An abstract, certified account of operational game semantics

PEIO BORTHELLE, Université Savoie Mont Blanc, CNRS, LAMA, France
TOM HIRSCHOWITZ, Université Savoie Mont Blanc, CNRS, LAMA, France
GUILHEM JABER, Nantes Université, LS2N, France
YANNICK ZAKOWSKI, ENS de Lyon, INRIA, CNRS, Lyon 1, LIP, France

Operational game semantics (OGS) is a method for interpreting programs as strategies in suitable games, or more precisely as labelled transition systems over suitable games, in the sense of Levy and Staton. Such an interpretation is called sound when, for any two given programs, bisimilarity of associated strategies entails contextual equivalence. OGS has been applied to a variety of languages, with rather tedious soundness proofs.

In this paper, we contribute to the unification and mechanisation of OGS. Indeed, we propose an abstract notion of language with evaluator, for which we construct a generic OGS interpretation, which we prove sound. Our framework covers a variety of simply-typed and untyped lambda-calculi, notably featuring fixed points, continuations, and recursive datatypes. All constructions and proofs are entirely mechanised in the Coq proof assistant.

Additional Key Words and Phrases: operational game semantics, certified mathematics

ACM Reference Format:
Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. 2023. An abstract, certified account of operational game semantics. In . ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Operational game semantics [18, 20] (OGS) is a method for constructing models of programming languages. OGS finds a sweet spot between contextual equivalence, which is entirely syntactic, and game semantics [2, 11], which is syntax-free. OGS models are generally easier to construct than proper game models, e.g., some languages have OGS models but no known proper game model [17]. Furthermore, OGS models are useful for proving the correctness of decision procedures for contextual equivalence [4, 14, 22].

OGS proceeds by interpreting open programs as labelled transition systems (LTSs) and comparing the resulting LTSs w.r.t. weak bisimilarity. The labelled transitions model the interactions of the considered program with any possible environment, i.e., a substitution and an evaluation context. Crucially, the constructed LTSs are first-order, even for higher-order languages.

Each OGS model comes with a *soundness* proof, i.e., a proof that for all pairs of programs, weak bisimilarity of induced strategies entails contextual equivalence. Such proofs are rather difficult, and are done by hand on a case-by-case basis. They rely on a notion of *composition* of LTSs.

A few authors have attempted to design a unified theory of OGS. Notably, Levy and Staton [21] offered a high-level categorical framework. More recently, Laird [19] proposed a unifying framework for OGS, in which he proves that weak bisimilarity of LTSs is a congruence w.r.t. composition, a standard lemma towards soundness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESOP '25, 2025, Hamilton, Canada

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06…$gratos
https://doi.org/XXXXXXX.XXXXXXX

Contribution. In this work, we go further, and prove a generic soundness result, mechanised in Coq. We thus contribute to both unification and mechanisation of OGS. Our contributions to unification are as follows.

- We introduce an abstract notion of language with evaluator, called a *language machine*, which notably covers several λ-calculi and variants of $\bar{\lambda}\mu\tilde{\mu}$-calculi [7, 8].
- For any language machine, we construct an OGS model.
- We prove that this model is sound w.r.t. some abstract analogue of contextual equivalence, under suitable hypotheses.

We furthermore provide a complete Coq mechanisation of our results, to emphasise their computational aspects and firmly ground our model in a constructive meta-theory. We favour a traditional, code-less, exposition along the paper for clarity. For the interested reader, we however systematically use hyperlinks represented by ()[1] to link definitions and theorems to their mechanised counterpart. The Coq development is inspired by Levy and Staton's transition systems over games [21], and includes notably the following main contributions.

- We present OGS using the well-scoped approach (), in the sense that everything is indexed by typing contexts, and variables are accessed as de Bruijn indices. This is in sharp contrast with previous work, which uses nominal style.
- We instantiate our abstract notion of language on several concrete examples: a simply-typed call-by-value λ-calculus with recursion (), a pure untyped call-by-value λ-calculus (), the $\bar{\lambda}\mu\tilde{\mu}_Q$-calculus [7] (), the polarised System L (), and the polarised System D () from Downen and Ariola [8].
- We implement () an indexed variant of the *interaction trees* library [25], which we use to define LTSs coinductively — as opposed to the more traditional, relational definition.
- We introduce () a new fixed-point combinator over a system of so-called *eventually guarded* equations, whose solution is unique w.r.t. strong bisimilarity. We use this combinator to define composition of OGS LTSs, which is a crucial ingredient to the soundness proof.

Before diving into the details, let us provide a high-level overview of the technical development.

Overview. Our first step, in §2, consists in reformulating the standard contextual equivalence used in OGS, namely *Closed Instantiation of Use* (CIU) equivalence [23, 24]. The definition of CIU equivalence is similar to contextual equivalence, but w.r.t. a simpler notion of context. Mason and Talcott proved that CIU equivalence in fact coincides with contextual equivalence.

Standard CIU equivalence checks that its arguments, say M and N, behave the same under any closed instantiation of their free variables, in any closed evaluation context of some fixed, basic type like the booleans: $E[M[\sigma]] \cong E[N[\sigma]]$, for some notion of observation of closed booleans. We argue that, up to mild expository adjustments, this may in fact be viewed as M and N behaving the same under any closed instantiation of their free variables. The idea is to make the ambient evaluation context explicit, modelling it as a "context variable". Thus, we move from terms M to so-called *named terms* $\alpha[M]$ for some context variable α, or, in $\mu\tilde{\mu}$-style notation, $\langle M \mid \alpha \rangle$, and putting M in some evaluation context E is now modelled as a mere instantiation $\{\alpha \mapsto E\}$. There is a slight twist to handle the top-level context, but ignoring this for now, up to such expository adjustments, one may consider CIU equivalence as comparing its arguments under any closed instantiation. We call this *substitution equivalence.* This slightly non-standard presentation— conflating both usual values and evaluation contexts into a single syntactic category of generalized values—is instrumental to simplify the axiomatisation of a language and the generic OGS construction.

[1] To the anonymous reviewers: for the purpose of this submission, we link to an entirely anonymous github repository. They can be soundly followed without breach of anonymity.

After briefly recalling OGS on a simple language in §3, we introduce our abstract notion of language with evaluator, called *language machine* in §4. For this, there is a basic, straightforward infrastructure, and then what we view as the core of OGS, namely how evaluation shapes the messages exchanged by the program and environment. Roughly, the basic infrastructure consists of

- a set of *types* — *contexts* are then defined as sequences of types,
- a family of *configurations*, indexed by contexts, and
- a family of *values*, indexed by *sequents*, i.e., pairs of a context and a type.

In this presentation, configurations replace the more usual syntactic category of *terms* and values subsume both the usual values and evaluation contexts. Configurations and values come with a *substitution* operation that replaces variables with values.

Beyond this basic infrastructure, language machines further feature an evaluation map, whose shape is inspired by the treatment of messages (a.k.a. moves) in traditional OGS. There, when a normal form is reached, it is immediately split into:

- the *head* variable, on which the normal form is stuck,
- the *observation* performed on it, which may introduce fresh variables, and
- the *filling*, which assigns values to these fresh variables.

A *message* is then sent to the other player, consisting of the head variable and the observation; the filling is stored for later use.

Example 1.1. The simplest example is perhaps when the normal form is a value v. With explicit context variables, this is modelled as $\langle v \mid \alpha \rangle$, for some context variable α. In this case:

- The head variable is α.
- The observation is a "return" observation $\langle x \mid \square \rangle$, saying that we are returning a value to the head variable, here denoted by $\square$. The returned value is not given in the observation but instead referred to by the fresh variable x, to be understood as a binder.
- Finally, the filling is the assignment $\{x \mapsto v\}$.

Example 1.2. A slightly less trivial, yet typical example is when the normal form has the shape $E[x\ v]$, for some evaluation context E and value v. In our notation, this looks like $\langle x \mid \bullet v; E \rangle$. In this case:

- The head variable is x.
- The observation is an "apply" observation $\langle \square \mid \bullet y; \alpha \rangle$, saying that we are applying the head variable $\square$ to some argument y in the context α.
- Finally, the filling is the assignment $\{y \mapsto v, \alpha \mapsto E\}$.

We model this in the abstract setting by introducing the notion of *observation structure*. An observation structure is a type-indexed family of *observations*, equipped with a map dom associating a context to each observation. Evaluation is then postulated as a partial map from configurations to triples of

- a *head* variable x in the current context,
- an observation o over the type of x, and
- a value of the expected type for each variable in $\mathsf{dom}(o)$,

satisfying some coherence axioms.

Remark 1.3. In this case, what is meant by *partial map* is a map $A \to \mathscr{D}\,B$, where $\mathscr{D}$ denotes Capretta's *delay* monad [5], see Definition 4.5 below.

This axiomatisation suffices for defining an abstract counterpart of the substitution equivalence introduced in §2 (Definition 4.24), and of OGS (§5). For the latter, we first introduce an abstract

notion of two-player game, essentially following Levy and Staton [21], and construct such a game from any language machine. Moves (a.k.a. messages) in this game are pairs of a variable and an observation over its type. We then define the interpretation of the language's configurations as strategies in this game.

We are then in a position to state our main result (Theorem 6.14), in §6, which states that any two configurations that have weakly bisimilar interpretations as strategies, are in fact substitution equivalent. For the result to actually hold, however, we need to make an additional hypothesis. This was surprising to us, as this hypothesis is trivially satisfied in most languages we know. To the best of our knowledge, this is the first time this condition is explicitly identified.

Let us note that this hypothesis is adjacent but orthogonal to the well-known "infinite chattering" problem in game semantics or process calculi, i.e., the composition of total strategies. In OGS settings, an important reasoning step is to exhibit a bisimulation between a composition of two strategies on one side and a configuration evaluation on the other side. While the evaluation unsurprisingly features only (silent) reduction steps, the composition features both reduction steps and (again silent) interaction steps. As such, the bisimulation needs to "skip over" interaction steps. To justify this skipping over, the main stake is to find reduction steps often enough, in between P and O interaction steps. Essentially, the problem is that upon receiving a message, a player will replace the head variable with a value, but this may not yield a reducible active configuration. One such case, dubbed *chattering*, is when the replacing value is again a variable. This is easily dealt with in our proof with some type indexing, structurally enforcing acyclicity. Our new hypothesis is needed in the other case, i.e. when the replacing value is not a variable, as nothing can be deduced about reduction steps of the obtained configuration.

For example, let us consider the "return" observation above $m \coloneqq \langle x \mid \square \rangle$. Upon reception of a message of the form (α', m), a player starts by instantiating the head variable α'. If this is in turn a variable, then the other player finds themselves in a similar situation, and so on, until some non-variable value is reached, by finiteness of chattering as discussed above. The active configuration now has the shape $\langle x \mid E \rangle$, for some E. However, it might well be the case that $\langle x \mid E \rangle$ still does not reduce[2]. In the meantime since the first message, not a single reduction step has happened. In order to ensure that the interaction is productive w.r.t. reduction steps, we need to make sure that this instantiation process terminates: this is precisely the purpose of our additional hypothesis, postulating well-foundedness of the relation describing these problematic instanciations.

Plan. After explaining the idea of substitution equivalence in §2 and briefly recalling OGS in §3, we introduce language machines and the general notion of substitution equivalence in §4. We then define the OGS model of any language machine in §5. We sketch the soundness proof in §6, with a focus on our fixed-point combinator for eventually guarded equations. Finally, we provide a comparison with the existing literature in 8, and conclude and give some perspectives in §9.

