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 However, directly adopting the implementation [8] in practice is prohibitively difficult due to its dependency on an outdated version of Metarocq [14].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 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 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.

```
(* Exp.v *)
Inductive Exp :=
(* Exp type can be seen as 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.
Fixpoint thrm : forall (e : Exp) → …
intros. destruct e.
(* close recursion *)
  – apply (thrm_ite Exp thrm …).
  – apply (thrm_lam Exp thrm …).
Defined.
(* ITE.v *)
Section exp_ite.
Exp : Type.
Inductive exp_ite :=
  | ite : Exp → Exp → Exp
  | boolt_lit : bool → Exp.
(* assume overarching property *)
Variable thrm : forall (e : Exp) → … .
(* Prove corresponding per-functor property *)
Definition thrm_ite : forall (e : exp_lam) → … .
End exp_ite.
(* Lambda.v: ommited, 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 semantics[2] 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 then 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

[1] `Fix (Λ A. A -> False)` allows to obtain proof of `False`. See appendix Figure 2

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

on various editions of the exhaustive natural language specification ECMA-262 [5]) with support for new features is impossible without manually rewriting a significant portion of an existing codebase.

Being one of the most-used languages, JavaScript provides a fertile ground for evaluating approaches for modular reasoning: its lack of a type system streamlines the encoding and makes it easier to *gradually* prove language properties, while keeping the mechanization open to extension.

With modularity, proofs about the core language will be reusable to mechanize the most recent additions to the language (i.e., the upcoming features – even before they become part of the ECMA-262 specification). It will also be possible to reuse proofs for further properties of the language's dialects (e.g., TypeScript), as well as various JavaScript frameworks (e.g., React, Solid, Angular, Vue, etc).

We are particularly interested in mechanizing mutability, exceptions, asynchrony and reactivity of JavaScript—in light of the *Signals* proposal [13], which is currently being considered by the JavaScript committee.

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

```
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'.
```

Our ongoing Rocq mechanization is available at https://github.com/FrogOfJuly/js-a-la-Carte. The implementation 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. Even this early in the development with such simple language, feature interaction(error handling in this case) requires isolation. To the best of our knowledge, this has not been previously investigated, and it is interesting to see how this issue will shape our mechanization.

4 Related work

There are other solutions to increase modularity of proofs.

Extensible metatheory mechanization via family polymorphism. FPOP[10] and Rocqet[4] are a Rocq language extensions which compile modular inductive types into Rocq modules allowing extension with the help of OO-inspired family polymorphism with late bindings.

Despite its success in formalizing CompCert and CakeML its modular type representation as modules is virtually incompatible with the rest of Rocq which increases learning curve and locks development to this particular dialect.

Am I misinterpreting all of this and it's OK and actually the best solution?

Program Logics à la Carte[19] achieves modularity 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[20]: coinductively defined datatype for representing effectful computations. Development provides several program logics fragment which correspond to commonly encountered pieces in different programming languages.

From the practical point of view however, this approach introduces considerable indirection to encoding and proofs, especially if it is required to define new program logic fragments.

Does it?

Intrinsically-typed definitional interpreters à la carte[18] leverages finatary 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 it's own, this approach also suffers from indirectness of modular type encoding. Learning curve for container machinery is quite steep and thus makes it hard to adopt for proofs about larger languages.

Should be worded better.

5 Discussion

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 proofs for modular representation and equivalent inductive types. There is an existing work[3] that could enable such transfer of proofs.

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

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] 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] 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.

[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-languge 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

```
(* Rocq won't allow direct definition
   as showed in Data types a la carte:
   data Fix f = In (f (Fix f))
*)
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` existence implies falsehood.