HyperAST: Incrementally Mining Large Source Code Repositories

Quentin Le Dilavrec
q.ledilavrec@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

***Abstract*—Modern software systems are large, with a project like Chromium reaching more than 30 million lines of code. Analyzing these large-scale projects over multiple versions rapidly becomes very expensive, and creating tools that can work at this scale is a challenge. This paper presents the *HyperAST* approach, that exploits the locality and redundancy of source code, to maintain thousands of Abstract Syntax Tree (AST) versions in memory. In particular, we contribute a programmatic interface to *HyperAST* that helps define the incremental computation of code metrics and efficient explorations of the fine-grained abstract syntax representation of source code.**

***Index Terms*—software repository mining, software maintenance.**

I. Introduction

Analyzing large-scale software systems is hard, because solutions to analyze source code histories both efficiently and accurately are still limited. Either code is analyzed like text with little access to the code structure and semantics, using tools such as ElasticSearch or directly through the Version Control System (VCS), *e.g.,* `git blame`. Alternatively, source code is analyzed through the intermediate compilation artifacts, giving access to semantics in exchange for more distance to the source and significant overhead, *e.g.,* the *Java Development Tools (JDT)*, or *CodeQL* [1], in the form of an Abstract Syntax Tree (AST) or a Control Flow Graph (CFG). To strike a middle ground between the scalability of VCSs and expressivity of ASTs, we identified the following challenges:

- provide full access to the fine-grained recursive structure of source code, for precise and robust analyses
- minimize memory footprint and reduce random memory accesses, to maintain as much code as possible in fast memory
- make processing steps modular to minimize unneeded computations

To address these challenges and provide more efficient means of computing code metrics (*e.g.,* McCabe Cyclomatic Complexity (Mcc), number of Lines of Code (LoC)), we propose the *HyperAST* framework. At its core the *HyperAST* [2] is a simple recursive data structure that stores source code represented as concrete syntax. The efficiency and scalability of *HyperAST* dealing with source code histories stems from two observations: 1) fine-grained source code is vastly redundant in and across versions, and 2) local intermediate results can be persisted and reused to make analyses more efficient. *HyperAST* materializes these observations through major design choices that we detail in Section II; at the same time we will discuss implications on functionality and performance of source code analyses.

While we explored some of the implications of *HyperAST* in earlier work [2], [3], in this paper we propose a sandboxed scripting abstraction for *HyperAST* to more easily 1) compute intermediate results (i.e., *preprocessing*), and 2) traverse versions using the intermediate results (i.e., *querying*).

We showcase this abstraction to compute the McCabe Cyclomatic Complexity and Lines of Code metrics (preprocessing), and explore source code exhibiting these metrics (querying); we showcase the scalability and performance benefits of *HyperAST* when exploring large software repositories with years of development history, *e.g.,* Hadoop (>1M LoCs, >30K commits).

II. The *HyperAST* Processing Model

In order to realize our vision of scalable and efficient source code analysis, we built the processing model of *HyperAST* around three fundamental design decisions:

- Subtrees of code are deduplicated through structural identity. A subtree of code is a node defined recursively with three identifying attributes: its type (*e.g.,* `method_declaration`), possibly a label (*e.g.,* the name of a method), and references to its children. A snapshot is consequently a subtree at the root of a commit.
- Local intermediate results (*e.g.,* tree size, Mcc, LoCs) are computed once per unique subtree and persisted contiguously to the primary attributes, we call them derived attributes. A subtree is ready to be used immediately once intermediate results have been computed and inserted into the *HyperAST*.
- Subtrees with similar attributes are stored in contiguous memory regions. Making memory more uniform and facilitating its management, while enabling the capability of storing attributes with identical values at the region level (instead of each subtree).

The functional and performance implications of these design choices over source code analyses are numerous, and constitute, to the best of our knowledge, a singular processing model among existing approaches [4], [5], whereby intermediate results are computed on AST subtrees without access to its context. To test the potential of this processing model, we have previously explored two source code analyses in earlier work, and established important functional and performance outcomes [2], [3]: 1) we found that deduplicating subtrees in and across versions drastically reduces the overall memory footprint of the *HyperAST*, stemming from the observation that changes to consecutive commits are mostly small, 2) we observed that computing intermediate results once per deduplicated subtree makes it an incremental process, increasing efficiency, 3) we established that intermediate results can be used to bubble up information from deep inside subtrees, enabling the preemptive skipping of unfruitful subtrees during traversal, thus reducing random memory accesses at the cost of persisting the required intermediate results; storing them contiguously further reduces random memory accesses, and 4) we found that storing nodes with similar attributes (*e.g.,* same type, same layout) in contiguous memory regions, makes memory more uniform and facilitates its management; more interestingly it also enables to store properties on regions instead of individual nodes, further improving memory utilization.

