Bridging Language Barriers: A Comparative Review and Empirical Evaluation of Source-To-Source Transpilers

André Freitas
ALGORITMI Research Centre / LASI, Dept. of Informatics, University of Minho, Braga, Portugal

Tiago Baptista
ALGORITMI Research Centre / LASI, Dept. of Informatics, University of Minho, Braga, Portugal

Pedro Rangel Henriques
ALGORITMI Research Centre / LASI, Dept. of Informatics, University of Minho, Braga, Portugal

Abstract

Source-to-source transpilation plays a pivotal role in modern software engineering by enabling code migration, feature adoption, and cross-language interoperability without sacrificing semantic integrity. The contributions discussed in this paper can be split into two. The first is a comprehensive literature review that aims at defining what transpilers are, traces their historical evolution from early Fortran/COBOL preprocessors to more recent tools like Babel and TypeScript, and examines key parsing methodologies, AST representations, and transformation strategies. The second is an experimental investigation which assesses several popular transpilers – selected by GitHub popularity and unique language-pair capabilities, when applied to an equivalent code snippet designed to sum even numbers and identify the maximum element. The metrics evaluated were the execution time, CPU, memory consumption, output accuracy and usability.

2012 ACM Subject Classification General and reference → Empirical studies; General and reference → Surveys and overviews; Software and its engineering → Syntax; Software and its engineering → Semantics; Software and its engineering → Translator writing systems and compiler generators; Software and its engineering → Source code generation; Software and its engineering → Parsers; Software and its engineering → Interpreters

Keywords and phrases Source-to-source translation, Code transformation, Parsing, Lexical analysis, Syntax analysis, Semantic analysis, Transpilation

Digital Object Identifier 10.4230/OASIcs.SLATE.2025.11

Supplementary Material *Software (Tools/Scripts)*: `https://justandre02.github.io/Comparative-Review-and-Empirical-Evaluation-about-transpilers/`

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/Centro ALGORITMI.

1 Introduction

Context and Motivation

As modern software systems grow in size and complexity, developers face ever-increasing challenges in maintaining, migrating and interoperating code across heterogeneous platforms and language versions. Another challenge that companies face is related to the modernization and maintenance of legacy systems that use older languages such as COBOL (according to a survey from Micro Focus from 2022, 800 billion lines of code of COBOL are used in production) since there is a shortage of developers to maintain it and the costs of running them are high due to the need of having specific hardware infrastructure (IBM mainframe for example).

14th Symposium on Languages, Applications and Technologies (SLATE 2025).
Editors: Jorge Baptista and José Barateiro; Article No. 11; pp. 11:1–11:16
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Source-to-source compilers – commonly known as *transpilers* – have emerged as significantly important tools in this context, enabling the automated translation of code from one language or dialect into another while preserving its original semantics and intent.

Objectives

This paper has two main investigation objectives. First, it presents a comprehensive literature review, that traces the conceptual foundations of transpilation: from early **Fortran** and **COBOL** processors in the 1960s, through the advent of AST-driven program transformations, to contemporary tools such as **Babel** and the **TypeScript Compiler**. Second, we complement this theoretical foundation with an independent empirical study that evaluates a curated set of fourteen transpilers – selected by **GitHub** popularity and functional scope – across uniform test cases measuring *execution time, CPU/memory footprint, output precision and usability.*

Document Structure

By joining these two perspectives – the broad bibliographic synthesis and the hands-on performance benchmarks – the research project here reported delivers both a context on the transpilation landscape and concrete guidelines for tool selection in real-world projects. Section 2 details the review methodology and taxonomy of transpilation techniques. Section 3 describes the design of the empirical evaluation, including test configuration and metric definitions. Section 4 presents and analyses the quantitative findings, while Section 5 discusses their implications and trade-offs. The paper concludes in Section 6 with a summary of contributions and possible future work, such as AI-augmented, context-aware code transformations and deeper paradigm-shifting transpilation.

2 Literature Review: Transpilation Techniques and Frameworks

Transpilers: definition and scope

Transpilers – also known as source-to-source compilers – translate code between programming languages or language versions while preserving its original semantics. Contrarily to compilers, rather than emitting machine code, they operate at a higher level through a three-stage process: lexical and syntactic analysis to construct an Abstract Syntax Tree (AST), semantic and syntactic transformations, and generation of equivalent target code [6, 25]. This approach allows developers to adopt new language features yet maintain backward compatibility and interoperability across diverse platforms [11, 19]. Beyond pure translation, transpilers are crucial for modernizing legacy systems by supporting gradual migrations and reducing technical debt [12].