2 CIU EQUIVALENCE THROUGH SUBSTITUTION EQUIVALENCE

In this section, we explain the idea of substitution equivalence, and the necessary pre-processing step that come with it, on a simple example, namely simply-typed, call-by-value λ-calculus with a unit type and recursive functions. Terms are generated by the following grammar

$$\text{values} \ni v, w \Coloneqq x \mid \{\} \mid \lambda^{\mathbf{rec}} f, x.\, p$$
$$\text{programs} \ni p, q \Coloneqq v \mid p\, q$$

[2]For example if x is of function type and E is a context starting by applying the hole to some value.

where λ^{rec} binds f and x in t, as usual. The language is typed. Types and typing contexts are generated by the following grammar,

$$A, B ::= \top \mid A \to B \qquad\qquad \Gamma ::= \varepsilon \mid \Gamma, x\colon \tau$$

with the following, standard typing rules.

$$\frac{x\colon \tau \in \Gamma}{\Gamma \vdash x\colon \tau} \qquad \frac{}{\Gamma \vdash \{\}\colon \top} \qquad \frac{\Gamma, f\colon A \to B, x\colon A \vdash p\colon B}{\Gamma \vdash \lambda^{\mathrm{rec}} f, x.\, p\colon A \to B} \qquad \frac{\Gamma \vdash p\colon A \to B \qquad \Gamma \vdash q\colon A}{\Gamma \vdash p\, q\colon B}$$

From now on, all terms are implicitly considered as coming with a typing derivation. Capture-avoiding substitution is defined as usual, and we evaluation contexts are defined by the following grammar

$$\text{contexts} \ni E ::= \square \mid p\, E \mid E\, v.$$

Context application is defined accordingly. Finally, evaluation is defined by the following inference rules.

$$\frac{}{(\lambda^{\mathrm{rec}} f, x.\, p)\, v \to p[f \mapsto (\lambda^{\mathrm{rec}} f, x.\, p), x \mapsto v]} \qquad \frac{p \to q}{E[p] \to E[q]}$$

As explained in the introduction, CIU equivalence of p and q is defined to mean $E[p[\sigma]] \cong E[q[\sigma]]$, for all closing substitutions σ and boolean contexts E, for some fixed equivalence relation $\cong$ between booleans closed programs. More precisely,

Notation 2.1. We write $\Gamma \vdash \sigma\colon \Delta$ for assignments to each variable $x\colon \tau \in \Delta$ of a value of type τ in context Γ.

Definition 2.2. Two programs $\vdash p, q$ of type $\top$ are said to *coterminate* whenever the evaluation of p ends iff the evaluation of q ends, or more precisely, $(p \to^* \{\}) \Leftrightarrow (q \to^* \{\})$.

Definition 2.3. For any context Γ and type A, two programs $\Gamma \vdash p, q\colon A$ are *CIU-equivalent*, which we denote by $p \approx_{\mathrm{CIU}} q$, iff for all assignments $\varepsilon \vdash \sigma\colon \Gamma$, and closed evaluation contexts E of type unit with a hole of type A, $E[p[\sigma]]$ and $E[q[\sigma]]$ coterminate.

With the main purpose of unifying notions, and hence simplifying the abstract framework, we want to put context application $E[-]$ and substitution $(-)[\sigma]$ on an equal footing in this definition.

Let us first give a bird's eye view of the process, and only then get into some technical detail.

The overall idea is to compile our simply-typed, call-by-value λ-calculus down to a slightly lower-level language [7, 8], in which

- evaluation contexts are first-class values,
- there are context variables, which may be replaced by evaluation contexts in substitutions, just like program variables may be replaced by values, and
- evaluation gets closer in style to an abstract machine, as it now operates on "configurations" $\langle p \mid \pi \rangle$, i.e., pairs of a program p and an evaluation context π.

This change in style is reflected in the syntax of evaluation contexts by writing

- $\bullet v; \pi$ instead of $\pi[\square\, v]$, and
- $p\bullet; \pi$ instead of $\pi[p\, \square]$.

Thus, e.g., a reduction of the form

$$(\lambda^{\mathrm{rec}} f, x.\, x)\, v\, w \to v\, w$$

will be interpreted as taking place in a suitably-typed, variable evaluation context α, and compiled down to

$$\begin{aligned}
\langle (\lambda^{\mathbf{rec}} f, x.\, x)\, v\, w \mid \alpha \rangle &\to \langle w \mid ((\lambda^{\mathbf{rec}} f, x.\, x)\, v)\bullet; \alpha \rangle && \text{evaluate argument} \\
&\to \langle (\lambda^{\mathbf{rec}} f, x.\, x)\, v \mid \bullet w; \alpha \rangle && \text{push argument} \\
&\to \langle v \mid (\lambda^{\mathbf{rec}} f, x.\, x)\bullet; \bullet w; \alpha \rangle && \text{evaluate argument} \\
&\to \langle \lambda^{\mathbf{rec}} f, x.\, x \mid \bullet v; \bullet w; \alpha \rangle && \text{push argument} \\
&\to \langle v \mid \bullet w; \alpha \rangle && \beta \text{ reduce.}
\end{aligned}$$

The main point of this is that, upon compilation into this lower-level language, comparing programs p and q becomes comparing configurations $\langle p \mid \alpha \rangle$ and $\langle q \mid \alpha \rangle$, and the effect of $E[-[\sigma]]$ in the source language may be achieved by the mere substitution

$$-[\sigma, \alpha \mapsto E].$$

Let us now give a bit more technical detail on the lower-level language.

Low-level types τ are either simple types A, or negated simple types $\neg A$. Programs have simple types A, while evaluation contexts have negated types $\neg A$. We have syntactic categories for programs and evaluation contexts, and a configuration is a pair of a program of some type A and of an evaluation context of type $\neg A$. Values and programs are defined and typed exactly as before. Evaluation contexts and configurations are specified by the following grammar.

$$\begin{aligned}
\text{values} \ni v, w &::= x \mid \{\} \mid \lambda^{\mathbf{rec}} f, x.\, p \\
\text{programs} \ni p, q &::= v \mid p\, q \\
\text{contexts} \ni \pi, \kappa &::= x \mid \bullet v; \pi \mid p\bullet; \pi \\
\text{configurations} \ni c, d &::= \langle\, p \mid \pi \,\rangle
\end{aligned}$$

The typing rules for values and programs are again exactly as before (except that typing contexts may now comprise evaluation context variables). Furthermore, the variable typing rule now covers the fact that any evaluation context variable $\alpha : \neg A$ is an evaluation context of type $\neg A$. Further typing rules, for evaluation contexts and configurations, are shown in the first part of Figure 1. Capture-avoiding substitution is defined straightforwardly, and evaluation rules are in the second part of Figure 1. We may now introduce substitution equivalence.

$$\frac{\Gamma \vdash v \colon A \qquad \Gamma \vdash \pi \colon \neg B}{\Gamma \vdash \bullet v; \pi \colon \neg(A \to B)} \qquad \frac{\Gamma \vdash p \colon A \to B \qquad \Gamma \vdash \pi \colon \neg B}{\Gamma \vdash p\bullet; \pi \colon \neg A} \qquad \frac{\Gamma \vdash p \colon A \qquad \Gamma \vdash \pi \colon \neg A}{\Gamma \vdash \langle\, p \mid \pi \,\rangle}$$

$$\begin{aligned}
\langle\, p\, q \mid \pi \,\rangle &\to \langle\, q \mid p\bullet; \pi \,\rangle \\
\langle\, v \mid p\bullet; \pi \,\rangle &\to \langle\, p \mid \bullet v; \pi \,\rangle \\
\langle\, \lambda^{\mathbf{rec}} f, x.\, p \mid \bullet v; \pi \,\rangle &\to \langle\, p[f \mapsto (\lambda^{\mathbf{rec}} f, x.\, p), x \mapsto v] \mid \pi \,\rangle
\end{aligned}$$

Fig. 1. Typing and evaluation rules for the lower-level variant of call-by-value λ-calculus

Definition 2.4. For any context Γ, two configurations $\Gamma \vdash c, d$ and d are *substitution equivalent*, which we denote by $c \approx_{\text{SUB}} d$, iff for all assignments $(\alpha \colon \neg\top) \vdash \sigma \colon \Gamma$, $c[\sigma]$ and $d[\sigma]$ *coterminate*, in the sense that $(c \to^* \langle\, \{\} \mid \alpha \,\rangle) \Leftrightarrow (d \to^* \langle\, \{\} \mid \alpha \,\rangle)$.

The main point of this section is:

Proposition 2.5. *For any context* Γ *and type* A *of the source language, two programs* $\Gamma \vdash p, q : A$ *are* ciu*-equivalent iff, in the typing context* $(\Gamma, \beta : \neg A)$ *(with* β *fresh w.r.t.* Γ*),* $\langle\, p \mid \beta \,\rangle$ *and* $\langle\, q \mid \beta \,\rangle$ *are substitution equivalent.*

Proof. There is a one-to-one correspondence between assignments $(\alpha : \neg\top) \vdash \sigma : (\Gamma, \beta : \neg A)$ in the lower-level language and pairs of an assignment $\varepsilon \vdash \gamma : \Gamma$ and an evaluation E context of type $\top$ with a hole of type A in the source language. Furthermore, for such assignments and context, $\langle\, p \mid \beta \,\rangle[\sigma]$ and $E[p[\gamma]]$ coterminate in the obvious sense, so result follows. □

3 A PRIMER ON OGS

In this section, we explain OGS on the example language introduced in the previous section.

The rough idea of OGS is that of a game between Proponent (P), and Opponent (O), each of which holds complementary program fragments, and follows a strategy dictated by evaluation. P and O both name their program fragments with variables, and moves in the game allow them to send variables, albeit in a constrained way, but no proper terms.

There are thus two kinds of positions in the game, in which one player is *active* and the other is *waiting*. Intuitively, the active player is evaluating some configuration, while the *waiting* one is storing their values and waiting to be solicited by the active player. When the active player, say A, gets stuck on a variable held by the waiting player, say W, then A sends a message to W, which becomes active.

Messages are determined by the normal forms hit by evaluation and proceed by splitting them into a head variable, an observation and an assignment. The message will consist of the head variable and the observation, while the assignment is kept private, only sharing its variables. Assuming the configuration is well typed, normal forms may take the following forms.

- $\langle\, x' \mid \bullet v ; \pi \,\rangle$ — Player A wants to compute the application $x'\, v$, but does not have access to the function x'; in this case they create fresh variables, say y and α, record the assignment $\{y \mapsto v, \alpha \mapsto \pi\}$, and send the message $(x', \langle\, \square \mid \bullet y ; \alpha \,\rangle)$;
- $\langle\, v \mid \alpha' \,\rangle$ — Player A has reached a value, but only W knows what to do with it next; in this case, A creates a fresh variable, say y, records the assignment $\{y \mapsto v\}$, and sends the message $(\alpha', \langle\, y \mid \square \,\rangle)$.

Notation 3.1. Although messages are really the pair of a head variable and an observation, we conflate them with their embedding into (normal) configuration. As such, the above messages $(x', \langle\, \square \mid \bullet y ; \alpha \,\rangle)$ and $(\alpha', \langle\, y \mid \square \,\rangle)$ are also written $\langle\, x' \mid \bullet y ; \alpha \,\rangle$ and $\langle\, y \mid \alpha' \,\rangle$, respectively. It may not be immediately obvious to the reader how to figure out which is the head variable in this notation: we use it when the context makes it clear which player is active, and which player generated which variable; in this case, the head variable in a message by some player is the single variable generated by the other player.

Furthermore, in order to clarify which player generated which variable, we tend to prime all variables generated by the (initially) waiting player.

Remark 3.2. A message like $x \cdot \langle\, \square \mid \bullet\, y ; \alpha \,\rangle$ is often noted in the literature on OGS as $x(y, \alpha)$, following a process-calculi notation, or as $call_x(y, \alpha)$. Such a message corresponds to a *question* of the channel x, providing the y as an argument and α as the return channel. A message like $\alpha \cdot \langle\, x \mid \square \,\rangle$ is then noted as $\alpha(x)$ or as $ret_\alpha(x)$, and corresponds to an *answer* of x via the channel α.

When the interaction is *well-bracketed*, as this is the case for languages without control operators, continuation names like α are often omitted. Here, we choose to work in the more general setting of possible non-well-bracketed interactions, so that we crucially use them.

Returning to our explanation of OGS, let us now explain how messages are received:

- upon receiving a message $\langle x' \mid \bullet y; \alpha \rangle$, player W looks up the value, say v', of x' in their store, and evaluates $\langle v' \mid \bullet y; \alpha \rangle$;
- upon receiving a message $\langle y \mid \alpha' \rangle$, player W looks up the value, say π', of α' in their store, and evaluates $\langle y \mid \pi' \rangle$.

Since our mechanisation is well-scoped, we record the *context increments* Θ incurred by each message type:

- for a message of the form $\langle x' \mid \bullet y; \alpha \rangle$, with $x' \colon A \to B$, the fresh variables are y and α, so the context increment is $\Theta_{A \to B} := (y \colon A, \alpha \colon \neg B)$;
- for a message of the form $\langle y \mid \alpha' \rangle$, with $\alpha' \colon \neg A$, the fresh variable is y, so the context increment is $\Theta_{\neg A} := (y \colon A)$.

