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, weak 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 with various evaluation strategies. These calculi notably feature recursive definitions, first-class continuations, and a wide variety of datatypes. All constructions and proofs are entirely mechanised in the Coq proof assistant.

Additional Key Words and Phrases: programming language semantics, mechanized semantics

ACM Reference Format:
Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. 2023. An abstract, certified account of operational game semantics. In *Proceedings of European Symposium on Programming (ESOP '25).* ACM, New York, NY, USA, 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Operational game semantics [23, 25] (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, 16], 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 [12, 22]. Furthermore, OGS models are useful for proving the correctness of decision procedures for contextual equivalence [4, 19, 27].

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.

Authors' addresses: Peio Borthelle, peio.borthelle@univ-smb.fr, Université Savoie Mont Blanc, CNRS, LAMA, Chambéry, France, 73000; Tom Hirschowitz, tom.hirschowitz@univ-smb.fr, Université Savoie Mont Blanc, CNRS, LAMA, Chambéry, France, 73000; Guilhem Jaber, guilhem.jaber@univ-nantes.fr, Nantes Université, LS2N, Nantes, France; Yannick Zakowski, yannick.zakowski@inria.fr, ENS de Lyon, INRIA, CNRS, Lyon 1, LIP, Lyon, France.

ESOP '25, 2025, Hamilton, Canada

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

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 [26] offered a high-level categorical framework. More recently, Laird [24] 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.

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 $\overline{\lambda}\mu\tilde{\mu}$-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 [26], 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 $\overline{\lambda}\mu\tilde{\mu}_Q$-calculus [7] () and the polarised System D () from Downen and Ariola [8].
- We implement () an indexed variant of the *interaction trees* library [34], 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 [28, 31]. 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 two programs, say p and q, behave the same under any closed instantiation of their free variables, in any closed evaluation context of some fixed, basic type like the booleans: $E[p[\sigma]] \cong E[q[\sigma]]$, for some notion of observation of closed booleans. We argue that, up to mild expository adjustments, this may in fact be viewed as p and q 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 programs p to so-called

[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.

named programs $\alpha[p]$ for some context variable α, or, in $\mu\tilde{\mu}$-style notation, $\langle\, p \mid \alpha \,\rangle$, and putting p 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 generalised values—is instrumental to simplify the axiomatisation of a language and the generic OGS construction.

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 to normal-forms shapes the messages exchanged by the program and environment. Roughly, the basic infrastructure consists of

- a set of *types* — *typing contexts* are then defined as sequences of types,
- a family of *configurations*, indexed by typing contexts, and
- a family of *values*, indexed by *sequents*, i.e., pairs of a typing 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 $\langle\, x \mid \Box \,\rangle$, saying that we are returning a value to the head variable, which has been replaced by the placeholder $\Box$. 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 $\langle\, \Box \mid \bullet y; \alpha \,\rangle$, saying that we are applying the head variable to some argument y in the evaluation context α. Here again, the head variable has been replaced by the placeholder $\Box$.
- 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 typing context to each observation. Evaluation is then postulated as a partial map from configurations to triples of

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

satisfying some coherence axioms.

Remark 1.3. Since we are working in a total meta-theory, namely the type theory underlying the Coq proof assistant, the representation of the evaluation function has to be carefully designed. Having access to coinductive types in this meta-theory, what is meant by the *partial map* representing the evaluation 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 [26], 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 6.23), 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 all languages we know. To the best of our knowledge, this is the first time this condition is explicitly identified.

Let us briefly explain it. A key lemma in virtually any OGS soundness proof roughly states the following: given any suitable program p and evaluation context E of type unit, the program $E[p]$ behaves the same as letting the strategies associated to p and E play against one another. A bit more formally, one defines a *composition* operation. The result of composition may either perform some (potentially diverging) computation steps, or play a final "return" move. And the key lemma states that $E[p]$ diverges iff the composition does, and, if not, both play the same final return move.

The difficulty lies in the definition of composition. In principle, it should work as follows. An invariant is that one of the two given strategies is *active* while the other is *waiting*, in a suitable sense. The composition is then computed like so:

- If the active strategy performs a reduction step or the final return move, then so does the composition.
- If the active strategy performs a non-final move, then the waiting strategy reacts to it, the roles are switched, and we start over, coinductively.

The problem is that the second clause does not produce anything in the result. This matches the behaviour of $E[p]$ as a whole, because this clause models communication between E and p, which is invisible in $E[p]$. In order for the definition to be safely guarded, we thus need to make sure that no two strategies get stuck in a loop involving only the second clause.

In the OGS model, we consider strategies consisting of some suitable family of program fragments from the language machine, and moves also are program fragments. When reacting to some non-final move, the waiting strategy suitably plugs the observation into one of its program fragments, and evaluates the result.

The problem is that this process could diverge without producing a single computation step, for two subtly different reasons:

- First, if the fragments are too trivial, plugging the given observation into them might not change the shape of the evaluated program. Assume that, as in Example 1.2, the active player reaches the configuration $\langle\, x \mid \bullet v; E \,\rangle$. Splitting it into the head variable x, observation $\langle\, \square \mid \bullet y; \alpha \,\rangle$, and filling

$\{y \mapsto v, \alpha \mapsto E\}$. Then the waiting strategy plugs the observation $\langle\, \Box \mid \bullet y; \alpha \,\rangle$ into the value it associates with x. If this value is again a variable, say x', it gets $\langle\, x' \mid \bullet y; \alpha \,\rangle$, which is already a normal form, hence unproductively splits into x', the same observation $\langle\, \Box \mid \bullet y; \alpha \,\rangle$, and the identity filling. No computation has occurred, hence the process could diverge unproductively.

- The second kind of divergence occurs when the head variable is instantiated by some proper term, but this does not suffice to trigger computation. E.g., starting from some configuration of the shape $\langle\, x \mid \alpha \,\rangle$, it takes two instantiations (of x and α) to reach a configuration of the shape $\langle\, \lambda y.p \mid \bullet v; \beta \,\rangle$, which performs a proper computation step. Now imagine a language in which instantiating head variables never leads to any configuration actually performing some reduction steps. In such a language, composition could unproductively get stuck in a loop in the second clause.

