КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ

UDC 004.4`4
DOI https://doi.org/10.36994/2788-5518-2025-01-09-08

SEPARATING STRUCTURE AND DATA IN ABSTRACT SYNTAX TREES FOR EFFICIENT IMMUTABLE TRANSFORMATIONS

Bilyk V. R., *IT STEP University, Lviv, Ukraine. https://orcid.org/0009-0002-0173-0120, vladbilyk@posteo.net*

Abstract. *In compiler design, the abstract syntax tree (AST) is the primary data structure used to represent the syntactic structure of a program during translation and analysis. Traditional AST representations model structure and data as a single entity, where structural relationships are encoded implicitly via object references, and data is embedded directly within the objects. While intuitive, this approach complicates the design and implementation of AST transformations, particularly under immutability constraints. Even simple modifications often require explicit AST traversal and extensive copying, introducing accidental complexity that may obscure the essential logic of transformations and reduce performance.*

This work explores an alternative approach that explicitly separates structure from data in the AST representation. The internal form of the AST is modeled as a pair of an array of data-only nodes and an adjacency matrix that explicitly encodes the AST structure. This separation enables efficient transformation by enabling independent updates to data without modifying the structure. Structural changes can also be performed in a principled manner by modifying the adjacency matrix and the node array independently. This separation of concerns simplifies reasoning about transformations and enables concise expression without the accidental traversal logic.

To demonstrate the utility of this design, a compiler for a minimal data query language is presented. Three representative AST operations are explored: verification, data-only updates, and structural modification, illustrating how different kinds of transformations benefit from the proposed model. The proposed approach is compared to traditional object-oriented representations and existing rewriting systems. The extensibility and first-class role of the proposed design makes it applicable to a wide range of compiler tasks beyond the example presented. This model serves as a useful foundation for building compilers and transformation tools that are both robust and practical. Future directions include optimizing the internal representation and supporting alternative traversal strategies.

Key words: *compilers, abstract-syntax tree, transformation, immutability, Java.*

ВІДОКРЕМЛЕННЯ СТРУКТУРИ ВІД ДАНИХ В АБСТРАКТНИХ СИНТАКСИЧНИХ ДЕРЕВАХ ДЛЯ ЕФЕКТИВНИХ ПЕРЕТВОРЕНЬ НЕЗМІННИХ ОБ'ЄКТІВ

Владислав Білик*, ІТ СТЕП Університет, Львів, Україна. https://orcid.org/0009-0002-0173-0120, vladbilyk@posteo.net*

Анотація. *У дизайні компіляторів абстрактне синтаксичне дерево (АСД) є основною структурою даних, яка використовується для представлення синтаксичної структури програми під час трансляції й аналізу. Традиційні представлення АСД моделюють структуру й дані як єдине ціле, де структурні зв'язки неявно кодуються через посилання на об'єкти, а дані є частиною самих об'єктів. Хоча цей підхід є інтуїтивним, він ускладнює моделювання та реалізацію перетворень АСД, особливо для незмінних об'єктів. Навіть прості модифікації часто потребують явного обходу АСД та значного копіювання, що додає другорядну складність до перетворень і знижує швидкодію.*

У роботі досліджується альтернативний підхід, який явно відокремлює структуру від даних у представленні АСД. Внутрішня форма АСД моделюється як пара масиву вершин, що містять лише дані, і матриці суміжності, яка явно кодує структуру АСД. Така модель уможливлює ефективні перетворення завдяки незалежним оновленням даних без зміни структури. Структурні зміни також є можливими, виконуються незалежно, змінюючи матрицю суміжності й масив вершин. Таке відокремлення спрощує розуміння перетворень та уможливлює лаконічне вираження без другорядної складності, що стосується обходу АСД.

Аби продемонструвати застосовність наведеного підходу, представлено компілятор для мінімальної мови запитів до даних. Дослідили три типові операції над АСД: верифікацію, оновлення лише даних, структурну модифікацію, ілюструючи, як така модель сприяє різним видам перетво-

рень. Запропонований підхід порівнюється з традиційними об'єктно-орієнтованими представленнями й наявними системами рерайтингу. Розширюваність і першокласна природа запропонованої моделі роблять її застосовною до низки компіляційних процесів, що виходять за рамки представленого прикладу. Ця модель слугує основою для створення надійних, практичних компіляторів і засобів перетворення. Серед майбутніх напрямів дослідження – оптимізація внутрішнього представлення АСД та підтримка альтернативних стратегій обходу.

