Artifact Report: an Abstract, Certified Account of Operational Game Semantics

Peio Borthelle (Université Savoie Mont Blanc), Tom Hirschowitz (CNRS), Guilhem Jaber (Nantes Université), and Yannick Zakowski (Inria)

This artifact report is a companion to the ESOP'25 paper *An abstract, certified account of operational game semantics* [3]. The paper describes the construction of a *sound model* for an abstract notion of language. The model is built using a semantic technique named *Operational Game Semantics (OGS)*.

All our results are formalized in the Coq proof assistant: this formalization[1] constitutes the artifact we discuss in the present document. More specifically, our formalization covers our main result, the soundness of the abstract OGS model w.r.t. substitution equivalence (Theorem 8), as well as four example calculi: two variants of call-by-value λ-calculi and two variants of $\mu\tilde{\mu}$-calculi [4,5]. The only axiom used is the Axiom K [17] for equality proof irrelevance[2].

The `README` details the installation process, and the structure of the code. An online rendering[3] is available thanks to Alectryon [12]. Furthermore, our paper provides *systematic hyperlinks* from statements to their formal Coq counterparts. We encourage the interested reader to use these tools to navigate the code.

In this document, we focus first on users: how to read and instantiate our main result. In a second time, we detail salient technical aspects of our formalization.

1 The `OGS` Library from the Perspective of a User

Our library is intended to be reusable. In this section, we describe to the interested user how to understand our result, and how to instantiate it.

Soundness Theorem The main theorem (Thm. 8) is proven in `OGS/Soundness.v`. It quantifies over any language machine verifying the required hypotheses, that is, an axiomatization in the style of abstract machines of substitution and evaluation. Slightly unfolding the definitions, it is typed as follows.

```
Theorem ogs_correction {Γ} Ω (x y : conf Γ)
  : m_strat _ (inj_init_act Ω x) ≈ m_strat _ (inj_init_act Ω y)
  -> forall γ : Γ =[val]> Ω, evalₒ (x ₜ⊛ γ) ≈ evalₒ (y ₜ⊛ γ).
```

In plain words, for any final scope Ω and language machine configurations x and y, whenever the two OGS strategies obtained by embedding x and y into initial strategy states are weakly bisimilar, then, for any assignment γ, substituting x and y by γ and evaluating them to a final observation yields two weakly bisimilar computations. In other words, either both substituted configurations diverge or both return the same final result.

[1] `https://github.com/Lapin0t/ogs/tree/esop25`
[2] To ease dependent pattern matching due to the intrinsically scoped representation.
[3] `https://lapin0t.github.io/ogs/Readme.html`

Language Machine Instantiation Several instances of language machines verifying the OGS soundness hypotheses are provided in the `Examples/` folder. Instanciating our generic semantics and proof with one's favorite language always follows the same blueprint. First define the standard syntax and substitution. Then, because the language evaluator must be presented as an abstract machine, define a family of configurations for this machine. Note that this abstract machine must reduce *open* configurations and these must also support substitution. For some languages such as $\mu\tilde{\mu}$-calculi [4], or say, Jump-With-Argument [10], configurations are already a standard notion, but for others such as λ-calculi, they are generally obtained by pairing a term with an evaluation context. Further, define the observation structure. This amounts to deciding which part of normal form *configurations* should be considered observable for the purpose of observational equivalence testing, and which part should be considered as a set of opaque *values*. Finally, when all these choices are made and the evaluator written, it sufficies to verify the theorem hypotheses. In our experience, they are always proven quite directly, without any surprising lemma required.

2 Inside the Beast: Implementation Details of Interest

We detail two salient aspects of our development. First, we have implemented strategies as shallow monadic computations, which led us to develop an indexed variant of the Interaction Tree (ITree) structure. Second, we have followed a well-scoped approach, relying on the `Equation` library and the `SProp` universe. This section is intended for experts readers familiar with the companion paper.

2.1 Strategies as Indexed Interaction Trees

Indexed Interaction Trees Library Rather than *specifying* labelled transition systems relationally in `Prop`, we have chosen a shallow Coq embedding for implementing strategies, understood as possibly non-terminating computations featuring uninterpreted actions (move exchanges). This suggests reusing the ITrees [18], which is designed for this purpose. It is also in line with Hancock and Hyvernat's similar construction of *interaction structures* [9] which represent agents in client-server protocols as coinductive trees.

However, the ITree's axiomatization of the possible actions is not expressive enough for our purpose. In essence, they capture games where the set of allowed client moves is constant throughout the game. To fix this, instead of describing possible actions by a polynomial functor on **Set**, we describe them by an *indexed* polynomial functor on $\mathbf{Set}^I$, for some set I representing active game positions. The `ITree/` folder thus contains a succinct port of the ITree library to this new indexed setting. This part of the artifact could certainly be useful independently of the OGS construction. We hope to extract it into a self-contained library.

This small library contains the coinductive definition of ITrees, their monadic structure, and the standard iteration operator. We provide definitions for weak and strong bisimilarity together with the main reasoning principles: strong bisimulation up-to equivalence and weak and strong bisimulation up-to monadic bind.

