© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716842

A Comparative Review of Data Representation and Data Types in Modern Programming Languages

VICTORIA MICHAEL UDOH[1], ONYEKACHUKWU CHRISTIAN UZOIGWE[2], DAMILARE ADEFEMIWA QUADRY[3], LOLADE AKINROLABU[4], ADETUNJI PHILIP ADEWOLE[5]
[1,2,3,4,5]*Department of Computer Science, University of Lagos.*

Abstract- ***The representation and classification of data remain central to the design, implementation, and evolution of modern programming languages. This study presents a comprehensive comparative review of data types and data representation across contemporary programming paradigms, situating current practices within their historical and theoretical foundations. Drawing from systematic literature spanning 2015-2025 and guided by PRISMA-based evidence synthesis, the review examines primitive, composite, and advanced data types, alongside the underlying type systems that govern their behavior. The analysis highlights the dual role of data types as both restrictive and enabling constructs, tracing their evolution from early languages such as FORTRAN and ALGOL to modern systems such as Python, Java, and Rust. Key distinctions between static and dynamic typing, as well as strong versus weak type enforcement, are critically evaluated, revealing inherent trade-offs between flexibility, safety, and performance. Furthermore, the review synthesizes recent advances in type system design, including Generalized Algebraic Data Types (GADTs), ownership-based memory models, and declarative collection abstractions, demonstrating their impact on program correctness and efficiency. The study also extends the discussion to modern data-intensive domains, where data representation frameworks underpin developments in array programming, multimodal data integration, and machine learning architectures. Despite noteworthy progress, the findings reveal persistent gaps between theoretical advances in type systems and their practical application in data science workflows. The divergence between safety-oriented systems programming and flexibility-driven data analytics environments shows the need for integrative approaches such as gradual typing, schema-aware type systems, and shape-polymorphic abstractions. This review contributes to the discourse by providing a unified perspective on data types and representation, offering insights that are critical for advancing programming language design and data-centric computing.***

Index Terms: ***Data Type, Types, Data Representation, Programming Languages, Composite Data***

I. INTRODUCTION

The concept of data types and data representation is one of the core concepts of computer science, shaping how information is structured, stored (Martini), and manipulated across programming paradigms. From the earliest days of computing, when machines operated directly on binary representations, to today's sophisticated, type-safe, and abstracted systems, the evolution of data types reflects broader shifts in computational theory, hardware capabilities, and software engineering practices (Sieczkowski et al.).

Historically, early programming in machine code and assembly language offered little abstraction, requiring programmers to manage raw binary and memory addresses explicitly. The introduction of high-level languages such as FORTRAN in the 1950s marked a turning point, enabling the formalization of primitive data types like integers and floating-point numbers (Durnová and Alberts). Subsequently, languages such as ALGOL introduced structured programming concepts and more rigorous type systems, paving the way for user-defined data types and scope rules (Durnová and Alberts). These developments were critical in abstracting hardware complexity and improving program correctness. As computing matured, the need for richer data representation led to the emergence of composite and abstract data types (ADTs). Languages like C provided structures and unions, allowing programmers to define complex data layouts while maintaining efficiency (Chavez et al.). Meanwhile, object-oriented languages such as Java and C++ extended this paradigm by encapsulating data and behavior within classes, thereby enhancing modularity and reuse (Nugroho and Sutanto).

In parallel, advances in type theory and programming language design led to the development of strong and static type systems, as seen in Haskell, and dynamically typed languages such as Python and JavaScript. These languages represent divergent philosophies in data representation, balancing safety, flexibility, and developer productivity (Martini). Furthermore, modern languages like Rust have introduced ownership models and memory-safe abstractions that fundamentally reshape how data is represented and accessed at runtime (Badaro et al.; Chavez et al.). Contemporary programming environments also reflect the increasing complexity of data itself. With the rise of distributed systems, big data, and artificial intelligence, data representation now extends beyond primitive and composite types to include serialization formats (JSON, Protocol Buffers), type inference systems, and domain-specific abstractions (Franke et al.; Sieczkowski et al.). These advancements underscore the necessity of understanding not only how data is defined within a language, but also how it is represented in memory, transmitted across systems, and optimized for performance. Despite these advancements, significant variations persist across modern programming languages in how they implement and enforce data types and representations. These differences raise critical questions for both researchers and practitioners seeking to balance efficiency, safety, and expressiveness. Accordingly, this review is guided by the following research questions:

1. How have data types and data representation evolved from early programming languages to modern multi-paradigm languages?
2. What are the fundamental differences in data type systems (dynamic, Static, Strong, Weak and Hybrid typing.) across contemporary programming languages?
3. How do different approaches to data representation impact program performance, memory management, and software reliability in modern computing environments?

II. LITERATURE REVIEW

From a data science and programming language theory perspective, the notion of a data type occupies a uniquely dual position within computational systems. Unlike its counterpart in mathematical logic, where types primarily serve to constrain expressiveness and avoid paradoxes, types in programming language's function both as restrictive mechanisms and enabling abstractions(Martini; Sterling and Harper). That is, they simultaneously prevent invalid operations while providing guarantees of correctness, structure, and computational safety (Martini). This duality underpins much of the evolution of programming languages and remains central to modern discussions on data representation. Historically, the development of data types reflects the convergence of two intellectual traditions: mathematical formalism and practical compiler design. Early languages such as FORTRAN employed the notion of modes to distinguish between numeric representations, primarily for optimization purposes. However, with the emergence of ALGOL (Durnová and Alberts), types became formalized as disjoint classes of values mapped to memory representations, forming the conceptual foundation for modern type systems (Martini). This shift marked the transition from machine-oriented computation to abstraction-driven programming. A central theme in the literature is the enduring dichotomy between static and dynamic typing, which represents one of the most fundamental design decisions in programming languages. Static typing enforces type constraints at compile time, while dynamic typing defers such checks to runtime (Singh et al.). Languages such as Java exemplify static typing, requiring explicit declarations and enabling early detection of type errors. In contrast, Python adopts dynamic typing, allowing variables to change type during execution, thereby enhancing flexibility and rapid prototyping (Alaria et al.). However, this binary classification is insufficient on its own. The concept of type safety introduces an orthogonal dimension, distinguishing between strong and weakly typed systems. Strong typing enforces strict compatibility rules between data types, whereas weak typing permits implicit conversions that may lead to unintended behavior (Singh et al.). Consequently, programming languages exist within a multidimensional design space, balancing flexibility, safety, and performance.

A pivotal advancement in data representation was the transition from primitive to structured data types.

Early programming languages were limited to basic types such as integers and floating-point numbers. The introduction of records and typed references, as proposed by Hoare in the 1960s, enabled the modeling of complex relationships and data structures (Martini). These innovations were subsequently realized in Simula, which introduced encapsulation and laid the groundwork for object-oriented programming. Building on this, (Batdalov et al. 2016) proposed a systematic classification distinguishing base types like integers, Booleans, characters from compound types such as arrays, records, sets, and classes (Batdalov et al.). This classification emphasizes structural relationships rather than implementation-specific distinctions, offering a more theoretically grounded perspective compared to language-specific categorizations such as primitive versus reference types in Java.

Modern programming languages have significantly extended the expressiveness of type systems. One notable development is the introduction of Generalized Algebraic Data Types (GADTs), which allow more precise type specifications by enabling constructors to produce values of different types. Languages such as Haskell and Scala have adopted GADTs to enhance type safety and expressiveness. Theoretical work demonstrates that GADTs are not merely syntactic enhancements but are fundamentally more expressive than traditional type constructs, enabling sophisticated compile-time guarantees through advanced pattern matching and type inference mechanisms (Sieczkowski et al.). This development reflects a broader trend toward increasingly expressive and mathematically grounded type systems. The most transformative recent development in type systems is the ownership model introduced in Rust. Unlike traditional garbage-collected or manually managed memory systems, Rust employs a compile-time ownership model that ensures memory safety without runtime overhead. Key concepts include Ownership-Each value has a single owner responsible for its lifecycle, borrowing; temporary references that do not transfer ownership, Lifetimes, Compile-time annotations ensuring reference validity. This model has been formally verified through the Rustbelt project, demonstrating that high-level safety guarantees can coexist with low-level control (Jung et al.). Importantly, Rust introduces an extensible safety framework, allowing controlled use of unsafe operations under verified conditions, thereby bridging the gap between theoretical rigor and practical systems programming.