Historical Evolution of Source-to-Source Translation

The roots of source-to-source translation trace back to **Fortran** and **COBOL** preprocessors of the 1960s, which provided macro facilities and dialect conversion to optimise code for different hardware architectures [21, 7]. Early program transformation research introduced AST representations to enhance portability and optimization [14].

During the 1980s and 1990s, the emergence of functional and object-oriented paradigms drove more sophisticated transformation frameworks capable of rewriting code while preserving functional equivalence [4, 10]. Enhanced parsing techniques and pattern-based rewrite algorithms laid the foundation for modern transpilers.

The exponential growth of web applications required tools to reconcile rapidly evolving **ECMAScript** standards with a variety of browsers. Babel[1] became a benchmark by enabling developers to author in **ESNext** and transpile to **ES5**, ensuring compatibility with older environments [1]. Concurrently, **TypeScript** introduced static typing on top of **JavaScript**, further enriching the transpilation ecosystem [20].

2.1 Parsing Methodologies and AST Representations

The parsing phase can be split into lexical analysis (tokenisation) of source text into identifiers, literals and symbols, and syntactic analysis with grammar validation and parse-tree construction [5]. Top-down parsers (like the descent) are straightforward to extend but cannot handle left recursion; bottom-up parsers (LR) support more complex grammars at the expense of implementation effort [8]. Generalised LR (GLR) parsing extends this flexibility, resolving ambiguities and accommodating context-sensitive constructs [15].

A parse tree is distilled into an AST, which removes redundant syntactic details and emphasizes the programme's semantic structure. Key characteristics include a hierarchical node representation, elimination of syntactic sugar, and explicit preservation of control and data flow dependencies which are essential for language-agnostic transformations [13, 10, 9, 23].

2.2 Code Transformation Strategies

Code transformation represents the core computational process of transpilation, where the AST is systematically modified to generate equivalent code in the target language or version [27]. This phase involves multiple strategies to ensure semantic preservation while adapting to the target language's specific syntactic and structural requirements [2].

The transformation process typically encompasses **several key strategies:**

- Structural transformation of language constructs;
- Semantic mapping between different language features;
- Handling of language-specific idioms and patterns;
- Generating optimized and compatible target code.

Complex transformations may involve multiple traversals over the AST, each one addressing different aspects of code translation [27]. These can include:

- Feature adaptation *(e.g., translating modern **JavaScript** features to **ES5**)*;
- Paradigm translation (converting functional programming constructs to imperative styles);
- Semantic equivalence preservation;
- Performance optimization.

2.3 Taxonomy of Transpilers

Reading the available literature, we found that the landscape of transpilation tools can be classified according to the type of translation they perform. They usually are categorised in three classes, as will be described below.

[1] `https://babeljs.io/docs/`

Language-to-Language Transpilers

Language-to-language transpilers represent an approach to cross-language code transformation, enabling developers to translate source code between fundamentally different programming languages. These transpilers address the challenge of linguistic heterogeneity in software ecosystems, facilitating code reuse, platform migration, and technological integration [28].

One of the most popular language-to-language transpilers is **Java2Python**[2] which transpiles from Java to Python. Although a popular tool, it was forgotten over the years due to the fact it only works with Python 2 and was left with no more updates since 9 years ago.

The translation process involves semantic mapping, addressing not just syntactic differences but also paradigmatic variations between source and target languages. For example, transpilers might translate between statically-typed and dynamically-typed languages, or between languages with different memory management approaches, requiring sophisticated computational strategies to preserve the original code's computational intent [22, 16].

Version Transpilers

Version transpilers focus on managing the evolutionary challenges of programming languages by enabling code migration between different versions of the same language. These tools are particularly useful in rapidly evolving language ecosystems where backward compatibility and feature adoption present a significant challenge for developers [3].

Modern version transpilation goes beyond syntax updates, addressing semantic changes, deprecation of language features, and introducing modern language capabilities to legacy codebases. **JavaScript** transpilers like **Babel** exemplify this approach, allowing developers to use next-generation **ECMAScript** features while maintaining compatibility with older browser environments [26].

Paradigm Transpilers

The most complex category, paradigm transpilers translate across programming paradigms: functional, imperative, declarative, demanding profound semantic transformations to preserve computational logic while adapting control flow, data abstractions and side-effect management [1].