The first kind of divergence is well-known, and usually called *infinite chattering*. It is usually dealt with by some sort of acyclicity argument. Instead, we are able to build acyclicity into the structure of positions, by refining the shape of typing contexts. The second kind of divergence never occurs in relevant instances, but we have to rule it out by imposing the above-mentioned additional hypothesis in the abstract framework.

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 comes 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

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

where $\lambda^{\mathbf{rec}}$ 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.

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

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

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

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

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

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 boolean closed programs. More precisely,

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

Definition 2.2. Two programs $\varepsilon \vdash p, q$ 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 $\Gamma \vdash p, q : A$ are *CIU-equivalent*, which we denote by $p \approx_{\text{CIU}} q$, iff for all assignments $\varepsilon \vdash \sigma : \Gamma$, 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 details.

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 them as stacks:

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

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

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

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.

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

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.

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

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

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

Definition 2.4. For any typing context Γ, two configurations $\Gamma \vdash c, d$ are *substitution equivalent*, which we denote by $c \approx_{\text{SUB}} d$, iff for all assignments $(\alpha : \neg\top) \vdash \sigma : \Gamma$, $c[\sigma]$ 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 typing 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 context E of type $\top$ with a hole of type A in the source language. Furthermore, for such assignments and evaluation contexts, $\langle\, p \mid \beta \,\rangle[\sigma]$ and $E[p[\gamma]]$ coterminate in the obvious sense, hence the result follows. □

3 A PRIMER ON OGS

In this section, we explain OGS on the low-level 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 shapes.

$\langle\, x' \mid \bullet v; \pi \,\rangle$ — 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 Opponent.

Remark 3.2. A message like $(x, \langle\, \square \mid \bullet\, y'; \alpha' \,\rangle)$ is often noted in the literature on OGS as $\overline{x}(y', \alpha')$, following a process-calculi notation, or as $x.\mathit{call}(y', \alpha')$ in object-oriented fashion. Such a message corresponds to a *question* of the channel x, providing the y' as an argument and α' as the return channel. Likewise, a message $(\alpha, \langle\, x' \mid \square \,\rangle)$ is usually noted as $\overline{\alpha}(x')$ or as $\alpha.\mathit{ret}(x')$, 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 as they can be reconstructed by maintaining a stack of evaluation contexts. Here, we choose to avoid this additional stack construction and work in the more general setting of possibly non-well-bracketed interactions, so that we crucially use these names.

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 $\langle\, y \mid \pi' \,\rangle$.

In this particular toy language, as there is at most one kind of observation per type of the head variable, and since fresh variables are nameless in the well-scoped approach with De Bruijn indices, observations on a type τ are truly just a proof that $\tau \neq \top$.

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

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

These context increments are then used to maintain two typing contexts during the play, the variables introduced by P and the variables introduced by O.

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

Definition 3.3. A position is a pair of typing contexts, together with a boolean tag indicating the active side. We denote positions where P is active by $(\Gamma, \Delta)^+$, and positions where P is waiting by $(\Gamma, \Delta)^-$.

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

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

We then define strategies coinductively:

Definition 3.4. 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 waiting position $(\Gamma, \Delta)^-$ is a map associating to each move m from it a *residual* strategy from the target of m.

Example 3.5. 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 $[\alpha \mapsto \langle\, \Box \mid \bullet\, \mathbf{tt}; \bullet\mathbf{not}; \kappa \,\rangle]$, 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:

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

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

Theorem 3.6. *If the strategies associated to two configurations are* weakly bisimilar*, then these configurations are substitution equivalent.*

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

Definition 3.7. 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 waiting 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 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, 11]. In the remainder of this

section, we define typing 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 Typing contexts, families and variables

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

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

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

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

Definition 4.1 ([11],🔥). 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 $u : \mathcal{X}\ \tau\ \Gamma$ may be thought of as an $\mathcal{X}$-term of type τ in typing 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 $f : \mathcal{X} \to\!\!\!\!\bullet\ \mathcal{Y}$ of families consists of a map $f : \forall \Gamma : \mathtt{Ctx}\ T,\ \mathcal{X}\ \Gamma \to \mathcal{Y}\ \Gamma$.
A morphism $f : \mathcal{X} \to\!\!\!\!\bullet\ \mathcal{Y}$ of sorted families consists of a map $f : \forall \tau : T,\ \Gamma : \mathtt{Ctx}\ T,\ \mathcal{X}\ \tau\ \Gamma \to \mathcal{Y}\ \tau\ \Gamma$.

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

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 the left coproduct injection, written $\eta\ x$ for some $x : X$, or the right coproduct injection, written $\tau\ a$ for some $a : \mathscr{D}\ X$, 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 $\mathtt{Fam}\ T$ and $\mathtt{Fam_s}\ T$. We write $\approx_{\mathscr{D}}$ for weak bisimilarity (🔥), i.e., either both sides loop or both terminate on the same element $x : X$. We write $\approx_{\mathscr{D}}$ 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* $C : \mathtt{Fam}\ T$ and $\mathcal{N} : \mathtt{Fam}\ T$ is given by maps:

$$\mathtt{eval} : C \nrightarrow \mathscr{D}\ \mathcal{N} \qquad\qquad \mathtt{emb} : \mathcal{N} \nrightarrow C$$

such that $\mathtt{eval} \circ \mathtt{emb} \approx_{\mathscr{D}} \eta$.

Example 4.8 (λ^{rec}, 🔥). `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 notion of context map from Fiore and Szamozvancev [11], which we call *assignments*.

Definition 4.9 (🔥). For any typing contexts $\Gamma, \Delta : \mathtt{Ctx}\ T$ and sorted family $\mathcal{X}$, an $\mathcal{X}$*-assignment* $\sigma : \Gamma \dashv[\mathcal{X}]\vdash \Delta$, 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

$$- \dashv[-]\vdash - : \mathtt{Ctx}\ T \to \mathtt{Fam}_{\mathtt{s}}\ T \to \mathtt{Ctx}\ T \to \mathtt{Set}$$
$$\Gamma \dashv[\mathcal{X}]\vdash \Delta \coloneqq \forall \tau : T,\ \tau \in \Gamma \to \mathcal{X}\ \tau\ \Delta.$$

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

$$\mathtt{sub} : \forall \tau : T, \forall \Gamma, \Delta : \mathtt{Ctx}\ T,\ \mathcal{X}\ \tau\ \Gamma \to (\Gamma \dashv[\mathcal{X}]\vdash \Delta) \to \mathcal{X}\ \tau\ \Delta$$

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

$$[\![-,-]\!]_s : \mathtt{Fam}_{\mathtt{s}}\ T \to \mathtt{Fam}_{\mathtt{s}}\ T \to \mathtt{Fam}_{\mathtt{s}}\ T$$
$$[\![\mathcal{X}, \mathcal{Y}]\!]_s\ \tau\ \Gamma = \forall \Delta : \mathtt{Ctx}\ T,\ (\Gamma \dashv[\mathcal{X}]\vdash \Delta) \to \mathcal{Y}\ \tau\ \Delta$$

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

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

subject to associativity and unitality laws.