A. Previous Applications of HyperAST

We initially demonstrated the applicability of the *HyperAST* model on the semantics of references of Java, most notably to enable finding all references to a given declaration (def-use relations), without any global association table [2]. We then introduced *HyperDiff*, in which we addressed the computation of tree diffs [3]. Combining *HyperAST* with the Gumtree diff algorithm [6], lazily decompressing the diffed snapshots into trees, *e.g.,* early matched subtrees are left untouched.

These two contributions enabled us to investigate strategies to adapt inherently complex AST features and algorithms on the *HyperAST*, demonstrating on large industrial software histories the possible efficiency and scalability gains over state-of-the-art solutions. To scale the reference search, we augmented an otherwise naive and exhaustive search, with means of preemptively skipping subtrees not containing the wanted references, leveraging precomputed sets of unresolved references on subtrees instead of maintaining global association tables [2]. In the tree diff case, we took a more parsimonious approach, providing the functional properties of a tree without mandating additional memory usage, and revisiting the Gumtree algorithm to make it only grow the required memory, *i.e.,* decompressing the DAG on demand. We thus demonstrated that *HyperAST* can be used to scale tree diffs to entire commits (∼20 million AST nodes) [3].

B. Challenges of conforming to the processing model

Fully exploiting the performance benefits of the *HyperAST* requires to conform to its processing model, which we acknowledge can be difficult, at least requiring additional facilities. Fundamentally, deduplicated subtrees do not have direct access to their parents, *i.e.,* the *HyperAST* is topologically a Direct Acyclic Graph (DAG). Moreover, results persisted on subtrees must satisfy the locality invariant, *i.e.,* $p_1 = p_2 \implies d_1 = d_2$, where for every subtree i, p_i is the set of primary attributes and d_i is the set of derived attributes. Consequently, certain analyses made for trees or graphs must be adapted and possibly redesigned to conform to these limitations.

Contrastingly, we also observed some accidental complexity over comparatively simpler to adapt code metrics, specifically when composing them while keeping the processing modular and minimal (we don't want to preprocess unnecessary metrics). The particular case of the offset of a code element (in a file) is illustrative. In a traditional AST, one would store the offset and length of code elements as attributes (alternatively the start and end position). Within *HyperAST*, only the length would be stored, then to compute the offset, one would sum the length of every left siblings up until reaching the ancestor representing the file. Moreover, there are multiple kinds of offsets (*e.g.,* in characters, in rows + columns, in topological order), corresponding to multiple positioning schemes (*e.g.,* a range in a file, a row/column in a file, a topological index). And converting between each of these schemes can be necessary to interoperate with other tools. Consequently, we would like to facilitate toggling and adding new positioning schemes (also for other metrics in general).

III. Programmatic Interface for *HyperAST*

To facilitate the overall usage of the *HyperAST*, we propose to formalize a programmatic interface. The first covered aspect is the preprocessing of subtrees (Section III-A). This is important to enforce the locality invariant, and compose different metrics and (possibly interdependent) preprocessing steps. The second aspect concerns the querying and traversal of a subtree (Section III-B). We need to provide a tree abstraction over the underlying DAG, helping to reduce the initial efforts of adapting an analysis. Finally, we provide the capability of using this programmatic interface through scripting languages (*e.g., Lua*), making *HyperAST* utilization interactive and robust. Using Sandboxed scripts also enables enforcing the locality invariant during preprocessing.

A. Defining Metrics and their Computation

To compute a metric M, an accumulator A is defined, it holds a partially computed metric while children are processed. We propose the 3 following phases to compute a metric on a given subtree:

init initializes A with a value, either default or derived from the type or label of the node

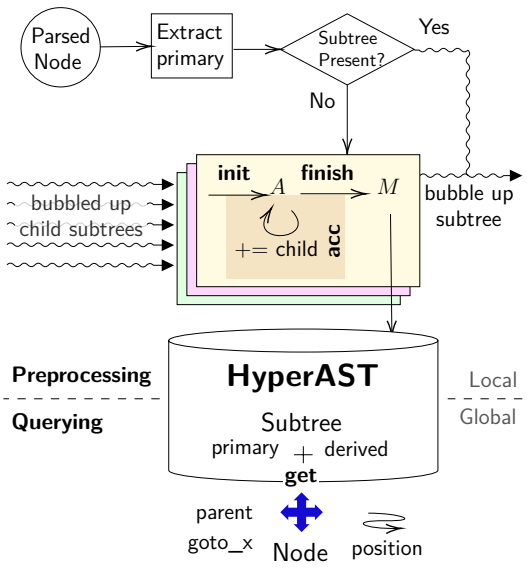

Figure 1: Architecture

(a) Mcc

```
1 mcc = if is_branch() then 1
      else 0
2 function acc(c)
3   mcc += c.mcc end
4 function finish()
5   if is_branch() then
6     mcc += 1 end
7   return {mcc: mcc} end
```

