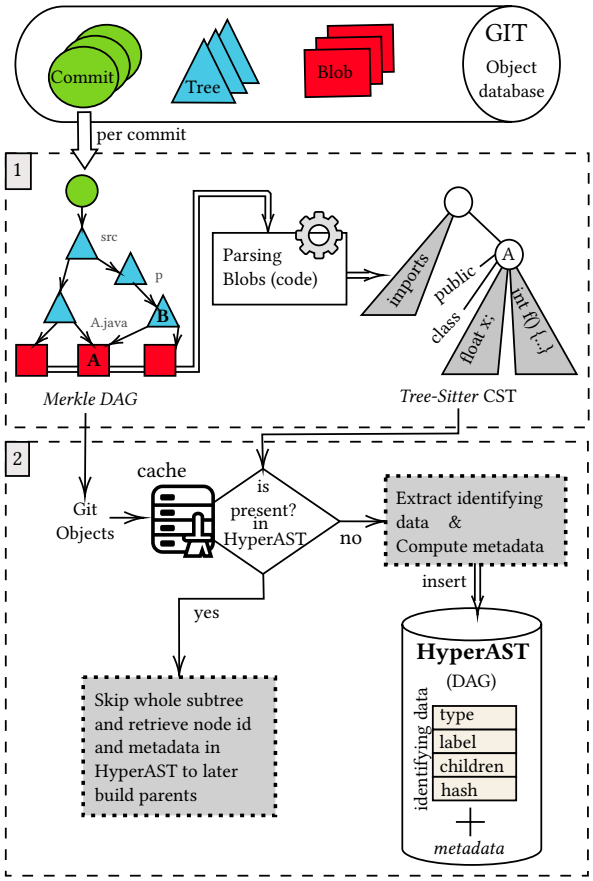

Figure 3: Overall approach of the *HyperAST*.

3 THE *HYPERAST* APPROACH

This section details the concept of *HyperAST*, that improves scalability when analyzing large software histories. The first enabling hypothesis is about 1) redundancy in space between code elements in a single version, and especially 2) redundancy in time among consecutive commits, where small changes are applied compared to the overall size of the code base [51]. The second hypothesis is that various computations can be done once on code subsets [11, 14] then reused multiple times in space and time.

This section first gives an overview of the *HyperAST* and then describes its structure. After that, it details how the *HyperAST* is constructed and updated with every new commit. Finally, we detail how to use the *HyperAST* on a code analysis use case.

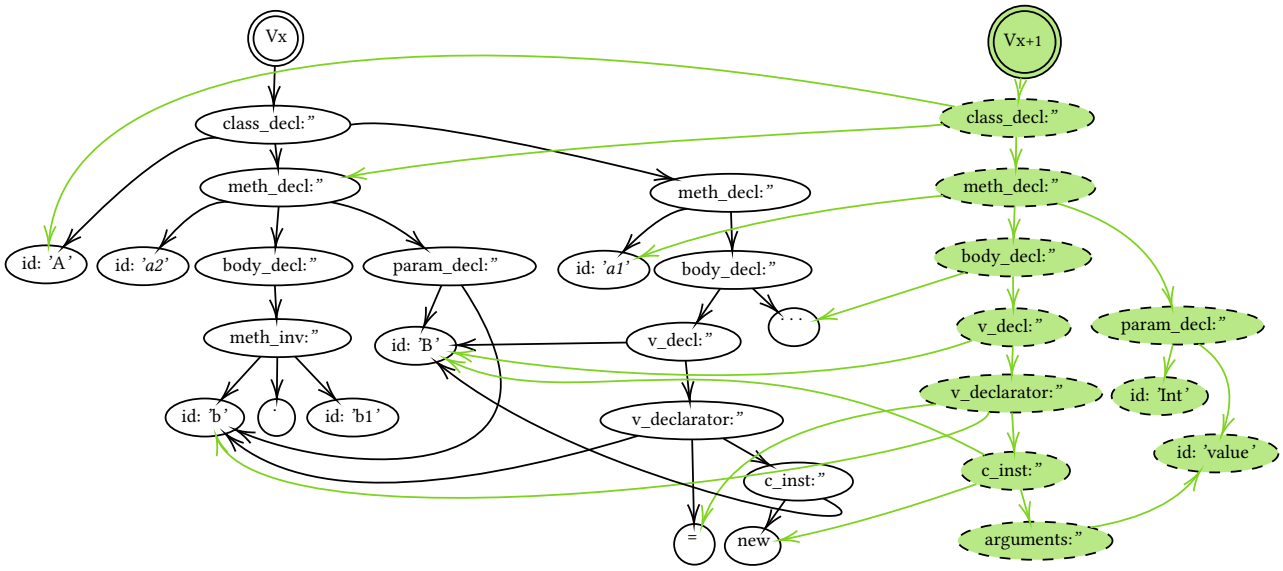

Figure 4: Example of a representation in the *HyperAST* for the first and second commits of class A in Figure 1.

3.1 Overview

Figure 3 shows the overall workflow of the approach of building the *HyperAST* incrementally on new commits. Given the Git the Merkle DAG of a software history, per commit (green circle), the construction process explores the hierarchy of directories through the Git Trees (blue triangles), and parses the code files represented by Git Blobs (red squares). We rely on Tree-Sitter [31] that parses the code into a CST (*Concrete Syntax Tree*). For each git object treated, the construction process associates its Object Identifier (OId) to its corresponding node in the *HyperAST* and stores it into a cache. Thus, if encountered again in following commits, those nodes are not inserted and simply reused across versions. Moreover, when processing an element to insert in the *HyperAST*, the process first extracts identifying data and checks whether the element is already inserted. If not, metadata are computed and inserted as well.

3.2 *HyperAST* structure

This section introduces the structure of the *HyperAST*.
A *HyperAST* is a Direct Acyclic Graph (DAG).
Let $HyperAST = (V, E)$, such that:
V is the set of nodes,
$E \subseteq V \times V$ the set of directed edges between nodes.
A node is a tuple $V = (type, label, children, metadata)$.

A *HyperAST* node is uniquely identified by its type, label, and children resulting of the code parsing. This characteristic is essential as it allows reusing structural clones (*i.e.,* unchanged part of the code) from a commit to other ones.

type: refers to a rule name of the language grammar, such as class, method declaration, assignment.

label: The textual content of a token in a grammar. It is mandatory for leaf nodes but optional for the parent nodes. For example with the given variable declaration "$int\ i\ =\ 0;$", i is the label of the variable identifier and 0 is the label corresponding to the literal assigned to i. The parent node's label of the variable declaration is empty.

children: A list of *references* to other nodes in the *HyperAST*.

metadata: A persistent storage for intermediate results or computation of code analysis on the local sub-tree throughout versions. It helps in reducing redundant and unnecessary computation when analyzing code histories. Its exact definition depends on how stakeholders plan to use and augment the *HyperAST* to fit their needs. For example, it could be complexity metrics or hashes.