Ключові слова: *компілятори, абстрактне синтаксичне дерево, трансформація, незмінні об'єкти, Java.*

Introduction

Programming languages provide the ability to describe computations at a high level of abstraction. For a program to be executed, it must ultimately be interpreted or translated into a lower-level language – typically machine code that will be executed directly by a CPU. This translation is the task of a compiler, which transforms source code from one representation into another. While it is typical for compilers to target machine code, the codomain of compilation may be arbitrary. A compiler may translate one high-level language into another to connect different domains, facilitating integration with external systems or leveraging specialized execution engines (e.g., compiling a domain-specific language embedded in Java [1] into SQL for database execution).

The compilation process is centered around an abstract syntax tree (AST) – a core data structure that represents the syntactic structure of the source program. After parsing, the AST becomes the intermediate representation upon which subsequent compiler stages operate. These stages typically include verification, optimization, and code generation, each of which may involve some form of analysis or transformation of the AST.

The process of transforming an AST is often the most complex and performance-critical part of a compiler. This work focuses on improving how AST transformations are defined and performed, specifically in the context of immutable data structures.

Two key challenges arise during AST transformation:

1. **Simplicity of definition and implementation.** Ideally, the definition of a transformation should be concise, tackling the core of the problem, without having to explicitly encode the rules for AST traversal. Approaches such as *rewriting systems* [2] and *source transformation systems* [3] try to achieve this goal by separating the core transformation logic from the details of the strategy for matching parts of an AST and applying the changes. This principle is in line with Brooks's distinction between *essential* and *accidental* complexity [4].

2. **Performance under immutability.** When all data is immutable – an increasingly common choice in modern software engineering [5] – informal reasoning becomes easier at the cost of performance penalties of varying degree. In compiler design, each AST transformation must produce a new AST object. Naïve implementations that rebuild entire trees on each change are inefficient. Techniques such as sharing [6], the Zipper [7], or flattened AST representations [8] aim to optimise operations on immutable data.

This work explores an alternative approach: decoupling AST structure and data into separate representations, enabling more localized and efficient updates while preserving immutability. We propose a model for AST design and transformation that explicitly separates structure from data. It is shown how this design simplifies the definition of AST transformations without incurring heavy performance penalties, and how it aligns with the goals of clarity, locality, and immutability. These ideas are illustrated using a minimal compiler for a data query language, implemented in Java.

Query Language

To illustrate the proposed approach, we introduce a minimal language for data querying, called Query Language (QL). QL serves as the source language in our compiler, with SQL as its target. While the final compilation stage that emits SQL code is beyond the scope of this work, we focus on intermediate stages involving AST transformations.

The syntax of QL is defined using a grammar in extended Backus–Naur form (BNF):

```
Query   ::= (from Source)? yield Yield+
Source  ::= <name:string>
Yield   ::= Expr (as <alias:string>)?
Expr    ::= Val | Column | Query
Val     ::= <value:object>
Column  ::= <name:string>
```

This grammar uses conventions extended from traditional BNF: capitalized identifiers represent non-terminals, lowercase ones represent terminals, and postfix operators "?", "*", and "+" denote optionality, repetition (zero or more), and repetition (one or more), respectively. Angle bracket notation *<label:type>* (e.g., *<name:string>*) denotes arbitrary data of a corresponding type with a label for descriptive purposes. *<value:object>* denotes a literal value of an arbitrary type supported by QL, which is expected to include common data types used in SQL: temporal types (date, datetime), numeric (integer, decimal), etc.

Here are a few example queries that conform to the QL grammar:

1. *from users yield name, email*
2. *yield 1*
3. *from users yield (yield 1) as id, “admin” as name*

To demonstrate AST transformation in this context, three key operations are defined that a QL compiler might perform:

1. **Scalar subquery verification.** A scalar subquery is a *Query* node that appears within a *Yield* node. Since such subqueries are intended to produce a single scalar value, they must contain exactly one *Yield*. If this condition is violated, compilation should fail with an error. This is an example of a *verification* operation.
2. **Alias renaming.** This operation changes the alias associated with a *Yield* node, without modifying its structure. This is an example of a *data-only transformation.*
3. **Subquery inlining.** If a scalar subquery yields a constant *Value* and does not use a *Source*, it is semantically equivalent to that value and can be inlined by replacing the *Query* with the *Value*. This is an example of a *structural transformation*.