(b) Lines of Code

```
1 local LoC, b = 0, false
2 function acc(c)
3   if c.is_comment() then
4     b = true
5   else if b then
6     LoC += c.LoC
7   else b = true end
8 end
9 function finish()
10  if is_comment() then LoC = 0
11  else if TY == "Spaces"
12  and L.find("\n") ~= nil
13  then LoC = 1 end
14  return {LoC: LoC} end
```

Listing 1: Preprocessing

```
1 max_mcc, method_mcc = 0, ""
2 local up = false
3 function query(n)
4   if n.mcc < max_mcc then
5     if n.goto_parent() then
6       up = true
7       return query(n)
8     else return max_mcc,
9                 method_mcc,
10    end
11  else
12    if n.mcc > max_mcc then
13        max_mcc = n.mcc
14        method_mcc = n.position()
15    end
16    if b and n.goto_first_child()
       then
17      up = false
18    if n.goto_next_sibling() then
19        up = true
20    elseif n.goto_parent() then
21      up = true end
22    return query(n)
23  end
24 end
```

Listing 2: Find largest method in terms of Mcc

acc accumulates results from children, such that each child c (a subtree) can be processed to mutate A.

finish finishes the accumulation process, A is converted to M.

Using the script evaluation scope, A can be defined with local variables (Listing 1) and passed to the functions representing each phase (making `acc` and `finish` closures). Listing 1 presents the preprocessing of two code metrics: Listing 1a computes the cyclomatic complexity (similarly presented in [5]), and Listing 1b presents a way of computing the number of lines of code, *i.e.,* it filters out comments and only counts newlines once through `is_nl`, that we provide to the scripting interface. Indeed, data and complex processing can be provided by implementing predicates (*e.g.,* `is_branch`, `is_comment`), and exposing derived data computed with the preexisting facilities (in Rust). These examples also show how we can propagate the derived data, for example, in Listing 1a, `finish` returns the Mcc in an association table, naming it (line 7); then in `acc`, through the child subtree passed as parameter it can be accessed as `c.mcc` (line 3).

B. *Querying* HyperAST

To query and traverse the *HyperAST*, we propose to overlay a tree interface over subtrees, namely a node. Nodes still provide access to all attributes to their subtree. Nodes add the capability of moving in the DAG with a simple cursor interface, just as if it was a standard AST, Nodes maintain and mutate their global position from the initial root, *e.g.,* a snapshot. Listing 2 illustrates this querying interface in terms of finding the largest method in terms of Mcc. This example uses the derived data computed by the example in Listing 1a, and demonstrates the use of a cursor interface to traverse a snapshot in *HyperAST*. Moreover, it uses `pos()` to obtain the position of the found code element, handling the transformation of the position from a list of offsets to a file and a character range.

IV. Using HyperAST

To demonstrate the concrete capabilities of *HyperAST* at mining common code metrics, as illustrated by Listings 1 and 2, we preprocessed the Mcc and LoC metrics and searched for the top method in terms of them. We used 20 source code histories from our previous *HyperAST* work [2], [3]. For each project we swiftly mined 100 commits on the main branch. To give a good idea of *HyperAST* responsiveness, we measured the cumulative time to reach a certain commit depth. The results were obtained on a 13" M1 MacBook Pro from 2020 with 16GB of RAM.

Figure 2 presents 11 projects (in order of LoCs of their last commit). Each project is presented separately, with commits shown in depth order over x-axis. Multiple y-axes are presented, one for the cumulated processing time, one for LoCs, and one for Mcc. We can observe that more complex queries are more expensive. Even on Hadoop the *HyperAST* was capable of computing 100 commits in 500 seconds.