3.3 *HyperAST* construction

This section details the construction of a *HyperAST* by first focusing on the tree construction, and then on metadata computations.

3.3.1 Tree construction. The construction process of a *HyperAST* is a tree-to-tree transformation. It starts by processing the commits and their trees of a git's Merkle DAG recursively. This thus maintains the structure and hierarchy of the history in terms of directories and code files, facilitating the serialization of the *HyperAST* back to the code. When reaching a Blob, it is parsed as a CST to be processed before to be inserted into the *HyperAST*. So, the produced *HyperAST* keeps the same structure as the original git's Merkle DAG where Blobs are replaced by CSTs. Compared to Git, this increase of granularity enables code analysis and allows further code deduplication.

The construction of the *HyperAST* can be considered as append-only. Thus, it is incremental in its nature since the approach processes each commit separately and appends all elements related to the commit in the *HyperAST*.

Moreover, node insertion and metadata computation are done only if the node is absent from the *HyperAST*. To check whether an element is already present in the *HyperAST*, the approach uses

its hash (see Hashes in Section 3.3.2) to detect a structurally similar node. In the following, we explain how to handle the different Git objects and the parsed CSTs to construct the *HyperAST*.

Handling a Commit: Each commit has a corresponding root node added to the *HyperAST*.

Handling a Tree: To insert a Tree in the *HyperAST*, the approach first inserts its children (*i.e.,* post-order mode). To computationally benefit from the Merkle DAG, the *HyperAST* keeps a temporary association table as a cache between an Oid (git Object Identifier) and references to *HyperAST* nodes. This allows quickly checking the presence of a node in the *HyperAST*. The construction approach handles each child of a given Tree as follows:

Tree Child is handled recursively.

Blob Child is parsed to produce a CST, which elements are handled recursively.

Finally, a Tree is inserted in the *HyperAST* as a node with its type set to *'directory'* and its label as the Tree name, and with references to its processed children.

Handling a CST element: A CST is also a recursive structure, which is processed following the same steps as for handling a Tree in post order. Finally, a processed CST element is inserted in the *HyperAST* as a node with its: original CST element type; label; and references to its processed children.

We now take the first two commits in Figure 1 as an example to show the constructed *HyperAST*. Figure 4 depicts the constructed *HyperAST* at the second commit $Vx+1$. The left part (white) depicts the *HyperAST* after its construction on the first commit Vx. The right part (green) depicts the new elements added to the *HyperAST* after the second commit is treated: the addition of the parameter `int value` in the method `a1()`, and the argument `value` in the constructor invocation `new B()`. This example shows how both commits exist in the same *HyperAST*. For sake of readability, Figure 4 does not illustrate the nodes' metadata. The entry points in the *HyperAST* corresponds to the handled commits (Vx and $Vx+1$). Thanks to the structural similarities between Vx and $Vx+1$, $Vx+1$ **can reuse all unchanged parts of the code and their already computed metadata**, depicted by the green arrows: this is reuse of **redundancy in time**. Finally, it is worth noting that **nodes are shared even in a single version, *i.e.,* redundancy in space**, such as the `id: 'value'` node in dashed green.

3.3.2 Metadata provisioning. The goal of metadata is to provide developers with intermediate pre-calculated results to be reused for a posteriori temporal code analysis across versions. We designed the *HyperAST* to be extensible to add other types of metadata either during or after its construction by appending the newly computed metadata. This section discusses two concrete kinds of metadata that we are going to use during the evaluation.

Similarity metrics. During the construction of a *HyperAST*, hashes [8, 9, 14] are computed as metadata to help in comparing sub-trees: a structural hash; a structural and label hash; a syntactical hash. The structural hash only uses the type of nodes, while the structural and label hash also uses nodes' labels. The creation of a *HyperAST* also computes for each sub-tree a syntactical hash that considers its serialized code. This metric aims to help in comparing and indexing versions of code.

Index of references. Given a declaration, finding all its references is a standard feature in most code analysis tools. It often implies maintaining an association table between declarations and references. This works fine in the case of a single version. However, maintaining such global tables and ASTs for multiple versions requires heavy maintenance work to keep those numerous tables synchronized. Instead, one could use an oracle capable of answering with certainty when a reference is absent from a given piece of code, *i.e.,* from a sub-tree. Thus, only exploring the sub-trees that may contain references.

To do so, we rely on Bloom filters [4], a probabilistic data structure efficient for supporting an oracle that checks whether an element is a member of a set, in our case a reference in a given sub-tree. Bloom filters also have a substantial space advantage over other data structures, as they do not store the data items at all (*i.e.,* the resolved references) as done in an association table. In essence, a bloom filter is an array of n bits all initially set to 0 and a set of hash functions. For each resolved reference, the Bloom filters use those hash functions to get a set of integers indexes. They set to 1 the corresponding indexes in the bit array to indicate the presence of the resolved reference in the sub-tree. Thus, rather than storing the reference, the *HyperAST* basically indexes it. The *HyperAST* stores the Bloom filters as a metadata with the inserted nodes. Thus, the reference resolution can be reused across multiple versions.

3.4 Use case usage: Finding References

This section discusses how to use the *HyperAST* and its metadata for a classical task: navigating in the code from a given declaration to all its references, in particular, *across the code history*. This is a basic functionality needed for numerous tasks, such as code smells [48], bug prediction [36], class stability [41], and co-evolution [25, 26]. Given declarations, and based on the indexed references, we reuse the bloom filters and their same hash functions to quickly query our oracle for the presence/absence of references in the various sub-trees, recursively until finding them. The goal of this use case is to show its feasibility. Furthermore, we will evaluate on its use case the performance and correctness of the *HyperAST* (Section 4).

3.5 Tool Implementation

We implemented the *HyperAST* in Rust, a performance-oriented programming language. The *HyperAST* is a DAG stored in an Entity-Component-System database [33], inspired from Silva et al. [43] that showed high performance gains. **The code of the *HyperAST* implementation is open source and freely available**[6]. The implementation interacts with Git to obtain the history of a given repository through its Merkel DAG. The implementation then uses Tree-Sitter [31] to parse the code of each Git *Blob*. Tree-Sitter is an incremental and resilient parser that tries to fix erroneous CSTs to a certain extent. Tree-Sitter is thus able to handle commits with erroneous code that a compiler would not.