These operations serve as specific use cases for the approach developed in the rest of this work.

A naïve approach

A common and straightforward way to model an AST is to tightly couple its structure and data in a single representation. For the Query Language (QL), such a model might look like the following:

```
interface Expr
record Query(Source source, List<Yield> yields) implements Expr
record Yield(Expr expr, String alias) implements Expr
```

In this encoding, *Query* and *Yield* are both nodes in the AST. The *Query* node is purely structural, consisting of a source and a list of yields, while each *Yield* node combines a structural sub-expression *expr* with a data component *alias*.

The limitation of this design becomes apparent when implementing AST transformations. Because the structure and data are coupled, any transformation, regardless of scope, must begin at the root node (Query) and recursively traverse the tree to locate and manipulate the desired parts.

For example, consider the implementation of operation 1, which verifies that each subquery yields exactly one value:

```
verifyScalarSubquery(Query q)
  for (Yield y : q.yields())
    if (y.expr() instanceof Query sq)
      verifyScalarSubquery_(sq);

verifyScalarSubquery_(Query q)
  if (q.yields().size() != 1)
    error();
  else // (1)
    verifyScalarSubquery(q);
```

In this implementation, the top-level *Query* node must be explicitly walked. The compiler must inspect every *Yield* node, detect subqueries, and recursively apply the same verification. Line (1) illustrates a subtle point: a subquery may itself contain nested subqueries, so the traversal logic must be repeated.

This naïve representation may suffice for a toy language like QL, but in a realistic scenario with more constructs (e.g., *Where*, *OrderBy*), the number of places where subqueries might occur would increase significantly. The programmer would have to manually extend the traversal logic for each new language construct. As a result, the complexity of defining even simple transformations grows rapidly because of the traversal overhead. This is an example of accidental complexity overshadowing the essential complexity of the task.

The proposed solution

To overcome the limitations of tightly coupled AST representations, we propose a design that explicitly separates structure from data, enabling AST transformations to focus on the essential complexity, simplifying both the definition and execution of transformations under immutability constraints.

The AST is defined using interfaces that distinguish structural and data components. This separation is expressed through method declarations, with methods representing data annotated using the *@Data* annotation:

```
interface Query
  Source source();
  List<Yield> yields();

interface Yield
  Expr expr();
  @Data String alias();
```

Structural relationships, such as between *Yield* and *Expr*, are represented by unannotated methods, while data components, such as *Yield.alias*, are explicitly marked. These interfaces are processed at runtime to produce an internal representation of the AST, which is hidden from the programmer. The implementation details of this internal data structure will be presented in a subsequent section.

Defining transformations

An AST is defined by a set of node types. An AST transformation is a function from ASTs to ASTs – an endomorphism over the AST domain.

For example, in the QL compiler, let the AST type be *QL*. Then, any transformation can be expressed as a function *f: QL → QL.* This general definition, while expressive, is often impractical for defining specific transformations that target specific node types instead of the whole AST. Therefore, a notion of specific transformations, which are then lifted into general transformations via structural traversal, is also required.

A specific AST transformation is characterized by:

1. **A node type as the entry point.** Let *g* be a transformation defined over a node type *N*. It can be lifted to an AST transformation *f* by applying *g* to every subtree whose root is a node of type *N*. For example, renaming a yield alias (operation 2) is defined over *Yield* nodes, but can be lifted to operate on the entire AST by applying it to all nodes of type *Yield.*

2. **The nature of the change.** Transformations may involve two kinds of changes: *data-only changes*, which modify just the contents of a node, without altering the tree structure; *structural changes*, which affect the relationships between nodes (e.g., replacing one subtree with another).

Some operations, such as verification, may have no structural or data changes, relying solely on side effects (e.g., error reporting). Below it is shown how each of the three QL transformations introduced earlier can be implemented using this model.

