A Rational Defense of Reasonable Reflection

Nada Amin
Harvard University, Cambridge, MA, USA

Abstract

Computational reflection is the ability of a system to inspect and modify itself. It has resisted static reasoning, especially in reflective languages, where semantics can change during program execution. We argue that reflection should be revisited in light of progress in verification and large language models. By requiring reflective modifications to carry proofs, it becomes possible to prove system properties even for reflective systems. This "reasonable reflection" could open up new ways of organizing self-improving systems.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases reflection, verification, AI

Digital Object Identifier 10.4230/LIPIcs.LICS.2026.1

Category Invited Paper

Supplementary Material *Software*: `https://github.com/namin/reasonable-reflection`

Funding Amazon Research Award and Google Cloud for compute credits.

Acknowledgements Thanks to Kenichi Asai, Matko Bošnjak, Will Byrd, Steve Chong, Kartik Chandra, Olivier Danvy, Michael Fogus, Walter Fontana, Fernanda Graciolli, Simon Henniger, Tzu-Han Hsu, Ohad Kammar, Anastasiya Kravchuk-Kirilyuk, Paul Krogmeier, Gabriel Poesia, Christopher Qiu, Matthew Retchin, Jack Rusher, Raffi Sanna, Gerry Sussman, Eric Tanter, Mitch Wand, Cameron Wong, Haoyi Zeng, and Stefan Zetzsche for discussions.

1 Reasonable Reflection

A computationally reflective system can inspect and modify itself. In the lineage of the reflective towers (3-LISP [20, 19, 21], Brown [9, 24], Blond [7, 6], Black [4, 3], Pink & Purple [2]), the meta level was data: evaluator components were ordinary bindings, modifiable from within. The capability was elegant and the systems resisted reasoning. Wand [23] showed why in one (syntactic) setting: first-class meta-level access collapses equational reasoning to alpha-equivalence. The same period saw Lenat's Eurisko [13], in which heuristics produced new heuristics, with the same problem on the soundness side. Reflection in both lineages was rich and ungovernable, and the field retreated to safer ground: staging, macros, constrained metaprogramming. Java reflection and JavaScript `eval` earned reflection a bad reputation in production languages.

Reflection became unreasonable not because it is inherently so, but because checking was out of reach. Now, two things have changed.

First, **the checker discipline is becoming practical at scale**. LCF [15, 16] carved out the pattern: an untrusted frontend produces evidence, a small trusted checker validates it. The mechanism varies: an abstract `thm` type (in LCF and Isabelle [18]), proof-term checking (Lean [17], Rocq [22]), semi-automation based on SMT solvers (Dafny [12], Verus [11]). Tactic-style metaprogramming extends the discipline to programs themselves: gated by the kernel, meta-level extensions become proof-carrying metaprogramming. Ideas from verified compilation [10] pave the way to verified reflection.