4 EVALUATION

This section presents the evaluation of the *HyperAST*. First, we present the research questions to then discuss the results. Then, we present the data set and evaluation process. We finally discuss

[6] https://github.com/quentinLeDilavrec/HyperAST

the threats to validity, limitations, and the scope of the approach. **All the material of this section and a replication package are available on our companion web page**[7].

We ran the implementation of our approach on the following hardware configuration: *2 x Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz; 187Gb ram; 1 T SSD*, running *Ubuntu 18.04.6*.

4.1 Research Questions

We now formulate the research questions as follow:

RQ1 **Can we compute the *HyperAST* correctly over several versions?** This research question aims to investigate the sound construction of the *HyperAST*.

RQ2 **How does the *HyperAST* perform and scale compared to a traditional approach?** This aims to position the scalability performance of our contribution over a long evolution history with an established state-of-the-art solution.

RQ3 **How does the *HyperAST* perform and scale on the code analysis task of finding references compared to a traditional approach?** This aims to investigate whether a code analysis task can leverage on the *HyperAST*, thus opening a new perspective to work with the evolution/time dimension rather than on only a single version at a time.

4.2 Data Set

This section presents the data set used in the evaluation and its selection process. The source of our data set is GitHub. We aimed to select real-world Java projects that compile, that are widely used, and continuously maintained with a rich evolution history. To gather such data, we follow this process:

(1) We first curated a list of software organizations that are present on GitHub. Among the organizations we selected are Apache, Google, QuarkusIO, AWS, Alibaba, Junit-team, Mockito, ReactiveX, Spring, Facebook, Bazel build, Jenkins, etc. To which we added other organizations that deliver known software, such as JavaParser, Netty, FasterXMl, Jacoco, Qos, Inria.

(2) To ensure that the software builds correctly, we focused on projects that use a build automation process that successfully passes. We selected Maven projects for this purpose. We also selected projects that support up to Java 14 included since it was imposed by the baseline tool to which we compare to.

(3) We selected the most popular projects (based on their number of 'stars' on Github) and with more than one thousand commits. We finally reached 18 real-world and representative software projects: industrial and academic projects, small to very large-sized projects, monolithic and modular, and simple to complex projects.

Table 1 describes our curated final list of software projects.

Table 1: Data set characteristics.

Projects	Java LOC	Java files	Commits	Contrib.	Stars
Apache Hadoop	1.63M	10.2k	25,749	435	12k
Apache Flink	1.5M	13.2k	30,587	1,037	1.8k
Netty	317k	2.78k	10,789	569	29k
AWS SDK Java v2	265k	3.15k	8,766	88	1.4k
Apache Dubbo	197k	2.81k	5,437	393	37k
Apache Log4j2	183k	2.32k	12,031	132	2.8k
Jenkins	181k	1.69k	32,252	701	19k
Javaparser	179k	1.67k	8,031	166	4.1k
Inria Spoon	154k	2.06k	3,891	106	1.3k
Apache Maven	92.5k	1.05k	11,567	150	3.1k
Apache Spark	85.6k	1.06k	32,821	1,805	33k
Apache SkyWalking	84.7k	1.58k	7,022	397	1.9k
Jackson Core	52.3k	283	2,025	59	200
Alibaba Arthas	44.2k	586	1,726	155	29k
Jacoco	38.8k	633	1,749	46	3.2k
JUnit4	31.2k	471	2,486	151	8.3k
Google gson	25.8k	212	1,650	124	21k
SLF4J	13.5k	256	1,956	61	1.9k

4.3 Results

We empirically evaluate the key properties of the *HyperAST*. To do so, we can classify the properties in two categories. A property can either be objectively checked with a simple assertion, *e.g.,* identity of parsing and re-serializing, or needs a comparison with an existing tool, *e.g.,* for CPU/memory performance and finding references. We choose to compare the *HyperAST* against Spoon [38] as it is a well-known tool in the community, which is used to analyze and modify Java code. Thus, we observe differences of performances between Spoon and the *HyperAST* both in time and in memory for construction and finding references.

4.3.1 RQ1. To answer RQ1 we implemented a serialization service to parse the code and pretty-printing it back to the file level. We can thus evaluate the sound incremental construction of the *HyperAST* with the following equivalence.

$$code \equiv prettyprint(parse(code))$$

Therefore, the goal is to show that the construction of the *HyperAST* is a sound transformation of the Git Merkle DAG and the code Blobs. To do that, we performed the above verification for each project using a set of Java file Blobs uniformly sampled from the Git object database.

On a total of 299 041 Java *Blobs* from all commits, the *HyperAST* was able to parse and to serialize 299 007 them back successfully with an exact equivalence. Thus, achieving 99.98 % correctness. We looked at the remaining 34 *Blobs* and found that it was caused by Tree-Sitter: these *Blobs* contain errors in the code with missing tokens, which the Tree-Sitter parser corrected by adding the most likely tokens that respect the Java grammar rules. This feature of Tree-Sitter allows the *HyperAST* to analyze even erroneous code in contrast to existing tools.

Therefore, we can answer RQ1 positively. Ensuring the sound construction of the *HyperAST* is essential for code transformation purposes, such as patch application or code refactoring.

> ***RQ*$_1$ insights:** Results show the soundness of the *HyperAST* construction with 99.98 % of correctness. It is a preserving transformation for correct code into the *HyperAST*. Erroneous code is slightly corrected at the token level in 0.02 %.

