

λ_{JS} à la Carte

Kirill Golubev
kirill.golubev@utu.fi
University of Turku
Turku, Finland

CCS Concepts: • Software and its engineering → Software verification.

Keywords: Language mechanization, modular proofs, JavaScript, reactivity.

1 Introduction

Half a page for overview and explaining motivation

In particular, we are interested in *reactivity* in JavaScript, which is highly popularized in various developer frameworks such as React [7], Solid [6], and others.

2 Background, à la Carte

Datatypes à la Carte [15] popularizes the idea of *extensible* inductive datatypes. One of the efforts to bring this idea to proof assistants was done by *Coq à la Carte* [8], which enables modularity in proofs. However, directly adopting the implementation [8] in practice is prohibitively difficult due to its dependency on an outdated version of Metarocq [14].

It is possible to represent an inductive type as a fixed point of a functor, but attempting such a decomposition naïvely leads to inconsistency¹. Therefore, it is not possible to use explicit fixed point operations for inductive type decomposition in theorem provers. As shown in the *Coq à la Carte* approach, this issue can be circumvented by *inlining* fixed point operations. Resulting functors will then allow open-recursion-style per-functor proofs, which will not rely on a particular structure of the decomposed type. Once the desired proofs are complete, the type can be assembled from the desired feature functors as a fixed point of their coproduct, and proofs can be assembled from per-functor proofs, by “closing” recursion over them.

These ideas are outlined in Figure 1. As can be seen in the code example, there is a considerable amount of boilerplate, and it can be automatically generated. Here we mention Coq-Elpi [16], which is a rule-based metalanguage for Rocq [17]; it gives a programmer an ability to generate tactics, inductive types and manipulate syntax with binders.

```
1 (* Exp.v *)
2 Inductive Exp :=
3   (* Exp type can be seen as inlined fixed point
4     of coproduct of feature functors exp_ite and exp_lam *)
5   | inj_ite: exp_ite Exp → Exp
6   | inj_lam: exp_lam Exp → Exp.
7 Fixpoint thrm: forall (e : Exp) → ...
8 intros. destruct e.
9 (* close recursion *)
10 - apply (thrm_ite Exp thrm ...).
11 - apply (thrm_lam Exp thrm ...).
12 Defined.
13 (* ITE.v *)
14 Section exp_ite.
15 Exp: Type.
16 Inductive exp_ite :=
17   | ite: Exp → Exp → Exp
18   | boolt_lit: bool → Exp.
19 (* assume overarching property *)
20 Variable thrm: forall (e : Exp) → ... .
21 (* Prove corresponding per-functor property *)
22 Definition thrm_ite: forall (e : exp_lam) → ... .
23 End exp_ite.
24 (* Lambda.v: omitted, similar to ITE.v *)
```

Figure 1. Minimal example of language extension with ITE constructions.

3 Targets for Mechanization

In our work, we investigate an extensible mechanization of the JavaScript semantics² by *specializing* the à la carte approach described above. This specialization will be focused on developing meta-tools that would target an impure imperative language without concurrency (in our case, JavaScript), rather than developing a generic framework suitable for a wide range of target languages.

Recently, a significant progress [4] was made on applying modular techniques to mechanize the semantics of major programming languages (cf. CompCert [12] and CakeML[11]), but none of the existing modular approaches targets the JavaScript semantics. We observe that extending existing JavaScript mechanizations (cf., e.g., [2, 9], which are based

¹Fix ($\Lambda A. A \rightarrow \text{False}$) allows to obtain proof of False. See appendix Figure 2

²We note that mechanizing the entirety of the JavaScript semantics and developing a generic modular framework are non-goals.

- Smith. 2014. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 87–100.
- [3] Cyril Cohen, Enzo Crance, and Assia Mahboubi. 2024. Trocq: proof transfer for free, with or without univalence. In *European Symposium on Programming*. Springer, 239–268.
- [4] Oghenevwoaga Ebesafe, Ian Zhao, Ende Jin, Arthur Bright, Charles Jian, and Yizhou Zhang. 2025. Certified compilers à la carte. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 372–395.
- [5] ECMA. 2015. *ECMA language specification*. ECMA International. <https://ecma-international.org/publications-and-standards/standards/ecma-262/>
- [6] Inc Facebook. 2015. *SolidJS: Reactive JavaScript library*. Open collective. <https://solidjs.com/>
- [7] Inc Facebook. 2019. *React: A JavaScript library for building user interfaces*. Facebook, Inc. <https://www.reactjs.org/>
- [8] Yannick Forster and Kathrin Stark. 2020. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 186–200.
- [9] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *ECOOP 2010—Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21–25, 2010. Proceedings 24*. Springer, 126–150.
- [10] Ende Jin, Nada Amin, and Yizhou Zhang. 2023. Extensible metatheory mechanization via family polymorphism. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1608–1632.
- [11] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices* 49, 1 (2014), 179–191.
- [12] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert—a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.
- [13] Daniel Ehrenberg Rob Eisenberg. 2023. *JavaScript Signals standard proposal*. <https://github.com/tc39/proposal-signals>
- [14] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The metacoq project. *Journal of automated reasoning* 64, 5 (2020), 947–999.
- [15] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.
- [16] Enrico Tassi. 2025. Elpi: rule-based meta-language for Rocq. In *CoqPL 2025—The Eleventh International Workshop on Coq for Programming Languages*.
- [17] The Coq Development Team. 2024. *The Coq Proof Assistant*. doi:10.5281/zenodo.14542673
- [18] Cas van der Rest, Casper Bach Poulsen, Arjen Rouvoet, Eelco Visser, and Peter Mosses. 2022. Intrinsically-typed definitional interpreters à la carte. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1903–1932.
- [19] Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 300–331.
- [20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

A Additional figures

```

1 (* Positivity checker won't allow direct definition
2    as showed in Data types a la carte:
3    data Fix f = In (f (Fix f))
4    *)
5 Axiom Fix : (Type → Type) → Type.
6 Axiom fix_def : forall {f : Type → Type}, Fix f = f (Fix f).
7
8 Theorem fix_implies_false : False.
9 Proof.
10   remember (Fix (fun A => A → False)) as A.
11   assert (not_a : A → False). {
12     intros. remember X as X'.
13     clear HeqX'.
14     rewrite HeqA in X.
15     rewrite fix_def in X.
16     rewrite ← HeqA in X.
17     exact (X X').
18   }
19   apply not_a.
20   rewrite HeqA.
21   rewrite fix_def.
22   rewrite ← HeqA.
23   exact not_a.
24 Qed.

```

Figure 2. Fix existence implies falsehood.