

Original software publication

Visualizing the semantics of gradual languages with Gredex[☆]Matías Toro^{ID}*, Éric Tanter^{ID}

PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Chile

ARTICLE INFO

Keywords:

Gradual typing
Static semantics
Operational semantics
Visualization
Derivation trees

ABSTRACT

We present Gredex, an interactive tool to visually explore the static and dynamic semantics of gradually-typed programming languages. Gredex supports a core gradual language with numbers, booleans, functions, pairs, sums, and fixpoints. Gredex displays typing derivations and step-by-step reduction of programs, showing how runtime type information tracking evolves, thereby providing a visual explanation of both successful and failing behaviors. Gredex has been used in multiple research projects over the last decade to design and experiment with gradual languages with features such as security typing, flexible union types, parametric polymorphism, sealing, and merging. Gredex offers a faithful, extensible implementation of gradual language semantics, supporting both experimentation and education.

Code metadata

Current code version
Permanent link to code/repository used for this code version
Permanent link to Reproducible Capsule
Legal Code License
Code versioning system used
Software code languages, tools, and services used
Compilation requirements, operating environments & dependencies

v1.1
<https://github.com/ElsevierSoftwareX/SOFTX-D-25-00545>
<https://doi.org/10.5281/zenodo.16371964>
MIT License
git
Scala, React, TypeScript, Apache Pekko HTTP, Material-UI, MathJax, npm, sbt.
The backend requires Scala 3.3.5 and sbt 1.10.11 for compilation, while the frontend requires Node.js (v20 or higher) and npm 10.8.1 for dependency management. The application is compatible with any modern web browsers (e.g., Chrome, Firefox). For ease of deployment, a Dockerfile is also provided to streamline the setup process.
<https://github.com/pleiad/gredex/blob/main/Readme.md>
mtoro@dcc.uchile.cl

If available Link to developer documentation/manual
Support email for questions

1. Motivation and significance

Gradual typing enables programmers to mix statically typed and dynamically typed code within the same program [1]. This is typically achieved by introducing an unknown type, usually noted $?$ or $*$, and adjusting typechecking to be optimistic when dealing with the unknown type. For instance, a function of the imprecise type $? \rightarrow \text{int}$ can be passed any argument, regardless of its statically-determined type. In order to preserve type soundness, a gradual language then needs to account for optimism in the typechecking phase with some runtime checking mechanism, such as casts [1], coercions [2], or evidence [3].

For instance, the following program is optimistically accepted by a gradual typechecker because the argument x of function f has the unknown type:

```
1 def f(x: float):
2   print("here")
3   return x * 2.5
4
5 a: ? = "weird" # well-typed:
6                # str consistent with ?
7 f(a)           # well-typed: ? consistent with
8                # float fails at runtime (str != float)
```

But at runtime, a check occurs at the function application $f(a)$ to validate that the value passed as argument is indeed a number.

[☆] This work has been partially funded by ANID FONDECYT Iniciación 11250054.

* Corresponding author.

E-mail addresses: mtoro@dcc.uchile.cl (M. Toro), etanter@dcc.uchile.cl (É. Tanter).

This check happens – and fails – before executing the body of `f`, so the printing statement on line 2 does not occur.¹

The flexibility afforded by gradual typing is valuable for software development, but designing gradual languages, with their static and dynamic semantics, can be challenging, particularly when dealing with features such as higher-order functions, polymorphism or subtyping.

To avoid relying solely on ad hoc language design, some formal methodologies for the systematic gradualization of statically-typed languages have been developed, most notably the Abstracting Gradual Typing (AGT) [3]. However, even with such a formal framework at hand, concretely implementing and experimenting with novel gradual languages is complex and error-prone. Researchers and educators require tools to prototype, validate, and communicate their designs interactively. Gredex fulfills this need by providing a unified platform for exploring the static and dynamic semantics of gradual languages. Loosely inspired by the PLT Redex tool for language design built in Racket for teaching and research about the dynamics of programming languages [4], Gredex supports a rich core language and offers interactive visualization of typing derivations and reduction steps.