It's mentioned as the most complex set of transpilers since the gap between languages of different paradigms is often hard to bridge, requiring first to understand and design how certain statements from a source language, which are not supported or don't exist in the target language, will be mapped into the target language and then apply multiple AST transformations in order to incrementally bridge the gap between both languages. **ClojureScript**[3] is a great example of this type of transpiler because it is a compiler for **Clojure** (dynamic, general-purpose programming language) that targets JavaScript.

[2] `https://github.com/natural/java2python`
[3] `https://clojurescript.org/`

2.4 Software Infrastructure Mapping in Transpilation

Infrastructure Dependencies and Library Ecosystem Challenges

The difficult task of mapping entire software infrastructures is included in the transpilation process, which goes far beyond translating syntactic and semantic code. Standard libraries, runtime environments, package management systems, and platform-specific components are all part of the extensive ecosystems that surround programming languages and make it possible to design and implement applications.

When transpiling real-world applications, developers face the fundamental challenge of relying on substantially different infrastructure foundations. A **Java** application, for instance, depends not only on the **Java** language syntax but also on the extensive **Java Standard Library**, **Java Virtual Machine runtime**, **Maven** or **Gradle** build systems, and Java-specific frameworks. Transpiling such an application to **Python** requires mapping these dependencies to equivalent **Python** infrastructure: the **Python Standard Library**, **CPython interpreter**, pip package manager, and corresponding **Python frameworks**.

Binary Component Integration and Platform Dependencies

When working with dependencies and binary components specific to the platform that cannot be translated immediately through transformation of the source code, the difficulty increases. Database drivers, system APIs, native libraries, and third-party components that are available as built binaries rather than source code are all integrated into many production applications. Pure source-to-source translation is unable to handle the extra layers of complexity created by these dependencies.

Successful application migration in such contexts requires hybrid approaches that combine code transpilation with infrastructure redesign. This process often involves identifying equivalent binary components in the target ecosystem, developing adapter layers to maintain interface compatibility, or replacing entire subsystems with functionally equivalent alternatives that align with the target platform's architectural patterns.

3 Independent Empirical Study: Methodology and Setup

The theoretical research (presented in Section 2) carried out on various transpilers provided valuable information on the characterisation and usage of these tools. However that task paved the way to a more practical and experimental study that could support a deeper analysis of their characteristics. This section describes an experiment aimed at providing concrete results that allow for several relevant conclusions to be identified, highlighting their advantages, limitations and ideal contexts of use. Table 1 exhibits the main observations drawn from the literature review, showing the parameters that better characterise the transpilation tools selected for the empirical study.

3.1 Test Cases and Selection Criteria

The empirical evaluation uses a uniform test suite in which each transpiler is asked to translate a short programme that (a) computes the sum of the even numbers in a given list and (b) finds the maximum value among the list elements. This logic was expressed in the various input languages supported by the tools under test: **JavaScript/TypeScript, Nim, Clojure, Haxe, C** and **Java**. The six code snippets written for each test can be found in

Appendix A. By holding the computational intent constant, we ensure that performance and correctness metrics reflect the capabilities of each transpiler rather than variations in algorithmic complexity.

The **compilers were selected primarily on the basis of the popularity in GitHub**, measured by star count, to capture tools with significant community adoption and support. Additionally, the **Java2Python** converter was included despite its lower visibility, as it fulfilled the specific requirement of translating **Java** code to **Python**. During setup, several candidates could not be installed or executed on our Linux environment due to outdated dependencies or compatibility issues; the final benchmark comprises only the subset of tools successfully integrated into the testbed.

Table 1 Key characteristics of the selected transpilers, with category placed after the tool name.