In this particular toy language, as there is at most one kind of observation per type of the head variable, and since the fresh variable's names are uniquely determined in the well-scoped approach with De Bruijn indices, messages are entirely determined by the "called" variable, i.e., x' and α' above. Assuming variables are split into those, say Γ, of A, and those, say Δ, of W, we may define messages that A may send to be all $x' \colon \tau \in \Delta$ where $\tau \neq \top$. Indeed:

- if τ is a non-negated type, then it must have the shape $A \to B$, and the message is $\langle x' \mid \bullet y; \alpha \rangle$ in the extended context pair $(\Gamma + \Theta_{A \to B}, \Delta)$;
- if τ is a negated type $\neg A$, then the message is $\langle y \mid x' \rangle$ in the extended context pair $(\Gamma + \Theta_{\neg A}, \Delta)$.

Notation 3.3. We denote by Θ_x the context increment Θ_τ corresponding to the type of the variable $x \colon \tau \in \Delta$.

From there, we may define the positions and moves of the game as follows.

Definition 3.4. A position is a pair of typing contexts, together with a boolean tag indicating activity. We denote active positions by $(\Gamma, \Delta)^+$, and passive ones by $(\Gamma, \Delta)^-$.

A move from an active position $(\Gamma, \Delta)^+$ is some $x' \colon \tau \in \Delta$ with $\tau \neq \top$; its target position is $(\Gamma + \Theta_{x'}, \Delta)^-$.

A move from a passive position $(\Gamma, \Delta)^-$ is the same with roles switched, i.e., some $x \colon \tau \in \Gamma$ with $\tau \neq \top$; its target position is $(\Gamma, \Delta + \Theta_x)^+$.

Remark 3.5. Each tagged pair $(\Gamma, \Delta)^+$ is only the position of one player, say P, at some point in the game. Variables in Γ are those generated by P. The position being active, moves are calls to variables in Δ. The position held by the other player at the same time is $(\Delta, \Gamma)^-$.

Example 3.6. Let us consider the position formed by the typing contexts $\Gamma = (\kappa : \mathbb{B})$ and $\Delta = (\alpha : (\mathbb{B} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{B}))$ with $\mathbb{B}$ the type of booleans $\mathbf{tt}, \mathbf{ff}$. The configuration $\langle \lambda x.x \mid \alpha \rangle$ formed by the identity function, in the position $(\Gamma, \Delta)^+$, may interact with the assignment $[\alpha \mapsto \langle \square \mid \bullet\, \mathbf{tt}; \bullet \mathbf{not}; \kappa \rangle]$, that is taken in the dual position $(\Delta, \Gamma)^+$, with **not** the negation operator of type $\mathbb{B} \to \mathbb{B}$. The interaction would produce the trace formed by the following sequence of moves:

$$\alpha \cdot \langle x_1 \mid \square \rangle \quad x_1 \cdot \langle \square \mid \bullet\, y_1; \alpha_2 \rangle \quad \alpha_2 \cdot \langle x_2 \mid \square \rangle \quad x_2 \cdot \langle \square \mid \bullet\, \mathbf{tt}; \alpha_3 \rangle$$
$$y_1 \cdot \langle \square \mid \bullet\, \mathbf{tt}; \beta_1 \rangle \quad \beta_1 \cdot \langle \mathbf{tt} \mid \square \rangle \quad \alpha_3 \cdot \langle \mathbf{ff} \mid \square \rangle \quad \kappa \cdot \langle \mathbf{ff} \mid \square \rangle$$

We then define strategies coinductively:

Definition 3.7. A strategy from some active position $(\Gamma, \Delta)^+$ consists either of a formal, silent move to the same position $(\Gamma, \Delta)^+$, or of a move m from it, together with a *residual* strategy from the target position of m.

A strategy from some passive position $(\Gamma, \Delta)^-$ is a map associating to each move m from it a *residual* strategy from the target of m.

The basic result that makes an OGS useful, and which we prove below in the abstract setting, is:

Theorem 3.8. *If two strategies are* weakly bisimilar*, then they are substitution equivalent.*

Here, weak bisimilarity between two strategies is also defined coinductively in the straightforward way:

Definition 3.9. Two strategies S and T from some active position are weakly bisimilar iff

- if S, after finitely many silent moves, plays some move m, then T also plays m after finitely many silent moves, and furthermore the residual strategies are again weakly bisimilar,
- and conversely.

Two strategies S and T from some passive position are weakly bisimilar iff, for each move, the residual strategies of S and T are weakly bisimilar.

Our task is now to carve out from this concrete example (and the few others we have in mind) what makes it work.

4 ABSTRACT MACHINES, ABSTRACTLY

In order to get to the core of the OGS method and its soundness proof, we start by introducing some abstractions decoupling us from the syntactic details of a particular language. Building upon this basis we will in later sections provide a high-level OGS construction (§5) and its soundness proof (§6). Following standard practice in dependently-typed programming we adopt an intrinsically typed, well scoped presentation of all syntactic constructions [3, 9]. In the remainder of this section, we define contexts and families (§4.1), introduce successively evaluators (§4.2), substitution structures on families (§4.3), and observations (§4.4), and finally introduce language machines and define substitution equivalence (§4.5).

We fix a set T of *types* for the whole section.

4.1 Contexts, families and variables

In intrinsically-typed settings, a context is simply a list of types. To match common practice, we append to the right. Variables are given by typed de Bruijn indices $i\colon \tau \in \Gamma$. Formally, contexts are defined inductively by the following two inference rules (),

$$\frac{}{\emptyset\colon \mathtt{Ctx}\ T} \qquad \frac{\Gamma\colon \mathtt{Ctx}\ T \qquad \tau : T}{\Gamma, \tau\colon \mathtt{Ctx}\ T}$$

while de Bruijn indices () are defined by the following ones.

$$\frac{}{\mathtt{top}\colon \tau \in \Gamma, \tau} \qquad \frac{i\colon \tau \in \Gamma}{\mathtt{pop}\ i\colon \tau \in \Gamma, \sigma}$$

Definition 4.1 ([9],). A *family* is a $\mathtt{Ctx}\ T$-indexed set, i.e., a map $\mathtt{Ctx}\ T \to \mathtt{Set}$.
A *sorted family* is a map of type $T \to \mathtt{Ctx}\ T \to \mathtt{Set}$.

For a sorted family $\mathcal{X}$, an element $u\colon \mathcal{X}\ \tau\ \Gamma$ may be thought of as an $\mathcal{X}$-term of type τ in context Γ.

Example 4.2. de Bruijn indices—i.e., variables—form a sorted family and so do values. On the other hand, configurations form a family.

Definition 4.3. A morphism $f\colon \mathcal{X} \rightarrow\!\!\!\triangleright\ \mathcal{Y}$ of families consists of a map $f\colon \forall \Gamma\colon \mathtt{Ctx}\ T,\ \mathcal{X}\ \Gamma \to \mathcal{Y}\ \Gamma$.
A morphism $f\colon \mathcal{X} \rightarrow\!\!\!\triangleright\ \mathcal{Y}$ of sorted families consists of a map $f\colon \forall \tau\colon T,\ \Gamma\colon \mathtt{Ctx}\ T,\ \mathcal{X}\ \tau\ \Gamma \to \mathcal{Y}\ \tau\ \Gamma$.

Proposition 4.4. *Families (resp. sorted families) and morphisms between them form a category* $\mathtt{Fam}\ T$ *(resp.* $\mathtt{Fam}_{\mathtt{s}}\ T$*).*

4.2 Evaluators

In this subsection, we introduce abstract evaluators. For our purposes, an evaluator should be some sort of abstract machine for our language. As in the introductory example (Fig. 1) the typing judgment of the configurations of this machine only mentions a typing context. For this reason they are represented by an *unsorted* family.

We wish to handle languages with non-termination, and it is the only effect we consider in this paper. Therefore it is natural to look at monadic evaluators in the delay monad [5], which we now recall.

Definition 4.5 (🐓). Let $\mathscr{D}\, X = \nu A.\, (X + A)$, where ν denotes the coinductive type former. Thus, an element of $\mathscr{D}\, X$ is either $\texttt{ret}\ x$ for some $x \colon X$, or $\tau.a$ for some $a \colon \mathscr{D}\, X$, coinductively.

Notation 4.6. We write interchangeably $(x \leftarrow u; v\ x)$ and $u \gg\!= v$ for the *bind* (🐓) operator of $\mathscr{D}$—evaluate u, and if it returns some x, then evaluate $v\ x$. We also denote by $\mathcal{D}$ the pointwise lifting of the delay monad to categories of families over some fixed set, typically $\texttt{Fam}\ T$ and $\texttt{Fam}_{\texttt{s}}\ T$. We write $\approx_{\mathscr{D}}$ for weak bisimilarity (🐓), i.e., either both sides loop or both terminate on the same element $x \colon X$. We write $\cong_{\mathscr{D}}$ for strong bisimilarity (🐓), i.e., either both sides loop or both terminate on the same element in the same number of τ steps. Finally, when R is a relation, we write $\mathscr{D}\ R$ for the lifting of the delay monad to the category of relations, relaxing the definition of weak bisimilarity and relating elements in $\mathscr{D}$ that both loop or both terminate on elements related by R.

In addition to the evaluator eventually computing a normal form, we postulate an embedding of normal forms into configurations, such that evaluating a normal form n returns n.

Definition 4.7 (🐓). An *evaluation structure on* $C \colon \texttt{Fam}\ T$ and $\mathcal{N} \colon \texttt{Fam}\ T$ is given by maps:

$$\texttt{eval} \colon C \twoheadrightarrow \mathscr{D}\, \mathcal{N} \qquad\qquad \texttt{emb} \colon \mathcal{N} \twoheadrightarrow C$$

such that $\texttt{eval} \circ \texttt{emb} \approx_{\mathscr{D}} \texttt{ret}$.

Example 4.8 ($\lambda^{\textbf{rec}}$, 🐓). $\texttt{eval}$ simply captures the evaluation rules described in Section 1, but turns their relational description into an executable big-step interpreter in the delay monad.

In the above definition C and $\mathcal{N}$ should be thought of respectively as the families of configurations and normal forms. Both will be further described in the following subsections.

4.3 Substitution

Let us now flesh out the role of variables. The idea is that, through indexing, variables occur in configurations, and may be substituted with some *values*. We explain in this section what it means for a sorted and unsorted family to have a substitution operation. As is standard in the well-scoped approach, we mean substitution in the parallel sense. The basic ingredient for this is the the notion of context map from Fiore and Szamozvancev [9], which we call *assignments*.

Definition 4.9 (🐓). For any contexts $\Gamma, \Delta \colon \texttt{Ctx}\ T$ and sorted family $\mathcal{X}$, an $\mathcal{X}$-*assignment* $\sigma \colon \Gamma \dashv\!\![\mathcal{X}]\!\!\mapsto \Delta$, or *assignment* when $\mathcal{X}$ is clear from context, consists of an element of $\mathcal{X}\ \tau\ \Delta$, for all $x : \tau \in \Gamma$, i.e., we have

$$- \dashv\!\![-]\!\!\mapsto - \colon \texttt{Ctx}\ T \to \texttt{Fam}_{\texttt{s}}\ T \to \texttt{Ctx}\ T \to \texttt{Set}$$
$$\Gamma \dashv\!\![\mathcal{X}]\!\!\mapsto \Delta \coloneqq \forall \tau \colon T,\ \tau \in \Gamma \to \mathcal{X}\ \tau\ \Delta.$$

Using assignments we can readily define a candidate type for substitution:

$$\texttt{sub} \colon \forall \tau \colon T, \Gamma, \Delta \colon \texttt{Ctx}\ T,\ \mathcal{X}\ \tau\ \Gamma \to (\Gamma \dashv\!\![\mathcal{X}]\!\!\mapsto \Delta) \to \mathcal{X}\ \tau\ \Delta$$

This type can be more succintly expressed as $\mathcal{X} \twoheadrightarrow [\![\mathcal{X}, \mathcal{X}]\!]_s$ using the following *internal substitution hom* of sorted families:

$$[\![-,-]\!]_s : \mathtt{Fam_s}\ T \longrightarrow \mathtt{Fam_s}\ T \longrightarrow \mathtt{Fam_s}\ T$$
$$[\![\mathcal{X}, \mathcal{Y}]\!]_s\ \tau\ \Gamma = \forall \Delta : \mathtt{Ctx}\ T,\ (\Gamma \dashv\!\![\mathcal{X}]\!\!\mapsto \Delta) \longrightarrow \mathcal{Y}\ \tau\ \Delta$$

Definition 4.10 (🐓). A *substitution monoid* $\mathcal{X} : \mathtt{Mon}\ T$ is a sorted family $\mathcal{X} : \mathtt{Fam_s}\ T$ with maps:

$$\mathtt{var} : -\!\in\!- \twoheadrightarrow \mathcal{X} \qquad\qquad \mathtt{sub} : \mathcal{X} \twoheadrightarrow [\![\mathcal{X}, \mathcal{X}]\!]_s$$

subject to associativity and unitality laws.