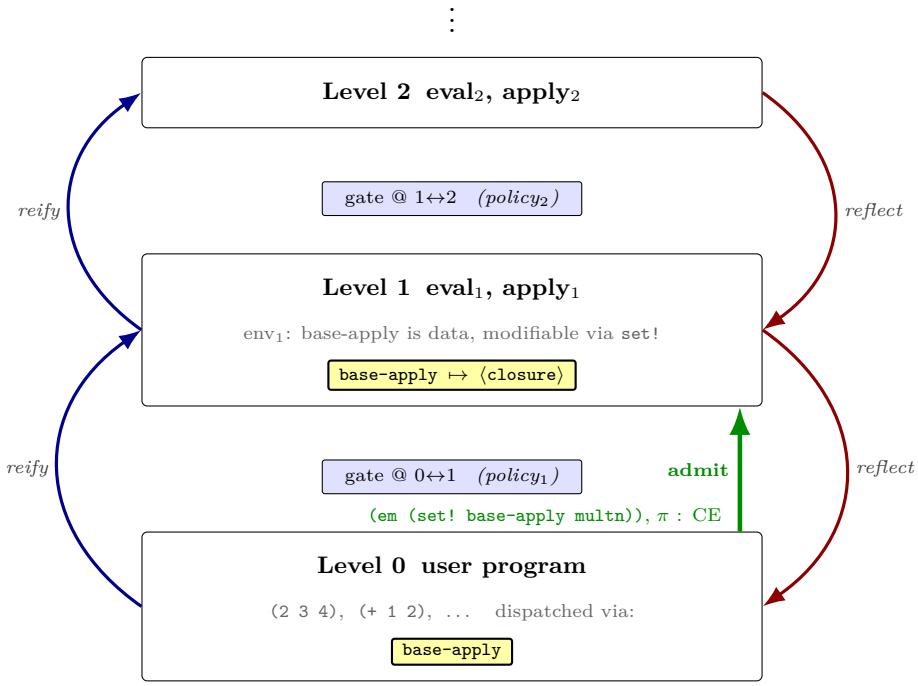

Figure 1 The gated reflective tower. Each level boundary admits two reciprocal motions: *reification* lifts running computation at level N into data at level $N{+}1$; *reflection* makes data at level $N{+}1$ the running computation at level N. Both pass through the gate at that boundary, which consults the policy stored at the level above – the policy is itself reflectively modifiable, but only through the gate one level up. The yellow cell holding `base-apply` appears twice, *causally connected* [14]: at level 1 it is data in the env (mutable by `set!` through a reflection step); at level 0 it is the dispatcher in effect for the user program's calls. The green path traces a concrete admission: `(em (set! base-apply multn))` at level 0 lifts the `set!` to level 1 (a reification of the proposal); accompanied by a conservative-extension (CE) proof π, it is admitted at gate@0↔1; level 1's `base-apply` is updated, and subsequent `(2 3 4)` at level 0 dispatches as multiplication.

Second, **machine-generated proposals are becoming practical**. Large language models can propose modifications along with proofs that the modifications will support the invariants and desiderata of the system, while only the mechanical kernel remains trusted.

This proof-carrying discipline enables **reasonable reflection**: arbitrary modification of the meta level, gated by an independent kernel.

2 Rational Defense

The pattern has three components:

- a **substrate**, the system to be modified, e.g., an interpreter, a program transformer, a derivation calculus, a heuristic agenda;
- a **gate**, ideally small, trusted, and mechanical, e.g., a sandbox, a type checker, a verifier, a proof kernel;
- a **proposer**, who proposes a modification together with the evidence the gate requires. The proposer can be a human, a language model, or a search procedure; the evidence takes whatever form the gate expects.

Instances vary along several properties of the gate and of the system:

- **Substrate kind**: what the gate deals with, e.g., an interpreter, a program transformer, a derivation calculus, a heuristic agenda.
- **Modification kind**: what change the gate approves or rejects, e.g., a heap-cell mutation, an axiom-schema extension, a rewrite, a defeasibility rule.
- **Evidence kind**: the artifact the gate examines, e.g., a sandbox trace, a type derivation, a discharged specification, a kernel-checked proof.
- **Policy**: the local criterion each admitted modification must satisfy, e.g., avoids unsafe constructs, type-checks, discharges the specification, is a conservative extension.
- **Guarantee**: the global property that emerges across composed admissions, e.g., overall non-crashing, preserved β-equivalence, type-safety invariant.
- **Reflective depth**: whether the gate itself is modifiable by the reflective system, e.g., a single static gate, or a multi-level tower in which the gate's admission criteria are themselves modifiable through the gate one level up.

3 Artifacts

Current artifacts are available from: `https://github.com/namin/reasonable-reflection`.

We highlight an instance (Figure 1). The **substrate kind** is a reflective tower of interpreters inspired by Black and CakeML.

- From Black [4, 3]: the apply rule is an ordinary heap cell in each level's meta-environment, modifiable via `set!`; the reflective shift `(em ...)` makes the meta-level data, and the causal connection between meta and object is the heap; nested `(em (em ...))` reaches multiple levels deep.
- From CakeML [10]: value bisimulation by data refinement, with closures related pointwise through their captured environments, is used to reason about conservative extension.

