λ_{JS} à la Carte

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

Mikhail Barash
mikhail.barash@uib.no
University of Bergen
Bergen, Norway

Jaakko Järvi
jaakko.jarvi@utu.fi
University of Turku
Turku, Finland

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

Keywords: Formal methods, modular proofs, JavaScript.

1 Introduction

A programming language formalization gives a single source of truth about the language behavior. Unless mechanized, a formalization can be erroneous and logically unsound. Mechanizing a formalization can guarantee that the semantics of the language is sound, and that the formalization is free of bugs. Mechanizing the formalization of a complex language, however, is laborious, as is maintaining an existing mechanization as the language evolves. Mechanizations are usually monolithic, such that each new feature requires manually rewriting existing code. At its core, this is due to the manifestation of the expression problem for inductive types.

Several solutions [8, 10, 20, 21] have been proposed to improve proof reuse in mechanizations, and some [5] have found use in practice (cf. CompCert [12] and CakeML [11]). Our work targets JavaScript semantics with one of these approaches, specializing *Coq à la Carte* [8]. We aim to show that it is possible to reason about modern heavily used languages in a modular fashion, keeping mechanization open to extension with new features.

2 Background

Datatypes à la Carte [17] popularizes the idea of *extensible* inductive datatypes. One effort to bring this idea to proof assistants was *Coq à la Carte* [8], which enables modularity in Rocq [19] proofs. However, directly using tools developed there is prohibitively difficult due to their dependency on an outdated version of Metarocq [16].

The key insight to extensibility is that it is possible to represent an inductive type as a fixed point of a particular functor. Unfortunately, attempting such a decomposition naïvely leads to inconsistency[1]. 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 avoided by *inlining* fixed-point operations. Resulting functors together with tight retracts[2] mechanism 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 corresponding 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.

```
(* ITE.v *)
Section exp_ite.
(* assume existance of overarching type *)
 Variable Exp : Type.
(* define feature functor *)
 Inductive exp_ite :=
   | ite : Exp ⟶ Exp ⟶ Exp
   | boolt_lit : bool ⟶ Exp.
(* assume tight retract for exp_ite *)
 Context `{Hretr : retract exp_ite exp}.
(* assume that property holds
   for the rest of the language *)
 Variable thrm : forall (e : Exp) ⟶ P e.
(* assume P-related retract for exp_ite *)
 Variable P_retracted : forall e, P (inj e) ⟶ P_ite (inj e).
(* Prove corresponding per-functor property *)
 Definition thrm_ite : forall (e : Exp) ⟶ P_ite e.
End exp_ite.
(* Lambda.v: omitted, similar to ITE.v *)
(* Exp.v *)
Inductive Exp :=
(* Exp type can be seen as an inlined fixed point
   of coproduct of feature functors exp_ite and exp_lam *)
 | inj_ite : exp_ite Exp ⟶ Exp
 | inj_lam : exp_lam Exp ⟶ Exp.
(* prove retracts *)
Instance r_exp_exp_lam : retract (exp_lam exp) exp := { ... }
Instance r_exp_exp_ite : retract (exp_ite exp) exp := { ... }
Definition P_retr_ite : forall (e : exp_ite), P (inj e) ⟶ ...
Definition P_retr_lam : forall (e : exp_lam), P (inj e) ⟶ ...
(* desired langugae property *)
Fixpoint thrm : forall (e : Exp) ⟶ P e.
Proof. intros. relevant_inversion; subst.
(* close recursion ----v *)
  - apply (thrm_ite Exp thrm P_retr_ite ...).
  - apply (thrm_lam Exp thrm P_retr_lam ...).
Defined.
```

Figure 1. Minimal example of language `Exp` extended with `ITE` constructions.

[1] `Fix (Λ A. A -> False)` implies `False`. See Appendix A, Figure 2.
[2] For tight retract definition, see Appendix A, Figure 3.

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 [18], a rule-based metalanguage for Rocq that gives a programmer the ability to generate tactics, inductive types and manipulate Rocq terms.

3 Targets for Mechanization

None of the existing modular approaches targets JavaScript semantics. We observe that extending existing JavaScript mechanizations (e.g., [2, 9]), which are based on various editions of the exhaustive natural language specification ECMA-262 [6], to support new language features is impossible without manually rewriting a significant portion of the mechanizations' existing codebase.

JavaScript is one of the most-used programming languages, it evolves fast, and has an actively maintained standard specification. JavaScript is thus an interesting and practically relevant test case for evaluating approaches for modular reasoning. The need for proof reuse and extensibility is evident, if one desires to tackle the language's many dialects (e.g. JSX, TypeScript), reason about various frameworks (e.g. React [7], Solid [3], etc.), and verify feature specifications while they are still in development.

We plan to follow the λ_{JS} formalization [9], and then extend it with the relevant features one by one, while maintaining the following safety theorems:

Look at OLang[15] and Wasm[13] for more useful properties

```
Theorem progress : forall c e,  lc e →
        isValue e ∨
        isError e ∨
        (exists c'  e',  step c e c'  e').

Theorem preservation : forall c e c'  e',  lc e
                                            → step c e c' e'
                                            → lc e'.
```

We are particularly interested in mechanizing mutability, exceptions, asynchrony, and reactivity of JavaScript, and we seek to establish soundness of the TC39 *Signals* proposal [14] that unifies reactive primitives across various developer frameworks.

