テーマ研究: 研究発表

ASTのフラット化による探索パフォーマンスの差異とその考察

学籍番号: 2101105460 市村 悠馬

目次

1. テーマの説明
2. 計測項目と手法
3. 得られた結果と考察
4. まとめ

テーマの説明

AST（抽象構文木）のフラット化による探索パフォーマンスの差異を計測し考察する

背景説明

Rust製JavaScriptツールチェインであるdeno_lintで実際に行われているパフォーマンスチューニングの手法

背景説明

AST(抽象構文木)とは

- プログラムコードを木構造で表したもの
- 静的な解析はこの木を読むことで行う

```
if(condition) { foo(); }
```

```
const ast = {
  type: "IfStatement",
  test: { type: "Identifier", value: "condition", optional: false },
  consequent: {
    type: "BlockStatement",
    stmts: [
      { type: "ExpressionStatement", expression: { /*...*/ } }
    ]
  }
};
```

背景説明

- deno_lintではJSプラグインでの拡張を提供しているため、RustとJSの間でASTをやり取りしなければならない
- デシリアライズのオーバーヘッドが大きくて時間がかかる
- → 「フラット化」によってそれを解消する

フラット化とは

AST構造をフラット（平坦）な配列に変換する手法

- 木構造をフラットな配列形式に変換
- 親子関係はインデックス番号で表現
  - 子のインデックス番号、兄弟のインデックス番号、親のインデックス番号を持つ
  - → n個目のノードのインデックス番号は `n * 4` で求まる
- 走査効率の向上が期待できる

フラット化とは

Pure-AST

```
const ast = {
  type: "IfStatement",
  test: { type: "Identifier", value: "condition", optional: false },
  consequent: {
    type: "BlockStatement",
    stmts: [
      { type: "ExpressionStatement", expression: { /*...*/ } }
    ]
  }
};
```

Flatten-AST

```
// フラット化されたAST（配列とインデックスで関係性を表現）
const ast = {
  stringTable: ["", "IfStatement", "Identifier", /*...*/],
  properties: [{}, { test: 2, consequent: 3, /*...*/ }, /*...*/],
  nodes: [0, 0, 0, 0, 1, 2, 0, 0, /*...*/]
};
```

研究内容の詳細

- フラット化前のAST構造とフラット化後のAST構造について、**同一の探索アルゴリズムを用いて性能比較**を行う
  - 同一のソースコードから生成したAST（通常・フラット化）に対して、同じ探索経路をたどる

余裕があったら...
なるべく大きなコード(例えばTSのchecker.tsなど)のASTで実験を行いたい

計測項目: 走査処理の総実行時間

計測対象

- 探索関数の開始から終了までの時間

理由

- パフォーマンス評価の基本指標
- 実用的な性能差の把握

手法

- Rustの `std::time::Instant` を使用
- 複数回の実行による平均値計測

```rust
use std::time::Instant;

fn measure_traverse_time(ast: &Ast) -> Duration {
    let start = Instant::now();
    traverse_ast(ast);
    let elapsed = start.elapsed();

    elapsed
}
```

計測項目: アロケーション回数

計測対象

- 走査中のメモリ割り当て回数

理由

- メモリ管理オーバーヘッドの評価
- 動的確保による性能影響の測定

手法

- カスタムアロケータによる計測
- `#[global_allocator]` 属性を利用

```
struct CountingAllocator;

static ALLOCATION_COUNTER: AtomicUsize =
    AtomicUsize::new(0);

unsafe impl GlobalAlloc for CountingAllocator {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        ALLOCATION_COUNTER.fetch_add(1,
            Ordering::SeqCst);
        System.alloc(layout)
    }
    // ...
}

#[global_allocator]
static ALLOCATOR: CountingAllocator =
    CountingAllocator;
```

得られた結果 -走査処理の総実行時間

実行時間の比較（5回測定）

実行回数	通常AST	フラットAST
1回目	45.9µs	8.9µs
2回目	27.0µs	10.0µs
3回目	25.0µs	8.8µs
4回目	25.3µs	8.7µs
5回目	26.0µs	13.9µs
平均	**29.84µs**	**10.06µs**

※ Release Profile（cargo run --release）での実行結果

※ コミットハッシュ `a38f4eeb` のソースコードでの結果

得られた結果 -走査処理の総実行時間

実行時間の比較（5回測定）

実行回数	通常AST	フラットAST
平均	**29.84µs**	**10.06µs**

約66%向上

考察: パフォーマンス向上の理由

メモリアクセスの局所性が高まったから

- pure-AST: 各ノードをヒープメモリ上に個別にアロケート
  - キャッシュミスが頻発
- flatten-AST: データをメモリ上で連続的に配置
  - CPUキャッシュを有効活用

まとめ

- フラット化前のAST構造とフラット化後のAST構造で同一の走査を行って性能を比較
- 計測項目は2種類
  - 走査処理の総実行時間
  - アロケーション回数
- 走査処理の総実行時間は約66%向上した
  - 理由はメモリアクセスの局所性が高まったからだと考えられる

参考文献

- 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)
- 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)