Even Lighter Than Lightweight: Augmenting Type Inference with Primitive Heuristics

Larisa Safina[1], Jan Bliznicenko[2] and Robert Pergl[3]

[1]Inria Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL F-59000 Lille, FR
[2]Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, Prague, 16000, CZ
[3]Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, Prague, 16000, CZ

Abstract

Type inference, as a technique of automatic deduction of types in programming languages, plays an important role in code correctness, maintainability, and performance optimization. In dynamically typed languages, type inference presents significant challenges due to their flexible, runtime-oriented typing mechanisms. This paper explores a novel set of primitive heuristics designed to augment type inference in Pharo. We demonstrate that even minimal hints, such as method naming conventions and collection patterns, can produce meaningful improvements in inferred type coverage.

1. Introduction

Type inference is a technique in programming languages for the automatic reconstruction of types before execution without the presence of explicit type annotations. It has been explored and successfully adopted in many statically typed programming languages [1, 2, 3], helping to improve their correctness, maintainability, and performance optimization [4, 5, 6]. However, it presents certain challenges when applied to dynamically typed languages due to the volatility of variable types and runtime-oriented typing mechanisms [7]. Despite these challenges, there have been many attempts to formalize and implement type inference in dynamically typed languages [8, 9, 10].

Smalltalk, as one of the dynamically typed programming languages, has not been exempt from the challenges of introducing type inference. The first approaches were presented in the early 1980s with the seminal works of Suzuki (1981) [11] and Borning and Ingalls [12]. The most active period of research in this area occurred during the late 1980s and early 1990s, with significant contributions by Johnson (1986) [13], Graver (1989) [14], Palsberg and Schwartzbach (1991) [15], Bracha and Griswold (1993) [16], Agesen (1995) [17], and others. The research focus changed in the 2000s towards more flexible approaches as pluggable type systems (Bracha (2004) [18]) and gradual typing (Allende et al. (2014) [19]). In recent years the interest has been renewed with the focus on the practical approach - developing new inference tools for IDE, enhancing code navigation and code analysis, e.g. RoelTyper by Pluquet and Wuyts (2009) [20], heuristic-based tool set by Lazarevic (2017) [21], followed by the most recent tool-integrating approach by Bliznicenko (2025) [22].

These approaches to introducing type systems, type checking, and type inference mechanisms in Smalltalk vary widely in complexity, developer usability, execution speed, and result precision. Due to the language's highly dynamic nature and the prevalence of polymorphic methods, type inference techniques often demand substantial computational resources and time to achieve an adequate level of precision. Conversely, more lightweight methods may sacrifice precision to remain practically usable. We believe that achieving a balance between these two extremes is one reason type inference has never been fully integrated into Smalltalk IDEs at a level where developers can rely on it interactively and efficiently.

IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland
larisa.safina@inria.fr (L. Safina); jan.bliznicenko@fit.cvut.cz (J. Bliznicenko); robert.pergl@fit.cvut.cz (R. Pergl)
0000-0002-4490-7451 (L. Safina); 0000-0002-5280-7151 (J. Bliznicenko); 0000-0003-2980-4400 (R. Pergl)

To address this challenge, we propose complementing the more precise and computationally heavy type inference techniques with information derived from lightweight type heuristics. We also suggest precalculating type information for code that is unlikely to change frequently. In our work, we focus on Pharo, as a modern open-source continuously developed and maintained Smalltalk dialect. We propose inferring in advance the types of the standard Pharo image and possibly some third-party libraries. The simple heuristics introduced in this paper, despite their minimal computational cost, can help infer return types for nearly half of all methods in the Pharo image. This heuristic-derived type information can serve as valuable input to accelerate more comprehensive inference systems while also enhancing coverage.

The paper is structured as follows. Section 2 reviews the state of the art in type inference (primarily in Smalltalk), with a focus on recently published and implemented techniques, including both traditional inference tools and AI-assisted methods. Section 3 presents the set of lightweight heuristics that have been implemented. Section 4 details the implementation, tool support, and validation of results. Section 5 discusses the limitations of the current approach and outlines potential directions for future work. Finally, Section 6 concludes the paper.

