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[12] and Coq à la Carte[7] to enable modularity. Sadly, direct use of the meta-programming tools developed in the latter is prohibitively difficult due to outdated Metarocq[11]. Howeverm 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 nievly will lead to incosistency, 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 its possible to assemble the type through inlined fixpoint application and proofs by "closing" the recursion over per-fucntor proofs.

The apporach is illustrated with an example at Fig 1 that roughtly 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 tedeous to write by hand. So, naturally it is tempting to automate its generation.

Coq-Elpi[13] is a rule-based meta-lanauge for Rocq[14] that gives a programmer an ability to generate tactics, inductive types and manipulate syntax with binders. We chose it over Metarocq for it's debugging infrostructure, binder handling and availability of onboarding material.

3 Targets for formalization

There is a significant progress in applying modular techniquies to existing programming languages(e.g. CompCert and CakeML[4]), however ...??

Turns out there are recent developments in applying modular approaches. That could have been akward..

```
(* Exp.v *)
Inductive Exp :=
(* Exp type can be seen as inlined fixpoint
   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 for brievity, but analogous to ite.v *)
```

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

There are several[8][2] developments that attempt to formalize and reason about JavaScript, however non of them is easy to extend with new features. Being one of the most used language, JavaScript provides a fertile ground for evaluating 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[5], an extensive specification in natural language, is also a very welcome addition.

Moreover, JavaScript has several frameworks[6] 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[9] 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}[8] formalization and then gradually extend it with features described in ECMA, while preserving the following 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'.
```

The aim of the project is niether 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 asynchronicy. Ongoing Rocq development is available here[1].

4 Discussion

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[16] 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[17]: 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 prohibitive degree of 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[15] 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 this way of doing proofs for larger languages.

Should be worded better.

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.

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

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. 2019. *React: A JavaScript library for building user interfaces.* Facebook, Inc. https://www.reactjs.org/

[7] 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.

[8] 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.

[9] Ecma International. 2025. *Ecma Technical Committee 39.* Ecma International. https://ecma-international.org/technical-committees/tc39/

[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] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.