Gredex was originally created to study evidence-based gradual languages that follow the AGT methodology, although it has since been used for different approaches and can be further extended. As such, Gredex can display both the initial elaboration of a source program (with initial evidence, cast insertion, etc.), and the evolution of runtime type tracking information (such as evidence, type tags, etc.) as a program executes. Gredex is particularly useful for debugging subtle issues, as it exposes the internal mechanisms of gradual typing, such as typing derivation trees, helping to understand dynamic errors. This transparency makes it an invaluable resource for both research and education.

To this date, Gredex has been used and directly contributed to the development of several gradually typed languages and their associated research. The first prototype was first developed to accompany the study of gradual unions [5], which defines a flexible semantics for union types, and the study of gradual security typing with references, with a language named GSLRef. [6]. It was subsequently used to study gradual parametricity, that is, the interaction between graduality [7,8] and parametricity [9] that manifests when designing a gradual counterpart to System F, under various forms, namely GSF [10,11], and Funk [12], a language with plausible sealing. During the same time, it was used to study different approaches to gradual typing with mutable references in λ_{REF} [13]. Recently, Gredex was used for the design and implementation of λM^* [14], a gradual language that supports merge types [15,16] and employs bidirectional type checking and typed-directed semantics (TDOS) [17,18]. Interestingly, λM^* does not follow the AGT methodology: its definition and implementation in Gredex are not based on evidence. Instead, the runtime semantics rely on a TDOS-style casting relation. This experience shows that many components of Gredex are reusable with different dynamic checking mechanisms, beyond the AGT approach. The Gredex version reported in this article was distilled from these different implementation efforts.

All these Gredex-based prototypes have consistently received positive evaluations and artifact evaluation badges (when applicable), demonstrating the practical impact of Gredex. By enabling researchers to experiment with and validate complex semantics, Gredex has facilitated advances in the state of the art of gradual typing. Its use in

published research attests to its importance and relevance in the field. In addition to its research applications, Gredex is a valuable educational tool. By making concepts such as evidence-based typing derivations and reduction steps concrete and interactive, it helps students and practitioners understand the intricacies of gradual typing. Gredex is used at our university for introducing students to the semantics of gradual typing. This dual role as a research and teaching tool highlights its broad significance and potential for further impact.

2. Software description

Gredex (<https://github.com/pleiad/gredex>) is a web-based application for the interactive exploration of formal programming language semantics, with a particular focus on typing derivations and program reduction steps for gradually-typed languages. The system provides a unified graphical interface to inspect derivation trees, observe runtime behavior via internal artifacts – such as evidence, coercions, and reduction steps – and experiment with novel type systems via user-defined extensions. Gredex separates the language logic from the presentation layer, and its build and deployment system packages both tiers into a single application. The software does not require any database or external service, and is entirely self-contained.

2.1. Software architecture

Gredex follows a client-server design composed of three primary components:

- **Frontend:** A single-page application written in React 19.1 [19], TypeScript 5.8 [20] and Material-UI 7.1 [21], responsible for rendering the user interface and communicating with the backend.
- **Web server:** A server implemented in Scala 3.3.5 [22] using the Pekko HTTP framework 1.0.1 [23] that exposes REST endpoints for typechecking and reduction, and serves static assets such as the compiled Javascript files.
- **Gradual language library (glang):** A modular Scala library that defines core functionality for parsing, typing and elaboration, and evaluation. This core gradual language can be extended to support new language mechanisms, semantics, typing disciplines, and runtime tracking mechanisms.

Fig. 1 depicts the architecture of Gredex. Initially, when the user inputs a source program in the code editor area, the frontend sends the source code to the backend, which parses it, and then performs typechecking by elaboration, and returns a typing derivation in LaTeX for the frontend to display in the typechecking area using MathJax 2.3 [24]. From then on, the user can request either full or step-by-step evaluation. To support step-by-step evaluation, the runtime system is implemented as a CEK machine [25] instead of a recursive interpreter. Once the execution of the requested number of steps is performed, the backend returns a list of evaluation steps as typing derivations for the frontend to display in the reduction area.