The **modification kind** is `set!` on the `base-apply` heap cell. The **evidence kind** is a kernel-checked Lean proof. The **policy** is per-modification conservative extension. The **guarantee** is that the substrate is always a conservative extension of the base. The **reflective depth** is multi-level: the gate is itself reflectively modifiable through the gate one level up. For example, we admit the application of a number as multiplication by that number, following an introductory example [1] in Black: `(2 3 4)` evaluates to 24, while `(+ 1 2)` remains 3. Conversely, a wrapper that would double every numeric result is refused at admission: no conservative-extension certificate exists, and the kernel rejects the record before it enters the runtime state. Admissions compose at a single gate and apply at deeper levels of the tower: a modification proposed via `(em (em ...))` is gated at level 2, affecting how level 1 dispatches.

4 Open Questions

The premise of the keynote is that the LICS community is positioned to develop the theory of these reflective systems as the practice of building them becomes accessible. Open questions span the classical axes: equational theories that survive gated reflection, semantics of evolving systems, proof theory of iterated reflection [8, 5], and a negative theory of which substrate-gate pairings are inherently incompatible. Reflection is not inherently a boundary of what we can reason about.

References

1 Nada Amin. Reflective towers of interpreters, SIGPLAN PL Perspectives, August 2021. URL: https://blog.sigplan.org/2021/08/12/reflective-towers-of-interpreters.

2 Nada Amin and Tiark Rompf. Collapsing towers of interpreters. *Proc. ACM Program. Lang.*, 2(POPL), 2018. doi:10.1145/3158140.

3 Kenichi Asai. *The Reflective Language Black*. PhD thesis, Department of Information Science, Graduate School of Science, University of Tokyo, February 1997.

4 Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation: for a better understanding of reflective languages. *Lisp Symb. Comput.*, 9(2–3):203–241, May 1996. doi:10.1007/BF01806113.

5 Lev D Beklemishev. Reflection principles and provability algebras in formal arithmetic. *Russian Mathematical Surveys*, 60(2):197–268, 2005.

6 Olivier Danvy and Karoline Malmkjær. A blond primer. Draft, DIKU, University of Copenhagen, Copenhagen, Denmark, February 1988.

7 Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 327–341, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/62678.62725.

8 Solomon Feferman. Transfinite recursive progressions of axiomatic theories. *The Journal of Symbolic Logic*, 27(3):259–316, 1962. doi:10.2307/2964649.

9 Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 348–355, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800055.802051.

10 Ramana Kumar. Self-compilation and self-verification. Technical Report UCAM-CL-TR-879, University of Cambridge, Computer Laboratory, February 2016. doi:10.48456/tr-879.

11 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), 2023. doi:10.1145/3586037.

12 K Rustan M Leino. *Program proofs*. MIT Press, 2023.

13 Douglas B Lenat. EURISKO: a program that learns new heuristics and domain concepts: the nature of heuristics III: program design and results. *Artificial Intelligence*, 21(1-2):61–98, 1983.

14 Pattie Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/38765.38821.

15 Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford University, Stanford, CA, USA, May 1972.

16 Robin Milner. LCF: A way of doing proofs with a machine. In Jiří Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, pages 146–159, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg. doi:10.1007/3-540-09526-8_11.

17 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-79876-5_37.

18 Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

19 Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-272.pdf.

20 Brian Cantwell Smith. Reflection and semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800017.800513.

21 Brian Cantwell Smith. *Computational Reflections*. The MIT Press, May 2026.

22 The Rocq Development Team. *The Rocq Prover Reference Manual, version 9.2.0*, 2026. URL: https://rocq-prover.org/doc/V9.2.0/refman/index.html.

23 Mitchell Wand. The theory of fexprs is trivial. *Lisp Symb. Comput.*, 10(3):189–199, 1998. doi:10.1023/A:1007720632734.

24 Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 298–307, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/319838.319871.