Both bisimilarity relations are defined using the `Coinduction` [14] library, which provides enhanced coinduction principles based on a lattice theoretic fixed point construction.[4] This relies on the impredicativity of the `Prop` universe.

Eventually Guarded Iteration Apart from indexation, the main novelty are two new iteration operators, respectively for *guarded* and *eventually guarded* iteration (Prop. 5). They have been crucial in our OGS soundness proof and could certainly be backported to the unindexed ITree library.

The standard unguarded operator can iterate any "loop body", but it must insert a silent step after each iteration to be well-defined. Hence, it only produces a fixed point of the loop body w.r.t. weak bisimilarity, and it is in general not unique. In contrast, our two new operators do not insert any silent step after iteration, and produce *unique* fixed points w.r.t. strong bisimilarity. For this to work, they respectively require the loop body to feature a guard (a computation step) at each iteration or infinitely often, i.e., after a finite number of iterations.

Theoretically, unguarded iteration can be axiomatized as a complete Elgot monad structure [7] on ITrees quotiented by weak bisimilarity, while our new guarded iteration operators yield two completely iterative monad structures with different notions of guardedness on ITrees quotiented by strong bisimilarity [8].

Relationship with Transition Systems over Games Following Levy and Staton [11], in the paper (Def. 19) we have defined strategies over some game $\mathcal{G} \colon \texttt{Game}\ I\ J$ as a pair of active and waiting state families $S^+ : \mathbf{Set}^I$ and $S^- : \mathbf{Set}^J$, together with action and reaction morphisms. This data is dubbed by Levy and Staton a *big-step system over* G and can be more succinctly expressed as a coalgebra for the following endofunctor on $\mathbf{Set}^I \times \mathbf{Set}^J$.

$$(S^+, S^-) \mapsto (\mathcal{D}(\texttt{final}_\mathcal{G} + [\![\texttt{client}_\mathcal{G}]\!]^+ S^-),\ [\![\texttt{server}_\mathcal{G}]\!]^- S^+)$$

Instead of working with arbitrary coalgebras, we can equivalently see strategies as their image in the *final* coalgebra, whose states consist of coinductive trees. We do not construct this final coalgebra directly, but instead express it using our indexed interaction tree construction. Given a polynomial endofunctor Σ on $\mathbf{Set}^I$ and an output family X, indexed interaction trees are given by the following final coalgebra:[5] $\texttt{itree}_\Sigma(X) \coloneqq \nu S.\ X + S + \Sigma(S)$.

The trick is to focus on strategies in an active positions, by considering the sequence of choosing a client move and waiting for the next server move as one unit. Indeed, the composition $[\![\texttt{client}_\mathcal{G}]\!]^+ \circ [\![\texttt{server}_\mathcal{G}]\!]^-$ is a polynomial functor, and the states of the final coalgebra of strategies over $\mathcal{G}$ can be computed as:[6]

$$\begin{aligned}\texttt{strat}^+_\mathcal{G} &\coloneqq \texttt{itree}_{([\![\texttt{client}_\mathcal{G}]\!]^+ \circ [\![\texttt{server}_\mathcal{G}]\!]^-)} \texttt{final}_\mathcal{G} \\ \texttt{strat}^-_\mathcal{G} &\coloneqq [\![\texttt{server}_\mathcal{G}]\!]^- \texttt{strat}^+_\mathcal{G}\end{aligned}$$

[4] `Coinduction` recently upgraded from the *companion* to a *tower induction* construction [15]. We have not made this port yet, and hence compile against Coq (8.17).

[5] Note that this is exactly the same construction as given by Xia et al. [18], simply taking place in the category $\mathbf{Set}^I$ instead of $\mathbf{Set}$.

[6] We do not formally prove it computes the announced final coalgebra, but this can be shown by straightforward fixed point calculation, recalling that $\mathcal{D}(X) \coloneqq \nu A.X + A$.

Our implementation choices are then straightforward. We encode polynomial endofunctors on $\mathbf{Set}^I$ as *indexed containers* [2], which we dub *events*.

```
Record event (I : Type) := Event {
  e_qry : I -> Type ;
  e_rsp : forall i, e_qry i -> Type ;
  e_nxt : forall i (q : e_qry i), e_rsp i q -> I }.
```

For some indexed container `E : event I` and output family `X : I -> Type`, the interaction tree endofunctor and its final coalgebra are respectively:

```
Variant itreeF (REC : I -> Type) (i : I) :=
| RetF (r : X i)
| TauF (t : REC i)
| VisF (q : E.(e_qry) i) (k : forall r : E.(e_rsp) q, REC (E.(e_nxt) r)).
CoInductive itree (i : I) := go { observe : itreeF itree i }.
```

2.2 Scope Structures

Our development of intrinsically-typed-and-scoped syntax largely follows the standard practice [6,1]. As this formalization style is heavy on dependent pattern matching, we make great use of the `Equations` plugin [16]. A notable novelty is that we abstract over the concrete representation of scopes and variables, which are usually fixed to lists of object language types and well-typed de Bruijn indices. Our motivation was pragmatic: using tailor made variable representations drastically reduced the amount of boilerplate in the complex $\mu\tilde{\mu}$-calculi instances.