Tool	Category	Input Languages	Output Languages	AST Handling	Ease of Use	Stability & Known Issues
TypeScript Compiler	Version	TypeScript	JavaScript	Complete	Simple	High stability, few errors
Babel	Version	JavaScript, TypeScript	JavaScript	Complete	Intuitive	High stability, occasional errors
eslint	Version	JavaScript, TypeScript	–	Partial	Moderate	High stability, frequent reports
Nim	Language-to-Language	Nim	C, C++, JavaScript	Complete	Moderate	Stable, actively developed
ClojureScript	Paradigm	Clojure	JavaScript	Complete	Hard	Stable, some reported errors
jscodeshift	Version	JavaScript, TypeScript	JavaScript	Complete	Easy	Moderately stable, occasional errors
ast-grep	Version	JavaScript, TypeScript	–	Simple	Moderate	Moderately stable
Haxe	Language-to-Language	Haxe	JavaScript, Python	Complete	Moderate	Stable, frequent updates
c2rust	Paradigm	C	Rust	Complete	Moderate	Stable, but complex bugs
Fennel	Paradigm	Fennel (Lisp dialect)	Lua	Complete	Simple	Stable, sporadic maintenance
Cito	Language-to-Language	C	JavaScript	Partial	Moderate	Moderately stable
TypeScriptToLua	Language-to-Language	TypeScript	Lua	Complete	Moderate	Stable, minor known bugs
godzilla	Language-to-Language	JavaScript	Lua	Complete	Moderate	Moderately stable
jsweet	Language-to-Language	Java	JavaScript, TypeScript	Complete	Moderate	Moderately stable, some bugs
j2cl	Language-to-Language	Java	JavaScript	Complete	Moderate	Stable, interoperability bugs
Java2Python	Language-to-Language	Java	Python (2.x)	Simple	Easy	Stable, but obsolete (Python 2 only)

Column Definitions

In this table, the **Tool** column lists the name of each transpiler under evaluation for reference. The **Category** column classifies each tool according to its conversion type – *"Language-to-Language"* for those translating between different languages, *"Version"* for those handling differences between versions of the same language, and *"Paradigm"* for those

mapping between distinct programming paradigms. The **Input Languages and Output Languages** columns indicate, respectively, which source languages the tool accepts and which target languages it can generate. The **AST handling** column describes the extent of AST support: *"Complete"* denotes full coverage of nodes and complex transformation passes; *"Partial"* signifies limited support for specific nodes or operations; and *"Simple"* refers to basic functionality for pattern matching or data extraction without deep rewriting capabilities. The Ease of Use column provides a qualitative assessment of the learning curve and interface clarity: *"Simple"* or *"Easy"* indicates straightforward, well-documented APIs requiring minimal configuration; *"Intuitive"* implies a coherent workflow that is naturally accessible despite some complexity; and *"Moderate"* signals that additional configuration steps or intermediate concepts must be mastered before the tool can be utilised to its full potential and *"Hard"* signifies that independent research must be conducted before using the tool. And finally the **Stability & Known Issues** column summarises the general maturity and reliability of each tool, noting whether it is actively maintained, prone to occasional errors or complex bugs, and any recurring issues that users should be aware of.

3.2 Experimental Environment and Tool Installation

All experiments were conducted on a Linux workstation equipped with a multi-core (8 cores) CPU and 32 GB of RAM. Transpilers were installed via their standard package managers or repositories *(e.g. npm for **JavaScript**-based tools, cargo for **Rust**-based tools)*. Where necessary, minor adjustments (such as version pinning or patching build scripts) were applied to resolve compatibility issues. Despite these efforts, some tools remained unusable and were excluded from the study.

Each transpiler was invoked with default settings except where command-line options were required to specify input and output languages. Execution time, CPU utilization and peak memory usage were recorded using system profiling utilities, like the *time* tool integrated in Linux. Output code was then compiled or interpreted to verify correctness, and a precision score from 0 to 100 was assigned based on successful execution and fidelity to the original specification.

4 Results

Table 2 summarizes the four parameters – *execution time, CPU used, memory consumed, and accuracy* – measured to compare the nine Transpilers chosen to conduct the experiment described in Section 3. The following subsections analyse in detail performance, precision, resource consumption, and usability. The last subsection sums up the study conclusions.

4.1 Performance and Efficiency

Execution times varied markedly across tools. **Nim** – transpiling to **C, C++** and **JavaScript** – achieved average runtimes of 0.61 s, 0.59 s and 0.12 s respectively, with moderate memory use (77.8 MB, 79.4 MB, 24.5 MB). In contrast, **ClojureScript** exhibited the slowest performance at 27.44 s despite low memory consumption (36.0 MB), rendering it unsuitable for time-sensitive workflows. The **jscodeshift** transformer delivered the fastest translation (0.06 s) and minimal memory footprint (1.76 MB), although this speed came at the expense of lower precision (see Section 4.2).

Table 2 Performance, resource usage and accuracy results for each transpiler.

