



**HAL**  
open science

## Analyser efficacement de grands historiques de code avec HyperAST : une démonstration

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

► **To cite this version:**

Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, Jean-Marc Jézéquel. Analyser efficacement de grands historiques de code avec HyperAST : une démonstration. 2023, pp.2. hal-04163509

**HAL Id: hal-04163509**

<https://inria.hal.science/hal-04163509v1>

Submitted on 17 Jul 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# Analyser efficacement de grands historiques de code avec *HyperAST*: une démonstration

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

Univ Rennes, IRISA, Inria, CNRS, INSA, Rennes  
{prénom.nom}@irisa.fr

**Abstract.** Cette démonstration présente l'*HyperAST* [1], une approche d'analyse d'historiques de code performante, se fondant sur la redondance du code à travers le temps et l'espace, ainsi que sur les possibilités d'analyses partielles de code. Actuellement, l'analyse des historiques de code se fait en mode « batch » : chaque version est traitée indépendamment des autres pour calculer un ensemble de métriques ; à la fin de l'analyse ces métriques sont utilisées pour observer l'évolution de la base de code au cours du temps. Notre approche propose de traiter plus finement l'historique de code, au niveau de l'AST, et de partager les éléments identiques dans et entre chaque version. Cette démonstration vise à expliquer ce principe au travers de trois scénarios.

**Keywords:** Mining Software Repositories · Temporal Code Analysis · Code Quality · GT VL · GT CLAP.

## 1 Présentation de l'*HyperAST*

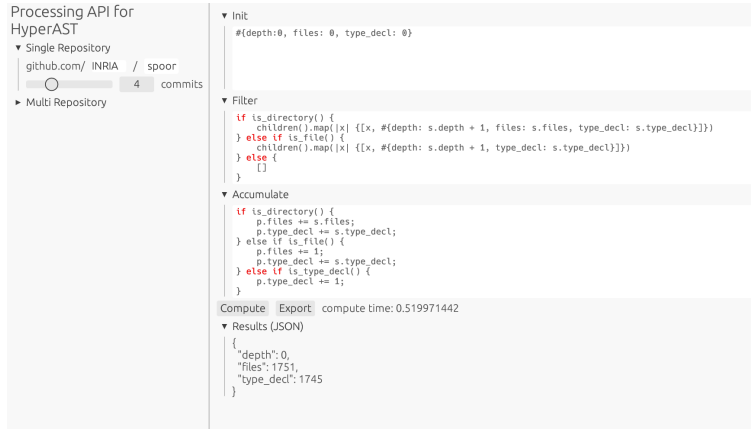
L'*HyperAST* propose de traiter les historiques de code au niveau de l'arbre de syntaxe abstraite (AST), et de partager les éléments identiques dans et entre chaque version afin de pouvoir stocker de manière optimisée de larges historiques de code. Cela aboutit à une structure de graphe direct acyclique, i.e. un arbre où chaque sous-arbre peut avoir plusieurs parents. Il est ensuite possible de persister les résultats intermédiaires calculés sur les sous-arbres de code, et ainsi rendre incrémental le calcul de métriques sur un historique de code.

Dans son état actuel, l'*HyperAST* permet de traiter un historique de code Git, notamment des projets Java ou C++. Il incorpore des métriques simples de code, et dans le cas du Java une analyse des def-use (trouver toutes les références à une déclaration). Dernièrement, il permet aussi de suivre des éléments de code dans le temps et de calculer la liste des changements entre deux commits. Finalement, nous avons développé une interface graphique pour faciliter la prise en main de différentes fonctionnalités.

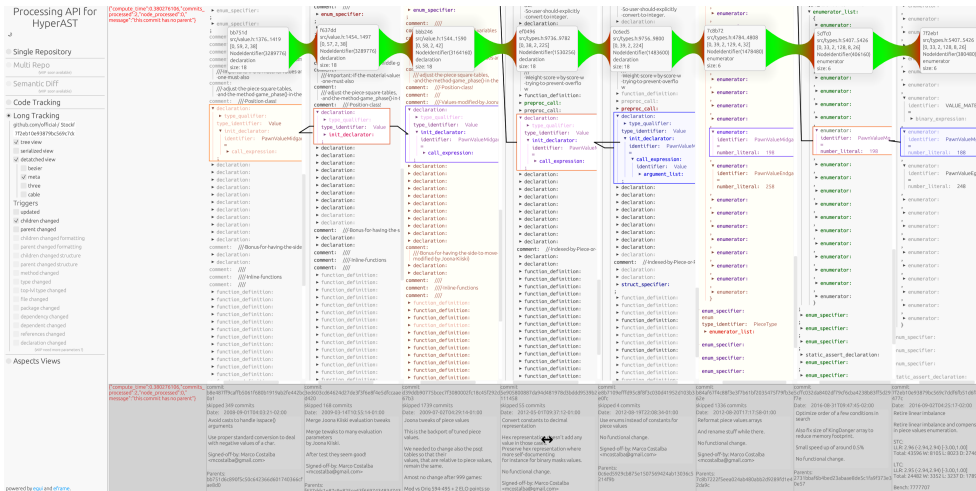
## 2 Description de la Démonstration

La démonstration impliquera trois scénarios d'utilisation de l'*HyperAST*.

- Le premier scénario montre comment calculer une métrique de code sur un commit. Il est d'abord possible de modifier un petit script et de l'exécuter sur un ou plusieurs commits, puis d'observer le résultat rendu et son temps de calcul. Ensuite, il est possible d'utiliser des métadonnées pré-calculé (i.e. des résultats intermédiaires de calcul stockés dans l'*HyperAST*) pour obtenir le même résultat, mais en un temps réduit. Finalement, il est possible d'ajouter une métadonnée à pré-calculer, pour ensuite s'en servir. Cet exemple, en plus de montrer un calcul typique de métrique sur une base de code, met en lumière l'augmentation du coût en calcul avec le nombre de commits et la taille du code, pour ensuite exposer comment limiter cette augmentation avec des métadonnées stockées dans l'*HyperAST*.



(a) Édition et calcul d'une métrique



(b) Tracking d'un élément de code sur plus de 3000 commits

Fig. 1: Captures d'écran de l'interface graphique de l'HyperAST

- Le second exemple montre comment ajouter une grammaire d'un langage à l'HyperAST. Plus particulièrement la réutilisation de grammaires Tree-sitter<sup>1</sup>. Ainsi, nous montrons comment se servir de notre outil sur de nouveaux historiques de code, non pris en charge précédemment.
- Le troisième exemple montre comment suivre un élément dans un historique de code, une des dernières fonctionnalités ajoutées à l'HyperAST. Cette fonctionnalité peut être utilisée en remplacement du « blame » de Git. Contrairement à l'approche traditionnelle qui se fonde sur le texte, notre approche se fonde sur l'AST. Nous montrons les améliorations ainsi permises, notamment pour suivre le déplacement d'un élément d'un fichier à un autre, mais aussi en contrôlant finement quand l'élément ou son entourage ont su samment changé pour en notifier l'utilisateur.

L'outil est disponible en accès ouvert sur <https://github.com/quentinLeDilavrec/HyperAST>, et contient des exemples en plus des instructions pour démarrer l'environnement graphique.

## References

1. Quentin Le Dilavrec, Djamel Eddine Khelladi, Arnaud Blouin, and Jean-Marc Jézéquel. HyperAST: Enabling Efficient Analysis of Software Histories at Scale. In *Proc. of ASE 2022*, pages 1–12. IEEE, 2022. URL: <https://hal.inria.fr/hal-03764541>.

<sup>1</sup> <https://tree-sitter.github.io/>