Our ongoing Rocq mechanization is available on GitHub[3]. The development includes feature functors for call-by-value untyped lambda calculus core, mutability with ML-style references, `if-then-else` constructions, as well as simple error handling that interacts with the core, mutability, and ITE. To the best of our knowledge, feature functor interactions have not been previously investigated, and it is interesting to see how they will shape our mechanization.

[3]https://github.com/FrogOfJuly/js-a-la-Carte

4 Related work

There are other solutions that address proof extensibility.

Extensible metatheory mechanization via family polymorphism. FPOP [10] and Rocqet [5] are Rocq language extensions that compile inductive types into Rocq modules, allowing extensibility with the help of OO-inspired family polymorphism with late bindings. Despite its success in formalizing CompCert and CakeML, its extensible inductive type representation as Rocq modules is virtually incompatible with usual inductive types, which worsens the learning curve and locks development to this particular dialect.

Program Logics à la Carte. Vistrup et al. [21] achieve extensibility by encoding effect-emitting (e.g., non-termination, exception, concurrency, etc.) components of the language as fragments of program logic. Language semantics is expressed as interaction trees [22]: a coinductively defined datatype for representing effectful computations. Development provides several program logic fragments that correspond to commonly encountered pieces in different programming languages. From the practical point of view, however, this approach introduces considerable indirection to language encoding and proofs, especially if it is required to define new program logic fragments.

Intrinsically-typed definitional interpreters à la carte. Van der Rest et al. [20] leverage finitary containers [1] and algebras over them to introduce "intrinsically-typed language fragments". Those bring together common syntax and semantics of chosen language features, while allowing composition. While very satisfying on its own, this approach also suffers from the indirectness of type encoding. Learning curve for container machinery is quite steep, which makes it difficult to adopt the approach to larger languages.

Talk about recent mechanizations that reject modularity: [13, 15]

5 Discussion

In all the techniques discussed above, proof modularity comes with the cost of departing from the usual way of reasoning about inductive types. In the case of Coq à la Carte, and consequently in our work, the departure is not quite dramatic, but it is still makes proofs less intuitive.

A promising way forward would be to establish a correspondence between proofs based on our modular representation and the equivalent monolithic proofs, and implement a transfer mechanism between them. Trocq [4] is a recently developed tool that we expect to enable this transfer automatically. If we are successful, we can expose a familiar interface for those uses that do not require extensibility.

References

[1] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015), e5.

[2] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 87–100.

[3] Ryan Carniato. 2015. *SolidJS: Reactive JavaScript library*. SolidJS project and Core Team. https://solidjs.com/

[4] 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.

[5] Oghenevwogaga Ebresafe, 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.

[6] ECMA. 2015. *ECMA language specification*. ECMA International. https://ecma-international.org/publications-and-standards/standards/ecma-262/

[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] Xiaojia Rao, Stefan Radziuk, Conrad Watt, and Philippa Gardner. 2025. Progressful Interpreters for Efficient WebAssembly Mechanisation. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 627–655.

[14] Daniel Ehrenberg Rob Eisenberg. 2023. *JavaScript Signals standard proposal*. https://github.com/tc39/proposal-signals

[15] Remy Seassau, Irene Yoon, Jean-Marie Madiot, and François Pottier. 2025. Formal Semantics and Program Logics for a Fragment of OCaml. *Proceedings of the ACM on Programming Languages* 9, ICFP (2025), 128–159.

[16] 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.

[17] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

[18] Enrico Tassi. 2025. Elpi: rule-based meta-languge for Rocq. In *CoqPL 2025-The Eleventh International Workshop on Coq for Programming Languages*.

[19] The Coq Development Team. 2024. *The Coq Proof Assistant*. doi:10.5281/zenodo.14542673

[20] 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.

[21] 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.

[22] 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

```
(* Rocq won't allow direct definition
   as showed in Data types a la carte:
   data Fix f = In (f (Fix f))
   so we mimic it with axioms.
*)
Axiom Fix : (Type → Type) → Type.
Axiom fix_def : forall {f : Type → Type}, Fix f = f (Fix f).

Theorem fix_implies_false : False.
Proof.
    remember (Fix (fun A ⇒ A → False)) as A eqn:HeqA.
    assert (not_a : A → False). {
        intros X. assert (X' := X).
        rewrite HeqA in X.
        rewrite fix_def in X.
        rewrite ← HeqA in X.
        exact (X X').
    }
    apply not_a.
    rewrite HeqA.
    rewrite fix_def.
    rewrite ← HeqA.
    exact not_a.
Qed.
```

Figure 2. `Fix` is inconsistent.

```
Class retract X Y :=
  {
    retract_I : X → Y ;
    retract_R : Y → option X;
    retract_works :
        forall x, retract_R (retract_I x) = Some x;
    retract_tight :
        forall x y, retract_R y = Some x → retract_I x = y
  }.
```

Figure 3. Tight retract definition.