Remark 4.11. Each functor $[\![\mathcal{Y}, -]\!]_s$ has a left adjoint $- \odot_s \mathcal{Y}$, and since maps $\mathcal{X} \nrightarrow [\![\mathcal{Y}, \mathcal{Z}]\!]_s$ are naturally isomorphic to maps $\mathcal{X} \odot_s \mathcal{Y} \nrightarrow \mathcal{Z}$, we recognise the internal-hom presentation of monoids. Categorically, a substitution monoid amounts to a monoid object in the skew-monoidal category $(\mathtt{Fam}_{\mathtt{s}}\ T, - \in -, - \odot_s -)$, see [11]. Preferring the internal hom to the monoidal product is motivated by mechanisation and programming, namely preferring working with subsets rather than quotients and with curried functions rather uncurried ones.

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 subject 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 another "exponentiation", this time of an unsorted family:

$$[\![-,-]\!] : \mathtt{Fam}_{\mathtt{s}}\ T \longrightarrow \mathtt{Fam}\ T \longrightarrow \mathtt{Fam}\ T$$
$$[\![\mathcal{X},\mathcal{Y}]\!]\ \Gamma \coloneqq \forall \Delta : \mathtt{Ctx}\ T,\ (\Gamma +\!\![\mathcal{X}]\!\!\rightarrow \Delta) \longrightarrow \mathcal{Y}\ \Delta$$

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

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:

$$\mathsf{sub} : \mathcal{X} \multimap [\![\mathcal{M}, \mathcal{X}]\!]$$

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 $(\mathtt{Fam}_{\mathtt{s}}\ T, - \in -, - \odot_s -)$. 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 aspect of 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, intended 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 $\mathcal{O} : T \longrightarrow \mathtt{Set}$ together with a map:

$$\mathsf{dom} : \forall \tau : T,\ \mathcal{O}\ \tau \longrightarrow \mathtt{Ctx}\ T$$

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

Remark 4.15. Concrete observation structures occur in the literature under various names: *ultimate pattern* [25], *continuation patterns* [35], *atomic value* [23], *abstract value* [20].

Example 4.16 ($\lambda^{\mathbf{rec}}$,). We recall elements of §3 in light of these abstract definitions. Since all types are negative in $\lambda^{\mathbf{rec}}$, we simply take as set of types T all $\tau : A \mid \neg A$. There is a single observation at each type but $\top$, 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 $A \longrightarrow B$, the domain carries fresh variables at $\neg B$ for the evaluation context and at A for the value.

Remark 4.17. Categorically, an observation structure is a map $T \to \mathtt{Set}/(\mathtt{Ctx}\ T)$. Instead of using a slice, observations could alternatively be presented as a sorted family, consistent with the view that patterns are a particular subset of values (notably with linear use of variables). For convenience in the mechanisation we prefer the above definition based on slices, consistent with the view that patterns are binders.

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

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

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

Its *domain* map is defined by:

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

It remains to incorporate the filling assignment:

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

$$\mathcal{N}_{\mathcal{O},\mathcal{V}}\ \Gamma \coloneqq \exists o : \mathcal{O}^\bullet\ \Gamma, \mathtt{dom}^\bullet\ o \multimap [\mathcal{V}] \rightarrowtail \Gamma.$$

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

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 $\mathcal{V}_{\mathcal{M}} : \mathtt{Fam_s}\ T$ of *values*,
- a substitution $\mathcal{V}_{\mathcal{M}}$-module $\mathcal{C}_{\mathcal{M}} : \mathtt{Fam}\ T$ of *configurations*,
- an observation structure $\mathcal{O}_{\mathcal{M}} : \mathtt{Obs}\ T$, and
- an evaluation structure $(\mathtt{eval}_{\mathcal{M}}, \mathtt{emb}_{\mathcal{M}})$ on $\mathcal{C}_{\mathcal{M}}$ and $\mathcal{N}_{\mathcal{O}_{\mathcal{M}},\mathcal{V}_{\mathcal{M}}}$.

Let $\mathtt{Machine}\ T$ 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 $\mathcal{N}_{\mathcal{O}_{\mathcal{M}},\mathcal{V}_{\mathcal{M}}}$
- We denote both substitution structures, on values and configurations, by $-[-]$, relying on context to disambiguate.
- We write $\Gamma \to_{\mathcal{M}} \Delta$ for $\Gamma \multimap [\mathcal{V}_{\mathcal{M}}] \rightarrowtail \Delta$.
- 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 $x : \tau \in \Gamma$ 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 (🔥). We define $\mathtt{eval}^{\mathrm{o}}_{\mathcal{M}} : C_{\mathcal{M}} \nrightarrow O^{\bullet}_{\mathcal{M}}$ to be the composite

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

Using this, let us at last define substitution equivalence.

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

$$\forall \gamma : \Gamma \to_{\mathcal{M}} \Delta,\ \mathtt{eval}^{\mathrm{o}}\ (u[\gamma]) \approx_{\mathscr{D}} \mathtt{eval}^{\mathrm{o}}\ (w[\gamma])$$

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 typing context Δ; second we do not compare normal forms, but only their projection to pointed observations, i.e., the head variable and the observation ($\mathtt{eval}^{\mathrm{o}}$ 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. In our example setting, this would amount to setting $\Delta = \emptyset, \neg\top$. 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 explicitly 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 $\mathcal{M} : \mathtt{Machine}\ T$, *evaluation respects substitution* iff the following two conditions hold.