Note that the CEK machine can be used with different mechanisms for runtime tracking of type information. While Gredex has been mostly used with evidence-based gradual languages (based on the AGT methodology), we have also developed an alternative tracking mechanism based on the type-directed operational semantics (TDOS) of Huang et al. [26]. Instead of decorating values with their typing evidence, this approach reduces an ascribed term in a runtime context that exposes the ascribed type, so that certain mechanisms (such as accesses to a merged value) can use it appropriately. Other components for runtime type tracking, such as casts or coercions, could be developed, but this is left for future work.

In terms of reuse, the frontend and web server are oblivious to the actual language considered (except for the Syntax Helper popup which should appropriately describe the syntax of the considered language,

¹ Note that this requirement to dynamically check optimistic assumptions, present in the seminal formulation of gradual typing [1] and subsequent developments, is also sometimes called “sound gradual typing”, in order to contrast it with type systems that support optimistic checking based on a wildcard type but do not perform any runtime check to validate static typing assumptions; popular examples of such optional type systems are TypeScript and all mainstream typecheckers for Python 3 type hints (Mypy, Pyright, Pyre, etc.). In these languages, our example program would yield an error *after* printing (Python) or silently return NaN (TypeScript).

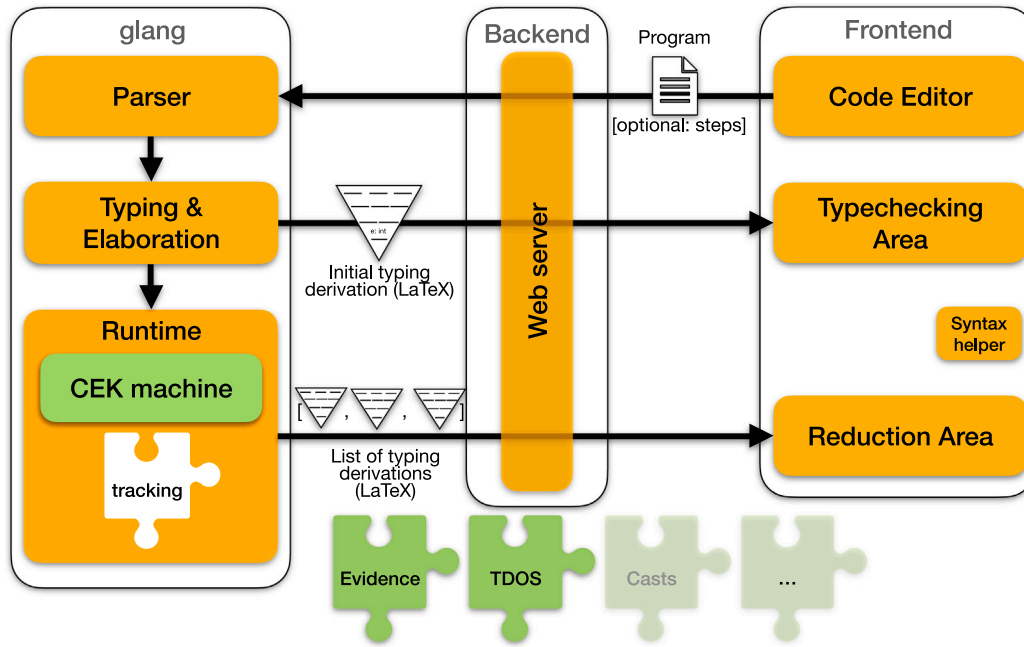


Fig. 1. Architecture of Gredex.

along with any additional information). The backend, on the other hand, is most dependent on the considered language mechanisms, typing discipline and runtime tracking mechanism, although some level of reuse is possible depending on the considered extension or variation. For instance, experimenting with the type-directed reduction semantics (TDOS) is an internal change that does not affect components such as parsing, typing, or most of the reduction rules. Adding a new user-facing language feature to the `glang` library, such as algebraic data types, involves: (a) extending the parser with the new AST nodes and parsing rules; (b) adding typing and elaboration rules, and consistency checks to the typing and elaboration module; and (c) implementing the new reductions in the CEK machine.