2. State of the art

While the introduction outlined a broad spectrum of type inference approaches proposed over the years for Smalltalk, this section narrows the focus to the most recent and practically validated contributions. Specifically, we examine works that have resulted in concrete implementations and have been published in peer-reviewed academic literature, so we excluded some implementations that have been developed within the community and are in use, but have not been published thus having no academic validation.

Spoon and Shivers [23] introduced DDP, a type inference algorithm for Smalltalk. It is partially based on the Cartesian Product Algorithm (CPA) by Agesen [24] that infers types only for requested program parts and prunes irrelevant goals to improve scalability. The evaluation presented in the paper shows an inference precision of about 30-40%, and while higher precision is achievable, it requires executing a larger number of cross-dependent computations, making exhaustive analysis increasingly impractical. The authors proposed several directions for improving performance by reusing previously computed subgoals and parallelizing inference queries.

Pluquet et al. created the RoelTyper [20], a type reconstruction tool focusing more on the speed of analysis rather than on precision, aiming to provide fast and practical support within the IDE. It reconstructs types for local, instance, and argument variables by analyzing message sends within a method, avoiding deeper class hierarchies or global analysis to maintain responsiveness.

Passerini et al. [25] developed J2Inferer, a constraint-based type inference tool for Pharo, similar in spirit to the work of Spoon and Shivers, but without the use of subgoal pruning. It was implemented as a pluggable type system and supporting advanced features like blocks and generics. It targets real industrial level environment, the practical-oriented approach we also aim to achieve. According to the paper, J2Inferer has been evaluated only on small codebase, our tests show certain delays in extracting types for the whole Pharo image.

2.1. Use of type heuristics

We acknowledge that many tools and inference engines may internally apply similar naming- or pattern-based heuristics, such as assuming `is*` or `==` returns a Boolean. However, due to their simplicity, these heuristics are often not explicitly documented or evaluated in publications. As such, our comparison is limited to the subset of tools and approaches that describe these strategies in their academic work. Several prior works have recognized that message sends such as `==`, `<`, includes and others implying a `Boolean` return type (e.g [20, 17]). However, these signals are typically treated in isolation, without forming part of a broader, systematic heuristic strategy.

The work most closely related to ours is that of Lazarević [21], who proposed four heuristics to improve the precision of simple type inference techniques in highly polymorphic code. Her approach

is based upon RoelTyper and the CPA, selected for their speed and practical use. Two heuristics rely on static methods-to-class instantiation frequency and class name occurrence, while a third uses dynamic data from inline caches. The last heuristic extracts type hints from method argument names to compensate for losses due to reflection and dynamic features.

3. Heuristic-Based Type Inference and Design Assumptions

3.1. Lightweight Heuristics

To complement existing inference techniques, we introduce a set of simple heuristics based on superficial but broadly applicable properties of methods in Pharo 13. These heuristics are designed to be extremely lightweight (in terms of implementation and usage complexity) and computationally cheap (regarding the runtime cost) while still providing useful signals for type inference, particularly in large-scale analyses. The heuristics cover approximately 73,000 of the 127,000 methods present in a standard Pharo 13 image. The full list of heuristics is given in the Table 2 of the Appendix.

3.1.1. Methods Without Explicit Returns

Roughly 57,000 methods in the Pharo 13 image do not contain an `OCReturnNode` and therefore implicitly return `self`. These are typically utility methods that either mutate internal state or enable method chaining. Since they always return the receiver's type, they contribute little to interprocedural type inference. However, if their return types remain unknown, they can still introduce uncertainty and impede more complex analyses by breaking inference chains or adding ambiguity. Given their simplicity and predictability, it is both feasible and beneficial to infer their types upfront, thereby reducing noise and improving the overall reliability of higher-level inference tools.

