Towards Hybrid and Reusable Type Inference for Dynamically-Typed Languages

Larisa Safina

[1]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France

larisa.safina@inria.fr

Abstract. *Type inference is a technique for automatically reconstructing types before program execution in the absence of explicit type annotations. While it has been adopted in many statically typed languages, applying type inference to dynamically-typed languages, poses challenges due to their runtime-oriented typing mechanisms. In this paper, we present* `HYPHE`*, a Hybrid Type Inference and Validation Framework for dynamically-typed languages, designed to combine within a single environment various approaches to type inference to leverage the strengths of each.* `HYPHE` *infers, stores, compares, and refines type information in a continuous loop integrated with developers. A preliminary evaluation in the Pharo Smalltalk environment indicates that combining multiple inference sources, as established in prior works, achieves broad type coverage while shifting computationally intensive analyses to offline processing and reusing accumulated inference data.*

1. Introduction and Background

Type inference is a fundamental technique in software engineering for automatically reconstructing types before execution in the absence of explicit annotations, important for program comprehension, performance optimization, and the early detection of defects [Kleinschmager et al. 2012]. While it has been explored and successfully adopted in many statically typed languages [Gundry et al. 2010], applying type inference in dynamically-typed languages poses specific challenges due to their runtime-oriented typing mechanisms where types are not enforced at compile time, and objects are not explicitly annotated with types [Castagna et al. 2024]. These languages are widely used in both research and industry (software development, scientific computing, artificial intelligence, etc) because of their flexibility for rapid prototyping and interactive programming. The absence of explicit type information, however, makes the software written in such languages harder to understand, analyze, optimize, and trust, restricting its usability by limiting tool support, and increasing the risk of runtime errors. Advancing type inference in such languages can lead to safer, faster, and more maintainable software, benefiting millions of developers in research and industry.

Over the years, numerous attempts have been made to formalize and implement type-related mechanisms in response to these challenges [Aiken and Murphy 1991, Aycock 2000]. For example, in Smalltalk, one of the earliest object-oriented dynamically-typed languages, the first approaches appeared as early as the 1980s [Suzuki 1981], and have since evolved into a wide range of type-related techniques and tools [Johnson 1986, Agesen 1995] that vary in methodology, complexity, usability, execution speed, and

precision. Currently, several state-of-the-art tools exist across different Smalltalk dialects, such as RoelTyper [Pluquet et al. 2009], J2Inferer [Passerini et al. 2014], or more recent heuristic-based approaches [Milojković 2017, Safina et al. 2025]. Each of these tools, however, addresses only part of the problem: their results are incomplete, sometimes inconsistent, and difficult to integrate into a unified representation of types. A recent contribution in this direction is TypeInfoTools (TIT) [Bliznicenko and Pergl 2025], which integrates several existing type inference tools and aggregates their results for easier comparison. TIT, however, focuses mainly on aggregation and visualization and does not define a homogeneous type model nor supports hybrid inference combined of various approaches and tools. Moreover, many of these inference techniques are computationally expensive, making them impractical to execute at runtime when developers expect immediate feedback.

To address the presented issues, we design HYPHE , a Hybrid Type Inference and Validation Framework for dynamically-typed languages. HYPHE aims to achieve the following goals:

(1) Unified integration of heterogeneous type inference sources. HYPHE provides an extensible framework that integrates static, dynamic, and LLM-based inference strategies within a single pipeline, enabling the production, storage, validation, and comparison of type information across multiple inference sources.

(2) A confidence-aware type model for reconciling heterogeneous results. Each inferred type is enriched with confidence scores and provenance metadata, and a dedicated merging mechanism resolves conflicts and consolidates type information originating from different inference techniques.

(3) A novel ”offline-online” inference workflow enabling incremental and community-driven refinement. HYPHE introduces a methodology in which computationally expensive inference and validation steps are executed offline, while previously accumulated and curated type information is reused to provide lightweight, on-demand analyses during development. This design enables continuous, recursive improvement of type information through runtime monitoring and contributions from the developer community.