Operation 1: Scalar subquery verification

```
verifyScalarSubquery(Yield yield)
  if (yield.expr() instanceof Query q && q.yields().size() != 1)
    error();
```

This operation is defined on *Yield* nodes and reports an error whenever a subquery does not match the requirements. Since the traversal is automatically handled by the abstraction, the implementation is minimal and free of accidental complexity. Only the essential logic needs to be specified, no traversal logic is required. This resembles a simple form of pattern matching in rewriting systems, where the transformation is applied only at relevant nodes, decoupled from traversal concerns.

Operation 2: Alias renaming

```
renameYield(Yield yield, String oldAlias, String newAlias, Yield.Factory yieldFactory)
  if (yield.alias().equals(oldAlias))
    replace(yield, yieldFactory.make(newAlias));
```

This operation demonstrates a data-only transformation: it changes the alias of a *Yield* without affecting its structure. The abstraction also provides a factory for each node type, which is simply a data constructor. This enables the programmer to construct new nodes using only the relevant data components. In this case, since *Yield* has only one data component (*alias*), the factory method has a single corresponding parameter. The structural component (*expr*) remains unchanged and can be reused in the resulting AST.

Operation 3: Subquery inlining

```
inlineSubquery(Yield y)
  if (y.expr() instanceof Query q && q.yields().size() == 1 && q.yields().getFirst() instanceof Value v)
    replace(y.expr(), v);
```

This example illustrates a structural transformation. When a scalar subquery yields a constant value and has no source, it can be inlined by replacing the subquery with the value it yields. Thanks to the abstraction, this replacement modifies the AST structure without requiring the programmer to manually reconstruct the entire tree. Notably, the transformation only replaces the relationship between a *Yield* and its *Expr*, leaving the *Yield* node itself unchanged. This illustrates the benefit of separating structure from data: structural updates can be performed without touching the data, while retaining the ability to compose them with data changes.

AST encoding

The internal representation of an AST consists of two components:

1. An array of nodes, where each element contains only data. The nodes are stored in the order of a depth-first pre-order traversal of the AST. This array captures the contents of the AST while remaining independent of its structure.

2. An adjacency matrix that encodes the structural relationships between nodes. The matrix is of size *n x n*, where *n* is the number of nodes in the array. For a given parent-child relationship between nodes *x* and *y*, the matrix entry at position *(i, j)* is present if node *x = a[i]* is the parent of node *y = a[j]*.

An AST is defined as a tuple *(a, m)*, where *a* is the array of data-only nodes, and *m* is the adjacency matrix capturing the structure.

Alternatively, an adjacency list could be used instead of an adjacency matrix, which would be most optimal for large but sparse ASTs, where the number of nodes tends to be large but the number of connections between nodes tends to be small. If ASTs do not have a clear tendency towards sparsity or density, an adaptive approach could be used, where the representation would be chosen dynamically based on the properties of the parser's input, such as its length. For illustrative purposes, an adjacency matrix is used, as it is trivial to map the transformations from one representation onto another.

Data-only changes

When performing data-only transformations, the structure of the AST remains unchanged. Therefore, the adjacency matrix *m* can be reused. To update an AST *(a, m)* by replacing a node *x* with a new version *x'* that differs only in its data components, we construct a new array *a'* as a copy of *a* with *x* replaced by *x'* at the same index. The resulting AST is *(a', m)*.

For example, consider the AST in Figure 1. A data-only change that renames the alias *"n"* to *"num"* in the *Yield* at index 2 produces a new AST as shown in Figure 2. Since the structure is unchanged, the same adjacency matrix is retained, and only the array of nodes is updated. This example illustrates the alias renaming operation presented earlier.

The following algorithm describes the necessary steps to perform this AST transformation:

1. Copy the array of nodes *a* into a new array *a'*.
2. For each *Yield y* in *a*, call the transformation function, and let its result be *y'*.
3. For each *y,* let its index in the array be *i*. Write *y'* to *a'[i].*
4. The result is a new AST – *(a', m)*, where *m* is the original adjacency matrix.