3.1.2. Methods With Heuristically Meaningful Names

We use simple naming patterns to infer likely return types for around 10,000 methods whose selectors strongly indicate their intent. These name-based heuristics are inspired by conventions commonly observed in Smalltalk libraries:

- Boolean-returning predicates (approx. 6500):
  - Methods starting with `is` (e.g., `isEmpty`, `isValid`)
  - Methods starting with `has` (e.g., `hasChildren`, `hasElements`)
  - Methods like `includes` or having the similar logic
  - Comparison operators, e.g `=`, `<`, `>`
- String-returning methods e.g `asString` (750 methods)
- Numerical methods, e.g `hash`, `size` (approx. 550)

These heuristics exploit strong naming and behavioral conventions in the Smalltalk ecosystem, making them both predictable and effective in practice. While they do not offer deep inference, they significantly improve baseline type guesses in the absence of more complex signals.

3.1.3. Heuristics for Collection Methods

Collections are ubiquitous in Smalltalk, and we identified 440 collection-related methods that exhibit consistent and easily classifiable return patterns. These patterns allow for reliable type inference based on common collection behavior. Here, we excluded from the list heuristics that overlap with the name-based ones, e.g. `size`, `includes:`.

- Numeric-returning methods, e.g. `indexOf:`, `findIndexForKey:`

- String-returning methods, e.g. `,` (as concatenation)
- Boolean-returning methods, e.g. `anySatisfy:`
- Transformation methods ("asSomething"), e.g. `asSet`, `asOrderedCollection`, `asBag`
- Collection-preserving methods: `addAll:`, `copy`, `groupedBy:`

3.1.4. Methods With Single Return

In addition to name-based and collection-specific heuristics, we identify a set of return patterns that offer strong and easily extractable types. Notably, if a method contains only a single return statement, we can safely assume that the return type of the method corresponds directly to the expression returned. Here we do not calculate the total of all inferred methods as they map often with the methods identified by other heuristics presented earlier. We consider methods returning:

- A class reference (e.g., `^SomeClass`) returns `SomeClass class` type (982 methods)
- A class instance pattern `SomeClass new` returns `SomeClass` type (394 methods)
- A `LiteralNode` (e.g., string, number, symbol): These literals directly map to their corresponding return types e.g. `String`, `Integer`, `Symbol`, etc. (8745 methods)
- `self`: This is a prevalent pattern with approx. 57000 methods. The method returns the same type as the class it is defined in.
- `self new`: This pattern, while less frequent (34 methods), implies that the method acts as a factory and returns a new instance of the receiver's class
- `nil` returns `UndefinedObject` type (278 methods)

These return-based heuristics are particularly appealing because they require no external analysis, only a syntactic check of the return expression. They are also highly reliable, as the returned expression often directly encodes the type information the inference engine seeks.

3.1.5. Return Types of Test Methods

There are approximately 22,500 test-related methods in the Pharo 13 image that we did not include in the total count of analyzed methods. These are typically implemented as methods within subclasses of `TestCase` (e.g., `setUp`) or follow a naming convention beginning with `test`. By convention, most of these methods return `self`, `Boolean`, or `nil`. However, they do not play a significant role in type analysis, as they rarely participate in the program logic beyond asserting test results. Consequently, they are not useful for type propagation. We nevertheless track their presence in order to reduce the scope of methods requiring deeper analysis and to avoid unnecessary computation.

3.2. On Type Semantics in Smalltalk

Smalltalk, and Pharo in particular, is a dynamically typed language with no built-in static type system. This means that types are not enforced at compile time, and objects are not annotated with declared types. Instead, Smalltalk relies on duck typing - if an object responds to a message, it is considered appropriate for use in that context. This aligns with the principles of structural typing, in which type compatibility is determined by the presence of required methods rather than by inheritance or nominal declarations.

Given this, Smalltalk does not implement a formal type system in the traditional static sense. However, for the purposes of type inference, various tools (such as RoelTyper and J2Inferer) approximate type information by analyzing method definitions, message sends, and class hierarchies. Each of these tools interprets the notion of type slightly differently, depending on its inference goals and constraints.