With the development of HYPHE , we target both fundamental and practical contributions. Fundamentally, we propose a novel inference methodology that separates expensive analysis from interactive use and treats type information as an evolving, reusable artifact. Practically, we deliver an extensible toolchain and an open, community-maintained knowledge base of inferred types to support developers working with dynamically-typed languages.

2. Methodology

2.1. Conceptual overview

To implement HYPHE, we have chosen Smalltalk as an example of a dynamically-typed language for our case study, and more specifically its open-source dialect called Pharo [Black et al. 2009] as it provides several key advantages for this project: (1) Full access to the language kernel, virtual machine, and compiler infrastructure, which facilitates low-level experimentation with type inference and validation mechanisms.

(2) A vast ecosystem of libraries, including those for refactoring, testing, and program analysis, making it an ideal environment for evaluating hybrid approaches. (3) Used both in research and industry, built around an active Pharo consortium and community of developments, which will serve as a valuable source of feedback and external validation throughout the project's evolution.

HYPHE is provided as an external library that can be installed into the Pharo IDE and augments the development environment with type inference, validation, and type-checking capabilities. After installation, an engineer should be able to download previously collected type information for the Pharo system libraries and for external libraries currently loaded in the image. This information is retrieved from a shared public repository and stored locally. The IDE presents the accumulated type information directly during development, for example by displaying available types when a developer is typing a method name or inspecting code, and makes this information available for type checking and program comprehension.

For packages where type information is missing, incomplete, or requires refinement, the engineer should be able to initiate additional analyses through the provided graphical user interface. HYPHE supports selecting and configuring different analysis techniques, including static inference tools, runtime monitoring, and LLM-based inference. The scope and cost of the analysis can be controlled by choosing appropriate techniques, adjusting confidence levels, merging strategies, and optionally providing user-defined type annotations. Runtime monitoring captures both offline feedback, collected during program execution and analyzed afterward, and online feedback, recording type mismatches observed at runtime for later analysis.

All inferred and collected type information is integrated by a type-merging component, which reconciles heterogeneous results, selects the most probable types, and stores them in a local type repository. The framework supports recursive refinement by feeding newly inferred type information back into subsequent analyses. Finally, validated results can be persisted and shared with the community by contributing them back to the shared repository, enabling reuse and continuous improvement of type knowledge across projects.

Implementing such a framework requires careful work on four fronts: (1) designing a type model that can be used across tools with heterogeneous outputs, (2) integrating existing tools into a unified pipeline and developing new ones, (3) incorporating LLM-based support, and (4) adding continuous integration and monitoring capabilities. Below, we discuss each of these directions in more detail.

2.2. Designing a Unified Model for Storing Type Information

HYPHE requires a confidence-based [Kazerounian et al. 2020] type model as an essential foundation for representing type information. This model is not intended as a full-fledged formal type system, but rather as a meta-level structure that collects heterogeneous results from static inference tools, runtime monitors, and LLM-based suggestions. Yet the model should be capable of defining the types of basic entities in the Pharo language (e.g., classes, methods, arguments, return values), as well as the relations between these types. Since Pharo relies on duck typing (where an object is considered suitable if it responds to a given message) which naturally aligns with the principles of structural typing, in

which type compatibility is determined by the presence of required methods rather than by belonging to a declared class hierarchy. Consequently, the model should be capable of capturing both inheritance relations and polymorphism, support unions of types, and define relaxed typing rules that reflect this structural perspective.

To tackle the challenge of resolving discrepancies between the outputs of existing tool, each type result in the model is assigned with a source and a confidence level determining how reliable is it. Confidence level plays important role in implementation of type merging algorithm that should serve for unification of type results, e.g it helps to choose between contradicting types, and it has impact on filtering and visualization of type results provided for a developer. However, if the discrepancies prove to be too large to be resolved within a single model, we consider maintaining multiple variants or forking tools where necessary.