[7] https://github.com/quentinLeDilavrec/ASE2022

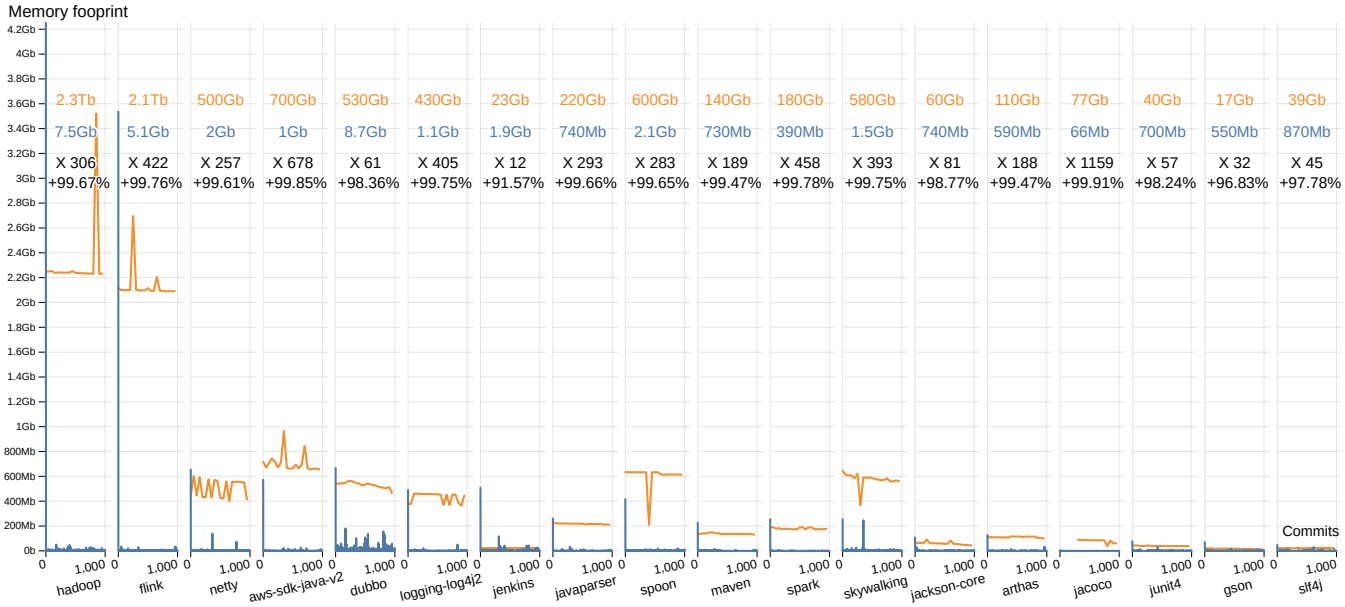

Figure 5: Megabytes of memory taken for each commit, legend: orange=spoon, blue=hyperAST, with total, x order-of-magnitude difference, and + gain. Note: we do the analysis starting from the last commit published corresponding to the commit n°0.

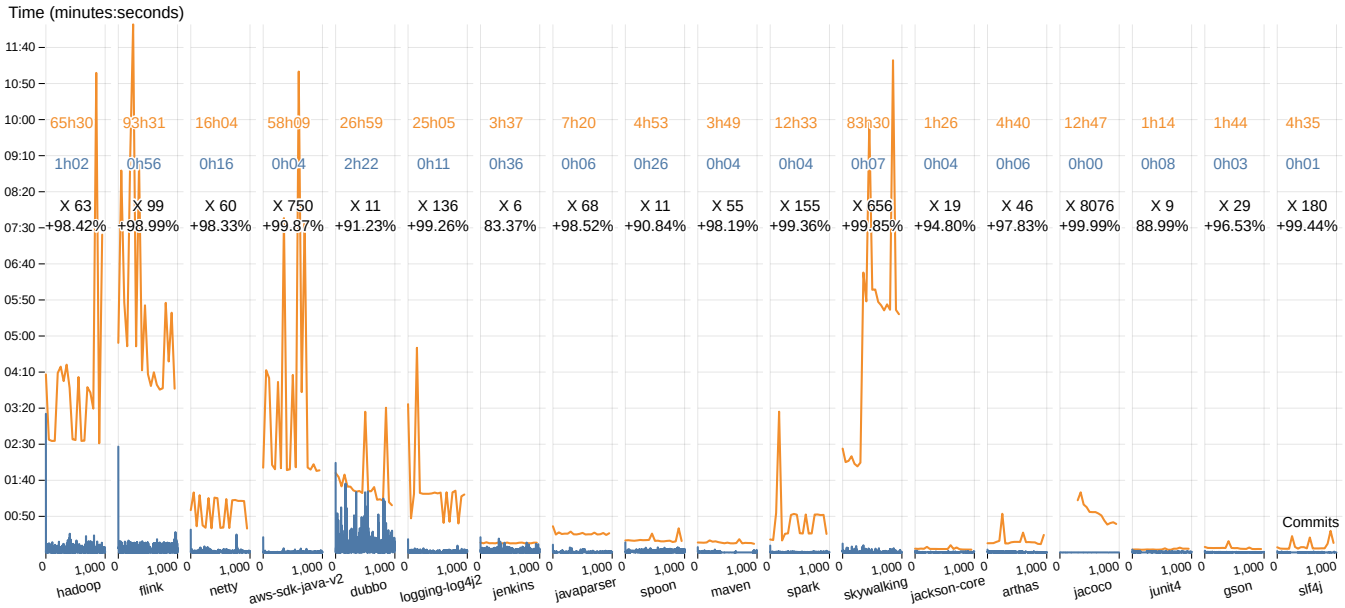

Figure 6: CPU time taken to construct each commit, legend: orange=spoon, blue=hyperAST, with total, x order-of-magnitude difference, and + gain. Note: we do the analysis starting from the last commit published corresponding to the commit n°0.

4.3.2 RQ2. This RQ investigates the difference between the *HyperAST* and traditional approaches in terms of memory footprint and execution time for building ASTs over versions. To do so, we selected the most recent thousand of commits in each project to analyze for a fair comparison between Spoon and the *HyperAST*. For the memory footprint, we measure the heap with and without the constructed structures, *i.e.,* the *HyperAST* and the Spoon ASTs. For construction time, we measure the time in CPU cycles from the start to the end of the construction.

With the *HyperAST* we use the Rust functions $Instant :: now()$ and $Instant :: elapsed().as\_nanos()$ for time measurement and $jemalloc_{ctl}$ library for memory measurement. With Spoon, we use the Java function $System.nanoTime()$ for time measurement and the functions $runtime.totalMemory()$ and $runtime.freeMemory()$ for memory measurement.