In this work, we adopt a pragmatic and minimalist notion of type, used solely to support comparison across inference tools. Specifically, we make the following assumptions:

- Return types only. We focus exclusively on inferring the return type of methods. Argument types, block parameters, and variable types are not considered.
- Single type assumption. We assume that each method has a single dominant return type. We do not currently support multiple disjoint return types (e.g., `Integer|Exception`).
- Simplified type equivalence. When comparing results from different tools, we allow a relaxed interpretation of type compatibility. For instance, we consider `Integer` and `Number` to be related through a common supertype or behavioral subset, even though Smalltalk does not enforce such a relationship statically.

This simplification allows us to focus on the inference mechanisms and outputs of different tools, without committing to a specific formal type system for Smalltalk. We believe that our tool could benefit in the future from some type-related extensions and a more rigorous type model in general (e.g., supporting union types), however such extensions are for the moment not in the scope of this paper.

4. Validation and Tool Support

The proposed heuristics have been implemented and tested within the Pharo 13 image. The entire analysis of the full image, which contains approximately 127,000 methods, completes in around 1 minute on a Mac M2 with 24 GB of RAM. This confirms the practicality of applying the heuristics even to large codebases in an interactive development context.

4.1. Internal Validation

As a basic self-validation mechanism, we provide a sanity check on the heuristics to identify internal contradictions between them. Specifically, if multiple heuristics suggest types for the same method, the results are cross-verified to ensure consistency. Contradictions (e.g., one heuristic suggesting `Boolean`, another suggesting `String`) are flagged for further inspection.

4.2. Comparison With Other Tools

To evaluate the results provided by our heuristics, we compared them with those produced by TypeInfoTools (TIT), a framework inspired by previous research on combining type inference techniques [22] that gathers and combines type information from various sources. Since TIT already integrates several existing tools and techniques, the comparison is greatly simplified. To clarify our use of TIT, a description of its main principles follows.

The core of the TIT framework uses multiple weighted information sources: the static type inference tools RoelTyper and J2Inferer, simple heuristics based on comparing variable names with class names (e.g., variables named `count` are usually `Integer`), and package relations (if an inferred package depends on another package, its classes are more likely to be relevant as data types).

Although there are more static type inference tools than RoelTyper and J2Inferer, most of them focus on variables and do not provide a way to infer a return type. Furthermore, name-based validation is only applied to methods categorised as getters (i.e., in the accessing protocol and having no arguments).

All these sources return a list of possible types instead of a single suggestion, so their results are merged, and their weights are summed. Static type inference tools have higher weights than simpler heuristics like package relations. The maximum weight is then determined, and all types with that maximum weight (generally meaning they are flagged by the largest number of the most relevant information sources) are selected. Since only one type is needed as a suggestion, the common superclass of these selected types is found. A result is only returned if this type is not `Object`, `ProtoObject`, or `nil`.

There is a limitation caused by the use of older type inference tools. These tools were developed many years ago for earlier versions of Pharo, and despite some fixes and updates, there were many

cases where they failed, often causing infinite loops, infinite recursive calls, or unskippable errors. As a consequence, we restricted our comparison to selected libraries[1].

4.2.1. Type Inference Coverage

The data on the amount of the methods for which the types have been inferred shows a trade-off between the speed and the completeness. TIT consistently identifies more method types across packages, sometimes covering over 70% of methods, when the heuristic-based tool covers fewer (approx. 45–55%) but being significantly faster. However, the overlap between the two tools is substantial, often being more than 40% of the total methods per package, indicating that a large amount of TIT's results can be substituted by the ones provided by heuristics. Thus we can reinforce both approaches: using fast heuristics to be fed to the tool chain to narrow the scope for deeper inference by TIT, optimizing overall performance without sacrificing the coverage.

4.2.2. Execution time

Figure 1 shows the execution time of both tools with respect to the number of methods in a package. The execution time of the TIT tool grows faster than linear for larger packages with some super-linear spikes that could be explain with the number of methods in a package (depending on internal package complexity or algorithm branching), suggesting, however, predictable scalability within typical package sizes but with a potential to become a concern for larger packages or in cross-package analysis. In contrast to TIT, the heuristic-based tool shows the stable, near-constant execution time across packages, typically completing its analysis around 0.5-1 second regardless of the package size suggesting that its performance is decoupled from the codebase complexity or method count. These difference in the execution time complexity shows that TIT can profit from using of pre-calculated heuristic results to reduce the calculation cost.