Remark 4.11. Each functor $[\![\mathcal{Y}, -]\!]$ has a left adjoint $- \odot_s \mathcal{Y}$, and since maps $\mathcal{X} \twoheadrightarrow [\![\mathcal{Y}, \mathcal{Z}]\!]_s$ are naturally isomorphic to maps $\mathcal{X} \odot_s \mathcal{Y} \twoheadrightarrow \mathcal{Z}$, we recognise the internal-hom presentation of monoids. Categorically, a substitution monoid amounts to a monoid object in the skew-monoidal category $(\mathtt{Fam_s}\ T, -\!\in\!-, -\!\odot_s\!-)$, see [9].

We now know what it means for a sorted family—for example the *values* of our language—to have a substitution: to be a monoid. To extend this notion to unsorted families, like configurations, we want to consider a slightly different notion of substitution, where the object of the substitution and the replacements are not of the same family, and do not even live in the same category. In particular, we wish to model the substitution of all the variables of a configuration by values. To do this we can realise that our internal substitution hom actually arises as the pointwise lifting of an other "exponentiation", this time of an unsorted family:

$$[\![-,-]\!] : \mathtt{Fam_s}\ T \longrightarrow \mathtt{Fam}\ T \longrightarrow \mathtt{Fam}\ T$$
$$[\![\mathcal{X}, \mathcal{Y}]\!]\ \Gamma \coloneqq \forall \Delta : \mathtt{Ctx}\ T,\ (\Gamma \dashv\!\![\mathcal{X}]\!\!\mapsto \Delta) \longrightarrow \mathcal{Y}\ \Delta$$

Our substitution of variables in configurations C by values $\mathcal{V}$ may now be typed as $C \twoheadrightarrow [\![\mathcal{V}, C]\!]$.

Definition 4.12 (🐓). Given a substitution monoid $\mathcal{M} : \mathtt{Mon}\ T$, a *substitution $\mathcal{M}$-module* $\mathcal{X} : \mathcal{M}\text{-}\mathtt{Mod}$ is a family $\mathcal{X} : \mathtt{Fam}\ T$ equipped with a map:

$$\mathtt{sub} : \mathcal{X} \twoheadrightarrow [\![\mathcal{M}, \mathcal{X}]\!]$$

subject to associativity and unitality laws.

Remark 4.13. Each functor $[\![\mathcal{V}, -]\!]$ also has a left adjoint $- \odot \mathcal{V}$ and much like the substitution of sorted families is a monoid in disguise, substitution of families is a monoidal action in disguise.

Categorically, a substitution module is a form of module over a monoid. Indeed $(\mathtt{Fam}\ T, -\!\odot\!-)$ constructs a skew right-module category over the skew monoidal category $(\mathtt{Fam_s}\ T, -\!\in\!-, -\!\odot_s\!-)$. Such a module category is sometimes called an *actegory* when not skew. In this setting, a substitution module is a skew right-module object in $(\mathtt{Fam}\ T, -\!\odot\!-)$.

4.4 Observations

We now tackle perhaps the most important for the OGS construction: the messages that Proponent and Opponent will exchange. As informally explained in §3, these messages arise from splitting normal forms into their outer, "inert" part, and their inner part to be hidden. More precisely, we postulate that any normal configuration decomposes into three parts: the *head variable* on which it is stuck, an *observation*, and a collection of values *filling* the observation, i.e., an assignment out of the *domain* or holes of the observation.

Variables and assignments being already described, it remains to axiomatise observations. Once this is done, we will axiomatise evaluators as evaluation structure with suitable substitution, in which normal forms are given by suitably typed triples of a variable, an observation, and an

assignment. Our axiomatisation of observations relies on the following notion of an observation structure.

Definition 4.14 (). An *observation structure* $\mathcal{O}$ is a type-indexed set $\mathcal{O} \colon T \to \texttt{Set}$ together with a map:

$$\texttt{dom} : \forall \tau \colon T,\ \mathcal{O}\ \tau \to \texttt{Ctx}\ T$$

We denote by $\texttt{Obs}\ T$ the set of observation structures on T.

Remark 4.15. Observations occur in the literature under various names [15, 18, 20, 26].

Example 4.16 (λ^{rec},). We recall elements of §3 in light of these abstract definition. Since all types are negative in λ^{rec}, we simply take as set of types T all $\tau : A \mid \neg A$. There is a single observation at each type, whose domain captures the associated context increment. At a negated type $\neg A$, a fresh variable at type A intended to carry the returned value is introduced. At a non-negated types, i.e., at an arrow type $A \to B$, the domain carries fresh variables at $\neg B$ for the context and at A for the value.

Remark 4.17. Observation structures could be presented as a sorted family. We choose the above equivalent definition presenting them as binders for convenience in the mechanisation.

Let us now define compatible pairs of a head variable and an observation:

Definition 4.18 (). Given an observation structure $\mathcal{O} \colon \texttt{Obs}\ T$, the family $\mathcal{O}^\bullet$ of *pointed observations* is defined by

$$\mathcal{O}^\bullet\ \Gamma \coloneqq \exists \tau \colon T,\ (\tau \in \Gamma) \times \mathcal{O}\ \tau.$$

Its *domain* map is defined by:

$$\texttt{dom}^\bullet \colon \forall \Gamma, \mathcal{O}^\bullet\ \Gamma \to \texttt{Ctx}\ T$$
$$\texttt{dom}^\bullet\ (\tau, (i, o)) \coloneqq \texttt{dom}\ o.$$

It remains to incorporate the filling assignment:

Definition 4.19 (). Given any observation structure $\mathcal{O} \colon \texttt{Obs}\ T$, sorted family of *values* $\mathcal{V} \colon \texttt{Fam}_{\texttt{s}}\ T$, we define a family of *normal forms* $\mathcal{N}_{\mathcal{O},\mathcal{V}}$ consisting of pairs of a pointed observation and an assignement filling its domain with $\mathcal{V}$-terms.

$$\mathcal{N}_{\mathcal{O},\mathcal{V}}\ \Gamma \coloneqq \exists o \colon \mathcal{O}^\bullet\ \Gamma, \texttt{dom}^\bullet\ o \mathrel{-\!\![\mathcal{V}]\!\!\rightarrow} \Gamma.$$

Notation 4.20. We denote normal forms $((x, o), \gamma) \colon \mathcal{N}_{\mathcal{O},\mathcal{V}}\ \Gamma$ by $x \cdot o(\gamma)$, thinking of o as a postfix projection, or method call with principal x and arguments γ. E.g., $p \cdot \texttt{fst}()$, $f \cdot \texttt{app}(x)$ or $k \cdot \texttt{ret}(\texttt{true})$.

4.5 Abstract evaluators and substitution equivalence

We are at last in a position to bring everything together and introduce our abstract notion of language with evaluator, called *language machine.*

Definition 4.21. A *language machine* over any set T consists of

- a substitution monoid $\mathcal{V}_\mathcal{M}$ of *values,*
- a substitution $\mathcal{V}_\mathcal{M}$-module $C_\mathcal{M}$ of *configurations,*
- an observation structure $\mathcal{O}_\mathcal{M}$, and
- an evaluation structure $(\texttt{eval}_\mathcal{M}, \texttt{emb}_\mathcal{M})$ on $C_\mathcal{M}$ and $\mathcal{N}_{\mathcal{O}_\mathcal{M},\mathcal{V}_\mathcal{M}}$.

Let $\texttt{Machine}\ T$ denote the set of language machines over T.

Notation 4.22. For any language machine $\mathcal{M}$:

- We use the shorthand $\mathcal{N}_{\mathcal{M}}$ instead of $\mathcal{N}_{O_{\mathcal{M}},\mathcal{V}_{\mathcal{M}}}$
- We denote both substitution structures, on values and configurations, by $-[-]$, relying on context to disambiguate.
- We write $\Gamma \rightarrow_{\mathcal{M}} \Delta$ for $\Gamma \dashv\![\mathcal{V}_{\mathcal{M}}]\!\mapsto \Delta$.
- We extend the notation for substitution from values to assignments: for any composable assignments $\Gamma \xrightarrow{\gamma}_{\mathcal{M}} \Delta \xrightarrow{\sigma}_{\mathcal{M}} \Theta$, we denote by $\gamma[\sigma]$ the assignment mapping any $x\colon \tau \in \Gamma$ to $(\gamma\, x)[\sigma]$.
- We often consider $\mathtt{emb}_{\mathcal{M}}$ as an implicit coercion, hence confuse normal forms $x \cdot o(\gamma)$ with the corresponding configurations.
- We extend the notation $x \cdot o(\gamma)$ to $v \cdot o(\gamma)$, for any suitable value v: using the previous point, this denotes the configuration $(x \cdot o(\gamma))[x \mapsto v]$, for some fresh x.

Finally, subscripts are often omitted when $\mathcal{M}$ is clear from context.

Let us now define *substitution equivalence*, our variant of CIU equivalence. For this, we need to fix the way we observe "closed" configurations. In fact, in applications, configurations are never actually closed, because of our presentation style. Typically, in the language of §2, a "closed" configuration of any type A in fact has a free continuation variable $\alpha : \neg A$. We define observation on such configurations to be the projection of their evaluation to pointed observations:

Definition 4.23 (🐓). Let $\mathtt{eval}^{\circ}_{\mathcal{M}} \colon C_{\mathcal{M}} \rightarrow O^{\bullet}_{\mathcal{M}}$ be defined by $\mathtt{eval}^{\circ}_{\mathcal{M}} \coloneqq \mathscr{D}\, \pi_1 \circ \mathtt{eval}_{\mathcal{M}}$.

Using this, let us at last define substitution equivalence.

Definition 4.24 (🐓). Given a language machine $\mathcal{M}\colon \mathtt{Machine}\ T$ and contexts $\Gamma, \Delta\colon \mathtt{Ctx}\ T$, two configurations $u, w\colon C_{\mathcal{M}}\ \Gamma$ are *substitution equivalent at* Δ, written $u \approx_{\text{SUB}} w$ iff:

$$\forall \gamma\colon \Gamma \rightarrow_{\mathcal{M}} \Delta,\ \mathtt{eval}^{\circ}\ (u[\gamma]) \approx_{\mathscr{D}} \mathtt{eval}^{\circ}\ (w[\gamma])$$

Remark 4.25. There are two peculiarities with our *substitution equivalence*. First the substitutions we are quantifying over are not *closing*, but rather target an externally quantified context Δ; second we do not compare normal forms, but only their projection to pointed observations, i.e., the head variable and the observation ($\mathtt{eval}^{\circ}$ vs $\mathtt{eval}$).

These two aspects are linked. Indeed in more standard CIU or contextual equivalence, one must choose a particular observation type τ, usually chosen as the type of booleans, or the unit type when it is sufficient which in our setting would amount to setting $\Delta = \emptyset, \neg\tau$. These *final types* are usually positive types, so that observations at $\neg\tau$ do not bind variables: they have empty domain. As such, in the usual case of positive final types, normal forms are equivalent to pointed observations. Instead of introducing all these restrictions and somewhat arbitrary choices, we rather parameterize on any collection of final types Δ, and project to pointed observations. This way, we uphold the principle of never syntactically comparing anything but observation.

To conclude this section we introduce the core hypothesis on a language machine, relating evaluation and substitution, which is crucial for the soundness proof of §6.

Definition 4.26 (🐓). In a machine $\mathcal{M}\colon \mathtt{Machine}\ T$, *evaluation respects substitution* iff the following two conditions hold.