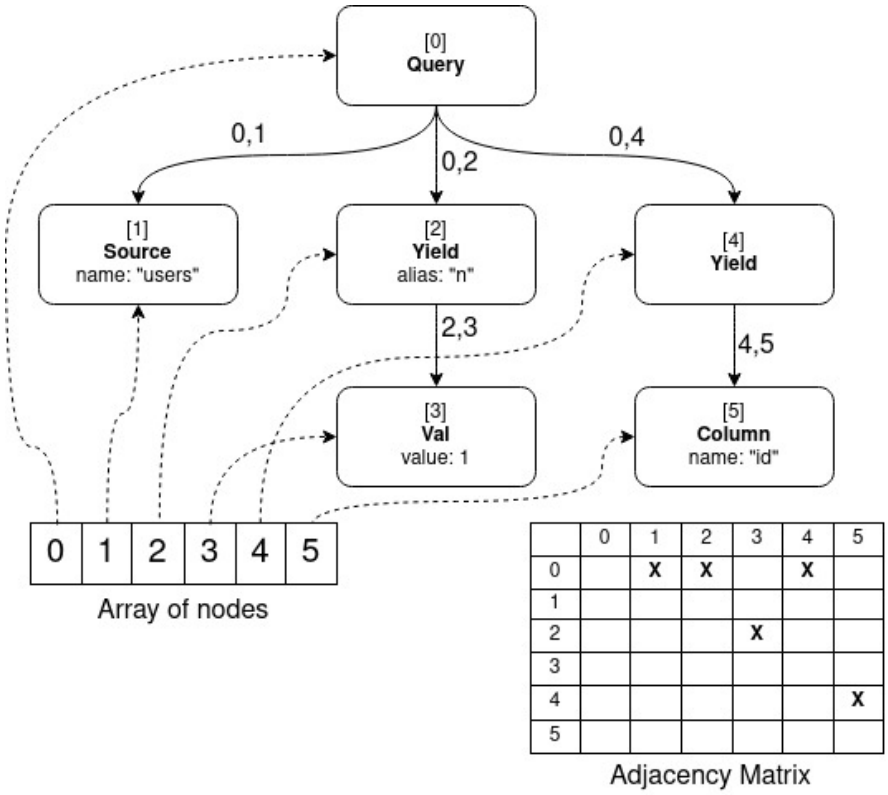

Fig. 1. AST before the data-only change

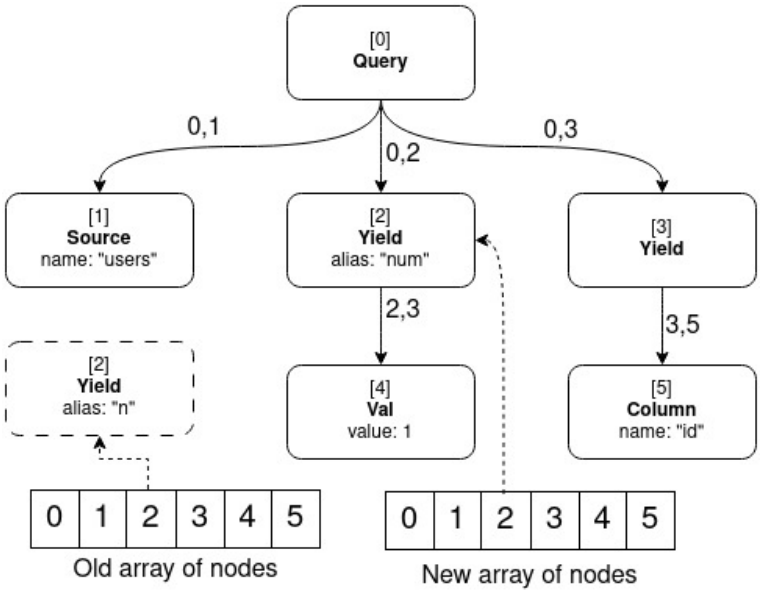

Fig. 2. AST after the data-only change

Structural changes

In contrast, structural transformations affect the shape of an AST and thus require modifications to both the node array and the adjacency matrix.

To replace a subtree rooted at node *x* with a new subtree rooted at *y*, the following steps are taken:

1. Identify subtrees *Tx* and *Ty*, rooted at *x* and *y*, respectively.
2. Construct a new AST *(a', m')* where *a'* is a copy of *a*, with the nodes in *Tx* removed and the nodes in *Ty* inserted in their place; *m'* is a copy of *m*, with the rows and columns corresponding to nodes in *Tx* replaced by those for *Ty*.