The root cause is that most standard OGS examples involve separating object language types into so-called *positive* and *negative*, with only variables of *negative* type being shared between OGS players and observed. Given a *strict* predicate `is_neg : ty -> SProp`, negative types can be constructed as the strict subset `{ t : ty | is_neg t }`. The prime benefit is that definitional equality of negative types is exactly definitional equality of the underlying "vanilla" types. For scopes containing only negative types, we lose this nice property if we represent them naively as `list { t : ty | is_neg t }`. It is vastly more convenient to work with the subset `{ ts : list ty | allS is_neg ts }`, where `allS` denotes the strict universal quantifier on lists.

To allow for such "non-standard" scope representations and their custom notion of well-typed variable, we devise a notion of *scope structure* close in spirit to the *Nameless, Painless* approach [13]. A scope structure on $S\colon \mathbf{Set}$ for object language types T consists of an element $\varnothing\colon S$, a binary operation $\oplus\colon S \to S \to S$ and a family of variables $\ni\colon S \to \mathbf{Set}^T$, subject to the two isomorphisms $\varnothing \ni t \approx \bot$ and $(\Gamma \oplus \Delta) \ni t \approx (\Gamma \ni t) \uplus (\Delta \ni t)$. The category of contexts is then taken to be the full image of $\ni$. In other words, objects are elements of S and renamings $\Gamma \to \Delta$ are given by functions $\forall t, \Gamma \ni t \to \Delta \ni t$. This interface can then be instantiated both by lists and de Bruijn indices as well as by our "subset scopes". The substitution metatheory is left mostly unchanged.

References

1. Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type and scope safe universe of syntaxes with binding: their semantics and proofs. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (Jul 2018). `https://doi.org/10.1145/3236785`
2. Altenkirch, T., Ghani, N., Hancock, P.G., McBride, C., Morris, P.: Indexed containers. J. Funct. Program. **25** (2015). `https://doi.org/10.1017/S095679681500009X`
3. Borthelle, P., Hirschowitz, T., Jaber, G., Zakowski, Y.: An abstract, certified account of operational game semantics. In: ESOP (2025)
4. Curien, P., Herbelin, H.: The duality of computation. In: ICFP. pp. 233–243. ACM (2000). `https://doi.org/10.1145/351240.351262`
5. Downen, P., Ariola, Z.M.: Compiling with classical connectives. Log. Methods Comput. Sci. **16**(3) (2020). `https://doi.org/10.23638/LMCS-16(3:13)2020`
6. Fiore, M., Szamozvancev, D.: Formal metatheory of second-order abstract syntax. Proc. ACM Program. Lang. **6**(POPL), 1–29 (2022). `https://doi.org/10.1145/3498715`
7. Goncharov, S., Milius, S., Rauch, C.: Complete elgot monads and coalgebraic resumptions. In: MFPS. Electron. Note Theor. Comput. Sci., vol. 325, pp. 147–168. Elsevier (2016). `https://doi.org/10.1016/J.ENTCS.2016.09.036`
8. Goncharov, S., Schröder, L., Rauch, C., Piróg, M.: Guarded and unguarded iteration for generalized processes. Log. Methods Comput. Sci. **15**(3) (2019). `https://doi.org/10.23638/LMCS-15(3:1)2019`
9. Hancock, P., Hyvernat, P.: Programming interfaces and basic topology. Ann. Pure Appl. Log. **137**(1-3), 189–239. `https://doi.org/10.1016/J.APAL.2005.05.022`
10. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semant. Struct. Comput., vol. 2. Springer (2004)
11. Levy, P.B., Staton, S.: Transition systems over games. In: LICS. pp. 64:1–64:10. ACM (2014). `https://doi.org/10.1145/2603088.2603150`
12. Pit-Claudel, C.: Untangling mechanized proofs. In: SLE. pp. 155—-174. ACM (2020). `https://doi.org/10.1145/3426425.3426940`
13. Pouillard, N.: Nameless, painless. In: ICFP. pp. 320–332. ACM (2011). `https://doi.org/10.1145/2034773.2034817`
14. Pous, D.: Coinduction all the way up. In: LICS. pp. 307–316. ACM (2016). `https://doi.org/10.1145/2933575.2934564`
15. Schäfer, S., Smolka, G.: Tower induction and up-to techniques for CCS with fixed points. In: RAMiCS. Lect. Note Comput. Sci., vol. 10226, pp. 274–289 (2017). `https://doi.org/10.1007/978-3-319-57418-9_17`
16. Sozeau, M., Mangin, C.: Equations reloaded: high-level dependently-typed functional programming and proving in coq. Proc. ACM Program. Lang. **3**(ICFP), 86:1–86:29 (2019). `https://doi.org/10.1145/3341690`
17. Streicher, T.: Investigations into intensional type theory. Habilitiation Thesis, Ludwig Maximilian Universität (1993)
18. Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. **4**(POPL), 51:1–51:32 (2020). `https://doi.org/10.1145/3371119`