Tool	Execution Time (s)	CPU Utilisation (%)	Memory Used (MB)	Accuracy (0-100)
TypeScript Compiler	1.79	86	188.0	80.72
Babel (JS → JS)	8.66	16	92.4	80.72
Babel (TS → JS)	3.29	20	82.4	80.72
Nim (→ C)	0.61	101	77.8	86.25
Nim (→ C++)	0.59	99	79.4	86.25
Nim (→ JS)	0.12	90	24.5	89.50
ClojureScript	27.44	62	36.0	57.70
jscodeshift	0.06	235	1.76	75.00
c2rust	0.16	81	135.3	75.00
Fennel	0.10	98	7.8	79.75
TypeScriptToLua	5.93	59	302.3	72.25
Java2Python	0.09	25	39.9	55.00

4.2 Output Precision

Because the experiment was carried out in multiple tools involving multiple programming languages, it was not possible to use a single application that would evaluate all codes using the same parameters.

So in order to quantify how faithfully each transpiler preserved the semantics of the original programs, we applied a composite scoring methodology on a 0–100 scale. We developed custom evaluation scripts in **Python, JavaScript, Nim,** and other host languages, one per transpiled output, to execute identical input sets and compare results against reference implementations. Although the code under test spanned multiple languages, all source files implement the same algorithmic logic, ensuring a uniform basis for comparison.

The overall precision score $S \in [0, 100]$ for each tool is computed as a weighted sum of five orthogonal categories:

$$S = 40\% \times \mathit{FunctionalCorrectness} + 25\% \times \mathit{SemanticEquivalence} \tag{1}$$

$$+ 15\% \times \mathit{CodeQuality} + 10\% \times \mathit{StructuralSimilarity} + 10\% \times \mathit{ErrorHandling}. \tag{2}$$

Each sub-score is normalized to the range $[0, 100]$ before weighting:

- **Functional Correctness** (40%): Does the transpiled program compile/run and produce the correct outputs for all test inputs? This is the highest-weight category, since a single failure yields a 0 in this dimension.

- **Semantic Equivalence** (25%): Do intermediate states, control flow structure, and side effects match the behaviour of the original program with equivalent inputs?
- **Code Quality** (15%): Does the output follow best practices and idiomatic style for the target language (e.g., Pythonic constructs, proper naming, concise expressions)?
- **Structural Similarity** (10%): Is the high-level structure (functions, classes, loops) aligned with the source?
- **Error Handling** (10%): Are edge cases and exceptions handled appropriately (e.g., null checks, boundary conditions)?

A perfect score (100/100) indicates that the transpiler produced code that compiles/runs without modification, maintains equivalent logic and control flow, adheres to target-language idioms, preserves structural patterns, and robustly handles errors across all test inputs.

Detailed Example: Java2Python (55.00/100)

The **Java2Python** tool translated our sample *'MaxFinder.java'* to *'max_finder.py'* with near-perfect style, structure, and error handling (100% in the last four categories, except the last one that scored 50%), but failed 'Functional Correctness' entirely (0/100 in that category) because of a single method call mismatch:

Original **Java** file:

```
int maxNumber = numbers.get(0);
```

Transpiled **Python** code:

```
maxNumber = numbers.get(0)
```

Since **Python** lists do not implement *'get()'*, this line causes a runtime exception for all test inputs. Correct behaviour requires:

```
maxNumber = numbers[0]
```

As a result, **Java2Python's** functional correctness sub-score is 0, producing a final precision of

$$0.40 \times 0 + 0.25 \times 100 + 0.15 \times 100 + 0.10 \times 100 + 0.10 \times 100 = 55.00.$$

All reported scores reflect the same composite evaluation. Tools scoring below 70-75 may therefore require manual correction or additional validation before deployment.

4.3 Resource Consumption

Memory usage showed significant divergence. **TypeScriptToLua** consumed the most RAM (302.3 MB), followed by the **TypeScript Compiler** (180.2 MB) and **c2rust** (135.3 MB). Conversely, **jscodeshift** (1.76 MB) and **Fennel** (7.8 MB) demonstrated exceptional frugality, making them attractive for resource-constrained environments. CPU utilisation generally remained below 100% of a single core; **jscodeshift** peaked at 235%, reflecting its multi-threaded execution across several cores.

Why in some cases the CPU usage exceeded 100% ?

CPU usage can exceed 100% because modern systems have several CPU cores and the percentage is calculated in relation to the capacity of a single core. When the percentage gets above 100%, it means that the process is using several CPU cores simultaneously.