For instance, consider the AST in Figure 3, where the *Query* at index 3 is replaced by the *Value* at index 5. The result is shown in Figure 4. Nodes associated with the original subtree are removed from the array, the corresponding rows and columns in the adjacency matrix are pruned, and matrix entries are updated to reflect the new structure. This example illustrates the subquery inlining operation presented earlier.

The following algorithm describes the necessary steps to perform this AST transformation:

1. Copy the array of nodes *a* into a new array *a'*, and copy the adjacency matrix *m* into a new matrix *m'*.
2. For each *Yield y* in *a*, call the transformation function, and let its result be *(old, new)*, where *old* is the node to be removed from the AST, and *new* is to be inserted in its place.
3. Find the index of *old* in *a*, and let it be *i*.
4. Write *new* to *a'[i]*.
5. Find the index of *new* in *a*, and let it be *j*.
6. Write the contents of *m[j]* to *m'[i]*.
7. Trace and delete orphaned nodes in *a'* and their corresponding entries in *m'*.
8. The result is a new AST – *(a', m')*.

Related work

Sampson's work [8] explores an alternative AST representation where all nodes are stored in a contiguous array, and structural relationships are maintained through integer indices rather than pointers. This "flattened" layout improves memory locality, and enables efficient allocation and deallocation. While this representation is driven by performance and memory constraints, our approach complements performance gains by explicitly separating structure and data, providing additional flexibility and clarity in transformation definitions.

Balland et al. [9] present Tom, a strategy-based term rewriting system embedded in Java. Their work emphasizes the separation of rewriting rules from traversal strategies, enabling modular and declarative transformations. While Tom is a standalone language built on top of Java, our approach fits into the Java language and can be implemented as a library. Tom relies on pattern matching over algebraic terms and strategy interpretation at runtime, while our method encodes the AST structure explicitly via an adjacency matrix and provides an interface to control data and structure separately.

Hemel et al. [10] present an architecture for domain-specific language compilation centered around code generation via model-to-model transformation based on AST rewriting. Their work is focused on modular transformation pipelines that enable modular normalization rules that operate on immutable data. Similarly

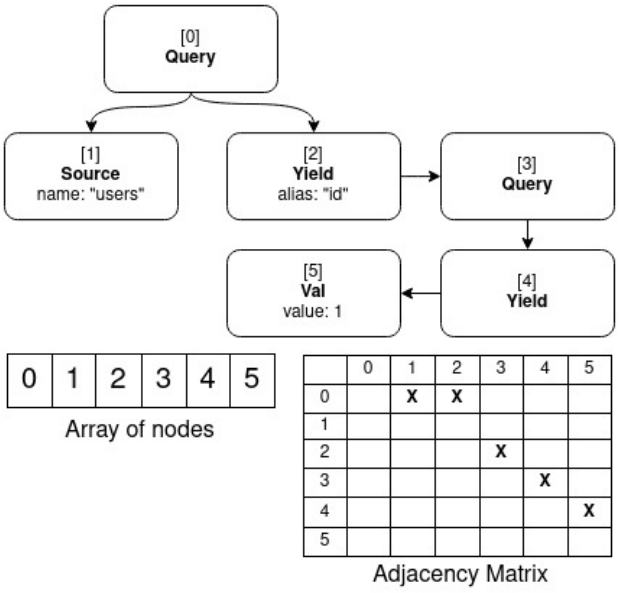

Fig. 3. AST before the structural change

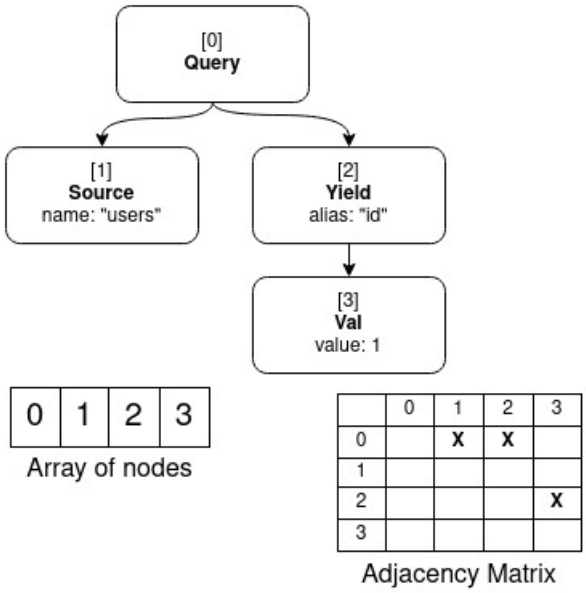

Fig. 4. AST after the structural change

to our approach, they seek to separate the essential complexity of AST transformation from traversal logic. While they focus on a standalone transformation language, our approach presents an encoding that can be integrated into Java directly.