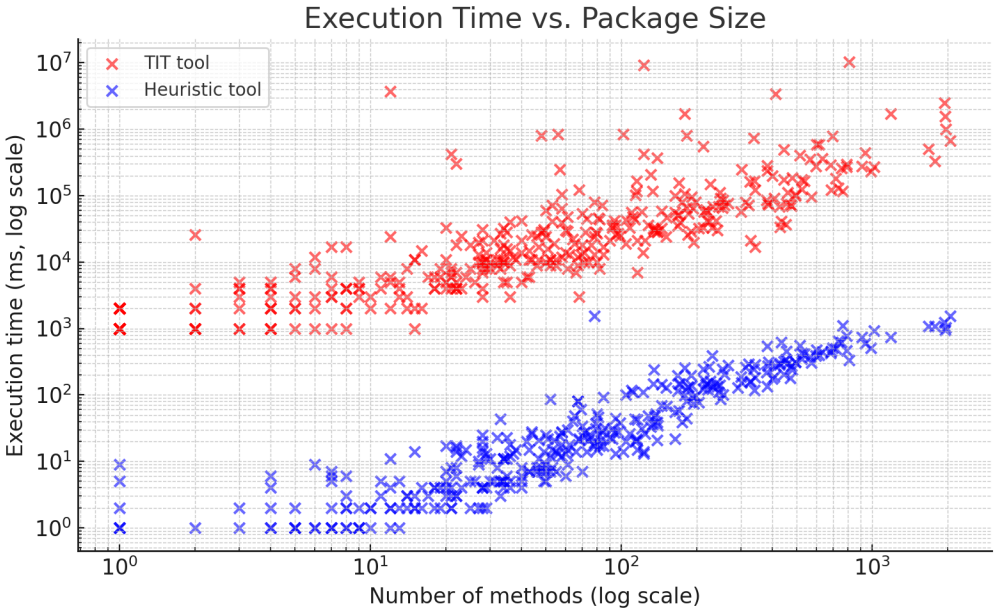

Figure 1: Execution time of TIT and heuristic-based tools vs. number of methods per package.

[1]The file with the results can be accessed on GitHub

4.2.3. High Coverage but Costly TIT Cases

There are certain cases where computation time of TIT grows superlinearly showing at the same time high coverage (see Table 1). This suggests that such packages can be good candidates for optimization with pre-calculated types.

Table 1
High-coverage, high-cost TIT examples

Library	Methods	Meth. discovered by TIT	TIT meth. coverage	TIT exec. time (ms)
Spec2-Layout	326	310	95.1%	163000
Roassal-Layouts	631	479	76%	360000
Roassal	525	329	62%	174000

4.3. Visualization Support

To assist with analysis and debugging, we added runtime visualization of the inferred type information stored in the CSV file (can be integrated to the image to avoid reading file overhead). We display the type for the requested method if it is present in our library (see Fig. 2). This visualization has negligible performance impact and can be enabled on demand during live exploration of the codebase.

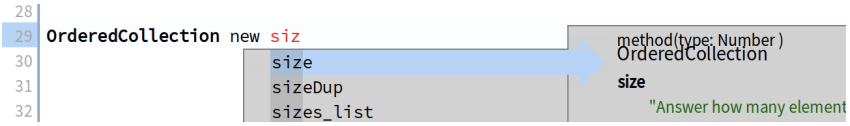

Figure 2: Inferred method type in the Pharo image.

5. Future Work and Limitations

Our primary goal for future work is to integrate our heuristic-based solution with existing type inference tools and evaluate its impact on both execution speed and type coverage. In addition to this main objective, we also identify several other promising directions for future research (as well as certain limitations) especially in the areas of scalability, interactivity, and AI-assisted inference.

5.1. Interactive Type Refinement

Future iterations may introduce an interactive system where the developer can consult or even correct or confirm inferred types. For ambiguous results multiple suggestions can be displayed (from heuristics, TIT, preferably integrated into one framework) and allow user selection. The code can be annotated with inferred types after a delay, code changes (in methods or classes) can trigger type reanalysis.