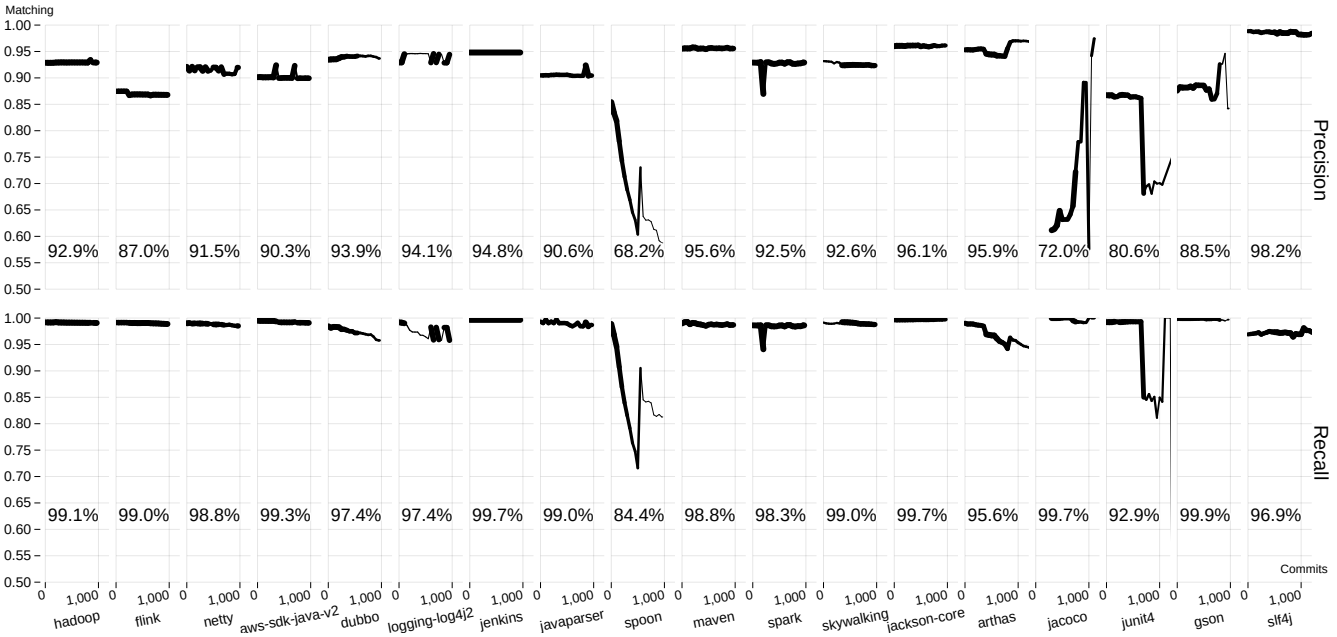

Figure 7: Validity of finding references with precision (top) and recall (bottom) per commit, with respective average.

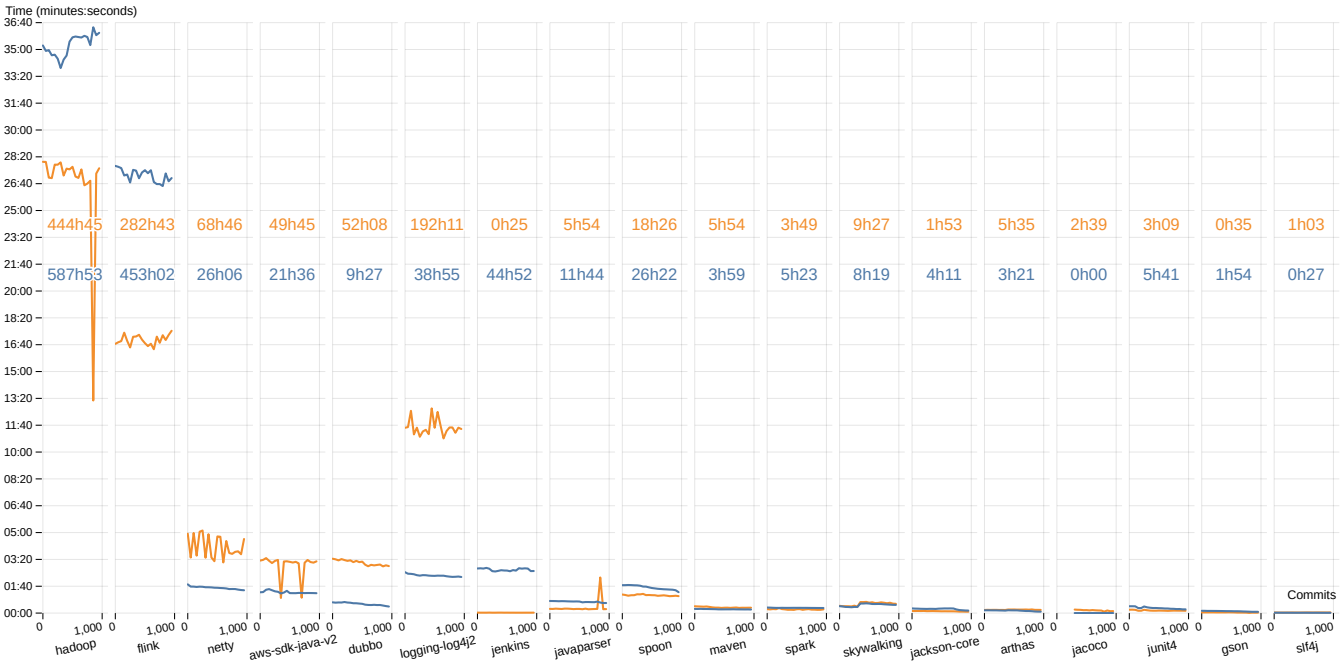

Figure 8: CPU time of finding references for each commit, legend: orange=spoon, blue=hyperAST, with total.

Figures 5 and 6 give the measured memory footprint and the construction time for both Spoon and the *HyperAST*. Note that since we started by treating the latest commit in the history and went back in time to treat the other commits, the depicted figures show the results in that order of commits, *i.e.,* left (0) being the most recent commit. Overall, the *HyperAST* outperforms Spoon. We observe less memory footprint and less construction time for *HyperAST* than Spoon.

Regarding the memory footprint, the *HyperAST* outperforms Spoon in all projects, as shown in Figure 5. On minimum and maximum, respectively, the *HyperAST* consumed 63 MB in Jacoco and 7.5 GB in Hadoop of memory compared to 17 GB in Gson and 2.3 TB

in Hadoop for Spoon. We observed the smallest and the biggest gains in memory footprint of +91.8 % and +99.9 %, respectively, in the Jenkins and Jacoco projects. The order-of-magnitude difference in memory footprint between the *HyperAST* and Spoon varied from ×12 in Jenkins up to ×1159 in Jacoco. Note that the sudden drop in the Spoon project is due to erroneous code in the commits that could not compile, whereas the *HyperAST* can still handle them thanks to the robustness and resilience of Tree-Sitter.

Regarding the construction time, we also observe benefits for the *HyperAST* over Spoon, as shown in Figure 6. The *HyperAST* outperforms Spoon for the analyzed projects. On minimum and maximum, respectively, the *HyperAST* took 1 min in Slf4j and 2 h and 22 min in Dubbo of construction time, compared to 1 h 14 min in Jacoco and 93 h 31 min in Flink for Spoon. We observed the smallest and the biggest gains in construction time of +83.4 % and +99.99 %, respectively, in the Jenkins and Jacoco. The order-of-magnitude difference in construction time between the *HyperAST* and Spoon varied from ×6 in Jenkins up to ×8076 in Jacoco. It is worth noting that the *HyperAST* outperforms Spoon even thought we run it on a single CPU core (no parallelization) in contrast to Spoon with the JDT. Thus, we actually consumed less computing power per commit. Parallelization would improve the construction time *i.e.,* latency, but, is left as future work.