- For all typing contexts Γ, configurations $u : C\ \Gamma$, and assignments $\gamma : \Gamma \to_{\mathcal{M}} \Delta$, 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, typing contexts Γ, Δ, observations $o : O\ \tau$ and assignments $\gamma : \mathtt{dom}\ o \to_{\mathcal{M}} \Gamma$ and $\delta : \Gamma \to_{\mathcal{M}} \Delta$,
$$(\mathtt{emb}\ (x \cdot o(\gamma))[\delta, x \mapsto x] = \mathtt{emb}\ (x \cdot o(\gamma[\delta]))$$
where $x : \tau$ 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 substitution

$\mathcal{V}$-module structure in the Kleisli category $(\mathtt{Fam}\, T)_{\mathscr{D}}$ of the delay monad:

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

Evaluation respects substitution iff the above is indeed a module structure, and if emb and eval are both *module morphisms*, between this module structure and the lifting of the substitution module structure of C from $\mathtt{Fam}\, T$ to $(\mathtt{Fam}\, T)_{\mathscr{D}}$. 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 [26]. 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 *client* and *server positions*, respectively. The definition then proceeds to postulate families of client moves from I to J, and server 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 Proponent on $\mathcal{G}$ consists of

- families $A : I \to \mathtt{Set}$ and $W : J \to \mathtt{Set}$ of active and waiting states, respectively,
- a partial "action" map, which to any active state $a : A\ i$ associates—when it is defined—either a final move in $\mathtt{final}_{\mathcal{G}}$, or a next move $m : \mathtt{move}^{P}_{\mathcal{G}}\ i$ in the client half-game of $\mathcal{G}$, together with a next waiting state in $W\ (\mathtt{next}_{\mathcal{G}}\ m)$, and
- a "reaction" map, which to any waiting state $w : W\ j$ and move $m : \mathtt{move}^{O}_{\mathcal{G}}\ j$ 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 $\mathcal{H} : \mathtt{HGame}\ I\ J$, we define two functors $(J \to \mathtt{Set}) \to (I \to \mathtt{Set})$, the *action* functor $[\![\mathcal{H}]\!]^{+}$ and the *reaction* functor $[\![\mathcal{H}]\!]^{-}$:

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

We may now define strategies.

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

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

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

- and a *reaction* morphism

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

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

Remark 5.10. The occurrence of the delay monad in the action morphism means that we allow Proponent 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 induced strategies entails substitution equivalence. Let us define weak bisimilarity.

Definition 5.11. Given two strategies $\mathcal{S}, \mathcal{T} : \mathtt{Strat}_{\mathcal{G}}$, a *weak bisimulation* $\alpha : \mathcal{S} \approx_{\mathcal{G}} \mathcal{T}$ 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 waiting states,

together with the following conditions:

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

where $u \lessapprox [R \to S] \gtrapprox v$ is shorthand for $\forall i\ x\ y,\ R\ i\ x\ y \to S\ i\ (u\ x)\ (v\ y)$, and we lift functors to relations in the straightforward way. Let *weak bisimilarity*, denoted by $\approx_{\mathcal{G}}$, be the largest weak 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 $\mathcal{M} = (\mathcal{V}, C, O, \mathtt{eval}, \mathtt{emb})$, 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. Naively, one then could define positions of the OGS game to consist of pairs of typing contexts.

However, as explained in the introduction, we refine this naive definition to build acyclicity into the model, and thus avoid the first kind of diverging composition. For this, we proceed by recording in the position the order in which variables were introduced. 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 the currently active, resp. waiting player:

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

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

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 waiting 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* $\mathtt{HOGS}$ of type

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

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

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

Furthermore, the *OGS game* $\mathtt{OGS}$ is defined by

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

5.3 The machine strategy

Recalling that we have fixed a language machine $\mathcal{M} : \mathtt{Machine}\ T$ at the beginning of §5.2, we now introduce the strategy, called the *machine strategy*, induced by $\mathcal{M}$ and Δ in the game $\mathtt{OGS}$. Unfolding

definitions, such a strategy should comprise two families $\widetilde{\mathcal{M}}^+$ and $\widetilde{\mathcal{M}}^-$, and morphisms

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

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 waiting player, in terms of the variables shared *by* them.
- Waiting 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 active player, in terms of the variables shared by them.
- 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 $x \cdot o(\delta)$; the strategy then 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 waiting 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, in accordance with interleaved contexts as defined in §5.2, we build acyclicity into the model by refining this naive definition. This is possible for assignments since, at each step, the value stored by some player for variables in a Γ_i may only depend 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 *active* and *waiting interleaved assignments* $\mathtt{Env}^+, \mathtt{Env}^-\colon \mathtt{ICtx}\ T \to \mathtt{Set}$ by mutual induction as follows.

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

Remark 5.17. Concretely, for any even interleaved context $\Phi = (\Gamma_1, \ldots, \Gamma_{2n})$, active and waiting interleaved assignments 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 waiting interleaved assignments 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 interleaved assignments thus end with $\cdot$: they do not store the values of variables introduced by the last move. Symmetrically, waiting interleaved assignments end with a proper assignment γ_i. Beware, of course, that the active player in an even interleaved context is Proponent, while the active player in an odd interleaved context is Opponent.

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

Definition 5.18 ( ). The *collapsing* functions for interleaved assignments are defined by mutual induction as follows.

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

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

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

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

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 $\widetilde{\mathcal{M}} : \mathtt{Strat}_{\mathsf{OGS}}$ is defined as follows:

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

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

- the reaction morphism is defined by

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

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. waiting) machine strategy states ( ),

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

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, weak bisimilarity of induced strategies entails substitution equivalence:

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

where

- $\approx^+_{\mathsf{OGS}}$ denotes the positive component of weak 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.

First in §6.1 we give an intuitive characterisation of composition and show how soundness follows from two main properties of composition: adequacy and congruence. As proving adequacy involves intricate coinductive reasoning, we will need to do an excursion into the theory of fixed points of equations in the delay monad and go through two variants of the construction of composition. In §6.2 we give the composition equations and construct a first naive definition of composition for which we prove congruence. Then in §6.3 we introduce the notion of *eventually guarded* equations which enjoy uniqueness of fixed points and use this to give an alternative characterisation of composition, for which we prove adequacy.

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

6.1 Soundness from composition

Composition makes sense for any game $\mathcal{G}$: although we will only use it in the specific case of composing two machine strategies in the OGS game, we present it in full generality for clarity.