- For all contexts Γ, configurations $u\colon C\ \Gamma$, and assignments $\gamma\colon \Gamma \rightarrow_{\mathcal{M}} \Delta$, we have

$$\mathtt{eval}\ (u[\gamma]) \approx_{\mathscr{D}} \left(\begin{array}{l} n \leftarrow \mathtt{eval}\ u\ ; \\ \mathtt{eval}\ ((\mathtt{emb}\ n)[\gamma]) \end{array} \right).$$

- For all types τ, contexts Γ, Δ, observations $o\colon \mathcal{O}\ \tau$ and assignments $\gamma\colon \mathtt{dom}\ o \to_{\mathcal{M}} \Gamma$ and $\delta\colon \Gamma \to_{\mathcal{M}} \Delta$,

$$(\mathtt{emb}\ (x, o, \gamma))[\delta, x \mapsto x] = \mathtt{emb}\ (x, o, \gamma[\delta])$$

where $x\colon \tau$ is fresh w.r.t. Γ and Δ.

Remark 4.27. The first equation of the above definition can be thought of as saying that for a configuration, "substituting then evaluating" is equivalent to "evaluating then substituting and evaluating". The second equation can be thought of as saying that for a normal form, "embedding then substituting *while keeping the head variable fixed*" is equivalent to "substituting then embedding".

Remark 4.28. Let us mention in passing the following categorical characterisation of substitution-respecting evaluation. The starting point is that normal forms feature a candidate $\mathcal{V}$-module structure in the Kleisli category $(\mathtt{Fam}\ T)_{\mathcal{D}}$ of the delay monad:

$$-[-]\colon \mathcal{N} \rightharpoonup [\![\mathcal{V}, \mathcal{D}\ \mathcal{N}]\!]$$
$$n[\sigma] \coloneqq \mathtt{eval}\ ((\mathtt{emb}\ n)[\sigma]).$$

Evaluation respects substitution iff the above is indeed a module structure, and if emb and eval are both *module morphism*, between this module structure and the lifting of the substitution module structure of C from $\mathtt{Fam}\,T$ to $(\mathtt{Fam}\,T)_{\mathcal{D}}$. Because of the particular expression of normal forms, all these facts reduce to the two simple equations of Def. 4.26, explaining how emb and eval commute with substitutions.

5 OPERATIONAL GAME SEMANTICS IN THE ABSTRACT

We now turn to the construction of the OGS, for any language machine, in the sense of Definition 4.21. First, in §5.1, we recall a general notion of game due to Levy and Staton [21]. We then introduce the OGS game (§5.2), together with the *machine strategy* arising from a language machine (§5.3).

5.1 Games and strategies

Levy and Staton's notion of game is parameterised by sets I and J of *active* and *passive positions*, respectively. The definition then proceeds to postulate families of active moves from I to J, and passive moves from J to I. As this is symmetric, we start by introducing a notion of "half-games", and then define games as pairs thereof.

Definition 5.1 (🐓). A *half-game* $\mathcal{H}$ over sets I and J consists of

- an I-indexed family of *moves* $\mathtt{move}_{\mathcal{H}}\colon I \to \mathtt{Set}$, and
- a *next* map $\mathtt{next}_{\mathcal{H}}\colon \forall i\colon I,\ \mathtt{move}_{\mathcal{H}}\ i \to J$.

We denote by $\mathtt{HGame}\ I\ J$ the set of half-games over I and J.

Remark 5.2. Concretely, in any active position $i\colon I$, $\mathtt{next}_{\mathcal{H}}\ i\ m$ returns the position reached after playing some move $m\colon \mathtt{move}_{\mathcal{H}}\ i$.

Notation 5.3. We generally omit the position in $\mathtt{next}_{\mathcal{H}}\ i\ m$, merely writing $\mathtt{next}_{\mathcal{H}}\ m$.

Let us now define games.

Definition 5.4 (🐓). A *game* $\mathcal{G}$ over sets I and J consists of

- a *client* half-game $\mathtt{client}_{\mathcal{G}}\colon \mathtt{HGame}\ I\ J$,
- a *server* half-game $\mathtt{server}_{\mathcal{G}}\colon \mathtt{HGame}\ J\ I$, and
- a set of *final moves* $\mathtt{final}_{\mathcal{G}}\colon \mathtt{Set}$.

We denote by $\mathtt{Game}\ I\ J$ the set of games over I and J.

Remark 5.5. The idea is that the moves of $\mathtt{client}_{\mathcal{G}}$ and $\mathtt{server}_{\mathcal{G}}$ are played between P and O, while the final moves are played "to the outside".

Notation 5.6. Given any game $\mathcal{G}\colon \mathtt{Game}\ I\ J$, we write $\mathtt{move}^{P}_{\mathcal{G}}$ for client moves and $\mathtt{move}^{O}_{\mathcal{G}}$ for server moves, overloading $\mathtt{next}_{\mathcal{G}}$ for both.

Warming up, we can define the dual of a game by swapping its two half-games:

Definition 5.7. The dual $\mathcal{G}^{\perp}\colon \mathtt{Game}\ J\ I$ of a game $\mathcal{G}\colon \mathtt{Game}\ I\ J$ is defined by swapping the client and server: $\mathtt{client}_{\mathcal{G}^{\perp}} \coloneqq \mathtt{server}_{\mathcal{G}}$, $\mathtt{server}_{\mathcal{G}^{\perp}} \coloneqq \mathtt{client}_{\mathcal{G}}$ and $\mathtt{final}_{\mathcal{G}^{\perp}} \coloneqq \mathtt{final}_{\mathcal{G}}$.

Now that games are defined, we turn to defining strategies in a game $\mathcal{G}\colon \mathtt{Game}\ I\ J$. The idea is that a strategy for P on $\mathcal{G}$ consists of

- families $A\colon I \to \mathtt{Set}$ and $P\colon J \to \mathtt{Set}$ of active and passive states, respectively,
- a partial "action" map, which for any active state $a\colon A\ i$ associates—when it is defined—either a final move in $\mathtt{final}_{\mathcal{G}}$, or a next move $m\colon \mathtt{move}^{P}_{\mathcal{G}}\ i$ in the client half-game of $\mathcal{G}$, together with a next passive state in $P\ (\mathtt{next}_{\mathcal{G}}\ m)$, and
- a "reaction" map, which to any passive state $p\colon P\ j$ and move $m\colon \mathtt{move}^{O}_{\mathcal{G}}\ j$ in the server half-game of $\mathcal{G}$ associates a next active state in $A\ (\mathtt{next}_{\mathcal{G}}\ m)$.

A strategy for Opponent on $\mathcal{G}$, or counter-strategy, is dual to this: it is simply a strategy on $\mathcal{G}^{\perp}$.

Let us now formalise all this in coalgebraic style, for which we need to define the following functors.

Definition 5.8 (🐓). Given any half-game $\mathcal{H}\ \colon \mathtt{HGame}\ I\ J$ we define two functors $(J \to \mathtt{Set}) \to (I \to \mathtt{Set})$, the *action* functor $[\![\mathcal{H}]\!]^{+}$ and the *reaction* functor $[\![\mathcal{H}]\!]^{-}$:

$$[\![\mathcal{H}]\!]^{+}\ \mathcal{X}\ i \coloneqq \exists m\colon \mathtt{move}_{\mathcal{H}}\ i,\ \mathcal{X}\ (\mathtt{next}_{\mathcal{H}}\ m)$$
$$[\![\mathcal{H}]\!]^{-}\ \mathcal{X}\ i \coloneqq \forall m\colon \mathtt{move}_{\mathcal{H}}\ i,\ \mathcal{X}\ (\mathtt{next}_{\mathcal{H}}\ m).$$

We may now define strategies.

Definition 5.9. Given a game $\mathcal{G}\colon \mathtt{Game}\ I\ J$, a *strategy* $\mathcal{S}$ for $\mathcal{G}$ consists of

- a family $\mathcal{S}^{+}\colon I \to \mathtt{Set}$ of *active* states,
- a family $\mathcal{S}^{-}\colon J \to \mathtt{Set}$ of *passive* states,
- an *action* morphism

$$\mathtt{play}_{\mathcal{S}}\colon \mathcal{S}^{+} \to \mathscr{D}\ (\mathtt{final}_{\mathcal{G}} + [\![\mathtt{client}_{\mathcal{G}}]\!]^{+}\ \mathcal{S}^{-}),$$

- and a *reaction* morphism

$$\mathtt{coplay}_{\mathcal{S}}\colon \mathcal{S}^{-} \to [\![\mathtt{server}_{\mathcal{G}}]\!]^{-}\ \mathcal{S}^{+}.$$

We denote by $\mathcal{S}\colon \mathtt{Strat}_{\mathcal{G}}$ the set of strategies for $\mathcal{G}$.

Remark 5.10. The occurrence of the delay monad in the action morphism means that we allow P to "think forever" and never actually play. This is crucial for interpreting languages with general recursion.

The main point of OGS consists in interpreting configurations as strategies in some game, in the hope that weak bisimilarity between configurations entails substitution equivalence. Let us define weak bisimilarity.

Definition 5.11. Given two strategies $\mathcal{S}, \mathcal{T}\colon \mathtt{Strat}_{\mathcal{G}}$, a *bisimulation* $\alpha\colon \mathcal{S} \approx_{\mathcal{G}} \mathcal{T}$ consists of

- an I-indexed relation $\alpha^+ : \mathcal{S}^+\, i \to \mathcal{T}^+\, i \to \mathtt{Prop}$ between active states,
- a J-indexed relation $\alpha^- : \mathcal{S}^-\, j \to \mathcal{T}^-\, j \to \mathtt{Prop}$ between passive states,

together with the following conditions:

$$\begin{cases} \mathtt{play}_\alpha : \mathtt{play}_\mathcal{S} \approx\!\![\alpha^+ \to \mathscr{D}(\mathtt{Eq}_\mathcal{R} + [\![\mathtt{client}_\mathcal{G}]\!]^+\, \alpha^-)]\!\!\approx \mathtt{play}_\mathcal{T} \\ \mathtt{coplay}_\alpha : \mathtt{coplay}_\mathcal{S} \approx\!\![\alpha^- \to [\![\mathtt{server}_\mathcal{G}]\!]^-\, \alpha^+]\!\!\approx \mathtt{coplay}_\mathcal{T} \end{cases}$$

where we write $u \approx\!\![R \to S]\!\!\approx v = \forall i\ x\ y,\ R\ i\ x\ y \to S\ i\ (u\ x)\ (v\ y)$ and lift functors to relations in the straightforward way. As is usual, bisimilarity is defined as the largest bisimulation.

5.2 The OGS game

In this subsection, we define the OGS game corresponding to any language machine. For §5.2–5.3, we fix a set T of types and a language machine $\mathcal{M} = (\mathcal{V}, \mathcal{C}, \mathcal{O}, \mathtt{eval}, \mathtt{emb})$, dropping the subscript.

To start with, final moves in the OGS game are pointed observations in some fixed context $\Delta : \mathtt{Ctx}\ T$, which we fix for §5.2–5.3. A typical such context is the singleton $\alpha : \neg\mathbb{B}$, so that a pointed observation on Δ is either $(\alpha, \mathtt{true})$ or $(\alpha, \mathtt{false})$.

As hinted in the primer (§3), moves in the OGS game consist of pointed observations, each of which introduces fresh shared variables. Accordingly, positions of the OGS game consist of lists $(\Gamma_1, \ldots, \Gamma_n)$ of contexts, each Γ_i modelling the variables introduced by the ith move (intuitively in some abstract play leading up to the current position).

Definition 5.12 (🐓). An *interleaved context* is a list of contexts. We denote by $\mathtt{ICtx}\ T = \mathtt{Ctx}\ (\mathtt{Ctx}\ T)$ the set of interleaved contexts.