Figures 5 and 6 confirm that the memory footprint and the construction time of the *HyperAST* are always paid initially and then are much lower for later commits. Therefore, the *HyperAST* depends on the code size only for the first commit and then only depends on the commit size, *i.e.,* code changes. For Spoon, it is rather dependent on the code size as expected, where the larger the code is, the more memory and time it consumes to construct its AST.

Finally, we clearly see the issue of scalability with Spoon only on a thousand of commits. However, a thousand of commits did not stress test the *HyperAST*. To do so, we took the Hadoop project that showed to be the most costly to build, and we built the *HyperAST* on all its history of 25 749 commits. The measured memory footprint was 71 GB and the construction time was 18 h 21 min. Compared to Spoon with a thousand of commits, we are still outperforming it with gains of +72 % in construction time and +97.4 % in memory footprint. Resulting in an order-of-magnitude difference of ×3.6 for construction time and ×38 for memory footprint.

> **RQ_2 insights:** The *HyperAST* shows significant results in performance compared to Spoon. It provides scalability gains in memory footprint from +91.8 % to +99.9 % and in construction time from +83.4 % and +99.99 %. Same observation holds when the *HyperAST* takes all the 25 749 commits of Hadoop.

4.3.3 RQ3. This RQ aims to compare the *HyperAST* and Spoon on the same task of code analysis, namely finding references of declarations. For the *HyperAST*, we use our prototype solution based on the oracle of the bloom filters, as explained in Section 3.4. For Spoon, we rely on its constructed association tables.

We first check the validity of our prototype solution. Then, we compare the performances of finding references. We consider Spoon as a ground truth. Thus, we can compute *precision* and *recall* for the *HyperAST* in finding references. Precision and recall vary from 0 to 1, *i.e.,* 0 % to 100 %. They are defined as follows:

$$precision = \frac{ResolvedReferences \cap ExpectedReferences}{ResolvedResolutions}$$

$$recall = \frac{ResolvedReferences \cap ExpectedReferences}{ExpectedReferences}$$

The comparison of references is not based on their labels, as this is not a guarantee of finding the same references. Rather, the comparison is based on the *start* en *end* characters in the code. This is a more restrictive comparison that further would discriminate our approach while increasing confidence in the results.

Figure 7 shows the different measured precision and recall in our case studies for the *HyperAST*. The measurements are per commit where the left (0) represent the most recent commit as we went back in time up to the thousand commit. We observed on average 90 % precision and 97 % recall. Precision ranged from 68.2 % to 98.2 % in Spoon and Slf4j projects, while recall ranged from 84.4 % to 99.7 % in Spoon and Jenkins projects. In the projects where precision was low, namely Spoon, Jacoco, and Junit4, we investigated the found references by the *HyperAST* and Spoon. We found that many discrepancies are due to minor shift in the start and end characters, mainly due to the inclusion of comments to the found references. However, other cases were due to an over-estimation of the *HyperAST* for possible shadowed or overridden references.

Figure 8 shows how the *HyperAST* compares to Spoon in terms of searching time to find all references. It shows that it is similar on average to Spoon for small-sized projects up to 180 k of LOC. We observe that our prototype of finding reference over preforms for medium-sized projects up to 300 k of LOC and it under performs for large-sized projects with millions of LOC. However, in the Haddop and Flink projects, Spoon out performs the *HyperAST* by roughly 10 min per commit. Nonetheless, in contrast to Spoon that is only capable of calculating all references for all declarations at once, the *HyperAST* can calculate the references for a single declaration at a time whenever needed. Therefore, depending on the usage scenario, the *HyperAST* will outperform Spoon in case of finding references for some given declarations and not all of them.

> **RQ_3 insights:**
> Results show the validity of finding references based on the *HyperAST*. We observed an average 90 % precision and 97 % recall without a significant difference in search time but a difference of 10 min in large-sized projects.

4.4 Discussion and implications

Our evaluation shows promising results at scaling the representation of the software history at the AST level. Hence, the *HyperAST* could be used as a basis to scale temporal code analysis. We have identified several use cases that could benefit from the *HyperAST* and that we will explore in future work.

Use case #1: The first practical use case that could benefit from the *HyperAST* are the syntactical code analyses that are performed in a batch way on the software history, *i.e.,* sequentially repeated for every commit without requiring relations between those ASTs. For example, counting syntactical structures of individual files to compute the cyclomatic complexity of classes [1], or detect bad smells in Java files [19, 35, 48].

Use case #2: The second practical use case benefiting from the *HyperAST* are the semantic code analyses that are also performed in a batch way on the software history. For example resolving reference relations with compilers [38], or parsers with reference solvers [18, 42, 45] to provide goto functionalities and enable static impact analysis.

Use case #3: The third practical use case that would benefit from the *HyperAST* are the temporal code analyses linking and tracing code elements across history. For example, understand when a null pointer issue or a bad smell appeared in a commit [10], tracking the evolutions of a given method [15, 17], measuring the stability metric of a class [41] [14, 15, 17], or understanding the evolution of a method complexity [28], computing change- and error-proneness of a given code element [5].

Use case #4: The fourth practical use case that could benefit from the *HyperAST* are the temporal code analyses that investigate complex relationship between code evolutions. In particular, between impacting evolutions and repairing evolution. For example, studying when and how code and tests were co-evolved [25, 26] or identifying causes for bugs code repair [27, 32].

4.5 Threats to Validity

4.5.1 Internal validity. Considering performance measurements, we choose to measure CPU and memory consumption for each processed commit. Our goal was to show to what extent the *HyperAST* scales on complex real-world software with large histories while quantifying it per commit to observe the scalability tendencies. Measuring once for all commits would not have accurately reflected variations of analysis costs, as during development a software might change in quality, complexity and size. Measuring in finer grain than a commit, say for a module, a package or a class, may also be biased, as the measurements could overweight the actual processing cost itself. Thus, measuring performance per commit gives more confidence in the results.