Table 1 summarizes the key properties of the approach presented in this work in comparison with existing solutions. While the comparisons are mainly qualitative, focusing on expressiveness, ease of integration, and performance, they highlight the essential differences between the approaches.

Table 1

Comparison of AST transformation frameworks

Framework	Integration	Interface for transformations	Performance properties
This work	Library	Concise, high-level definition, decoupled from traversal	Adaptive choice of the AST representation based on language properties
Sampson [8]	Manual, must be adopted as a programming style	Unspecified	Helps avoid recursion during traversal, exploits cache locality
Balland et al. [9]	Standalone language	Rewrite rules based on pattern matching, decoupled from traversal	Transformations are optimized and compiled into Java bytecode
Hemel et al. [10]	Standalone language	Rewrite rules with custom traversal strategies	Multi-stage compilation of transformations

Conclusion

This work presented an approach to AST transformation that separates structure from data, enabling efficient operations on immutable data structures. By storing all AST nodes in an array and structural relationships in an adjacency matrix, the encoding supports localized updates without heavy performance penalties. Through AST transformation for a minimal query language as an example, it was shown how this approach eliminates accidental complexity and simplifies the implementation, supporting data-only and structural changes to the AST.

Compared to traditional object-oriented design, where transformation logic is often coupled with traversal concerns, our abstraction takes care of the accidental complexity. This facilitates informal reasoning, which is particularly important in modern software engineering. Unlike external systems that rely on additional rewriting languages, our approach stays within the Java language, providing the benefits of static typing and integration.

Future work may explore optimizations to the internal AST encoding and composability of transformations. The principles and architecture presented in this work provide a foundation for building compilers that are both simple and robust.

References

1. Ghosh, D. (2010). DSLs in action. Manning.
2. Bravenboer, M., Kalleberg, K. T., Vermaas, R., & Visser, E. (2006). Stratego/XT 0.16. In PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, 95–99. https://doi.org/10.1145/1111542.1111558.
3. Cordy, J. R., Dean, T. R., Malton, A. J., & Schneider, K. A. (2002). Source transformation in software engineering using the TXL transformation system. Information and Software Technology, 44(13), 827–837. https://doi.org/10.1016/s0950-5849(02)00104-0.
4. Brooks, F. (1987). No Silver Bullet Essence and Accidents of Software Engineering. Computer, 20(4), 10–19. https://doi.org/10.1109/mc.1987.1663532.
5. Helland, P. (2015). Immutability changes everything. Communications of the ACM, 59(1), 64–70. https://doi.org/10.1145/2844112.
6. Okasaki, C. (1998). Purely functional data structures. Cambridge University Press. https://doi.org/10.1017/cbo9780511530104.
7. Huet, G. (1997). The Zipper. Journal of Functional Programming, 7(5), 549–554. https://doi.org/10.1017/s0956796897002864.
8. Sampson, A. (2023, May 1). Flattening ASTs (and Other Compiler Data Structures). https://www.cs.cornell.edu/~asampson/blog/flattening.html.
9. Balland, E., Moreau, P., Reilles, A. (2008). Rewriting strategies in Java. Electronic Notes in Theoretical Computer Science, 219, 97–111. https://doi.org/10.1016/j.entcs.2008.10.037.
10. Hemel, Z., Kats, L. C. L., Groenewegen, D. M., & Visser, E. (2009). Code generation by model transformation: a case study in transformation modularity. Software & Systems Modeling, 9(3), 375–402. https://doi.org/10.1007/s10270-009-0136-1.

Дата першого надходження рукопису до видання: 05.05.2025
Дата прийнятого до друку рукопису після рецензування: 04.06.2025
Дата публікації: 25.07.2025