Of course, we may extract from any interleaved context the variables introduced by P, and those introduced by O:

Definition 5.13 (🐓). We define two *collapsing* functions $\downarrow^+, \downarrow^- :\ \mathtt{ICtx}\ T \to \mathtt{Ctx}\ T$ as follows:

$$\begin{array}{lcl} \downarrow^+\emptyset & := & \emptyset \\ \downarrow^+(\Phi, \Gamma) & := & \downarrow^-\Phi + \Gamma \end{array} \qquad \begin{array}{lcl} \downarrow^-\emptyset & := & \emptyset \\ \downarrow^-(\Phi, \Gamma) & := & \downarrow^+\Phi. \end{array}$$

Remark 5.14. Intuitively, $\downarrow^+$ retains from an interleaved context the variables that are unknown to the currently active player, starting with the last introduced context. Symmetrically, $\downarrow^-$ retains those that are unknown to the passive player, which does not include the last introduced context. Concretely, we have:

$$\begin{array}{lcl} \downarrow^+(\Gamma_1, \ldots, \Gamma_{2n}) & = & \Gamma_2 + \Gamma_4 + \ldots + \Gamma_{2n} \\ \downarrow^+(\Gamma_1, \ldots, \Gamma_{2n+1}) & = & \Gamma_1 + \Gamma_3 + \ldots + \Gamma_{2n+1} \\ \downarrow^-(\Gamma_1, \ldots, \Gamma_{2n}) & = & \Gamma_1 + \Gamma_3 + \ldots + \Gamma_{2n-1} \\ \downarrow^-(\Gamma_1, \ldots, \Gamma_{2n+1}) & = & \Gamma_2 + \Gamma_4 + \ldots + \Gamma_{2n}. \end{array}$$

Let us now define the OGS game, as expected. This in fact only depends on the fixed context Δ and the observation structure of the considered language machine.

Definition 5.15 (🐓). The *OGS half-game* HOGS of type

$$\mathtt{HOGS} : \mathtt{HGame}\ (\mathtt{ICtx}\ T)\ (\mathtt{ICtx}\ T)$$

$$\mathtt{move}_\mathtt{HOGS}\ \Phi := \mathcal{O}^\bullet \downarrow^+\Phi$$

$$\mathtt{next}_\mathtt{HOGS}\ \Phi\ m := (\Phi,\ \mathtt{dom}^\bullet\ m).$$

Furthermore, the *OGS game* OGS is defined by

$$\mathtt{client}_\mathtt{OGS} := \mathtt{HOGS} \qquad \mathtt{server}_\mathtt{OGS} := \mathtt{HOGS} \qquad \mathtt{final}_\mathtt{OGS} := \mathcal{O}^\bullet\ \Delta.$$

5.3 The machine strategy

Recalling that we have fixed a language machine $\mathcal{M}$: `Machine` T at the beginning of §5.2, we now introduce the strategy, called the *machine strategy*, induced by $\mathcal{M}$ and Δ in the game `OGS`. Unfolding definitions, such a strategy should comprise two families $\widetilde{\mathcal{M}}^+$ and $\widetilde{\mathcal{M}}^-$, and morphisms

$$\widetilde{\mathcal{M}}^+\ \Phi \twoheadrightarrow \mathscr{D}\ (O^\bullet\ \Delta + \exists m\colon O^\bullet\ {\downarrow^+}\Phi,\ \widetilde{\mathcal{M}}^-(\Phi, \mathsf{dom}^\bullet\ m))$$
$$\widetilde{\mathcal{M}}^-\ \Phi \twoheadrightarrow \forall m\colon O^\bullet\ {\downarrow^+}\Phi,\ \widetilde{\mathcal{M}}^+(\Phi, \mathsf{dom}^\bullet\ m),$$

for all interleaved contexts Φ. A naive definition of the machine strategy is as follows:

- Active states in any $\widetilde{\mathcal{M}}^+\ \Phi$ are pairs (c, γ) of a language configuration $c\colon C\ (\Delta + {\downarrow^+}\Phi)$ and an assignment $\gamma\colon {\downarrow^-}\Phi \to_{\mathcal{M}} {\downarrow^+}\Phi$ recording the values of variables shared with the other player, in terms of their variables.
- Passive states in any $\widetilde{\mathcal{M}}^-\ \Phi$ are mere assignments ${\downarrow^+}\Phi \to_{\mathcal{M}} \Delta + {\downarrow^-}\Phi$ recording the values of variables shared with the other player, in terms of their variables.
- The next move played by any active state (c, γ) is obtained by evaluating c. If c diverges, then so does the strategy. Otherwise, c evaluates to some normal form $x \cdot o(\delta)$, then the strategy plays (x, o). If x is in Δ, then (x, o) is a final move in $O^\bullet\ \Delta$; otherwise, it is a non-final move in $O^\bullet\ {\downarrow^+}\Phi$, and the resulting passive state is the copairing $[\gamma, \delta]$.
- The reaction of any γ to some pointed observation (x, o) with domain Θ is merely $(x[\gamma] \cdot o(\delta), \gamma)$, where δ denotes the obvious assignment $\Theta \to_{\mathcal{M}} \Delta + {\downarrow^-}\Phi + \Theta$ (recalling Notation 4.22).

In fact, we refine this naive definition to explicitly account for dependencies between values and variables: the value stored by some player for variables in a Γ_i must depend only on previously introduced variables. We formalise this with the following refinement of the notion of assignment, recalling the "final" context Δ fixed at the beginning of §5.2.

Definition 5.16 (🐓). We define the active and passive *OGS environments* Env^+, Env^- : `ICtx` $T \to$ `Set` by mutual induction as follows.

$$\frac{}{\varepsilon\colon \mathsf{Env}^+\ \emptyset} \qquad \frac{}{\varepsilon\colon \mathsf{Env}^-\ \emptyset} \qquad \frac{e\colon \mathsf{Env}^-\ \Phi}{e, \cdot\colon \mathsf{Env}^+\ (\Phi, \Gamma)} \qquad \frac{e\colon \mathsf{Env}^+\ \Phi \qquad \gamma\colon \Gamma \to_{\mathcal{M}} (\Delta + {\downarrow^+}\Phi)}{e, \gamma\colon \mathsf{Env}^-\ (\Phi, \Gamma)}$$

Remark 5.17. Concretely, for any even interleaved context $\Phi = (\Gamma_1, \ldots, \Gamma_{2n})$, active and passive OGS environments have the form

$$(\varepsilon, \gamma_1, \cdot, \gamma_3, \cdot, \ldots, \gamma_{2n-1}, \cdot) \qquad \text{and} \qquad (\varepsilon, \cdot, \gamma_2, \cdot, \gamma_4, \ldots, \cdot, \gamma_{2n}),$$

respectively. Symmetrically, for any odd interleaved context $\Phi = (\Gamma_1, \ldots, \Gamma_{2n+1})$, active and passive OGS environments have the form

$$(\varepsilon, \cdot, \gamma_2, \cdot, \gamma_4, \ldots, \cdot, \gamma_{2n}, \cdot) \qquad \text{and} \qquad (\varepsilon, \gamma_1, \cdot, \gamma_3, \cdot, \ldots, \gamma_{2n+1}),$$

respectively. Intuitively, the active player is to play next, hence did *not* play the last move. Accordingly, active OGS environments thus end with $\cdot$: they do not store the values of variables introduced by the last move. Symmetrically, passive OGS environments end with a proper assignment γ_i. Beware, of course, that the active player in an even interleaved context is P, while the active player in an odd interleaved context is O.

Like the interleaved contexts by which they are indexed, OGS environments can be collapsed into basic assignments.

Definition 5.18 (🐓). The *collapsing* functions for OGS environments are defined by mutual induction as follows.

$$
\begin{array}{ll}
\downarrow^{+} : \mathsf{Env}^{+}\,\Phi & \to \downarrow^{-}\Phi \to_{\mathcal{M}} (\Delta + \downarrow^{+}\Phi) \\
\downarrow^{+}\,\varepsilon & := \mathtt{elim}_{\emptyset} \\
\downarrow^{+}\,(e,\cdot) & := (\downarrow^{-}\,e)[w], \\
\\
\downarrow^{-} : \mathsf{Env}^{-}\,\Phi & \to \downarrow^{+}\Phi \to_{\mathcal{M}} (\Delta + \downarrow^{-}\Phi) \\
\downarrow^{-}\,\varepsilon & := \mathtt{elim}_{\emptyset} \\
\downarrow^{-}\,(e,\gamma) & := [\downarrow^{+}\,e,\ \gamma]
\end{array}
$$

where wkn denotes the obvious weakening: we have $\Phi = (\Phi', \Gamma)$ for some Φ' and Γ, and the result is

$$\downarrow^{+}\Phi' \quad \xrightarrow{e}_{\mathcal{M}} \quad \Delta + \downarrow^{-}\Phi' \quad \xrightarrow{w}_{\mathcal{M}} \quad \Delta + \downarrow^{-}\Phi' + \Gamma,$$

recalling $\downarrow^{-}(\Phi', \Gamma) = \downarrow^{+}\Phi'$ and $\downarrow^{+}(\Phi', \Gamma) = \downarrow^{-}\Phi' + \Gamma$.

We now have everything in place to define the machine strategy.

Definition 5.19 (🐓). The machine strategy $\widetilde{\mathcal{M}} \colon \mathtt{Strat}_{\mathsf{OGS}}$ is defined as follows:

- the family of active states is $\widetilde{\mathcal{M}}^{+}\,\Phi := C\ (\Delta + \downarrow^{+}\Phi) \times \mathsf{Env}^{+}\,\Phi$;
- the family of passive states is $\widetilde{\mathcal{M}}^{-}\,\Phi := \mathsf{Env}^{-}\,\Phi$;
- the action morphism is defined by

$$
\begin{array}{l}
\mathtt{play}_{\widetilde{\mathcal{M}}} \colon \widetilde{\mathcal{M}}^{+} \to \mathscr{D}\ (O^{\bullet}(\Delta) + [\![\widetilde{\mathsf{OGS}}]\!]^{+}\ \widetilde{\mathcal{M}}^{-}) \\
\mathtt{play}_{\widetilde{\mathcal{M}}}\ \Phi\ (c, e) := \\
\qquad x \cdot o(\gamma) \leftarrow \mathtt{eval}\ c\,; \\
\qquad \begin{cases} \mathtt{ret}\ (\mathtt{inl}\ (x, o)) & \text{if } x \in \Delta \\ \mathtt{ret}\ (\mathtt{inr}\ ((x, o), (e, \gamma))) & \text{if } x \in \downarrow^{+}\Phi\,; \end{cases}
\end{array}
$$

- the reaction morphism is defined by

$$
\begin{array}{l}
\mathtt{coplay}_{\widetilde{\mathcal{M}}} \colon \widetilde{\mathcal{M}}^{-} \to [\![\widetilde{\mathsf{OGS}}]\!]^{-}\ \widetilde{\mathcal{M}}^{+} \\
\mathtt{coplay}_{\widetilde{\mathcal{M}}}\ \Phi\ e\ (x, o) := (x[\downarrow^{-} e] \cdot o(\delta_o), (e, \cdot)),
\end{array}
$$

where δ_o denotes the obvious assignment $\mathsf{dom}(o) \to_{\mathcal{M}} \downarrow^{-}\Phi + \mathsf{dom}(o)$.

To finish up, we define two functions injecting configurations (resp. assignments) into active (resp. passive) machine strategy states (🐓),

$$
\begin{array}{ll}
[\![-]\!]^{+} \colon C\ \Gamma \to \widetilde{\mathcal{M}}^{+}\ (\emptyset, \Gamma) & \qquad [\![-]\!]^{-} \colon (\Gamma \to_{\mathcal{M}} \Delta) \to \widetilde{\mathcal{M}}^{-}\ (\emptyset, \Gamma) \\
[\![u]\!]^{+} := (u[w], \varepsilon) & \qquad [\![\gamma]\!]^{-} := (\varepsilon, \gamma)
\end{array}
$$

where w denotes the obvious weakening $\Gamma \to_{\mathcal{M}} \Delta + \Gamma$.

6 SOUNDNESS

In this section, we prove the soundness of our model, i.e., for any pair of configurations u and w, bisimilarity of induced strategies entails substitution equivalence:

$$\forall u, w,\ [\![u]\!]^{+} \approx^{+}_{\mathrm{OGS}} [\![w]\!]^{+} \to u \approx_{\mathrm{SUB}} w,$$

where

- $\approx^{+}_{\mathrm{OGS}}$ denotes the positive component of bisimilarity between strategies (Definition 5.11) in the OGS game (Definition 5.15), and
- $\approx_{\mathrm{SUB}}$ denotes substitution equivalence (Definition 4.24).

Following standard OGS methods, our proof for this statement uses a crucial intermediate definition, namely composition of strategies with counter-strategies.