As a next step, an important challenge in unifying types retrieved from different sources would be the development of a more sophisticated merging algorithm. The current implementation prioritizes confidence levels and common ancestors, but a more advanced approach could incorporate both structural and nominal typing to define subtyping relations more precisely and produce finer-grained types. Although this direction is promising, it involves substantial formal work and considerable complexity, and therefore is not a short-term priority for the current stage of the project.

2.3. Tooling for Dynamic and Static Type Inference Integration

`HYPHE` builds upon multiple type inference projects (e.g. TIT project) and reuses the core tools originally developed within it. However, we have observed many other promising tools and ideas developed earlier being currently abandoned and outdated not necessarily due to conceptual shortcomings, but because they were created at a time when the required computational resources were unavailable, or because the original developers left the projects (e.g., after completing a PhD, finishing a postdoc contract, retiring, or after company acquisitions). We plan to re-examine these ideas with modern infrastructure and renewed maintenance may reveal valuable opportunities for further progress.

To manage the retrieved type information effectively, `HYPHE` requires an environment built around both external and local repositories, supported by Continuous Integration (CI) mechanisms for synchronization. This ensures that type information remains up to date as the code evolves and that any discrepancies are promptly detected and resolved. Specifically, we organize and store type information for different Pharo language versions (for system libraries) and for external libraries in a GitHub repository, using structured formats comparable to the Typeshed repository for Python [Typeshed 2025]. CI pipelines can be configured to re-run selected inference strategies whenever the target code is modified. Developers will be able to adjust the sensitivity of the inference process and optionally enable additional dynamic checks, as frequent analysis may introduce performance overhead.

2.4. LLM-Assisted Type Inference System

We are experimenting in using LLMs as an additional source of type inference. Recent advances in LLMs showing decent results in benchmarks for code-related tasks [Wang et al. 2025] give opportunities to infer types from context, call graphs, and

naming conventions. We plan to design prompt templates based on program analysis and systematically evaluate different LLMs according to cost, speed, and precision. The inferred results will be stored in the unified type repository and compared against more ”conservative” inference results (which have higher confidence level and will serve as a ground truth dataset in the absence of other results). A configurable UI (behind the controller) will be offered to developers to tune LLMs parameters and confidence-level agains other sources of type results.

However, this part of the project remains experimental and rather under-specified as the number of important aspects remain open. We need to define working prompt-engineering strategies, systematic ways to detect and mitigate hallucinations, and address the cost-limitations of LLMs (probably via batching mode when possible).

2.5. Monitoring and Feedback

With the mechanisms of runtime monitoring and feedback we close the loop of the complete type inference ecosystem and supporting continuous validation and refinement of type results which accuracy and precision will be progressively improved over time. More specifically, we will instrument the runtime to detect type errors during execution, to store types assigned to variables during execution, to integrate a feedback loop to update type information semi-automatically in case of trivial discrepancies, and build diagnostic tools to cross-check runtime behavior against the stored repository. The system will be able to generate error reports or even propose pull requests to repair outdated information. In parallel, sandboxing techniques will be explored to test candidate types in isolation. As runtime monitoring mechanisms may introduce performance overhead, we plan to mitigate it through selective instrumentation and careful measurement as part of the evaluation activities.

3. Preliminary Evaluation

To demonstrate the feasibility and operationalization of HYPHE's confidence-driven type model, we implemented a prototype tool in Pharo Smalltalk[1]. At this stage of the project, we have realized the unification of heterogeneous type information by defining a common JSON-based format for representing inferred types and by implementing several merging strategies based on confidence weighting, voting, and tiered prioritization, as described in prior work. In addition, we implemented Mycelium, a Typeshed-like repository for dynamically-typed languages[2] , which stores type information for Pharo system libraries and external libraries. The prototype integrates Mycelium with HYPHE, enabling automatic retrieval of relevant type information upon installation and supporting the contribution of newly inferred or refined types back to the shared repository.

