

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: JavaScript, formal methods, verification

1 Introduction

Half a page for overview and explaining motivation

2 Inlined functor fixpoints

We follow closely ideas from Data types à la Carte[9] and Coq à la Carte[5] to enable modularity. Sadly, direct use of the meta-programming tools developed in the latter is prohibitively difficult due to outdated Metarocq[8]. However overall structure of reasoning holds well even without them.

The main idea is to separate closed inductive types into modular feature functors and non modular fixpoint operation. Following this idea nicely will lead to inconsistency, so the fixpoint operation is not present explicitly, but inlined each time.

Resulting feature functors allow open-recursion-style per-functor proofs without relaying on a particular structure of overarching type. Once desired proofs are complete it's possible to assemble the type through inlined fixpoint application and proofs by "closing" the recursion over per-functor proofs.

The approach is illustrated with an example at Fig 1 that roughly outlines extension of untyped lambda calculus with with booleans and if-then-else constructions. However, there is much more to it that is possible to cover here.

It is noticable even in this very simple example that there is some boilerplate code that is tedious to write by hand. So, naturally it is tempting to automate its generation.

Coq-Elpi[10] is a rule-based meta-language for Rocq[11] that gives a programmer an ability to generate tactics, inductive types and manipulate syntax with binders. We chose it over Metarocq for its debugging infrastructure, binder handling and availability of onboarding material.

3 Targets for formalization

There is yet to be a significant test of modularity for mainstream programming languages formalization.

There are several[6][1] developments that attempt to formalize and reason about JavaScript, however none of them is easy to extend with new features. Being one of the most used language, JavaScript provides a fertile ground for evaluating

```
1 (* Exp.v *)
2 Inductive Exp :=
3   (* Exp type can be seen as inline fixpoint
4     of coproduct of feature functors *)
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 *)
25 (* Omitted for brevity, but analogous to ite.v *)
```

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

existing approaches for modular reasoning. Lack of sophisticated type system streamlines the encoding and makes it easier to gradually prove language properties, while keeping the formalization open to extension. ECMA[3], an extensive specification in natural language, is also a very welcome addition.

Moreover, JavaScript has several frameworks[4] and dialects (e.g. TypeScript) that enable different styles of programming. Ability to reuse proofs about core language for dialects would be a nice showcase of modularity. The t39 proposal process[7] is transparent and well documented thus permitting mechanization of ongoing specification of nightly features before they are adopted into core language.

There exists industrial cases where it is paramount to have reusable proofs. For example: React vs Signals proposal.

To the best of our knowledge no modular proof technique from above was successfully used for mechanisation of a mainstream programming language.

We plan to, at first, follow λ_{JS} [6] formalization and then gradually extend it with features described in ECMA, while preserving the progress theorem.

Theorem progress : forall c e, !c e \rightarrow
 isValue e \vee
 isError e \vee
 (exists c' e', step c e c' e').

The aim of the project is neither to formalize the whole existing JavaScript semantics nor to develop a generic modular framework. Rather the aim is to test how far one can go with mechanisation by specialising existing approach to JavaScript, while maintaining extendability. With this goal in mind the following features of JavaScript are of the most interest: mutability, exceptions, reactivity and asynchronicity. Ongoing Rocq development is available here¹.

4 Discussion

There are other solutions to increase modularity of proofs.

Extensible metatheory mechanization via family polymorphism.

Rocq plugin for type family polymorphism

Program Logics à la Carte.

Coinduction with ITrees.

Interpreters à la Carte.

Containers as functors for fixpoints

Argue about that indirect encoding is too taxing.

Proof modularity comes with the cost of departing from the usual way of reasoning about inductive types. Even if in case of Coq à la Carte the departure is not quite dramatic, it still requires to rethink how one approaches proofs.

The ideal solution would be to have a correspondence between modular and inductive proofs. There is an existing work [2] that could enable the proofs transfer between "equivalent" datatypes.

Talk about how they achieve that and is it possible to leverage that for functor representation of chosen datatype.

References

- [1] Martin Bodin, Arthur Charguéraud, Daniele Filaretto, Philippa Gardner, Sergio Maffei, 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.
- [2] 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.
- [3] ECMA. 2015. *ECMA language specification*. ECMA International. <https://ecma-international.org/publications-and-standards/standards/ecma-262/>
- [4] Inc Facebook. 2019. *React: A JavaScript library for building user interfaces*. Facebook, Inc. <https://www.reactjs.org/>
- [5] 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.
- [6] 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.
- [7] Ecma International. 2025. *Ecma Technical Committee 39*. Ecma International. <https://ecma-international.org/technical-committees/tc39/>
- [8] Matthieu Sozeau, Abhishek Anand, Simon Boulter, 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.
- [9] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.
- [10] Enrico Tassi. 2025. Elpi: rule-based meta-language for Rocq. In *CoqPL 2025-The Eleventh International Workshop on Coq for Programming Languages*.
- [11] The Coq Development Team. 2024. *The Coq Proof Assistant*. doi:10.5281/zenodo.14542673

¹<https://github.com/FrogOfJuly/js-a-la-Carte>