We first describe what composition should do, intuitively, and show how soundness follows from two main properties properties of composition, adequacy and congruence. It will then remain to actually define composition, and prove both properties.

As before, we fix a set T of types, a “final” context Δ, and a language machine $\mathcal{M} = (\mathcal{V}, C, O, \texttt{eval}, \texttt{emb})$, omitting subscripts.

6.1 Soundness from composition

Composition should take as arguments an active and a passive state over the same interleaved context in the machine strategy, and return (a computation of) a pointed observation over the final context Δ. It should thus be a map

$$\forall \Phi, (C\ (\Delta + \downarrow^{+}\Phi) \times \mathsf{Env}^{+}\ \Phi) \rightarrow \mathsf{Env}^{-}\ \Phi \rightarrow \mathscr{D}\ (O^{\bullet}\ \Delta).$$

Intuitively, given any $u\colon C\ (\Delta + \downarrow^{+}\Phi)$, $e\colon \mathsf{Env}^{+}\Phi$, and $f\colon \mathsf{Env}^{-}\Phi$, the composition $(u, e) \parallel f$ should work as follows. We start by evaluating u.

(1) If u diverges, then so does $(u, e) \parallel f$.
(2) If u reduces to $x \cdot o(\gamma)$ with x a final channel, then $(u, e) \parallel f$ finishes with the pointed observation $x \cdot o$.
(3) If u reduces to $x \cdot o(\gamma)$ with x in the support of f, then $(u, e) \parallel f$ continues, roughly, as $((f\ x) \cdot o, f) \parallel (e, \gamma)$.

The two main properties that we expect composition to satisfy are the following.

Definition 6.1. Suppose given a composition map $\parallel$ of the above type.

(1) Composition is *adequate* iff, for all configurations $c\colon C\ \Gamma$ and assignments $\gamma\colon \Gamma \rightarrow \Delta$, evaluating and observing $c[\gamma]$ yields the same result as composing $[\![c]\!]^{+}$ and $[\![\gamma]\!]^{-}$, up to weak bisimilarity, i.e., $\texttt{eval}^{o}_{\mathcal{M}}\ (c[\gamma]) \approx_{\mathscr{D}} [\![c]\!]^{+} \parallel [\![\gamma]\!]^{-}$.
(2) Weak bisimilarity is a *congruence* for composition iff, for any weakly bisimilar active states s_1 and s_2 and for any passive state t, $s_1 \parallel t$ and $s_2 \parallel t$ are weakly bisimilar, and symmetrically.

Remark 6.2. It might not obvious that adequacy type-checks at all. But by construction $[\![c]\!]^{+} \parallel [\![\gamma]\!]^{-}$ lives in $\mathscr{D}\ (O^{\bullet}\ \Delta)$, and so does $\texttt{eval}^{o}_{\mathcal{M}}\ (c[\gamma])$, since $\texttt{eval}^{o}_{\mathcal{M}}$ is the composite

$$C\ \Delta \xrightarrow{\texttt{eval}} \mathscr{D}\ (\mathcal{N}_{\mathcal{M}}\ \Delta) \xrightarrow{\mathscr{D}\ \pi_1} \mathscr{D}\ (O^{\bullet}\ \Delta).$$

Let us readily show that soundness follows from composition:

Proposition 6.3. *If any adequate composition for which weak bisimilarity is a congruence exists, then OGS is sound.*

Proof. For any configurations $c_1, c_2\colon C\ \Gamma$ with bisimilar interpretations $[\![c_1]\!]^{+}$ and $[\![c_2]\!]^{+}$, and any assignment $\gamma\colon \Gamma \rightarrow_{\mathcal{M}} \Delta$, we have

$$\begin{aligned}
\texttt{eval}^{o}_{\mathcal{M}}\ (c_1[\gamma]) &\approx_{\mathscr{D}} [\![c_1]\!]^{+} \parallel [\![\gamma]\!]^{-} && \text{by } 1\\
&\approx_{\mathscr{D}} [\![c_2]\!]^{+} \parallel [\![\gamma]\!]^{-} && \text{by } 2\\
&\approx_{\mathscr{D}} \texttt{eval}^{o}_{\mathcal{M}}\ (c_2[\gamma]) && \text{by } 1,
\end{aligned}$$

hence $c_1 \approx_{\text{SUB}} c_2$, as desired. □

6.2 Defining composition

We have shown that soundness follows from the existence of an adequate and congruent composition. In this section, we explain how to construct such a composition, up to suitable hypotheses.

The naive formalisation of the above intuitive definition of composition is *not* guarded: *a priori*, some composition $s \parallel t$ could be caught in a loop in the third case, with s and t exchanging moves indefinitely, and thus not producing any value in $\mathscr{D}\ (O^\bullet\ \Delta)$, even after infinitely many computation steps.

This is rather subtle: by definition, even a single computation step in the configuration of the active state counts as productive, even though it models some sort of "silent" step. Indeed, each silent step contributes to constructing a value in $\mathscr{D}\ (O^\bullet\ \Delta)$. The problem comes from potential infinite sequences of moves played without a single computation step, i.e., at each step, u does not reduce, but *is* $x \cdot o(\gamma)$.

At this point, it seems reasonable to ask: since adequacy is stated up to weak bisimilarity, isn't it possible to add a formal silent step in front of the third case? This does yield a valid definition, but we were not able to prove it adequate. Indeed, Coq's coinduction principle, which essentially states the existence and uniqueness of fixed points, only works up-to *strong* bisimilarity. And this is for good reason, because replacing strong bisimilarity with weak bisimilarity leads to a false statement, as illustrated by the following example.

Example 6.4. Letting $\mathbb{1} = \{\star\}$ and $\mathbb{2} = \{\mathtt{true}, \mathtt{false}\}$ denote respectively the one and two-element sets, consider the map $e \colon \mathbb{1} \to \mathscr{D}\ (\mathbb{1} + \mathbb{2})$ mapping any $x \colon \mathbb{1}$ to $\mathtt{ret}\ (\mathtt{inl}\ x)$. This map can be iterated and intuitively encodes the recursive definition of a map $f \colon \mathbb{1} \to \mathscr{D}\ \mathbb{2}$ caught in an infinite loop.

Indeed a fixed point of e w.r.t. weak bisimilarity is a map $f \colon \mathbb{1} \to \mathscr{D}\ \mathbb{2}$ such that, for all $x \colon \mathbb{1}$, we have

$$f\ x \approx_{\mathscr{D}} e\ x \ggg \begin{cases} \mathtt{inl}\ x' & \mapsto f\ x' \\ \mathtt{inr}\ y & \mapsto \mathtt{ret}\ y, \end{cases}$$

which by definition of e simplifies to $f\ x \approx_{\mathscr{D}} f\ x$.

This equation is solved by the standard iteration operator by defining $f\ x \coloneqq \mathtt{tau}\ (f\ x)$, i.e., $f\ x \coloneqq \mathtt{tau}^\omega$. But in fact it is also solved by any $f \colon \mathbb{1} \to \mathscr{D}\ \mathbb{2}$, in particular $f\ x \coloneqq \mathtt{ret}\ \mathtt{true}$ and $f\ x \coloneqq \mathtt{ret}\ \mathtt{false}$, so uniqueness cannot hold. The same phenomenon occurs for other equations, such as $e\ x \coloneqq \mathtt{tau}^n\ (\mathtt{inl}\ x)$.

This example shows that we cannot afford to merely add a formal silent step in front of the problematic third case. But it also shows that we should better aim for adequacy up to strong bisimilarity, as adequacy up to weak bisimilarity is beyond the reach of Coq's coinduction principle.

In order to do this, let us examine more closely the potential causes of divergence.

First, in the third case, t will replace x with its value. Divergence would happen if this value was a variable, and the value of that variable was again a variable, and so on. Fortunately, because OGS positions are interleaved contexts, the chain of variables obtained by such successive instantiations reaches older and older layers and, hence must stop eventually.

But then, even when the head variable is instantiated with some proper value, nothing guarantees that some redex is reached.

Example 6.5. Consider a configuration of the form $\langle\, \lambda x.v \mid \alpha' \,\rangle$ in the language of §2, where α' is a variable created by the passive player, say O. Then, the active player, say P, creates a variable, say y, for $\lambda x.v$, and plays $(\alpha', \langle\, y \mid \square \,\rangle)$. Roles are then switched, and O becomes active, maybe with a configuration of the form $\langle\, y \mid [w'] \cdot \pi' \,\rangle$. At this point, we have a proper instantiation of the played variable α', but no reduction occurs yet! Indeed, it is only when O plays – a move of the form

$(y, \langle \square \mid [w] \cdot \pi \rangle)$ – that P becomes active again, with a configuration of the form $\langle \lambda x.v \mid [w] \cdot \pi \rangle$, which reduces at last.

In the general case, nothing guarantees that any finite chain of instantiations reaches a redex. (The reader is invited to imagine some weird language in which some redexes are infinite patterns.)

To avoid such redex-free infinite instantiation chains, we make an additional hypothesis on the considered language machine, which we show is satisfied by our applications. There is some degree of freedom here. We choose a version that relies on observations only:

Definition 6.6 (🐓). Let $\prec$ denote the binary relation on $\exists \tau \colon T,\ \texttt{Obs}\ \tau$ defined by $(\tau_2, o_2) \prec (\tau_1, o_1)$ iff there is a non-variable value v such that applying the observation o_1 to v returns *without any reduction step* a normal form with observation o_2. More formally, this means that there exists a non-variable value $v \colon \mathcal{V}\ \Gamma\ \tau_1$, assignments γ, δ and a variable i such that $\texttt{eval}\ (v \cdot o_1(\gamma)) \approx_{\mathcal{D}} \texttt{ret}\ (i \cdot o_2(\delta))$.

The language machine $\mathcal{M}$ has *finite redexes* iff $\prec$ is well founded.

Example 6.7. Let us consider an extension of simply-typed λ-calculus with lists and deep pattern matching. Thus, types now include $\mathrm{list}(A)$, for all types A, we introduce a new syntactic category of *patterns*, ranged over by $p, q, \ldots$, and extend the definition of values and programs as follows:

$$
\begin{aligned}
p &::= x \mid [] \mid x :: p \\
v &::= \ldots \mid [] \mid v_1 :: v_2 \\
p &::= \ldots \mid \text{match } e_1 \text{ with } \ldots p_i \to e_i \ldots \text{ end}
\end{aligned}
$$

Furthermore, we add the following inference rules for the reduction relation.

$$
\frac{}{v \mid x \triangleleft [x \mapsto v]} \qquad \frac{}{[] \mid [] \triangleleft []} \qquad \frac{v_2 \mid p \triangleleft \gamma}{v_1 :: v_2 \mid x :: p \triangleleft (\gamma, x \mapsto v_1)}
$$

$$
\frac{v \mid p_1 \triangleleft \gamma}{\text{match } v \text{ with } p_1 \to e_1, \ldots \text{ end} \to e_1[\gamma]}
$$

$$
\frac{v \mid p_1 \not\triangleleft \gamma}{\text{match } v \text{ with } p_1 \to e_1, p_2 \to e_2, \ldots \text{ end} \to \text{match } v \text{ with } p_2 \to e_2, \ldots \text{ end}}
$$

Example 6.8. Let us consider a rather ad-hoc language machine where this relation $\prec$ can admit an arbitrarily long, yet finite chain. To do so, we consider an untyped version of the language of Figure 1, and extend the grammar as follows,

$$
\begin{aligned}
\text{values} \ni v &::= \ldots \mid [p] \\
\text{programs} \ni p, q &::= \ldots \mid \mathbf{fork}(p, q) \mid \mathbf{yield}_i \quad (i \in \mathbb{N}) \\
\text{contexts} \ni \pi &::= \ldots \mid [p] \cdot \pi
\end{aligned}
$$

and the evaluation rules as follows.

$$
\begin{aligned}
\langle \mathbf{fork}(p, q) \mid \pi \rangle &\longrightarrow \langle p \mid [q] \cdot \pi \rangle \\
\langle \mathbf{yield}_n \mid [p_1] \cdot \ldots \cdot [p_n] \cdot \pi \rangle &\longrightarrow \langle p_n \mid [p_1] \cdot \ldots \cdot [p_{n-1}] \cdot \pi \rangle.
\end{aligned}
$$

The $\mathbf{fork}(p, q)$ instruction pushes a suspended computation $[q]$ onto the stack, while the $\mathbf{yield}_n$ instruction resumes the nth suspended computation from the stack. The observation structure O is then extended with observations $\langle \mathbf{yield}_n \mid x_1 \cdot \ldots \cdot x_p \cdot \square \rangle$, for all $p < n \in \mathbb{N}$. This gives rise to the

following sequence of observations:

$$\langle \square \mid \alpha' \rangle \succ \langle \mathbf{yield}_n \mid \square \rangle \succ \langle \mathbf{yield}_n \mid x_1 \cdot \square \rangle$$
$$\succ \langle \mathbf{yield}_n \mid x_1 \cdot x_2 \cdot \square \rangle \succ \ldots \succ \langle \mathbf{yield}_n \mid x_1 \cdot x_2 \ldots x_{n-1} \cdot \square \rangle$$

where we omit the typing information, e.g., working in an untyped setting.