2.2. Software functionalities

Gredex offers the following key functionalities:

Extensible core gradual language. The minimalist and modular object-oriented design of `glang` allows users to extend or adapt it to support other language features and typing disciplines.

Typing derivation visualization. For a given program, users can inspect the complete typing derivation as a tree, with each node showing the rule applied to the corresponding term and each of its subterms. Each typing derivation can be expanded or collapsed, either locally—by clicking on a horizontal bar of the derivation—or globally—by clicking on the icon in the top right corner that appears when hovering over the derivation tree.

Step-wise reduction. The reduction process is visualized step by step. Users may reduce any finite number of steps. The active reducible expression (aka. *redex*) is highlighted in each configuration. Every intermediate term in the reduction trace can be inspected to reveal its typing derivation. If a runtime type error occurs, the system highlights where it occurred and which elements caused the error.

Configurable parameters. The frontend can communicate additional configuration options to the backend – such as flags for controlling evidence or cast visibility, number of reduction steps, and

visualization strategies for ascriptions – by extending the payload object sent to the backend. On the backend side, the `IOption` class should be modified to receive these parameters, which are then implicitly passed to the different `glang` modules.

Example gallery. Predefined examples are bundled with each language instantiation. These illustrate behaviors of interest, involving both successful and failing typing and reduction scenarios. Each example is editable and can be re-executed. The language implementor can extend and organize examples as needed.

3. Illustrative examples

This section presents an illustrative example that demonstrates the main features of Gredex: typing derivation visualization, step-by-step reduction, and inspection of runtime evidence (the runtime type tracking device of an AGT-derived language). The chosen example is deliberately simple, but representative of gradual typing behavior, highlighting how imprecise type information interacts with runtime type safety.

Program. Consider the following gradually typed program:

```
let x = true :: * in x + 1
```

This expression binds the boolean value `true` to variable `x`, explicitly ascribed with the unknown type `*`. The body attempts to perform arithmetic addition between `x` and the integer `1`. Due to the unknown type of `x`, the program typechecks thanks to consistency [1], and reduction proceeds with a runtime check that eventually fails, revealing the underlying type mismatch.

Typing derivation. Fig. 2 shows the typing derivation generated by Gredex for the above program. The derivation is visualized as a tree: each node corresponds to the application of a typing rule, and the use of consistency is accompanied by the corresponding evidence. In this example, consistency between `*` and `nat` is used to justify applying `+` to `x` and `1`. The derivation tree is interactive. A user can expand or collapse subtrees and inspect which rule, type, and evidence is used at each step.

Visual Type Derivations

Preloaded examples

-- Select an example from this list to load it in the scratchpad --

Program

```

1 let x = true :: * in
2 x + 1

```

TYPECHECK

☒ Hide evidences
 CLEAR RELOAD VIEW SYNTAX

Parsed program

$$\text{let } x = ((\text{true} :: \text{bool}) :: *) \text{ in } ((x :: \text{int}) + (1 :: \text{int}))$$

Type derivation and reduction

(TLIT)

$$\frac{}{\text{true} : \text{bool} \quad \text{bool} \sim \text{bool}}$$

(TYP-VAR)

$$\frac{}{x : * \quad * \sim \text{int}}$$

(TLIT)

$$\frac{}{1 : \text{int} \quad \text{int} \sim \text{int}}$$

(TASC)

$$\frac{}{(\text{true} :: \text{bool}) : \text{bool} \quad \text{bool} \sim *}$$

(TASC)

$$\frac{}{(x :: \text{int}) : \text{int}}$$

(TASC)

$$\frac{}{(1 :: \text{int}) : \text{int}}$$

(TBOP)

$$\frac{}{((x :: \text{int}) + (1 :: \text{int})) : \text{int}}$$

(TLET)

$$\frac{}{\text{let } x = ((\text{true} :: \text{bool}) :: *) \text{ in } ((x :: \text{int}) + (1 :: \text{int})) : \text{int}}$$

REDUCE!