We evaluated HYPHE across three datasets that differs in the level of methods nature and polymorphism: `WordArray` (18 methods, low-level primitives), `Collections-Sequenceable` (457 methods, standard collections), and `Kernel & Streams` (1,772 methods, core abstractions).

Coverage and Performance. Overall, HYPHE achieved inference coverage of 89% for WordArray, 100% for Collections-Sequenceable (457/457 methods), and 100%

[1]Available at [Hyalinos 2025] and described in detail in [Safina 2026].

[2]`https://github.com/hyalinos/mycelium`

for Kernel & Streams (1,772/1,772 methods). Lightweight heuristic inference alone covered 58% of Pharo core libraries in under one minute, demonstrating the practical efficiency of fast pre-analysis before invoking computationally expensive tools.

Source Agreement and Conflict Rates. Table 1 shows inferencer agreement patterns across datasets. High agreement rates (90-98% full consensus) indicate that for most methods, inference sources converge naturally, requiring minimal conflict resolution. However, 9.6% (Collections-Sequenceable) and 2.0% (Kernel & Streams) of multi-source methods exhibited type disagreements where our conflict resolution mechanisms were actively engaged.

Table 1. Inferencer agreement and conflict resolution statistics

Dataset	Methods	Full Agreement	Conflicts Resolved
WordArray	18	100%	0
Collections-Seq.	457	90.4%	44 (9.6%)
Kernel & Streams	1,772	98.0%	15 (2.0% of multi-source)

Confidence Calibration Results. Across all datasets, average confidence scores ranged from 0.28 (highly conservative Biased Consensus) to 0.99 (optimistic Weighted Voting), with most strategies producing confidences between 0.75-0.96. Where ground truth was available (106 methods in Collections-Sequenceable), inferred types were compatible in approximately 70% of validatable cases, with errors primarily from Type Info Tools misidentifying implementation details as semantic return types.

Performance and Computational Costs. We observed a wide variance in computational costs across different type inference approaches. Lightweight heuristic analyses completed in under one minute, making them suitable for interactive use, whereas constraint-based inference required several hours to analyze roughly half of the system methods, and offline test profiling consumed multiple minutes on test-heavy libraries which could be acceptable for CI pipelines but impractical for interactive sessions. Even relatively fast inference techniques introduced delays of hundreds of milliseconds per method when invoked synchronously, resulting in noticeable latency during type checking. These observations motivate HYPHE's tiered architecture, which pre-computes expensive analyses offline and serves their results from the type repository, while reserving only lightweight heuristics for on-demand inference to preserve developer responsiveness.

Validation. These results demonstrate that HYPHE's confidence-driven type model can be operationalized through concrete mechanisms for confidence computation and conflict resolution. The implemented tool successfully combines multiple inference sources, with conflict resolution actively engaged in approximately 10% of cases. Where ground truth exists, inference accuracy is reasonable (70%), though validating highly polymorphic methods remains challenging as they often lack concrete ground truth.

4. Future Work and Limitations

The project is still in its early stage and remains under active development. The immediate next step we plan is a systematic validation of the proposed approach, focusing on the trade-offs between inference correctness, computational cost, and developer usability. We

aim to make this validation process reusable as HYPHE grows and requires continuous and incremental assessing of its tool-set, inference techniques and datasets. This includes benchmarking precision and recall across inference strategies, measuring the overhead of runtime monitoring, and collecting developer feedback on usability and integration into everyday development workflows. Such a reusable evaluation infrastructure will allow the framework to evolve alongside the rapidly changing landscape of type inference techniques.

A key direction of future work is the implementation of incremental inference mechanisms that reuse previously collected type information to improve the results of other analyses. While the current prototype supports storage and retrieval of type data, enabling cross-tool feedback requires additional instrumentation so that newly inferred or validated types can guide subsequent analyses, refine confidence estimates, and reduce redundant computation. Another direction is the integration of large language models as an optional inference source. We plan to evaluate LLMs for type inference based on program context and usage patterns, focusing on accuracy, cost, and latency, and to mitigate hallucinations through comparison with conservative inference results.