Another direction is to integrate runtime type mismatch detection as a feedback mechanism for evaluating and refining type inference heuristics. In this case, the system would monitor program execution and record instances where a type-related runtime error occurs such as sending an unexpected message to an object or failing a type-sensitive operation. These mismatches could be correlated with previously inferred types, and if a discrepancy is detected, it may indicate that a heuristic produced an incorrect or overly general type. With the user's consent, such data could be logged anonymously and aggregated to improve future inference quality. Over time, this feedback loop could help highlight which heuristics are most error-prone and guide the design of more precise or context-aware rules.

This mechanism could be embedded into a continuous runtime monitoring framework, passively analyzing type behavior during development or testing phases. In environments like Pharo, where live programming is the norm, such runtime insight offers a valuable complement to the static analysis allowing the system to adapt and learn from real usage patterns.

5.2. Mining type information from the tests assertion

Another promising direction is to mine type information from the test code. Although at the moment we ignore the results of test methods and exclude them from type inference, we can use them in the future to collect implicit type information based on the execution result of the assertion statements, e.g., from the example below we can assume that the return type of the `doSomething` method is Integer

```
self assert: someObject doSomething = 100
```

5.3. Toward a Type Repository for Smalltalk

Inspired by Python's `typeshed` [26], we envision a type information repository for Smalltalk, containing external type declarations for core and community libraries. This would enable:

- Continuous integration pipelines that recheck types when code is updated
- Detection of type changes across Pharo versions or conflicting definitions in different packages
- Precomputed types for stable libraries and on-the-fly inference for user code

5.4. AI-Assisted Inference

As an experimental feature, we implemented a basic integration with several Large Language Models (LLMs) from OpenAI and Mistral to issue real-time queries for predicting the most likely return type of a given method, class, and package. The returned results did not differ from those obtained using our heuristics; however, we tested this only on a limited set of methods due to the real-time nature of the process.

We believe that integration with LLMs can be beneficial as an additional source for type inference or as a means of validating results. However, this approach has limitations that would require additional workarounds to address:

- Performance: Real-time queries to an external service are slow, blocking typing and navigation. Responses may not arrive in time to be useful during development.
- Incomplete Coverage: The assistant currently works only for code in the standard Pharo image. User-defined or freshly written methods are not analyzed.
- Outdated Knowledge: the LLMs' knowledge of Pharo may be out of date, especially for newer releases.
- Precision: Responses may lack certainty.
- Lack of Validation: LLMs might suggest hallucinated class names or incorrect types. A mechanism should be added to verify that suggested types exist in the system.
- Cost: Some models API, e,g OpenAI that we used for experiments, are not free and require a paid account.
- Limited Smalltalk Familiarity: Most LLMs are significantly more familiar with mainstream languages like Python, limiting their effectiveness with Pharo.

Despite these challenges, most of the above limitations can be successfully addressed with proper architectural and design strategies: e.g. cache management and batch processing for improving performance, providing a model with method sources for improving coverage and knowledge issues, fine-tuning a model for more precision, or combining with static analysis for better validation. These improvements open up a space for building complex hybrid solutions for type inference.

5.5. Open questions

Finally, there are still many open challenges remained which deserve careful consideration and can serve as a foundation for more future work.

- Can we efficiently fork, parallelize or even offload the type inference computations to the external image?
- Can we exclude irrelevant packages from analysis and how to identify them?
- Is there a possibility for using a secure sandbox in Pharo for running code during inference without effects?
- How do we ensure inferred types remain valid and consistent over time?

Despite the presence of open questions and limitations, we believe that there is a potential for a creating a combined, interactive, and adaptive approach to type inference in Pharo that can balance primitive heuristics, static analysis, and LLM-based inference in a unified workflow.