Step size

20

Fig. 2. Typing derivation for `let x = true :: * in x + 1`. Evidences are hidden for simplicity.

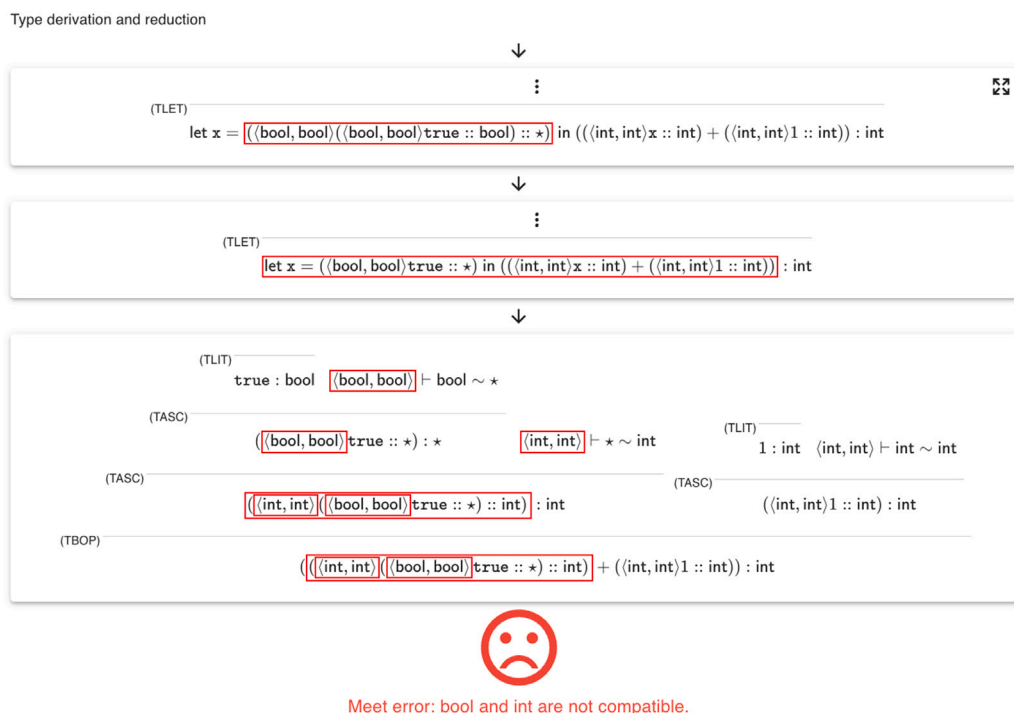


Fig. 3. Reduction trace for `let x = true :: * in x + 1`, ending in a runtime error. Evidences are shown for clarity.

Reduction trace and runtime error. Clicking the “Reduce” button executes the program step by step. Fig. 3 shows each redex boxed in red and displays the evolving evidence. In this case, an evidence composition from `bool` to `nat` fails at runtime. When an error occurs, Gredex highlights the failing expression, the responsible evidence, and the derivation step where the inconsistency was justified. This enables precise inspection of runtime failures in gradual typing.

4. Impact

In the last twenty years, since the seminal paper on gradual typing was published [1], gradual typing has been a very prolific area of research, both theoretically and practically. A major research question has been how to design gradual counterparts to existing static typing disciplines. In this context, Gredex has directly supported the design,

validation, and dissemination of several gradually typed languages, published in major venues in the Programming Languages community. Its impact lies in its ability to improve the development and exploratory work of existing research, and to affect the practice of validating and understanding the semantics of gradual languages.