Given a game $\mathcal{G}$, a strategy $\mathcal{S} : \mathtt{Strat}_{\mathcal{G}}$ and a counter-strategy $\mathcal{T} : \mathtt{Strat}_{\mathcal{G}^\perp}$, the composition of $\mathcal{S}$ and $\mathcal{T}$ should take as arguments an active state in $\mathcal{S}^+\ i$ and a waiting state in $\mathcal{T}^-\ i$ over the same active position i in $\mathcal{G}$, and return (the delay computation of) a final move $\mathtt{final}_{\mathcal{G}}$. It should thus be a map

$$\forall i, \mathcal{S}^+\ i \to \mathcal{T}^-\ i \to \mathscr{D}\ (\mathtt{final}_{\mathcal{G}}).$$

Intuitively, given any $u : \mathcal{S}^+\ i$ and $v : \mathcal{T}^-\ i$, the composition $u \parallel v$ is an iterative process that should work as follows. We start by evaluating the action morphism $\mathtt{play}_{\mathcal{S}}\ u$.

(1) If $\mathtt{play}_{\mathcal{S}}\ u$ diverges, then so does $u \parallel v$.
(2) If $\mathtt{play}_{\mathcal{S}}\ u$ returns a final move $r : \mathtt{final}_{\mathcal{G}}$, then so does $u \parallel v$.
(3) If $\mathtt{play}_{\mathcal{S}}\ u$ returns a pair (m, u') of a client move and a subsequent waiting state, then composition passes this move to the reaction morphism of v and continues. I.e., $u \parallel v$ continues as $\mathtt{coplay}_{\mathcal{T}}\ v\ m \parallel u'$, swapping roles between $\mathcal{S}$ and $\mathcal{T}$.

Specialised to the case of the OGS game and the composition of instances of the machine strategy, composition is thus a map

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

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 $c : C\ \Gamma$ and assignments $\gamma : \Gamma \to_{\mathcal{M}} \Delta$, evaluating and observing $c[\gamma]$ yields the same result as composing $[\![c]\!]^+$ and $[\![\gamma]\!]^-$, up to weak bisimilarity, i.e., $\mathtt{eval}^\circ_{\mathcal{M}}\ (c[\gamma]) \approx_{\mathscr{D}} [\![c]\!]^+ \parallel [\![\gamma]\!]^-$.
(2) Weak bisimilarity is a *congruence* (🔥) for composition iff, for any weakly bisimilar active states $s_1 \approx^+_{\mathrm{OGS}} s_2$ and compatible weakly bisimilar waiting states $t_1 \approx^-_{\mathrm{OGS}} t_2$, we have $s_1 \parallel t_1 \approx_{\mathscr{D}} s_2 \parallel t_2$.

Remark 6.2. It might not be obvious that adequacy type checks at all. But by construction $[\![c]\!]^+ \parallel [\![\gamma]\!]^-$ lives in $\mathscr{D}\ (O^\bullet\ \Delta)$, and so does $\mathtt{eval}^\circ_{\mathcal{M}}\ (c[\gamma])$ (Definition 4.23).

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 $c_1, c_2 : C\ \Gamma$ with weakly bisimilar interpretations $[\![c_1]\!]^+$ and $[\![c_2]\!]^+$, and any assignment $\gamma : \Gamma \to_{\mathcal{M}} \Delta$, we have

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

hence $c_1 \approx_{\mathrm{SUB}} c_2$, 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.

In the previous section we have loosely described composition as an iterative process and given a self-referencing description. What is meant is that being a monad of partial functions, the delay monad enjoys an *iteration* operator, constructing fixed points for arbitrary sets of equations, w.r.t. weak bisimilarity. Intuitively, this operator will start by unfolding the definition, and proceed by case analysis: if unfolding yields a self call, then iteration performs a τ step and continues, otherwise it ends with the given value. Let us formalise this intuition:

Definition 6.4 ( ). An *equation* consists of a set X of *variables*, a set Y of *constants*, and a *definition* function $X \to \mathscr{D}\ (Y + X)$.

Definition 6.5 ( ). Given an equation $f : X \to \mathscr{D}\ (Y+X)$, the *iteration* of f is a map $f^\dagger : X \to \mathscr{D}\ Y$ given coinductively by:

$$f^\dagger\ x \coloneqq f\ x \gg\!= \begin{cases} \mathtt{inl}\ y & \mapsto \eta\ y \\ \mathtt{inr}\ x' & \mapsto \tau\ (f^\dagger\ x'). \end{cases}$$

Proposition 6.6. *Given an equation $f : X \to \mathscr{D}\ (Y + X)$, the iteration of $f^\dagger$ is a fixed point of f, i.e., it satisfies the following property.*

$$\forall x, f^\dagger\ x \approx_{\mathscr{D}} f\ x \gg\!= [\eta, f^\dagger]$$

Proof. By direct coinduction. □

Using this, we can readily describe one step of composition by an equation and obtain composition as its iteration.

Definition 6.7 ( ). The composition equation for machine strategies in the OGS game is given by the following map.

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

With a slight twist due to uncurrying we can now define composition.

Definition 6.8 ( ). *Naive composition* is defined as $- \parallel - \coloneqq \mathtt{comp\text{-}eqn}^\dagger\ (-,-)$.

Proposition 6.9 ( ). *Weak bisimilarity is a congruence for naive composition.*

Proof. By coinduction: the binary relation on $\mathscr{D}(\mathcal{O}^\bullet\ \Delta)$ given by all pairs $(s_1 \parallel t_1, s_2 \parallel t_2)$ such that $s_1 \approx^+_{\mathrm{OGS}} s_2\ t_1 \approx^-_{\mathrm{OGS}} t_2$, is a weak bisimulation. □

6.3 Proving adequacy with eventual guardedness

In order to prove adequacy, we have to give a weak bisimulation between $\mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\ (c[\gamma])$ and $[\![c]\!]^{+} \parallel [\![\gamma]\!]^{-}$. When facing such an equational proof where one of the members is defined as a fixed point (here the right-hand side, composition), the prime reasoning scheme is uniqueness of fixed points: assuming the composition equation has a unique fixed point, we could deduce adequacy from the fact that $\mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\ (-[-])$ is a fixed point of the composition equation. Indeed, if substituting-then-evaluating-then-observing is a fixed point, by uniqueness it is then bisimilar to composition, which has been constructed as a fixed point. A first point to make more precise is the notion of equivalence considered when talking about fixed points. The statement of adequacy hints at weak bisimilarity, but in fact uniqueness of fixed points w.r.t. weak bisimilarity is not provable in our situation. Instead, from now on we will work mainly with strong bisimilarity, only connecting back to weak bisimilarity later on, with Proposition 6.18.