6. Conclusion

This paper presents a set of lightweight heuristics designed to enhance type inference in dynamically typed Smalltalk environments. Our approach is deliberately simple, designed for speed and ease of integration into interactive development environments. We demonstrated that even minimal hints, such as method naming conventions and collection patterns, can produce meaningful improvements in inferred type coverage. The tool has been tested on Pharo 13 image being able to infer the types of 58% of methods within 1 minute, thus showing the performance results that support its practicality.

In future work, we aim to integrate our heuristics with more sophisticated type inference tools to combine their broader results with our low execution time aiming to have a hybrid solution that is both fast and precise. Additional research directions include interactive type refinement, mining type information from tests, building a shared type repository for Smalltalk and Pharo, and integrating AI-assisted inference. We foresee both opportunities and challenges in pursuing these directions, but we believe that addressing them could lead to a powerful, scalable, and adaptive type inference workflow.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. Gundry, C. McBride, J. McKinna, Type inference in context, in: Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 43–54. URL: https://doi.org/10.1145/1863597.1863608. doi:10.1145/1863597.1863608.

[2] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82, Association for Computing Machinery, New York, NY, USA, 1982, p. 207–212. URL: https://doi.org/10.1145/582153.582176. doi:10.1145/582153.582176.

[3] M. Akhin, M. Belyaev, Kotlin language specification, chapter 14: Type inference, https://kotlinlang.org/spec/type-inference.html, 2020. Accessed: 2025-09-01.

[4] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, E. Tanter, Do static type systems improve the maintainability of software systems? an empirical study, in: 2012 20th IEEE International Conference on Program Comprehension (ICPC), 2012, pp. 153–162. doi:10.1109/ICPC.2012.6240483.

[5] A. Wright, M. Felleisen, A syntactic approach to type soundness, Inf. Comput. 115 (1994) 38–94. URL: https://doi.org/10.1006/inco.1994.1093. doi:10.1006/inco.1994.1093.

[6] A. Bloss, P. Hudak, J. Young, An optimising compiler for a modern functional language, Comput. J. 32 (1989) 152–161. URL: https://doi.org/10.1093/comjnl/32.2.152. doi:10.1093/comjnl/32.2.152.

[7] G. Castagna, M. Laurent, K. Nguyễn, Polymorphic type inference for dynamic languages, Proc. ACM Program. Lang. 8 (2024). URL: https://doi.org/10.1145/3632882. doi:10.1145/3632882.

[8] A. Aiken, B. Murphy, Static type inference in a dynamically typed language, in: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '91, Association for Computing Machinery, New York, NY, USA, 1991, p. 279–290. URL: https://doi.org/10.1145/99583.99621. doi:10.1145/99583.99621.

[9] O. Agesen, J. Palsberg, M. I. Schwartzbach, Type inference of self: Analysis of objects with dynamic and multiple inheritance, Software: Practice and Experience 25 (1995) 975–995. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250903. doi:https://doi.org/10.1002/spe.4380250903. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250903.

[10] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, Y. Choi, Type inference for static compilation of javascript, in: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 410–429. URL: https://doi.org/10.1145/2983990.2984017. doi:10.1145/2983990.2984017.

[11] N. Suzuki, Inferring types in smalltalk, in: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '81, Association for Computing Machinery, New York, NY, USA, 1981, p. 187–199. URL: https://doi.org/10.1145/567532.567553. doi:10.1145/567532.567553.

[12] A. H. Borning, D. H. H. Ingalls, A type declaration and inference system for smalltalk, in: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '82, Association for Computing Machinery, New York, NY, USA, 1982, p. 133–141. URL: https://doi.org/10.1145/582153.582168. doi:10.1145/582153.582168.

[13] R. E. Johnson, Type-checking smalltalk, in: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '86, Association for Computing Machinery, New York, NY, USA, 1986, p. 315–321. URL: https://doi.org/10.1145/28697.28728. doi:10.1145/28697.28728.

[14] J. O. Graver, Type checking and type inference for object-oriented programming languages, Ph.D. thesis, USA, 1989. AAI9010868.

[15] J. Palsberg, M. I. Schwartzbach, Object-oriented type inference, SIGPLAN Not. 26 (1991) 146–161. URL: https://doi.org/10.1145/118014.117965. doi:10.1145/118014.117965.