Modern data representation increasingly relies on collection abstractions, which organize and manipulate large datasets efficiently. Traditional frameworks such as those in Java and C++ often require developers to select specific data structures like lists and queues, leading to over-specification and reduced flexibility. The Collection Skeletons framework addresses this limitation by adopting a property-centric approach, where developers specify desired properties, such as ordering or duplication rather than concrete implementations. The system then determines the optimal data structure, resulting in improved performance and portability (Franke et al.). This shift represents a paradigm transition from operation-centric to declarative data representation.

In data science, array programming has emerged as a dominant paradigm for representing structured data. Libraries such as those in Python (NumPy) treat arrays as multidimensional tensors characterized by attributes such as shape, data type, and memory layout. These arrays support vectorized operations, eliminating explicit loops. Broadcasting, enabling operations across dimensions and Memory-efficient views, reducing duplication.

Such features enable efficient computation on large datasets and support interoperability across hardware platforms, including GPUs and distributed systems (Harris et al.). This paradigm is particularly critical in scientific computing and machine learning. Recent advances in machine learning have further expanded the scope of data representation. Transformer architectures, initially developed for natural language processing, are now being adapted for tabular data, requiring novel encoding strategies such as positional embeddings and table serialization (Badaro et al.). Simultaneously, the challenge of multimodal data representation integrating text, images, and sensor data has gained prominence. This requires bridging the heterogeneity gap, where different data types exhibit distinct statistical and structural properties. Current approaches involve extracting modality-specific features and projecting them into unified representational spaces (Gaonkar et al.).

In scientific and engineering domains, data representation must also address issues of standardization, interoperability, and reproducibility. For example, the evolution of flow cytometry data standards highlights the tension between accommodating heterogeneous data types and maintaining backward compatibility (Spidlen et al.). Similarly, genomic data tools emphasize the importance of reproducible workflows and provenance tracking in complex data pipelines (Q. Liu et al.). Empirical studies further challenge assumptions about data complexity. For instance, increasing the number of data types in multi-omics analysis does not necessarily improve predictive performance, suggesting that data quality and representation may be more critical than sheer diversity (J. Liu et al.; Q. Liu et al.).

Despite significant advancements, this study reveals several persistent gaps. There is limited integration between advanced type systems such as GADTs and ownership models, and practical data science workflows. Additionally, empirical studies comparing the real-world impact of type systems on software quality remain scarce (Alaria et al.; Batdalov et al.). Moreover, the divide between formal methods research and data science practice remains pronounced. While the former emphasizes correctness and guarantees, the latter prioritizes flexibility and rapid experimentation (Harris et al.; Jung et al.). Bridging this gap represents a critical challenge for future research.

III. METHODOLOGY

This systematic review was undertaken in accordance with the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) framework, as articulated by Page et al., (2021) and Haddaway et al., (2022). The PRISMA approach provides a robust and internationally recognized protocol for ensuring methodological transparency, reproducibility, and rigor in evidence synthesis studies. Guided by this framework, the review adopted a structured, multi-stage methodology. The process commenced with the formulation of a comprehensive search strategy aligned with the thematic focus of the study, namely, data types and their representation in modern programming languages. To ensure breadth and relevance, a systematic search was conducted across major academic databases using the title-abstract-keyword (TITLE-ABS-KEY) fields. This approach enabled the identification of studies that not only explicitly referenced the core concepts in their titles but also those whose abstracts and keywords reflected the underlying technical constructs.