Happily, substituting-then-evaluating-then-observing is indeed a fixed point of the composition equation w.r.t. strong bisimulation. To make this precise we need to strengthen the statement slightly: $\mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\ (-[-])$ takes as input a pair of *initial* OGS configurations consisting of a language configuration and an assignment, while the composition equation also operates on non-initial configurations. We thus need to generalise substitution to such non-initial configurations. We proceed by defining zipping functions on interleaved assignments, which map pairs of complementary interleaved assignments to plain assignments in the final typing context:

Definition 6.10 ( ). The *zipping* functions for interleaved assignments are defined by mutual induction as follows.

$$
\begin{aligned}
- \curvearrowleft - &: \mathsf{Env}^{+}\ \Phi \to \mathsf{Env}^{-}\ \Phi \to \downarrow^{+}\Phi \to_{\mathcal{M}} \Delta \\
\varepsilon \curvearrowleft \varepsilon &:= \mathtt{elim}_{\emptyset} \\
(a, \cdot) \curvearrowleft (b, \gamma) &:= [b \curvearrowright a, [\mathrm{id}, b \curvearrowleft a] \circ \gamma], \\
- \curvearrowright - &: \mathsf{Env}^{+}\ \Phi \to \mathsf{Env}^{-}\ \Phi \to \downarrow^{-}\Phi \to_{\mathcal{M}} \Delta \\
\varepsilon \curvearrowright \varepsilon &:= \mathtt{elim}_{\emptyset} \\
(a, \cdot) \curvearrowright (b, \gamma) &:= b \curvearrowleft a
\end{aligned}
$$

We may now define a generalisation of the above zipping-then-evaluating-then-observing map to non-initial states of the machine strategy.

Definition 6.11 ( ). We define z-e-obs as follows.

$$
\begin{aligned}
&\mathtt{z\text{-}e\text{-}obs} : \forall \Phi, C\ (\Delta + \downarrow^{+}\Phi) \times \mathsf{Env}^{+}\ \Phi \to \mathsf{Env}^{-}\ \Phi \to \mathscr{D}\ (O^{\bullet}\ \Delta) \\
&\mathtt{z\text{-}e\text{-}obs}\ (c, a)\ b \coloneqq \mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\ (c[\mathrm{id}, a \curvearrowleft b])
\end{aligned}
$$

The above definition readily satisfies $\mathtt{z\text{-}e\text{-}obs}\ [\![c]\!]^{+}\ [\![\gamma]\!]^{-} = \mathtt{eval}^{\mathrm{o}}\ (c[\gamma])$.

Proposition 6.12 ( ). *If for $\mathcal{M}$ evaluation respects substitution (Definition 4.26), then uncurried zipping-then-evaluating-then-observing is a strong fixed point of the composition equation, i.e., defining* $\mathtt{z\text{-}e\text{-}obs}'\ (u, v) \coloneqq \mathtt{z\text{-}e\text{-}obs}\ u\ v$, *for all* $a : \exists \Phi, \widetilde{\mathcal{M}}^{+}\ \Phi \times \widetilde{\mathcal{M}}^{-}\ \Phi$,

$$
\mathtt{z\text{-}e\text{-}obs}'\ a \approx_{\mathscr{D}} \mathtt{comp\text{-}eqn}\ a \gg\!= [\eta, \mathtt{z\text{-}e\text{-}obs}']
$$

While the proof of the above proposition is tedious, it is still a direct coinduction. We will only mention that the core reasoning step is provided by the language machine hypothesis stating that evaluation respects substitution and that the core argument for rearranging assignments is given by the following technical lemma.

Proposition 6.13 (🔥). *Zipping is a fixed point of collapsing, i.e. for all* $a : \mathsf{Env}^+\ \Phi$ *and* $b : \mathsf{Env}^-\ \Phi$, *the following equations hold:*

$$a \curvearrowleft b = [id, (a \curvearrowright b)] \circ \downarrow^- b$$
$$a \curvearrowright b = [id, (a \curvearrowleft b)] \circ \downarrow^+ a$$

Proof. By direct induction on Φ. □

Since zipping-then-evaluating-then-observing is a fixed point of the composition equations, adequacy will follow if we prove uniqueness of fixed points for the composition equation. Even w.r.t. strong bisimilarity, uniqueness of fixed points in the delay monad is a strong property which does not hold for every equation. Let us start with the following counter-example.

Example 6.14. Letting $\mathbb{1} = \{\star\}$ and $\mathbb{2} = \{\mathtt{true}, \mathtt{false}\}$ denote respectively the one and two-element sets, consider the map $e : \mathbb{1} \to \mathscr{D}\ (\mathbb{2} + \mathbb{1})$ mapping any $x : \mathbb{1}$ to $\eta\ (\mathtt{inr}\ x)$. This map can be iterated and intuitively encodes the recursive definition of a map $f : \mathbb{1} \to \mathscr{D}\ \mathbb{2}$ caught in an infinite loop.

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

$$f\ x \approx_{\mathscr{D}} e\ x \gg\!= \begin{cases} \mathtt{inl}\ y & \mapsto \eta\ y \\ \mathtt{inr}\ x' & \mapsto f\ x', \end{cases}$$

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

This equation is solved with the standard iteration operator by $e^\dagger\ x \coloneqq \tau\ (e^\dagger\ x)$, i.e., $f^\dagger\ x \coloneqq \tau^\omega$. But in fact it is also solved by any $f : \mathbb{1} \to \mathscr{D}\ \mathbb{2}$, in particular $f\ x \coloneqq \eta\ \mathtt{true}$ and $f\ x \coloneqq \eta\ \mathtt{false}$, so uniqueness cannot hold.

The reason why uniqueness fails in this example is because the equation does not do anything before the recursive call: no information is gained by computing one step. In the context of coinduction, the idea that new information should be gained before looping is called *productivity* and the most common family of criteria enforcing it are syntactic *guardedness* criteria.