[16] G. Bracha, D. Griswold, Strongtalk: typechecking smalltalk in a production environment, SIGPLAN Not. 28 (1993) 215–230. URL: https://doi.org/10.1145/167962.165893. doi:10.1145/167962.165893.

[17] O. Agesen, Concrete type inference: delivering object-oriented applications, Ph.D. thesis, Stanford, CA, USA, 1996. UMI Order No. GAX96-20452.

[18] G. Bracha, Pluggable type systems, 2004.

[19] E. Allende, O. Callaú, J. Fabry, E. Tanter, M. Denker, Gradual typing for smalltalk, Sci. Comput. Program. 96 (2014) 52–69. URL: https://doi.org/10.1016/j.scico.2013.06.006. doi:10.1016/j.scico.2013.06.006.

[20] F. Pluquet, A. Marot, R. Wuyts, Fast type reconstruction for dynamically typed programming languages, SIGPLAN Not. 44 (2009) 69–78. URL: https://doi.org/10.1145/1837513.1640145. doi:10.1145/1837513.1640145.

[21] N. Milojković, Augmenting Type Inference with Lightweight Heuristics, PhD thesis, University of Bern, 2017. URL: http://scg.unibe.ch/archive/phd/lazarevic-phd.pdf.

[22] J. Bliznicenko, R. Pergl, Combining type inference techniques for semi-automatic UML generation from pharo code, J. Comput. Lang. 82 (2025) 101300. URL: https://doi.org/10.1016/j.cola.2024.101300. doi:10.1016/J.COLA.2024.101300.

[23] S. A. Spoon, O. Shivers, Demand-driven type inference with subgoal pruning, Ph.D. thesis, USA, 2005. AAI3198609.

[24] O. Agesen, 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, Springer-Verlag, Berlin, Heidelberg, 1995, p. 2–26.

[25] N. Passerini, P. Tesone, S. Ducasse, An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types, in: International workshop of Smalltalk technologies, Cambridge, United Kingdom, 2014. URL: https://inria.hal.science/hal-03779638.

[26] Typeshed maintainers and contributors, Typeshed: Stubs for Python standard library and third-party libraries, https://github.com/python/typeshed, 2025. Accessed: 2025-09-01.

Heuristic based on	Heuristic category	Heuristic	Examples	Expected Type	Methods in Pharo13	Of them Abstract
Selector	Selector matches regex	`^includes[A-Z].*`	includesPoint:	Boolean	228	7
Selector	Selector matches regex	`^has[A-Z].*`	hasPlugin:	Boolean	1026	14
Selector	Selector matches regex	`^is[A-Z].*`	isAbstract	Boolean	4901	90
Selector	Selector equals	`"="`	= aNumber	Boolean	387	4
Selector	Selector equals	`"<"`	$<$	Boolean	31	2
Selector	Selector equals	`">"`	$>$	Boolean	20	0
Selector	Selector equals	`"<="`	$\leq$	Boolean	41	0
Selector	Selector equals	`">="`	$\geq$	Boolean	16	0
Selector	Selector equals	hash	hash	Integer	351	4
Selector	Selector equals	size	size	Number	117	2
Selector	Selector equals	priority	priority	Number	92	5
Selector	Selector ends with String	String	definitionString	String	736	14
Selector	Selector ends with Text	Text	errorText	String	432	4
Instance side body	Each return statement returns	an object of the same class	`^ OrderedCollection`	corresponding Meta Class	982	-
Instance side body	Each return statement returns	an object instantiated of the same class	`^ OrderedCollection new`	corresponding Class	394	-
Instance side body	Each return statement returns	nil	`^ nil`	UndefinedObject	278	-
Instance side body	Each return statement returns	ASTLiteralNode	`^ 1, ^ 'Hello', ^ true`	corresponding Class	8745	-
Instance side body	No return statement	No return statement	initialize size := 0	class to which method belongs	57364	0
Class side body	Each return statement returns	self new	default self new	class to which method belongs	34	-

Table 2: Heuristics for inferring method return types in Pharo13.