The search strategy incorporated carefully selected keywords and phrases, including “data types,” “data representation,” and “modern programming languages,” alongside their variants. Boolean operators (AND, OR) were applied to enhance both the sensitivity and specificity of the search, thereby ensuring comprehensive retrieval of relevant studies while minimizing unrelated results. The temporal scope of the search was restricted to publications between 2015 and 2025, allowing for an up-to-date examination of developments in data representation over the past decade. All identified records were exported into Zotero, which functioned as the central platform for data organization and management. Within this environment, records were systematically processed to remove duplicates through a combination of automated detection and manual verification. Metadata standardization, document classification, and tracking of inclusion decisions were also conducted to maintain a transparent audit trail. The initial search yielded 60 records, of which 29 duplicates were identified and removed, resulting in 31 unique studies. A preliminary screening of titles and abstracts was subsequently performed to assess alignment with the study’s conceptual scope. Inclusion criteria required that each study address at least one of the following dimensions: data types, data representation, or modern programming languages. Based on this screening, 11 studies were excluded, leaving 20 records for full-text retrieval. Of these, 6 studies were not successfully retrieved. Specifically, 4 were excluded due to publication dates falling outside the defined timeframe, while 2 were deemed insufficiently relevant to the thematic focus of the review. Consequently, 14 studies were retained for detailed evaluation, with an additional 1 record assessed and deemed eligible, resulting in a final sample of 15 studies included in the review.

Notably, the selection process revealed a limited number of studies within the defined period, suggesting a potential decline or fragmentation in

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716842

research specifically addressing data types and data representation within the context of modern programming languages. The overall selection process is illustrated in Figure 1, which presents the PRISMA flow diagram summarizing the identification, screening, eligibility, and inclusion stages.

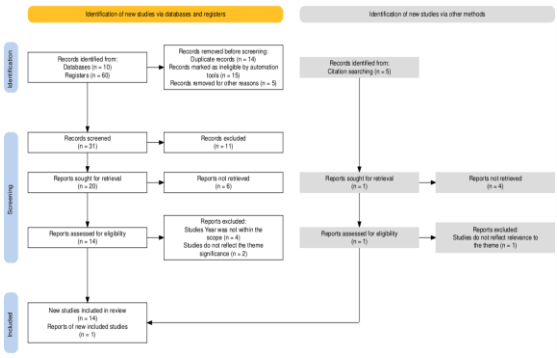

Figure 1: Prisma Flow Diagram

The corpus of studies reviewed comprises one conference proceeding and fourteen peer-reviewed journal articles, reflecting a predominance of rigorously validated academic contributions alongside a modest representation of conference-based, practice-oriented research outputs. As illustrated in Figure 2, this distribution shows the dual nature of data types and programming languages, drawing from both formal academic inquiry and insights shaped by evolving industry practices. Beyond mere numerical distribution, the review also examined the temporal progression of publications across the ten-year period (2015-2025). This longitudinal perspective reveals patterns in research intensity, indicating how scholarly attention to data types and their representations has evolved over time. Such trends are visually synthesized in Figure 3, which captures both categorical and temporal dynamics, illustrating the trajectory of academic engagement in this field from the foundational era of early programming languages such as FORTRAN, LISP, and COBOL to contemporary developments in type systems and data abstraction (Sieczkowski et al.).

To synthesize the findings, the study adopted an integrative review approach, enabling the consolidation of diverse research methodologies and perspectives into a coherent analytical framework. Emphasis was placed on extracting and comparing insights related to data types, data representation mechanisms, programming language paradigms, and human-centric considerations such as usability and developer interaction with type systems. Furthermore, the methodological rigor of the included studies was critically evaluated through an assessment of their research designs, analytical procedures, and internal validity. This evaluative step ensured that only robust and contextually relevant findings informed the synthesis. Consequently, the conclusions drawn from this review are grounded in high-quality evidence, enhancing their reliability and relevance to both academic research and practical applications in modern programming language design.