In the case of **jscodeshift**, CPU usage reaching 235% means that your transformation is effectively using around 2.35 CPU cores on average. This is possible because: **Node.js** (which runs **jscodeshift**) is multi-threaded through its event loop and job threads. The **V8 JavaScript** engine (used by **Node.js**) can parallelise certain operations.

4.4 Usability and Stability

Developer experience was assessed qualitatively. **Babel** and **jscodeshift** stood out for ease of use, due to intuitive command-line interfaces and extensive plugin ecosystems. The **TypeScript Compiler** and **Babel** also showed high stability with minimal runtime errors. **Nim**'s tooling proved reliable but is under active development, which may introduce future breaking changes. Although **Java2Python** delivered perfect accuracy and stability, its support for only **Python 2** renders it impractical for modern codebases.

4.5 Practical Recommendations

Based on these findings, the research reach the conclusion that:

- For **JavaScript/TypeScript** projects: use **Babel** or the **TypeScript Compiler** for a balance of precision, stability and moderate resource demands.
- Where raw performance is critical: employ **Nim** for its rapid and accurate transpilation to **C**, **C++** or **JavaScript**.
- For quick, lightweight transformations: opt for **jscodeshift**, acknowledging potential trade-offs in precision.
- In specialised migrations *(e.g. **C→Rust**, **Fennel→Lua**)*: consider **c2rust** or **Fennel** with the understanding that additional verification may be required.

5 Conclusion

In the context of a research project, an extensive review of the literature was carried out focusing on transpilation: definition, evolution, principles, techniques, and tools. Following that bibliographic study, an empirical study has been conducted to compare some of the most relevant transpilers found and available. In the next paragraphs, the most relevant outcomes are presented.

Insights from the Literature Review

The state of the art analysis confirms that modern transpilers have matured into sophisticated systems capable of transforming code across languages, versions and paradigms while preserving semantic integrity. Early preprocessors evolved into AST-centric frameworks, and tools such as **Babel** and the **TypeScript Compiler** now underpin large-scale web applications by reconciling cutting-edge language features with legacy environments. Key findings include:

- **Semantic Preservation**: Multi-stage AST transformations enable faithful code translation even between drastically different paradigms.

- **Performance Trade-Offs**: Parsing and code-generation introduce overheads that must be balanced against compatibility gains.
- **Tool Ecosystems**: Plugin architectures *(e.g. **Babel** plugins)* and rich analysis APIs foster extensibility and customisation.

Insights from the Empirical Study

The independent benchmarks highlight the practical trade-offs faced when choosing a transpiler:

- **Raw Performance**: **Nim** outperforms all other tools, delivering sub-second translations with perfect accuracy.
- **Balanced Options**: **Babel** and the **TypeScript Compiler** offer strong precision (91%) and stability with moderate resource demands.
- **Lightweight Transforms**: **jscodeshift** achieves near-instantaneous conversions at minimal memory cost, albeit with lower accuracy.
- **Specialised Tools**: Converters such as **c2rust** and **Java2Python** address niche migrations effectively but require careful validation and environment compatibility.

These results underscore the importance of aligning transpiler selection with project priorities – whether that be throughput, fidelity or ecosystem integration.

5.1 Future Work

Building on both theoretical and practical insights, several avenues for further research were identified:

AI-Augmented Transpilation

Integrating machine learning techniques to produce context-aware transformations promises higher fidelity and automatic correction of edge cases [18]. In addition,nual labor rlabord to create a transpiltranspiler is eliminated,nated, from building a parser to a source language to apply AST transformations and code generation for a target language.

Paradigm-Bridging Frameworks

Advancing paradigm transpilers to accurately convert between functional, imperative and declarative models remains an open challenge, with potential in new algorithmic strategies [17].

Distributed and Cloud-Native Scenarios

As software architectures grow more distributed, transpilers must support microservices and server less deployments, optimizing for networked environments and heterogeneous runtimes [24]. In order to, for instance, provide transpilation as a service without requiring a user to have a local system with hardware capabilities to handle the migration of projects with millions of lines of code.

Enhanced Benchmarking

Future empirical studies should incorporate multi-core scaling, plugin overhead, and real-world codebases to refine our understanding of performance versus precision trade-offs.

Toolchain Integration and Automation

Developing unified workflows that integrate transpilers with CI/CD pipelines and automated testing frameworks will streamline large-scale migrations and continuous modernization efforts.