In the longer term, HYPHE aims to generalize beyond Pharo by identifying language-agnostic abstractions in the type model and assessing their applicability to other dynamically-typed languages through a small cross-language case study. The integration of heterogeneous tools also introduces risks related to consistency and scalability, which we plan to address through continuous validation, caching, offline prioritization, and cross-source comparison.

5. Conclusion

In this work, we presented HYPHE, a framework that takes a step towards hybrid and reusable type inference for dynamically-typed languages. We demonstrated the feasibility of a confidence-driven approach by implementing a prototype in Pharo that unifies heterogeneous type information through a common representation and multiple merging strategies. We also introduced Mycelium, a Typeshed-like repository that enables reuse, sharing, and incremental refinement of type information across projects. Our preliminary evaluation on core Pharo libraries shows that combining multiple inference sources can achieve broad coverage while motivating an ”offline–online” workflow that balances analysis cost and developer responsiveness. Together, these results indicate that a multi-source, knowledge-oriented approach to type inference is both practical and promising, while leaving the space open for further research and validation.

References

Agesen, O. (1995). The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP ’95, page 2–26, Berlin, Heidelberg. Springer-Verlag.

Aiken, A. and Murphy, B. (1991). Static type inference in a dynamically typed language. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’91, page 279–290, New York, NY, USA. ACM.

Aycock, J. (2000). Aggressive type inference. International Python Conference.

Black, A. P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., and Denker, M. (2009). *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland.

Bliznicenko, J. and Pergl, R. (2025). Combining type inference techniques for semi-automatic UML generation from pharo code. *J. Comput. Lang.*, 82:101300.

Castagna, G., Laurent, M., and Nguy˜ên, K. (2024). Polymorphic type inference for dynamic languages. *Proc. ACM Program. Lang.*, 8(POPL).

Gundry, A., McBride, C., and McKinna, J. (2010). Type inference in context. In *Proceedings of MSFP '10*, page 43–54, New York, NY, USA. ACM.

Hyalinos (2025). Hyalinos: A transparent type inference framework for pharo smalltalk. `https://github.com/hyalinos`. Accessed: 2026-01-28.

Johnson, R. E. (1986). Type-checking smalltalk. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, page 315–321, New York, NY, USA. ACM.

Kazerounian, M., Ren, B. M., and Foster, J. S. (2020). Sound, heuristic type annotation inference for ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, page 112–125, New York, NY, USA. Association for Computing Machinery.

Kleinschmager, S., Robbes, R., Stefik, A., Hanenberg, S., and Tanter, E. (2012). Do static type systems improve the maintainability of software systems? an empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162.

Milojković, N. (2017). *Augmenting Type Inference with Lightweight Heuristics*. PhD thesis, University of Bern.

Passerini, N., Tesone, P., and Ducasse, S. (2014). An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types. In *International Workshop on Smalltalk Technologies (IWST'14)*.

Pluquet, F., Marot, A., and Wuyts, R. (2009). Fast type reconstruction for dynamically typed programming languages. *SIGPLAN Not.*, 44(12):69–78.

Safina, L. (2026). Hyalinos: From type inference to community knowledge. In *Proceedings of the 34th IEEE/ACM International Conference on Program Comprehension (ICPC '26)*, Rio de Janeiro, Brazil. ACM. Accepted for publication.

Safina, L., Blizničenko, J., and Pergl, R. (2025). Even lighter than lightweight: Augmenting type inference with primitive heuristics. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST)*, Gdańsk, Poland.

Suzuki, N. (1981). Inferring types in smalltalk. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 187–199, New York, NY, USA. ACM.

Typeshed (2025). `https://github.com/python/typeshed`. GitHub repository.

Wang, K., Li, T., Zhang, X., Wang, C., Sun, W., Liu, Y., and Shi, B. (2025). Software development life cycle perspective: A survey of benchmarks for code large language models and agents.