So now we know that the coinductive definition of composition is semantically productive, but it remains to convince Coq! For this, we introduce a novel form of iterator, which takes as input what we call an *eventually guarded* recursive definition. Intuitively, an eventually guarded recursive definition is a family of mutually recursive definitions, in which each definition becomes guarded after finitely many unfoldings. Let us make this more precise.

Definition 6.9 (🐓).

- An *equation* consists of a set X of *variables*, a set Y of *constants*, and a *definition* function $X \to \mathscr{D}\ (X + Y)$.
- An element $u \colon \mathscr{D}\ (X + Y)$ is *guarded* if it is not of the form $\mathtt{ret}\ (\mathtt{inl}\ x)$.
- Similarly, an equation $e \colon X \to \mathscr{D}\ (X + Y)$ is *pointwise guarded* if for all x, $e\ x$ is guarded.

Definition 6.10 (🐓). An equation $e \colon X \to \mathscr{D}\ (X + Y)$ is *pointwise eventually guarded* if for all x, there is an $n \colon \mathbb{N}$ such that $e^n\ x$ is guarded, where

$$e^0 x := \mathtt{ret}\ (\mathtt{inl}\ x)$$
$$e^{n+1}\ x := e\ x \gg\!= \begin{cases} \mathtt{inl}\ x' & \mapsto e^n\ x' \\ \mathtt{inr}\ y & \mapsto \mathtt{ret}\ y. \end{cases}$$

Our iterator for eventually guarded definitions is the proof of

Proposition 6.11 (🐓). *All pointwise eventually guarded equations admit a unique fixed point w.r.t. strong bisimilarity.*

We then define the equation for composition, as follows.

Definition 6.12 (🐓). The composition equation is given by the following map.

$$\mathsf{comp\text{-}eqn} \colon \widetilde{\mathcal{M}}^+ \| \widetilde{\mathcal{M}}^- \to \mathscr{D}\ (O^\bullet\ \Delta + \widetilde{\mathcal{M}}^+ \| \widetilde{\mathcal{M}}^-)$$
$$\mathsf{comp\text{-}eqn}\ (u, w) :=$$
$$\quad \mathtt{play}_{\widetilde{\mathcal{M}}}\ u \gg\!=$$
$$\quad \begin{cases} \mathtt{inl}\ r & \mapsto \mathtt{ret}\ (\mathtt{inl}\ r) \\ \mathtt{inr}\ (m, u') & \mapsto \mathtt{ret}\ (\mathtt{inr}\ ((\mathtt{coplay}_{\widetilde{\mathcal{M}}}\ w\ m), u')) \end{cases}$$

We prove (🐓) this equation is pointwise eventually guarded by combining the hypothesis that $\mathcal{M}$ has finite redexes, together with a well-founded induction on the age of the channel—depth in the interleaved context—the two composed strategies are interacting over. This enables finding a reduction step after some finite amount of interaction steps, i.e., iterations of the composition equation. This legitimates the following.

Definition 6.13 (🐓). Composition is the unique fixed point of comp-eqn w.r.t. strong bisimilarity.

The statement of adequacy of composition quantifies configurations and assignments. We generalize it to arbitrary active and passive states of the machine strategies and prove it by applying the uniqueness principle of the composition fixed point. After some equational reasoning on assignments we conclude with the hypothesis that for $\mathcal{M}$, evaluation respects substitution (Def. 4.26) and we have proved:

Theorem 6.14 (🐓). *If $\mathcal{M}$ has finite redexes and evaluation respects substitution, then bisimilarity of induced LTSs is sound w.r.t. substitution equivalence.*

7 IMPLEMENTATION DISCUSSION

Through this section, we highlight a few relevant aspects of our formal development.

Disagreement between the paper and mechanisation. We have made sure to remain as faithful to our mechanisation as possible, but two minor technical differences remain.

A first superficial distinction concerns the `emb` operator (Def. 4.7). In the mechanisation, instead of this embedding of normal forms—that is, of triplets $x \cdot o(\gamma)$—into configurations, we use a more expressive but equivalent operation embedding $v \cdot o(\gamma)$ triplets into configurations. This operator can be thought of as applying the observation o with arguments γ to the value v.

The second distinction concerns the representation of strategies. In Definition 5.9, we define strategies as a coalgebra, that is by a carrier ($\mathcal{S}^+$ and $\mathcal{S}^-$) and a morphism (*action* and *reaction*). In the Coq mechanisation, instead of working with such coalgebras, we opt to work with *elements of the final coalgebra*. In doing so, we introduce an indexed variant of interaction trees ([25], 🐓) which has been instrumental to precisely represent strategies.

Axioms and executability. All our results are admit free, relying only on the Coq's standard library's statement of axiom K.

We additionally point out that by working with coinductive representations of LTSs (as opposed to more traditional relational specifications), we have the possibility to extract directly executable strategies to OCaml. We believe it opens the perspective for developing new software artifacts around OGS.

8 RELATED WORK

OGS models that appear in the literature are built in two possible ways: (1) using an operational semantics defined as an environment-based abstract machine that implements the synchronisation process used to define composition of OGS states, as in [10, 18, 19]; (2) using an operational semantics defined as a small-step reduction relation, that is then shown to be bisimilar with the synchronisation process implemented as an abstract machine, as in [13, 16]. In our work, we have chosen to extend the second approach, using an abstract notion of language machine with axioms the operational semantics has to satisfy for the OGS model to be sound.

In the oldest proof of soundness of the OGS w.r.t. contextual equivalence, published in [18], the adequacy result (Proposition 2 of that paper) is only sketched so that the chattering problem is overlooked. The structure needed to collapse the composition of an active and a passive state into a language machine configuration was studied in [16, 19]. In these works, an acyclicity condition on the assignments of the two configurations is stated to prove that no infinite chattering is possible. Acyclicity is enforced by the existence of an order on the variables forming the domain of the assignments. Moreover, the preservation of this acyclicity condition during the interaction was established in [19] using a typing system for the interaction. In our work, we rather choose to give more structure to positions: using interleaved contexts, i.e., lists of typing contexts, provides a direct way to enforce acyclicity.

Our notion of games and strategy is directly taken from [21]. In that work, a form of adequacy, namely that composition computes the substitution, was established for a particular instance of language machine. In future work, we would be interested in understanding if this proof can be recast in our axiomatic approach.

The issue of infinite chattering was studied in game semantics [1, 6, 12] to provide *winning conditions* enforcing the preservation of totality of strategies by composition. This infinite chattering

is thus different from the one studied in our paper, where we allow programs to have infinite reduction paths. In our setting, an infinite chattering would be an artifact of the composition looping without even creating a reduction step.

9 CONCLUSION AND PERSPECTIVES

We have proposed an abstract notion of language with evaluator, for which we have constructed a generic OGS interpretation, which we have proved sound, in Coq.

An important direction for future work is to incorporate more language features into the framework. Notably, we plan to cover effectful evaluators by generalising from the delay monad to richer, well-behaved monads. It would also be useful to handle more sophisticated type systems, including, e.g., polymorphism or subtyping.

Another direction consists in investigating completeness in the abstract framework, be it by restricting attention to sufficiently effectful languages, or by refining the OGS model to make it fully abstract, by enforcing conditions like well-bracketing or visibility.

Finally, it might be fruitful to investigate the link between OGS and other models in the abstract framework, including denotational game semantics and Lassen's normal form bisimilarity.

REFERENCES

[1] Samson Abramsky et al. 1997. Semantics of interaction: an introduction to game semantics. *Semantics and Logics of Computation* 14, 1 (1997).

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Information and Computation* 163, 2 (2000), 409–470. https://doi.org/10.1006/inco.2000.2930

[3] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 1–30. https://doi.org/10.1145/3236785

[4] Benedict Bunting and Andrzej S. Murawski. 2023. Operational Algorithmic Game Semantics. In *LICS*. 1–13. https://doi.org/10.1109/LICS56636.2023.10175791

[5] Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

[6] Pierre Clairambault and Russ Harmer. 2010. Totality in arena games. *Ann. Pure Appl. Log.* 161, 5 (2010), 673–689. https://doi.org/10.1016/J.APAL.2009.07.016

[7] Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 233–243. https://doi.org/10.1145/351240.351262

[8] Paul Downen and Zena M. Ariola. 2020. Compiling With Classical Connectives. *Log. Methods Comput. Sci.* 16, 3 (2020). https://lmcs.episciences.org/6740

[9] Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498715

[10] Dan R. Ghica and Nikos Tzevelekos. 2012. A System-Level Game Semantics. In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2012, Bath, UK, June 6-9, 2012 (Electronic Notes in Theoretical Computer Science, Vol. 286)*, Ulrich Berger and Michael W. Mislove (Eds.). Elsevier, 191–211. https://doi.org/10.1016/J.ENTCS.2012.08.013

[11] J. M. E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/inco.2000.2917

[12] Martin Hyland. 1997. Game semantics. *Semantics and logics of computation* 14 (1997), 131.

[13] Guilhem Jaber. 2015. Operational Nominal Game Semantics. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9034)*, Andrew M. Pitts (Ed.). Springer, 264–278. https://doi.org/10.1007/978-3-662-46678-0_17

[14] Guilhem Jaber. 2020. SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References. *Proceedings of the ACM on Programming Languages* 28 (2020), 1–28. https://doi.org/10.1145/3371127

[15] Guilhem Jaber and Andrzej S. Murawski. 2021. Complete trace models of state and control. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*

(Lecture Notes in Computer Science, Vol. 12648), Nobuko Yoshida (Ed.). Springer, 348–374. https://doi.org/10.1007/978-3-030-72019-3_13

[16] Guilhem Jaber and Andrzej S. Murawski. 2021. Compositional relational reasoning via operational game semantics. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470524

[17] Guilhem Jaber and Nikos Tzevelekos. 2016. Trace semantics for polymorphic references. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 585–594. https://doi.org/10.1145/2933575.2934509

[18] James Laird. 2007. A Fully Abstract Trace Semantics for General References. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4596)*, Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki (Eds.). Springer, 667–679. https://doi.org/10.1007/978-3-540-73420-8_58

[19] James Laird. 2020. A Curry-style Semantics of Interaction: From Untyped to Second-Order Lazy $\lambda\mu$-Calculus. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 422–441. https://doi.org/10.1007/978-3-030-45231-5_22

[20] Søren B. Lassen and Paul Blain Levy. 2007. Typed Normal Form Bisimulation. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4646)*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, 283–297. https://doi.org/10.1007/978-3-540-74915-8_23

[21] Paul Blain Levy and Sam Staton. 2014. Transition systems over games. In *Proceeding of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 64:1–64:10. https://doi.org/10.1145/2603088.2603150

[22] Yu-Yang Lin and Nikos Tzevelekos. 2020. Symbolic Execution Game Semantics. In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference) (LIPIcs, Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:24. https://doi.org/10.4230/LIPICS.FSCD.2020.27

[23] Ian A. Mason and Carolyn L. Talcott. 1991. Equivalence in Functional Languages with Effects. *J. Funct. Program.* 1, 3 (1991), 287–327. https://doi.org/10.1017/S0956796800000125

[24] Robin Milner. 1977. Fully Abstract Models of Typed *lambda*-Calculi. *Theor. Comput. Sci.* 4, 1 (1977), 1–22. https://doi.org/10.1016/0304-3975(77)90053-6

[25] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[26] Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching.* Ph. D. Dissertation. USA. Advisor(s) Pfenning, Frank and Lee, Peter. AAI3358066.