Moreover, as spoon compiles the code, some commits could not be analyzed with Spoon. Whereas, we still were able to analyze them and insert them in the *HyperAST* thanks to the resilience of Tree-Sitter for ill-formed code. In addition, for the validity check with precision and recall, we had to select the commits that Spoon successfully compiled on which it computed the association tables. Thus, being able to compare the found references on both approaches. Our goal here was to show that the *HyperAST* can be used in a code analysis efficiently. However, we did not measure the effort of developing the service of finding references, as this was not the goal of the present contribution. Nonetheless, this is left for future work.

4.5.2 External validity. We implemented and evaluated the *HyperAST* approach for Java with maven build system. Our conclusions in theory could generalize to other programming languages with similar features than Java (*e.g.,* strong static nominal typing). Nonetheless, further experimentation remains necessary on other languages to generalize our results. Note that Tree-sitter is able to support other languages (*e.g.,* C, JavaScript). Thus, making the *HyperAST* extensible beyond Java by integrating the Tree-sitter grammars of other languages. However, the goal of this paper was not to support multiple languages but to show the scalability benefits provided by the *HyperAST*. Further experimenting with other languages is left for future work. Moreover, we relied on Spoon as a baseline, mainly due to its popularity as a frontend to the Java development tools (JDT). Hence, we cannot generalize the observed benefit of the *HyperAST* compared to Spoon for other AST toolings, such as Javaparser as it also resolves the references. This is left for future work to enhance the comparison results.

Finally, the *HyperAST* considers Git histories. As discussed in Section 2.2, the *HyperAST* can operate on any snapshot-based history similar to Git. For the other cases (e.g. SVN), a classical technique consists in converting a repository to Git similarly as Software Heritage [12] initiative does before archiving repositories.

4.5.3 Conclusion validity. Our evaluation gave promising results, showing that the *HyperAST* allows to save a lot, respectively, with an order-of-magnitude difference from 6 up to 872 in construction time and from 12 up to 1158 in memory footprint. The evaluation results also showed providing a reliable service for finding references with an average precision and recall respectively of 90% and 97%. Even though we evaluated it on 18 projects, we plan further evaluation is needed on more projects to have more insights and statistical evidence. Yet, our curated list of projects represents real-world complex software with very large histories.

5 RELATED WORK

Multiple approaches aim to perform temporal code analysis, but face scalability issues. Sousa et al. [44] proposed to detect multiple interrelated refactorings, called composite refactorings between v_i and v_{i+1}. They consist in detecting smells in version v_i. However, to do so, they had to run through the commits to find the bad smells, then to run again to find refactorings before linking them.

Cedrim et al. [7] similarly proposed to understand the impact of refactorings on bad smells. Thus, detecting refactorings in v_{i+j} and bad smells in version v_i. Thus, both [7, 44] having several ASTs for the studied commits, which limits the scalability of their studies.

Pantiuchina et al. [37] illustrate this challenge explicitly. They quoted several points: *"the choice of selecting a subset of the 303 projects was dictated by the computationally expensive data extraction process adopted in our study". and "This process took three months on a 56-core server". "we computed 42 product- and process-metrics (*e.g.*, code quality metrics, change-proneness of classes) for each of the* 213 102 *commits in the studied projects."*

The *HyperAST* aims to be as support for scaling, especially by reducing redundancy and reusing computed results for parts of the code that do not change across commits.

Moreover, Kim and Notkin [20] surveyed matching approaches of code elements that can be used for multi-version program analyses. With the *HyperAST*, this is partially done by construction where the same nodes are shared across versions. For the changed parts of the code, a matching per sub-tree can be applied, hence, only computing the matching on the changed parts of the code across versions.

Larose et al. [24] proposed to merge AST nodes into supernodes to speed up the run-time performance of Truffle-based interpreters [50] in GraalVM[8]. The idea behind is to reuse similar nodes in behavior and in a single version. Protzenko et al. [40] also proposed to merge the ASTs for the purpose of real-time collaboration. However, both [24, 40] reason on the space dimension of the AST and not on the time dimension. In the *HyperAST* we reuse structurally similar AST nodes and across versions.

Code analysis tools (use case #2 in section 4.4) such as Spoon and JavaParser propose to build the AST and to resolve the references with association tables. GitHub also developed tools, in particular related to our work Semantic [42] and Stack-graphs [45]. The former relies on Tree-sitter to parse the code and resolve the AST symbols for code navigation. The latter is inspired from Scope Graphs [21, 22, 49] to resolve name bindings necessary for resolving references of declarations. The *HyperAST* proposes the same (i.e., parsing and resolving references) but with a different strategy based on an oracle and bloom filters while taking advantage of the structure of the *HyperAST*.

Furthermore, Boldi et al. [6] proposed to compress the archive of Software Heritage [13]. However, they do it only on the file structure without considering the content and for storage purposes of the archive. Hartel et al. [16] used incrementalization theory to incrementalize the processing of repository resources and artefacts. Grund et al. [15] proposed an approach to track changes at the method level across time. Higo et al. [17] also proposed an approach to track methods changes based on Git. All [15–17] (use case #3 in section 4.4) are complementary to our approach as they can be implemented as strategy on top of the *HyperAST*.

One similar approach to the *HyperAST* aimed at improving temporal analysis by making use of the temporal coherency of revisions. Alexandru et al. [1] proposed an approach called LISA. It is a graph oriented database that relies on a vertex compression algorithm to share AST nodes among revisions, where structural relations between AST nodes are constrained by ranges of revisions [1]. Compared to the *HyperAST*, LISA misses a large reduction of redundant computation as metadata because it does not support sharing identical code subtrees independently of their position. In addition, Alexandru et al. [1] only discusses LISA in the context of the computation of syntactical metrics, such as cyclomatic complexity (use case #1 in section 4.4). Whereas we present a semantic analysis (i.e. resolution of references and declarations needed for impact analysis) that is more challenging to compute.

To the best of our knowledge, the *HyperAST* is the first approach to propose an AST that lives with the evolution history of a software. This was possible based on the redundancy both in space between AST elements and in time between versions.

6 CONCLUSION

This paper proposes a novel type of AST, namely the *HyperAST* that permits temporal code analyses to scale across large code histories. The evaluation on 18 large projects shows that we were able to scale with thousands of commits with an order-of-magnitude difference between the *HyperAST* and Spoon, from ×6 up to ×8076 in construction time and from ×12 up to ×1159 in memory footprint. While the *HyperAST* requires up to 2 h 22 min and 7.2 GB for the biggest project, Spoon requires up to 93 h and 31 min 2.2 TB. The gains in construction time varied from +83.4 % to +99.99 % and the gains in memory footprint varied from +91.8 % to +99.9 %. These benefits and gains were also observed when we built the *HyperAST* on the whole 25 749 commits history of the Hadoop project in contrast to only a thousand of commits with Spoon. We further compared the task of finding references of declarations with the *HyperAST* and Spoon. We observed on average 90 % precision and 97 % recall without a significant difference in search time.