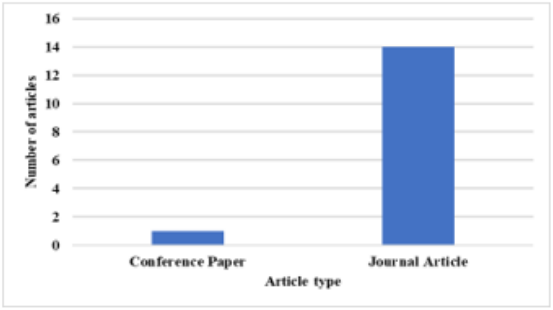

Figure 2: Article Types Diagram

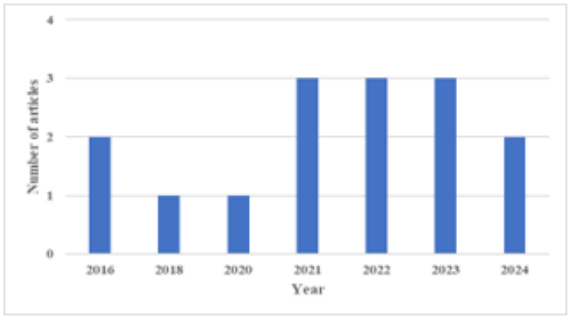

Figure 3: Article Year Breakdown Diagram

IV. DISCUSSION

Data types, formally defined as structured sets of permissible values together with the operations that can be performed on them, remains foundational to programming language design and implementation. A data type constrains how data is represented in memory and governs the semantics of operations applied to it. As articulated in the introduction, data

types serve not only as syntactic constructs but also as mechanisms for enforcing correctness, optimizing performance, and enabling abstraction in software systems (Martini). In modern programming languages, the scope and sophistication of data types have expanded significantly, reflecting advances in type theory, compiler design, and software engineering methodologies. This evolution is particularly evident when examining the three principal categories of data types, primitive (basic), composite, and user-defined data types. Understanding these categories in a comparative context across languages such as Java, Python, C#, and Rust provides critical insights into how contemporary systems balance performance, safety, and flexibility.

Primitive (Basic) Data Types

Primitive data types constitute the most elementary units of data representation. These types are typically directly supported by hardware architectures, ensuring efficient computation and minimal abstraction overhead. The principal primitive types include integers, floating-point numbers, and Booleans, each exhibiting nuanced differences across programming languages.

Integer Data Types.

Integer data types represent whole numbers and are among the earliest abstractions introduced in programming languages. Despite their apparent simplicity, their implementation varies significantly across languages, particularly in terms of bit-width, signedness, and memory allocation. Languages such as Java and C# provide multiple integer representations, byte, short, int, and long, each corresponding to different memory sizes. This multiplicity enables developers to optimize resource utilization and computational performance. In contrast, Python adopts an abstraction-oriented approach by offering a single int type with arbitrary precision. This design eliminates concerns about overflow but introduces additional computational overhead (Singh et al.). Such abstraction reflects Python's prioritization of developer productivity over low-level optimization. A more explicit and safety-oriented design is observed in Rust, where integer types such as i8, i16, i32, i64, i128, and isize explicitly encode size and signedness. This approach enhances predictability and prevents unintended behavior, aligning with Rust's emphasis on memory safety and compile-time error detection (Jung et al.). Similarly, Swift distinguishes between signed (Int) and unsigned (UInt) integers, reinforcing type safety. These variations underscore a fundamental trade-off: abstraction versus control. While abstraction simplifies development and reduces cognitive load, explicit typing facilitates fine-grained optimization, which is essential in systems programming and high-performance computing.

Floating-Point Data Types

Floating-point data types are designed to represent real numbers and are indispensable in domains such as scientific computing, financial modeling, and engineering simulations. Most modern programming languages implement floating-point arithmetic in accordance with the IEEE 754 standard, ensuring consistency in representation and operations. Languages such as Java and C# provide float and double types, corresponding to single and double precision, respectively. Notably, C# extends this capability with a decimal type, specifically designed for high-precision financial calculations, thereby mitigating rounding errors inherent in binary floating-point representations. Conversely, Python and PHP simplify the model by exposing primarily to a double-precision float. While this reduces complexity for developers, it can obscure precision-related issues, particularly in numerically sensitive applications. In Rust, floating-point types (f32, f64) explicitly denote precision, promoting clarity and deliberate usage. This explicitness reflects a broader design philosophy aimed at making potential computational inaccuracies visible during development (Jung et al.)