Coq, like other proof assistants with support for coinduction, only accepts looping coinductive definitions provided their body is syntactically guarded. Then, a natural unfolding equation between the definition and its body will hold w.r.t. strong bisimilarity, as it is the extensional notion of equality at coinductive types. As such, the strong, guarded fixed points that Coq allows us to write are always unique w.r.t. strong bisimilarity. Syntactic guardedness is the reason why the fixed point operator $-^\dagger$ inserts a τ node at each looping point. This constructs the unique fixed point of the guarded equation $e' \coloneqq e \gg\!= [\eta, \tau \circ \eta]$ w.r.t. strong bisimilarity. As e' is pointwise weakly bisimilar to e, a byproduct is that this also constructs a fixed point for e w.r.t. weak (but not strong) bisimilarity.

Considering these remarks, and as we were able to prove that zipping-then-evaluating-then-observing is a *strong* fixed point of composition, it seems natural to continue by seeking to construct the unique strong fixed point of the composition equation. The last obstacle is that the composition equation is neither syntactically nor even semantically guarded: for some composable OGS state pairs, one composition step might not produce any information. Luckily, with one last additional hypothesis on the language machine, we will be able to prove a weaker yet sufficient *eventual guardedness* property.

In the following, we recall guardedness formally, introduce eventual guardedness, and prove that it enables the construction of a unique fixed point w.r.t. strong bisimilarity. Then, we discuss why the composition equation is not guarded but only eventually guarded, introduce the last missing hypothesis, and finally conclude by proving eventual guardedness of composition.

Definition 6.15 ( ).

- An element $u : \mathscr{D}\ (Y + X)$ is *guarded* if it is not of the form $\eta\ (\mathtt{inr}\ x)$.
- Similarly, an equation $e : X \to \mathscr{D}\ (Y + X)$ is *guarded* if for all x, $e\ x$ is guarded.
- An equation $e : X \to \mathscr{D}\ (Y + X)$ is *pointwise eventually guarded* if for all x, there exists an $n : \mathbb{N}$ such that $e^n\ x$ is guarded, where

$$e^0 x \coloneqq \eta\ (\mathtt{inr}\ x)$$

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

Proposition 6.16 ( ). *All guarded equations admit a unique fixed point w.r.t. strong bisimilarity.*

Proof. Since the formal proof amounts to constructing a syntactically guarded definition given a (semantic) guardedness proof, it translates trivially on paper. Concretely, we define an iteration operator similar to $-^\dagger$, which does pattern-matching on the evaluated equation $e\ x$, making the guard explicit and eliminating the unguarded case by absurdity on the guardedness proof. The fixed point property and uniqueness are by direct coinduction. □

Proposition 6.17 ( ). *All eventually guarded equations admit a unique fixed point w.r.t. strong bisimilarity.*

Proof. Since, for all x, an eventually guarded equation e can be pointwise unrolled a finite number n_x of times into a guarded element, $e^{n_x}\ x$, we construct the eventually guarded fixed point as the guarded fixed point of $e'\ x \coloneqq e^{n_x}\ x$. Fixed point property w.r.t. the original equation e and uniqueness are by direct coinduction. □

Let us finally connect these new fixed point constructions with the previous iteration operator.

Proposition 6.18 ( ). *For any guarded (resp. eventually guarded) equation e, the fixed point of e w.r.t. strong bisimilarity is pointwise weakly bisimilar to the fixed point w.r.t. weak bisimilarity $e^\dagger$.*

Proof. By direct coinduction. □

It remains to prove eventual guardedness of the composition equation. Before introducing the missing hypothesis, let us give an intuitive understanding of the problematic case.

Reasoning backwards, given $(c, u) : \widetilde{\mathcal{M}}^+\ \Phi$ and $v : \widetilde{\mathcal{M}}^-\ \Phi$, for the composition equation $\mathtt{comp\text{-}eqn}\ ((c, u), v)$ *not* to be guarded, it must be that $\mathtt{play}_{\widetilde{\mathcal{M}}}\ (c, u) \approx_{\mathscr{D}} \eta\ (\mathtt{inr}(m, w))$ for some m and w. Unfolding the definition of $\mathtt{play}$ for the machine strategy, it means that $\mathtt{eval}\ c \approx_{\mathscr{D}} \eta\ (x \cdot o(\gamma))$ for some $x \in \downarrow^+ \Phi$, some o and γ. In other words, c is already in normal form $x \cdot o(\gamma)$, where x is not a final variable, but one introduced by Opponent.

While not possible at the beginning of the game, since in an initial configuration Opponent has not yet introduced any variables, this kind of configuration are routinely attained during plays. One such case is when one of the interleaved assignments contains a mapping $x \mapsto y$, i.e., when a player, say P, has introduced a variable x, mapping to a value which already was a variable y given by the other player, O. In this case, whenever O sends an observation o targeting x, P will resume on a normal configuration $y \cdot o$, and immediately address the same observation back to O, targeting y.

As stated in the introduction, we call this case *chattering*. Getting rid of the possibility of an infinite sequence of such chattering is why we have introduced the complex notion of interleaved assignments. Indeed, interleaved assignments enforce that stored values only refer to Opponent variables introduced *before*. Hence, after one step of chattering, the new variable will be strictly older than the old one. Since the past history is finite at any given point, only a finite amount of

such chattering is possible, after which the stored value will necessarily be either a final variable or a non-variable value (a *proper* value).

It now remains to argue eventual guardedness at $((c, u), v)$, in the case where c is a normal form, either of the form $x \cdot o(\gamma)$ for x a final variable, or of the form $v \cdot o(\gamma)$ where v is a proper value. The first case is trivially guarded since it will end the whole composition, but the second is problematic. Indeed, without further hypothesis on the language machine we are stuck. Let us look at an example.

Example 6.19. 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 waiting player, say O. At first, the active player, say P, creates a variable, say y, for $\lambda x.v$, and plays $(\alpha', \langle\, y \mid \Box \,\rangle)$. Roles are then switched, O becomes active and looks up its stored value for α', say $\bullet w'; \pi'$ for some w' and π'. This is indeed a *proper* value, but applying it to the argument y yields the configuration $\langle\, y \mid \bullet w'; \pi' \,\rangle$ which is still in normal form: precisely our problematic case.