As future work, we first plan to enlarge the scope of the *HyperAST* to other languages by integrating more grammars from Tree-Sitter. We will revisit existing approaches discussed in section 4.4 on top of the *HyperAST*. Finally, we will explore the possibility to offer an engine capable of running a temporal analysis taken as a query to be executed on the *HyperAST*. Thus, facilitating its usage by a wider audience.

[8] https://www.graalvm.org

ACKNOWLEDGMENT.

The research leading to these results has received funding from the *RENNES METROPOLE* under grant *AIS no. 19C0330* and from *ANR* agency under grant *ANR JCJC MC-EVO² 204687*.

REFERENCES

[1] Carol V Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C Gall. 2019. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering* 24, 1 (2019), 332–380.

[2] Thazin Win Win Aung, Huan Huo, and Yulei Sui. 2020. A literature review of automatic traceability links recovery for software change impact analysis. In *Proceedings of the 28th International Conference on Program Comprehension*. IEEE/ACM, 14–24.

[3] Gabriele Bavota, Luigi Colangelo, Andrea De Lucia, Sabato Fusco, Rocco Oliveto, and Annibale Panichella. 2012. TraceME: traceability management in eclipse. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 642–645.

[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[5] Arnaud Blouin, Valéria Lelli, Benoit Baudry, and Fabien Coulon. 2018. User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners. *Information and Software Technology* 102 (May 2018), 49–64.

[6] Paolo Boldi, Antoine Pietri, Sebastiano Vigna, and Stefano Zacchiroli. 2020. Ultra-large-scale repository analysis via graph compression. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 184–194.

[7] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. IEEE/ACM, 465–475.

[8] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th international conference on program comprehension*. IEEE, 243–247.

[9] Gregory Cobena, Serge Abiteboul, and Amelie Marian. 2002. Detecting changes in XML documents. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 41–52.

[10] Benoit Cornu, Earl T Barr, Lionel Seinturier, and Martin Monperrus. 2016. Casper: Automatic tracking of null dereferences to inception with causality traces. *Journal of Systems and Software* 122 (2016), 52–62.

[11] Barthélémy Dagenais and Martin P Robillard. 2014. Using traceability links to recommend adaptive changes for documentation evolution. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1126–1146.

[12] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES 2017 - 14th International Conference on Digital Preservation*. iPRES, 1–10.

[13] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software heritage: Why and how to preserve software source code. In *iPRES 2017-14th International Conference on Digital Preservation*. 1–10.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM/IEEE, 313–324.

[15] Felix Grund, Shaiful Alam Chowdhury, Nick C Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing method-level source code histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1510–1522.

[16] Johannes Härtel and Ralf Lämmel. 2020. Incremental map-reduce on repository history. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 320–331.

[17] Yoshiki Higo, Shinpei Hayashi, and Shinji Kusumoto. 2020. On tracking Java methods with Git mechanisms. *Journal of Systems and Software* 165 (2020), 110571.

[18] Roya Hosseini and Peter Brusilovsky. 2013. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, Vol. 1009. University of Pittsburgh, 60–63.

[19] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275.

[20] Miryung Kim and David Notkin. 2006. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*. IEEE/ACM, 58–64.

[21] Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7745)*, Krzysztof Czarnecki and Görel Hedin (Eds.). Springer, 311–331.

[22] Gabriël Konat, Vlad A. Vergu, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. The Spoofax Name Binding Language. In *Companion to the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2012, Tucson, AR, USA, October 19 - 26, 2012*. ACM.

[23] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25, 3 (2020), 1980–2024.

[24] Octave Larose, Sophie Kaleba, and Stefan Marr. 2022. Less Is More: Merging AST Nodes To Optimize Interpreters. In *MoreVMs'22: Workshop on Modern Language Runtimes, Ecosystems, and VMs*. ACM, 1.

[25] Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Untangling Spaghetti of Evolutions in Software Histories to Identify Code and Test Co-evolutions. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 206–216.

[26] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–46.

[27] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 329–340.

[28] Mateus Lopes and Andre Hora. 2022. How and why we end up with complex methods: a multi-language study. *Empirical Software Engineering* 27, 5 (2022), 1–42.

[29] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. 2016. JDeodorant: clone refactoring. In *Proceedings of the 38th international conference on software engineering companion*. IEEE/ACM, 613–616.

[30] Tom Mens. 2008. *Introduction and roadmap: History and challenges of software evolution*. Springer.

[31] Microsoft. 2022. Tree-Sitter. https://tree-sitter.github.io/tree-sitter/. Accessed: 2022-05-06.

[32] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *Journal of Systems and Software* 163 (2020), 110538.

[33] Robert Nystrom. 2014. *Game programming patterns*. Genever Benning.

[34] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2014. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* 41, 5 (2014), 462–489.

[35] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[36] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2017. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering* 45, 2 (2017), 194–218.

[37] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–30.

[38] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179.

[39] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (jun 2016), 78–87.

[40] Jonathan Protzenko, Sebastian Burckhardt, Michał Moskal, and Jedidiah McClurg. 2015. Implementing real-time collaboration in TouchDevelop using AST merges. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*. ACM, 25–27.

[41] D Rapu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. 2004. Using history information to improve design flaws detection. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* IEEE, 223–232.

[42] Github Semantic. accessed July 27, 2022. Title of Citation. https://github.com/github/semantic.

[43] Pablo Diego Silva da Silva, Rodrigo Oliveira Campos, and Carla Rocha. 2021. OSS Scripting System for Game Development in Rust. In *IFIP International Conference on Open Source Systems*. Springer, 51–58.

[44] Leonardo Sousa, Diego Cedrim, Alessandro Garcia, Willian Oizumi, Ana C Bibiano, Daniel Oliveira, Miryung Kim, and Anderson Oliveira. 2020. Characterizing and identifying composite refactorings: Concepts, heuristics and patterns. In *Proceedings of the 17th International Conference on Mining Software Repositories*. IEEE/ACM, 186–197.

[45] Github Stack-graphs. accessed July 27, 2022. Title of Citation. https://github.com/github/stack-graphs.

[46] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 1, 1 (2020), 1.

[47] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 403–414.

[48] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[49] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018).

[50] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 13–14.

[51] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. 2008. Mining software repositories to study co-evolution of production & test code. In *2008 1st international conference on software testing, verification, and validation*. IEEE, 220–229.