First, Gredex accelerates the development cycle for gradually typed languages. The design of gradual languages typically involves rigorous metatheoretical development, including the formulation and proof of properties such as type safety [27], gradual guarantees [7], and hyper-properties like noninterference [28] or parametricity [9]. However, proving such theorems is costly and often requires multiple iterations. Gredex enables researchers to experiment with new constructs and typing rules before fully committing to a formalization. Being able to inspect concrete derivations and reduction traces in specific examples, which would be difficult or tiresome to write down manually and debug, accelerates understanding and progress. As such, Gredex is also valuable during the formalization process, as it facilitates the rapid refutation of invalid conjectures by enabling concrete executions and visual inspection of reduction traces and evidence interactions. This bidirectional interaction between executable prototypes and formal models allows researchers to uncover subtle design flaws, clarify semantic choices, and refine proof strategies. This tight feedback loop has been instrumental in the creation of languages such as GTFL[®] [5], GSLRef. [6], GSF [10,11], Funk [12], and λ_{REF} [13]. All these languages used Gredex as a base to test the design, validate key metatheoretical claims, and illustrate stereotypical behavior. Artifacts based on Gredex received positive feedback, and the associated papers were awarded artifact evaluation badges (when applicable). In addition, Gredex was also used in variations beyond AGT-derived gradual languages, such as ObSec [29] and λM^* [14], showcasing the reach of the tool beyond its initially-envisioned setting.

Second, Gredex facilitates the way users explore and debug gradual languages. Rather than inspecting small-step semantics on paper, users can interactively explore how typing derivations are built and evolve at runtime. This leads to a better understanding and refinement of core semantics, and is also serving as an effective pedagogical tool when teaching gradual typing concepts to graduate students willing to get involved in programming language research.

5. Conclusions

Gredex is an interactive tool for designing and understanding the static and dynamic semantics of gradual languages. By exposing typing and reduction derivations, including the combination of runtime type tracking information, and failures, the tool supports the interactive development and refinement of sound and expressive gradual languages. Its use in published research in major programming language venues attests to its impact and practical relevance, in addition to its – so far, anecdotal – use in educational settings.

CRedit authorship contribution statement

Matías Toro: Writing – review & editing, Writing – original draft, Validation, Software, Project administration, Methodology, Investigation, Conceptualization. **Éric Tanter:** Writing – review & editing, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

