進捗_2025/07/02

研究テーマの内容と現状の進捗について報告する。
第 11 回のディベートに投稿した内容を包含しながらも、進捗等を含むより詳細な報告を付け足した内容となっている。

研究テーマについて

テーマ研究として AST のフラット化によるパフォーマンスの差異の計測とその結果に対する考察を行う。
Hagemeister2025 では、JavaScript ランタイムの1つ、 Deno のリンターである deno lint において、AST と呼ばれる木構造のデータに対して「フラット化」と呼ばれる処理を行うことにより、効率的に木の探索を行っている。
本研究ではフラット化前の AST とフラット化後の AST を同一の探索順で全探索し、そのパフォーマンスやコストを計測、差異を評価しそれに対する考察を行う。

計測内容について

Hagemeister2025 では JS Engine で AST のデシリアライズを排除し、受信したバッファを直接走査することでパフォーマンスを改善したと説明している、これはすなわち `JSON.parse` をによる再帰的なツリー構造を走査するコストとフラット化した AST を走査するコストを比べたとき、フラット化した AST の走査がパフォーマンス上優位な結果が得られたということである。
今回の研究ではこの点についての検証と考察を行うこととする。

AST とフラット化した AST それぞれに対して全体を一度だけ走査する関数を実装し、同一の AST に対して実行することで計測を行う。
具体的には、走査処理の総実行時間、AST のメモリ使用量、アロケーション回数の取得を考えている。
それぞれの計測項目の意義や手法について、現在調査できている内容を示す。

走査処理の総実行時間

Rust の標準ライブラリである `std::time::Instant` [1] を用いて関数の開始から終了までの経過時間を取得する。
パフォーマンスとして扱われる一番身近なもののひとつとして、総実行時間の計測を行う。

メモリ使用量

走査対象となる AST 、フラット化した AST それぞれをメモリ上に構築した場合に必要になるメモリ使用量を取得する。
データ構造の効率性を評価するためにメモリ使用量の計測を行う。

`std::mem::size_of_val` [2] を利用して動的にメモリ使用量を取得することができる。
一般的な話として、構造体で利用する型によって大きくメモリ使用量が変わってしまう可能性がある。
AST の構造体を定義する際は必要十分な機能を持つ型定義の中でメモリ使用量が最小になるように実装することを目指す。

アロケーション回数

走査中に発生したメモリアロケーションの回数を取得する。
アロケーション回数を計測することでメモリ管理のオーバーヘッドを計測することができる。
Hagemeister2025 で説明されていた facade class の実装によるノードの実体化を必要最低限に抑える手法もアロケーション回数を削減する手法の1つであり、アロケーション回数はデータ構造の違いによって生じるパフォーマンスの差異に関係するデータであると考える。

Shiisaa Moriai2020 によると、独自のアロケータ実装を `global_allocator` 属性で登録することでメモリアロケータとして独自実装を利用できる。
グローバルに定義したカウンタ変数をインクリメントした後で `std::alloc::System.alloc` [^3] を実行することで、デフォルトのメモリアロケータを実行しながら回数のカウントができる。
これを実装することで計測を行う。

実装の進捗

通常の AST に対して BFS を実行するコードは完成している。
`code\src\main.rs` に実装があり、 `bfs_ast` が BFS で走査を実行するための関数である。
独自で定義したミニマムな AST ノードを表す構造体で AST を表現し、限定的な実装になっている。
JavaScript のソースコードとして `if(condition) { foo(); }` を利用してテストを行っており、当該ソースコードから生成される AST での動作確認のみ完了している。

実装済みの AST ノードの種類が少なく、より大規模なソースコードを扱うには追加での実装が必要になる。
また、各種計測のためのコード、フラット化 AST に対して BFS で走査を行う関数、AST から フラット化 AST を生成する関数が不足している。

本レポート執筆時点でのソースコード全文は下記を参照いただきたい。
https://github.com/petaxa/cu-seminar/tree/887374ed31ab39b1b834467e0b8d323f8b9dc5f0/theme-research/code

今後の予定

計測内容について で示した通りの計測を行えるようにコードを書き足していく。
以下の優先順位に基づき、実装を進める。

1. フラット化 AST に対して BFS で走査を行う関数を実装
2. 走査処理の総実行時間を計測する関数を実装
3. メモリ使用量を計測する関数を実装
4. アロケーション回数を計測する関数を実装
5. フラット化 AST を生成する関数を実装
6. より多くの AST ノードに対応

参考文献

Hagemeister2025

Marvin Hagemeister. "Speeding up the JavaScript ecosystem - Rust and JavaScript Plugins". marvinh.dev. 2025. https://marvinh.dev/blog/speeding-up-javascript-ecosystem-part-11/. (参照 2025-06-24)

Moriai2020

Shiisaa Moriai. "Rustのメモリアロケーションをちょっとだけ掘ってみた". Qiita. 2020. https://qiita.com/moriai/items/4e2ec2d9c3b352394ef3. (参照 2025-07-03)

1. リファレンス: https://doc.rust-lang.org/std/time/struct.Instant.html ↩
2. リファレンス: https://doc.rust-lang.org/beta/std/mem/fn.size_of_val.html ↩