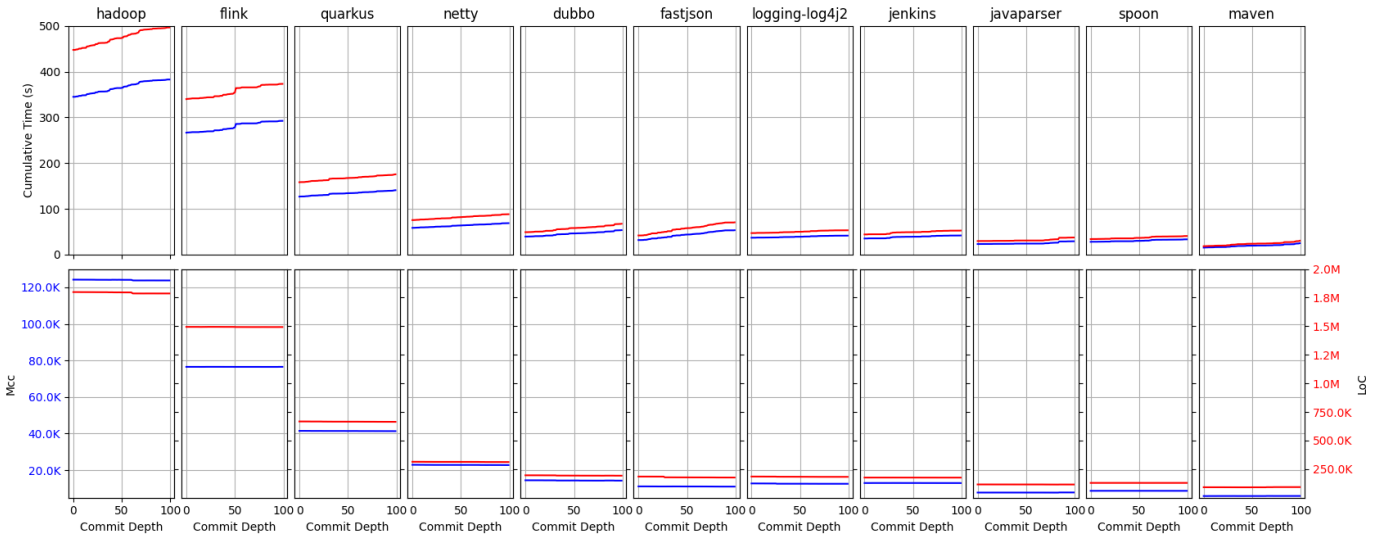

Figure 2: Using *HyperAST* to observe the evolution of Mcc and LoC metrics over 100 commits of main branches. Top plots show cumulated time taken to mine the metric. Bottom plots show the values of measured metrics. Legend: [Mcc: blue, LoC: red]

V. Availability

Our Tool is available on GitHub https://github.com/HyperAST/HyperAST, with a reproduction package on *Zenodo* [7] and through the main Rust package manager https://crates.io/crates/hyperast. The presented experiments can be found in the top level benchmarks. The different `examples/` and `bench/` directories found through the codebase are also good entry points to learn how to use the *HyperAST*.

VI. Summary

In this paper we have presented the *HyperAST* framework, which enables fine-grained temporal analyses of large code histories, bridging the gap between file mining (through commits in VCSs, *e.g.,* Git) and AST mining (that leverage compilers, *e.g.,* JDT). At its core *HyperAST* efficiently provides the AST representation of source code in large software histories. It supports designing incrementally computed code metrics and predictive AST searches, while enabling precise temporal code analyses through tree-based diffs. We presented an abstraction to properly define the computation of metrics and traversal using these metrics to skip subtrees. This abstraction is available through two sandboxed scripting languages, to guarantee the locality invariant of the *HyperAST*. It also enables to use *HyperAST* as a database with robust remote querying capabilities. To illustrate the capabilities of our approach we provide the dataset of selected metrics computed on commits of large source code histories. This dataset was made on a common laptop, *HyperAST* is intentionally focussed on efficiency, to illustrate how a large audience could build their own dataset and execute investigations without resorting to powerful computers.

We hypothesize that *HyperAST* has the potential to facilitate source code repository mining, enabling practitioners to focus on the fundamental aspects of code processing, specifically the trade-off between memory and compute, carefully balancing between eager and lazy computing of metadata. The *HyperAST* can already be used to measure fine-grained code metrics on large code bases over multiple years or decades of development. We also plan to continue extending the *HyperAST* capabilities — to further bridge the gap between textual and AST-based mining—, providing more languages, robust semantic analyses (name-resolution and def-uses), and more ergonomic code queries.

Acknowledgment

References

[1] T. Szabó, "Incrementalizing production codeql analyses," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1716–1726.

[2] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "Hyperast: Enabling efficient analysis of software histories at scale," in *37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 2022, pp. 1–12.

[3] ——, "Hyperdiff: Computing source code diffs at scale," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*, 2023.

[4] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Trans. Software Eng.*, vol. 48, no. 3, pp. 930–950, 2022.

[5] C. V. Alexandru, S. Panichella, S. Proksch, and H. C. Gall, "Redundancy-free analysis of multi-revision software artifacts," *Empirical Software Engineering*, vol. 24, no. 1, pp. 332–380, 2019.

[6] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 313–324.

[7] Q. Le Dilavrec, "Hyperast," Feb. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14810468