Boolean Data Types

The Boolean data type represents logical values, true or false and underpins control flow constructs such as conditionals and loops. Despite its simplicity, it is one of the most universally consistent data types across programming languages. Languages such as Java (Boolean), C# (bool), and Python (bool) provide standardized implementations. However, dynamically typed languages like PHP permit implicit type coercion between Booleans and other data types, potentially introducing ambiguity and runtime errors.

This distinction highlights a broader issue in type system design: the balance between flexibility and strictness. Strong enforcement reduces errors but may limit convenience, while permissiveness enhances flexibility at the cost of reliability (Martini).

Composite Data Types

Composite data types enable the aggregation of multiple data elements into unified structures, thereby facilitating more complex data modeling. These include arrays, lists, and associative structures such as dictionaries or maps.

Arrays and Lists

Arrays are fixed-size collections of homogeneous elements, representing one of the earliest composite data structures. Modern programming languages extend arrays with dynamic alternatives such as lists and vectors. In Java and C#, a clear distinction exists between static arrays and dynamic collections (ArrayList, List<T>), enabling both performance optimization and programming flexibility. In contrast, Python provides highly flexible structures such as list and tuple, capable of storing heterogeneous data types. While this flexibility accelerates development, it may incur performance penalties due to dynamic type checking and memory overhead (Harris et al.). Rust introduces Vec<T>, a dynamic array governed by strict ownership and borrowing rules. This ensures memory safety without relying on garbage collection, representing a significant advancement in systems-level data representation (Jung et al., 2018). Recent developments emphasize higher-level abstractions for handling collections. For instance, declarative collection models enable programmers to express operations on data more succinctly, improving both readability and maintainability (Franke et al.).

Dictionaries and Maps

Dictionaries (or maps) are data structures that store key–value pairs, enabling efficient lookup, insertion, and deletion operations. Their implementation varies significantly across programming languages. In statically typed languages such as Java (HashMap) and C# (Dictionary), type constraints are enforced on both keys and values, enhancing type safety and predictability. Conversely, Python (dict) and PHP (associative arrays) allow dynamic typing, supporting heterogeneous data but potentially reducing performance and increasing susceptibility to runtime errors. Rust (HashMap) provides a balanced approach by combining strong typing with ownership-based memory management, ensuring both efficiency and safety. The importance of such data structures extends into modern data-intensive applications, including machine learning and multimodal data processing, where efficient representation and retrieval are critical (Badaro et al.; Gaonkar et al.).

Type Systems in Modern Programming Languages

A type system is a formal framework that classifies data types and enforces constraints on their usage within a program. It plays a crucial role in ensuring program correctness and reliability. Programming languages can be broadly categorized as follows: dynamic, Static, Strong, Weak and Hybrid typing as shown in Table 1 Recent advances such as Generalized Algebraic Data Types (GADTs) enhance type expressiveness and enable more precise modeling of program invariants, thereby improving correctness and robustness (Sieczkowski et al.)

Table 1: Type system categorization of programming languages

	Static / Dynamic Type	Strong / Weak Type	Hybrid Features
C#	Static	Strong	Limited (via dynamic)
Dart	Static (with inference)	Strong	Yes (supports dynamic usage)
Java	Static	Strong	Minimal (reflection, limited dynamism)
PHP	Dynamic	Weak	Yes (optional type declarations)
Python	Dynamic	Strong	Yes (type hints - gradual typing)
Rust	Static	Strong	No (strict compile-time guarantees)
Swift	Static	Strong	Limited (type inference)

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1716842

Data Representation in Modern Computational Contexts

In contemporary computing environments, data representation extends beyond programming language constructs to encompass data storage formats, interoperability standards, and domain-specific representations. For example, standardized data formats in scientific computing, such as those used in flow cytometry, ensure consistency and reproducibility across experiments (Spidlen et al.). Similarly, tools for genomic data management emphasize reusable and structured data representations to support large-scale bioinformatics workflows (Liu et al., 2024). In the domain of artificial intelligence, efficient data representation is critical for model performance. Techniques for encoding tabular and multimodal data significantly influence the effectiveness of machine learning algorithms (Badaro et al.; Gaonkar et al.). These developments show a key transition that data types are no longer confined to programming language syntax but are also integral to broader computational ecosystems, influencing system design, interoperability, and scalability as shown in Table 2, the cross-language comparison of different programming languages.

