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, program equivalence

1 Introduction

Half a page for overview and explaining motivation

2 à-la-carte-ness

We chose Rocq[12] because of metatooling support

We follow closely ideas from Data types à la Carte[11] 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 Sadly, direct use of the meta programming tools developed in the latter is prohibitively difficult due to outdated Metarocq[10], but the overall structure of reasoning holds well even without them.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. 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 through "closing" the recursion over per-fucntor proofs.

The apporach is illustrated with an example 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.

Next piece that is important to discuss is metaprogramming tools that are available in Rocq.

Explain ELPI

Imporovements, what I want to do

```
(* Exp.v *)
Inductive Exp :=
(* Exp type can be seen as inline fixpoint
   of coproduct of feature functors  *)
  | inj_ite : exp_ite Exp → Exp
  | inj_lam : exp_lam Exp → Exp.
Fixpoint thrm : forall (e : Exp) → ...
intros. destruct e.
(* open 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 *)
(* Analogous to ite.v *)
```

Figure 1. minimal example

3 Targets for formalization

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

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 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[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[8] is transparent and well documented thus permitting mechanization of ongoing specification of nightly features before they are adopted into core language.

To the best of our knowledge no modular technique from above was ever 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, lc e →
        isValue e ∨
        isError e ∨
        (exists c' e', step c e c' e').
```

The aim of the project is not to formalize the whole existing JavaScript semantics, but rather to test how far one can go with mechanisation, 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].

One can do the same formalization as λ_{JS} for Featherweight Java[7].

4 Discussion

There are other solutions to increase modularity of proofs.

Proof modularity papaers

1. Family Polymorphism[9]

   Rocq plugin for type family polymorphism

2. Program Logics à la Carte[14]

   Coinduction with ITrees.

3. Interpreters à la Carte[13]

   Containers as functors for fixpoints

Argue about that indirect encoding is too taxing.

It's interesting to look into the possibility of gradually encoding calculus of inductive constructions in a modular fashion.

Proof modularity comes with the cost of departing from the usual way of reasoning about inductive types. Even in case of Coq à la Carte departure is not quite dramatic, but 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.

Is it possible to use containers as a meta language, while presreving ability to do actual reasoning with inductive types in Rocq?

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

References

[1] 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.

[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] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 3 (2001), 396–450.

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.