5.2 Additional information

If this article interests you, it is important to mention that there is a complementary website where you can find the article you read, the transpilers (and their version) used along with the links so you can easily access the same tools used in the experiment. After that, the webpage displays all the scripts used to evaluate the transpiled code, so that you can replicate the same experiment and further extend it. Right at the end there is a link to another document that dives into the state-of-the-art study about transpilers.

Here is the link:
`https://justandre02.github.io/Comparative-Review-and-Empirical-Evaluation-about-transpilers/`

References

1. Bastidas F. Andrés and María Pérez. Transpiler-based architecture for multi-platform web applications. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017. doi:10.1109/ETCM.2017.8247456.
2. Thomas Baar and Slaviša Marković. A graphical approach to prove the semantic preservation of uml/ocl refactoring rules. In *Perspectives of Systems Informatics: 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers 6*, pages 70–83. Springer, 2007.
3. Andrés Bastidas Fuertes, María Pérez, and Jaime Meza. Transpiler-based architecture design model for back-end layers in software development. *Applied Sciences*, 13(20):11371, 2023.
4. Victor Berdonosov and Alena Zhivotova. The evolution of the object-oriented programming languages. *Scholarly Notes of Komsomolsk-na-Amure State Technical University*, 1:35–43, June 2014. doi:10.17084/2014.II-1(18).5.
5. A. Cox and C. Clarke. Syntactic approximation using iterative lexical analysis. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 154–163, 2003. doi:10.1109/WPC.2003.1199199.
6. Andrés Bastidas Fuertes, María Pérez, and Jaime Meza Hormaza. Transpilers: A systematic mapping review of their usage in research and industry. *Applied Sciences*, 13(6):3667–3667, 2023. doi:10.3390/app13063667.
7. Nadja Gaudillière-Jami. AD Magazine: Mirroring the Development of the Computational Field in Architecture 1965–2020. In *ACADIA 2020: Distributed Proximities*, volume 1, pages 150–159, Online, France, October 2020. B. Slocum, V. Ago, S. Doyle, A. Marcus, M. Yablonina, M. del Campo. URL: https://hal.science/hal-04588619.
8. Joshua Goodman. Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 177–183, USA, 1996. Association for Computational Linguistics. doi:10.3115/981863.981887.
9. Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems*, 37, July 2015. doi:10.1145/2743016.
10. Dick Grune and Ceriel J. H. Jacobs. *Introduction to Parsing*, pages 61–102. Springer New York, 2008. doi:10.1007/978-0-387-68954-8_3.
11. Evgeniy Ilyushin and Dmitry Namiot. On source-to-source compilers. *International Journal of Open Information Technologies*, 4(5):48–51, 2016.

12 Philip Japikse, Kevin Grossnicklaus, and Ben Dewey. *Introduction to TypeScript*, pages 413–468. Springer, December 2019. doi:10.1007/978-1-4842-5352-6_10.

13 Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. An ast structure enhanced decoder for code generation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, PP:1–1, December 2021. doi:10.1109/TASLP.2021.3138717.

14 A. Johnson. Fortran preprocessors. *Computer Physics Communications*, 45:275–281, August 1987. doi:10.1016/0010-4655(87)90164-0.

15 Adrian Johnstone, Elizabeth Scott, and Giorgios Economopoulos. Evaluating glr parsing algorithms. *Science of Computer Programming*, 61:228–244, August 2006. doi:10.1016/j.scico.2006.04.004.

16 Kostadin Kratchanov and Efe Ergün. Language interoperability in control network programming, 2018.

17 Chenyang Lyu, Zefeng Du, Jitao Xu, Yitao Duan, Minghao Wu, Teresa Lynn, Alham Fikri Aji, Derek F Wong, Siyou Liu, and Longyue Wang. A paradigm shift: The future of machine translation lies with large language models. *arXiv preprint arXiv:2305.01181*, 2023.

18 André Melo, Nathan Earnest-Noble, and Francesco Tacchino. Pulse-efficient quantum machine learning. *Quantum*, 7:1130, 2023. doi:10.22331/Q-2023-10-09-1130.

19 Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, pages 1–12, 2023. doi:10.1109/ms.2023.3243858.

20 Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, 41(1):105–112, 2024. doi:10.1109/MS.2023.3243858.