Table 2: Cross language Feature Comparison

Feature	C#	Dart	Java	PHP	Python	Rust	Swift
Classes	✓	✓	✓	✓	✓	✕	✓
Structs	✓	✕	✕	✕	Limited	✓	✓
Enums	✓	✓	✓	✓	✓	✓	✓
Interfaces	✓	Implicit	✓	✓	Abstract Base Class (ABC)	Traits	Protocols
Type Aliases	✓	✓	Limited	Limited	✓	✓	✓

V. CONCLUSION

This review has examined the evolution, structure, and implications of data types and data representation within modern programming languages, revealing a field characterized by both enduring foundational principles and rapid innovation. From their origins as simple mechanisms for distinguishing numerical representations, data types have evolved into sophisticated constructs that underpin abstraction, enforce correctness, and enable efficient computation. The transition from early paradigms in FORTRAN and COBOL to contemporary languages such as Rust and Swift reflects a continuous effort to reconcile low-level control with high-level safety guarantee. A central finding of this study is the persistent tension between competing design philosophies. Static and strongly typed systems provide robustness, predictability, and performance optimization, while dynamic and flexible systems facilitate rapid development and adaptability, particularly in data science and machine learning contexts. Advances such as GADTs and ownership-based models demonstrate that it is possible to enhance expressiveness and safety simultaneously; however, these innovations remain underutilized in mainstream data-centric programming environments. Equally significant is the transformation of data representation beyond traditional programming constructs. In contemporary computational ecosystems, data types are deeply intertwined with storage formats, interoperability standards, and analytical frameworks. The rise of array programming, multimodal data representation, and transformer-based architecture underscores the increasing complexity and heterogeneity of data, necessitating more expressive and adaptable type systems. Nevertheless, the review identifies critical gaps that warrant further investigation. These include the limited integration of advanced type theory into practical data science workflows, the scarcity of empirical studies evaluating the real-world impact of type systems, and the disconnect between programming language research and applied data analytics. Addressing these challenges will require interdisciplinary collaboration and the development of hybrid approaches that bridge theoretical rigor with practical usability. The future of data types and data representation lies in the convergence of safety, flexibility, and scalability. Emerging directions such as gradual typing, schema-aware systems, and

ownership-inspired models for high-level languages hold significant promise. By aligning advances in programming language theory with the demands of modern data-driven applications, the field is poised to play a pivotal role in shaping the next generation of intelligent and reliable computational systems.

REFERENCES

[1] Alaria, Satish Kumar, et al. "Comparative Study of Java and Python: A Review." Industrial Engineering Journal, vol. 52, no. 04, 2023, pp. 2698–708. DOI.org (Crossref), https://doi.org/10.36893/IEJ.2023.V52I4.2698-2708.

[2] Badaro, Gilbert, et al. "Transformers for Tabular Data Representation: A Survey of Models and Applications." Transactions of the Association for Computational Linguistics, vol. 11, Mar. 2023, pp. 227–49. DOI.org (Crossref), https://doi.org/10.1162/tacl_a_00544.

[3] Batdalov, Ruslan, et al. "Extensible Model for Comparison of Expressiveness of Object-Oriented Programming Languages." Applied Computer Systems, vol. 20, no. 1, Dec. 2016, pp. 27–35. DOI.org (Crossref), https://doi.org/10.1515/acss-2016-0012.

[4] Chavez, Brethel B., et al. "C Programming Language: A Review." Journal of Universal Computer Science, vol. 7, no. 1, 2021.