- [1] Siek JG, Taha W. Gradual typing for functional languages. Scheme and Functional Programming Workshop 2006.
- [2] Herman D, Tomb A, Flanagan C. Space-efficient gradual typing. High-Order Symbolic Computation 2010;23(2):167–89.
- [3] Garcia R, Clark AM, Tanter É. Abstracting gradual typing. In: Bodik R, Majumdar R, editors. Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM; 2016, p. 429–42. <http://dx.doi.org/10.1145/2837614.2837670>.
- [4] Felleisen M, Findler RB, Flatt M. Semantics Engineering with PLT Redex. MIT Press; 2009, URL <http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885>.
- [5] Toro M, Tanter É. A gradual interpretation of union types. In: Proceedings of the 24th Static Analysis Symposium. Springer; 2017, p. 382–404.
- [6] Toro M, Garcia R, Tanter É. Type-driven gradual security with references. ACM Transactions on Programming Languages and Systems 2018;40(4):16:1–55.
- [7] Siek JG, Vitousek MM, Cimini M, Boyland JT. Refined criteria for gradual typing. In: 1st Summit on Advances in Programming Languages. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik; 2015, <http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.274>.
- [8] New MS, Jamner D, Ahmed A. Graduality and parametricity: together again for the first time. Proc of the ACM on Programming Languages 2020;4(POPL):46:1–32. <http://dx.doi.org/10.1145/3371114>.
- [9] Reynolds JC. Types, abstraction and parametric polymorphism. In: Mason REA, editor. Information Processing 83, Proceedings of the IFIP 9th World Computer Congress. North-Holland/IFIP; 1983, p. 513–23.
- [10] Toro M, Labrada E, Tanter É. Gradual parametricity, revisited. In: ACM SIGPLAN Symposium on Principles of Programming Languages. ACM; 2019, p. 17:1–30.
- [11] Labrada E, Toro M, Tanter É. Gradual system F. Journal of the ACM 2022;69(5):38:1–78.
- [12] Labrada E, Toro M, Tanter É, Devriese D. Plausible sealing for gradual parametricity. Proc of the ACM on Programming Languages 2022;6(OOPSLA1):70:1–28.
- [13] Toro M, Tanter É. Abstracting gradual references. Science of Computer Programming 2020;197:102496.
- [14] Ye W, Oliveira BCdS, Toro M. Merging gradual typing. Proc of the ACM on Programming Languages 2024;8(OOPSLA1):294:648–76.
- [15] Reynolds JC. Design of the programming language Forsythe. In: ALGOL-like languages. 1997, p. 173–233.
- [16] Dunfield J. Elaborating intersection and union types. Journal of Functional Programming (JFP) 2014;24(2–3):133–65. <http://dx.doi.org/10.1145/2398856.2364534>.
- [17] Oliveira BCdS, Shi Z, Alpuim J. Disjoint intersection types. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. ICFP 2016, New York, NY, USA: Association for Computing Machinery; 2016, p. 364–77. <http://dx.doi.org/10.1145/2951913.2951945>.
- [18] Huang X, Oliveira BCdS. A type-directed operational semantics for a calculus with a merge operator. In: Hirschfeld R, Pape T, editors. 34th European Conference on Object-Oriented Programming. Leibniz international proceedings in informatics (LIPIcs), vol. 166, Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik; 2020, p. 26:1–32. <http://dx.doi.org/10.4230/LIPIcs.ECOOP.2020.26>, URL <https://drops.dagstuhl.de/opus/volltexte/2020/13183>.
- [19] Facebook Open Source. React: A JavaScript library for building user interfaces. 2013, <https://react.dev>. Version 18.2, [Accessed 23 July 2025].
- [20] Microsoft. TypeScript language specification. 2024, <https://www.typescriptlang.org>. Version 5.5, [Accessed 23 July 2025].
- [21] MUICore Team. MUI: React components for faster web development. 2025, <https://mui.com/>. Version 7.1.1, [Accessed 23 July 2025].
- [22] Odersky M, Spoon L, Venners B. Programming in Scala. 5th ed. Artima Press; 2021.
- [23] Apache Pekko Project. Apache pekko HTTP. 2023, <https://pekko.apache.org>. Version 1.0.1, [Accessed July 2025].
- [24] MathJax Consortium. MathJax 3: A JavaScript display engine for mathematics. 2024, <https://www.mathjax.org>. Version 3.2, [Accessed 23 July 2025].
- [25] Felleisen M, Friedman DP. Control operators, the SECD-machine, and the λ -calculus. In: Wirsing M, editor. Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III. North-Holland; 1987, p. 193–222.
- [26] Huang X, Zhao J, Oliveira BCdS. Taming the merge operator. Journal of Functional Programming (JFP) 2021;31:e28. <http://dx.doi.org/10.1017/S0956796821000186>.

- [27] Wright AK, Felleisen M. A syntactic approach to type soundness. *Inform and Comput* 1994;115(1):38–94. <http://dx.doi.org/10.1006/INCO.1994.1093>.
- [28] Goguen JA, Meseguer J. Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy. IEEE Computer Society; 1982, p. 11–20. <http://dx.doi.org/10.1109/SP.1982.10014>.
- [29] Cruz R, Rezk T, Serpette B, Tanter É. Type abstraction for relaxed noninterference. In: 31st European Conference on Object-Oriented Programming. Schloss Dagstuhl; 2017, p. 7:1–27.