21 Alberto Pettorossi, Maurizio Proietti, Fabio Fioravanti, and Emanuele De Angelis. A historical perspective on program transformation and recent developments (invited contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, PEPM 2024, pages 16–38, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3635800.3637446.

22 David Plaisted. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications*, 06:30–40, January 2013. doi:10.4236/jsea.2013.64A005.

23 Hardik Raina, Aditya Kurele, Nikhil Balotra, and Krishna Asawa. Language agnostic neural machine translation. In *Proceedings of the 2024 Sixteenth International Conference on Contemporary Computing*, IC3-2024, pages 535–539, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3675888.3676109.

24 Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. Programming reconfigurable heterogeneous computing clusters using mpi with transpilation. In *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 1–9, November 2020. doi:10.1109/H2RC51942.2020.00006.

25 Larissa Schneider and Dominik Schultes. Evaluating swift-to-kotlin and kotlin-to-swift transpilers. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '22, pages 102–106, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524613.3527811.

26 P Thomas Schoenemann. Syntax as an emergent characteristic of the evolution of semantic complexity. *Minds and Machines*, 9:309–346, 1999. doi:10.1023/A:1008360020568.

27 Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. Autotransform: automated code transformation to support modern code review process. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 237–248, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510067.

28 Shanshan Wang and Xiaohui Wang. Cross-language translation and comparative study of english literature using machine translation algorithms. *Journal of Electrical Systems*, 2024. doi:10.52783/jes.3136.

A Code snippets used

To consolidate the experimental study we have conducted, and that was discussed in Sections 3 and 5, Listings 1 to 6, presented below, show the code snippets used for the transpilation across various tools.

Listing 1 JavaScript test case.

```
class Main {
    public static function processNumbers(numbers:Array<Int>):
                                    {sumEven:Int, maxNumber:Int} {
        var sumEven = 0;
        var maxNumber = numbers[0];

        for (num in numbers) {
            if (num % 2 == 0) {
                sumEven += num;
            }
            if (num > maxNumber) {
                maxNumber = num;
            }
        }

        return {sumEven: sumEven, maxNumber: maxNumber};
    }
}
```

Listing 2 Nim test case.

```
proc processNumbers(numbers: seq[int]):
                        tuple[sumEven: int, maxNumber: int] =
    var sumEven = 0
    var maxNumber = numbers[0]

    for num in numbers:
        if num mod 2 == 0:
            sumEven += num
        if num > maxNumber:
            maxNumber = num

    return (sumEven, maxNumber)

let numbers = @[1, 2, 3, 4, 5, 6]
let result = processNumbers(numbers)
```

Listing 3 Clojure test case.

```
(defn process-numbers [numbers]
  (let [sum-even (reduce + (filter even? numbers))
        max-number (reduce max numbers)]
    {:sum-even sum-even
     :max-number max-number}))
```

Listing 4 Haxe test case.

```
class Main {
    public static function processNumbers(numbers:Array<Int>):
                                    {sumEven:Int, maxNumber:Int} {
        var sumEven = 0;
        var maxNumber = numbers[0];

        for (num in numbers) {
            if (num % 2 == 0) {
                sumEven += num;
            }
            if (num > maxNumber) {
                maxNumber = num;
            }
        }

        return {sumEven: sumEven, maxNumber: maxNumber};
    }
}
```

Listing 5 C test case.

```
#include <stdio.h>

typedef struct {
    int sumEven;
    int maxNumber;
} Results;

Results processNumbers(int numbers[], int length) {
    int sumEven = 0;
    int maxNumber = numbers[0];

    for (int i = 0; i < length; i++) {
        if (numbers[i] % 2 == 0) {
            sumEven += numbers[i];
        }
        if (numbers[i] > maxNumber) {
            maxNumber = numbers[i];
        }
    }

    Results results = {sumEven, maxNumber};
    return results;
}
```

Listing 6 Java test case.

```
import java.util.List;

public class Main {
    public static Result processNumbers(List<Integer> numbers) {
        int sumEven = 0;
        int maxNumber = numbers.get(0);

        for (int num : numbers) {
            if (num % 2 == 0) {
                sumEven += num;
            }
            if (num > maxNumber) {
                maxNumber = num;
            }
        }

        return new Result(sumEven, maxNumber);
    }
}

class Result {
    public int sumEven;
    public int maxNumber;

    public Result(int sumEven, int maxNumber) {
        this.sumEven = sumEven;
        this.maxNumber = maxNumber;
    }
}
```