[5] Durnová, Helena, and Gerard Alberts. "Was Algol 60 the First Algorithmic Language?" IEEE Annals of the History of Computing, vol. 36, no. 4, Oct. 2014, pp. 104–104. DOI.org (Crossref), https://doi.org/10.1109/MAHC.2014.63.

[6] Franke, Björn, et al. "Collection Skeletons: Declarative Abstractions for Data Collections." Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering [Auckland New Zealand], 2022, pp. 189–201. DOI.org (Crossref), https://doi.org/10.1145/3567512.3567528.

[7] Gaonkar, Apeksha, et al. "A Comprehensive Survey on Multimodal Data Representation and Information Fusion Algorithms." 2021 International Conference on Intelligent Technologies (CONIT), 2021. sci-hub.africa, https://doi.org/10.1109/CONIT51480.2021.9498415.

[8] Haddaway, Neal R., et al. "PRISMA2020 : An R Package and Shiny App for Producing PRISMA 2020-compliant Flow Diagrams, with Interactivity for Optimised Digital Transparency and Open Synthesis." Campbell Systematic Reviews, vol. 18, no. 2, June 2022, p. e1230. DOI.org (Crossref), https://doi.org/10.1002/cl2.1230.

[9] Harris, Charles R., et al. "Array Programming with NumPy." Nature, vol. 585, no. 7825, Sept. 2020, pp. 357–62. DOI.org (Crossref), https://doi.org/10.1038/s41586-020-2649-2.

[10] Jung, Ralf, et al. "RustBelt: Securing the Foundations of the Rust Programming Language." Proceedings of the ACM on Programming Languages, vol. 2, no. POPL, Jan. 2018, pp. 1–34. DOI.org (Crossref), https://doi.org/10.1145/3158154.

[11] Liu, Jiyuan, et al. "Multiview Subspace Clustering via Co-Training Robust Data Representation." IEEE Transactions on Neural Networks and Learning Systems, vol. 33, no. 10, Oct. 2022, pp. 5177–89. DOI.org (Crossref), https://doi.org/10.1109/TNNLS.2021.3069424.

[12] Liu, Qian, et al. "ReUseData: An R/Bioconductor Tool for Reusable and Reproducible Genomic Data Management." BMC Bioinformatics, vol. 25, no. 1, Jan. 2024, p. 8. DOI.org (Crossref), https://doi.org/10.1186/s12859-023-05626-0.

[13] Martini, Simone. "Several Types of Types in Programming Languages." IFIP Advances in Information and Communication Technology, 2016. sci-hub.africa, https://doi.org/10.1007/978-3-319-47286-7_15.

[14] Nugroho, Agung Yuliyanto, and Nur Hamid Sutanto. Exploring the Code Foundation: A Literature Review of Data Structures in C++. 2024.

[15] Page, Matthew J., et al. "The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews." BMJ, Mar. 2021, p. n71. DOI.org (Crossref), https://doi.org/10.1136/bmj.n71.

[16] Sieczkowski, Filip, et al. "The Essence of Generalized Algebraic Data Types." Proc. ACM Program. Lang., vol. 8, no. POPL, Jan. 2024, p. 24:695-24:723. ACM Digital Library, https://doi.org/10.1145/3632866.

[17] Singh, Vidhan, et al. "Introduction to the Basic Data Types in Programming." Tuijin Jishu/Journal of Propulsion Technology, vol. 43, no. 4, Nov. 2023, pp. 246–49. DOI.org (Crossref), https://doi.org/10.52783/tjjpt.v43.i4.2345.

[18] Spidlen, Josef, et al. "Data File Standard for Flow Cytometry, Version FCS 3.2." Cytometry Part A, vol. 99, no. 1, Jan. 2021, pp. 100–02. DOI.org (Crossref), https://doi.org/10.1002/cyto.a.24225.

[19] Sterling, Jonathan, and Robert Harper. "Guarded Computational Type Theory." Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science [New York, NY, USA], LICS '18, 2018, pp. 879–88. ACM Digital Library, https://doi.org/10.1145/3209108.3209153.