Continuing, O plays the observation $\langle\, \Box \mid \bullet z'; \beta' \,\rangle$ targeted at y, after which P becomes active again, now with a configuration of the form $\langle\, \lambda x.v \mid \bullet z'; \beta' \,\rangle$, finally generating a redex, hence guarding the composition with a τ node.

From this example, we learn that while this last problematic case is possible, it is usually resolved by a couple more steps of composition. As in the general case no such reasoning is possible, we introduce our last hypothesis on the language machine, intuitively stating that there is no infinite chain of these last bad cases.

Definition 6.20 (🔥). Let $\prec$ denote the binary relation on $\exists \tau : T,\ \mathtt{Obs}\ \tau$ defined by $(\tau_2, o_2) \prec (\tau_1, o_1)$ iff there is a proper 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 proper value $v : \mathcal{V}\ \Gamma\ \tau_1$, assignments γ, δ and a variable i such that $\mathtt{eval}\ (v \cdot o_1(\gamma)) \approx_{\mathscr{D}} \eta\ (i \cdot o_2(\delta))$.

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

We can now prove eventual guardedness and conclude our soundness proof.

Proposition 6.21 (🔥). *If the language machine $\mathcal{M}$ has* finite redexes, *then* comp-eqn *is eventually guarded.*

Proof. As the Coq proof is a rather tedious translation of the intuitive case reasoning done above, we only sketch the proof here.

First we change the statement slightly, to proving instead that for all x, o, u, v and some renaming r, comp-eqn $((x \cdot o(r), u), v)$ is eventually guarded, from which we conclude by case distinction on the active configuration.

Then, by induction on the accessibility proof of the relation $\prec$ at o, introduce the induction hypothesis. Then, by induction on the *age* of x–the number of moves after which x was introduced—introduce the induction hypothesis.

Then, $x \cdot o(r)$ is in normal form, so composition is not guarded yet. Let us unfold one step of comp-eqn. We now need to prove that comp-eqn $((w \cdot o(r'), v), (u, r))$ is eventually guarded, where w is the value obtained by looking x up in v and r' is some renaming.

Then by case on the value w.

- If w is a final variable, the composition ends and is guarded.
- If w is a non-final variable y, then y is strictly older than x, and we apply the second induction hypothesis.
- Else w is a proper value. If $w \cdot o(r')$ is some normal form $y \cdot o'(\gamma)$, then we can exhibit $o' \prec o$ and apply the first induction hypothesis. Else it is not a normal form, hence there is a redex, i.e. composition is guarded.

□

Let us write $- \|_g -$ for the strong fixed point of the now eventually guarded composition equation.

Proposition 6.22 (🔥). *If the language machine $\mathcal{M}$ has* finite redexes*, then the strong fixed point of* `comp-eqn` *is adequate.*

Proof. Let $c : C\ \Gamma$ and $\gamma : \Gamma \to \Delta$.

$$\begin{array}{lll} [\![c]\!]^+ \| [\![\gamma]\!]^- & \approx_{\mathscr{D}} & [\![c]\!]^+ \|_g [\![\gamma]\!]^- \qquad \text{by Proposition 6.18} \\ & \cong_{\mathscr{D}} & \texttt{z-e-obs}\ [\![c]\!]^+\ [\![\gamma]\!]^- \qquad \text{by Propositions 6.12 and 6.17} \\ & = & \texttt{eval}^{\mathrm{o}}\ (c[\gamma]) \qquad \text{by definition} \end{array}$$

Hence $\texttt{eval}^{\mathrm{o}}\ (c[\gamma]) \approx_{\mathscr{D}} [\![c]\!]^+ \| [\![\gamma]\!]^-$, i.e. composition is adequate. □

Theorem 6.23 (🔥). *If $\mathcal{M}$ has finite redexes and evaluation respects substitution, then weak bisimilarity of induced OGS strategies is sound w.r.t. substitution equivalence.*

Proof. By Propositions 6.3,6.9 and 6.22. □

Let us comment on our newly introduces hypothesis, finite redexes. The first observation is that the appearance of this hypothesis was a surprise to us, as to the best of our knowledge nothing like this was present in the particular instances of OGS soundness proofs in the literature. Indeed, in all standard instances we could think of, the $\prec$ relation only has chains of length at most 1. In fact for properly focused languages—such as call-by-push-value and polarised $\mu\tilde{\mu}$—with no arguable spurious redex-searching phase in the evaluation, the hypothesis is properly trivial as observing a proper value always creates a redex. Yet one can easily create contrived examples such as the following where $\prec$ has arbitrarily long chain.

Example 6.24. 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,

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

and the evaluation rules as follows.

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

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

$$\begin{array}{l} \langle\, \Box \mid \alpha' \,\rangle \succ \langle\, \mathbf{yield}_n \mid \Box \,\rangle \succ \langle\, \mathbf{yield}_n \mid x_1 \cdot \Box \,\rangle \\ \qquad \succ \langle\, \mathbf{yield}_n \mid x_1 \cdot x_2 \cdot \Box \,\rangle \succ \ldots \succ \langle\, \mathbf{yield}_n \mid x_1 \cdot x_2 \ldots x_{n-1} \cdot \Box \,\rangle \end{array}$$

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

Perhaps this example gives a hint for our naming choice "finite redexes". Indeed as we can see, the length of chains in $\prec$ is directly linked to the syntactical depth of the premises of reduction rules. As replacing a variable with a proper value necessarily increases the depth of a term, after

some finite amount of such steps, a reduction rule will be fired. As such, $\prec$ being well-founded can be interpreted as enforcing that no premise of a reduction rule has infinite depth: every redex is of finite size.

By virtue of syntax concretely being an inductive construction, or at least a finite data structure, we believe this hypothesis to be very easy to verify in practice and could not imagine a meaningful case where it would not be satisfied. Yet this hypothesis is key to a soundness theorem oblivious to the internal structure of the language syntax and the language evaluator. Indeed, our whole development rests only on semantic hypotheses on the language machine and does not make any assumption on how the various components have been *constructed*.

7 IMPLEMENTATION DISCUSSION

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

Differences 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 operation embedding $v \cdot o(\gamma)$ triplets into configurations. This new operator can be thought of as applying the observation o with arguments γ to the value v. Both are mutually expressible, but for clarity and ease of exposition we have kept the more minimal embedding in the paper.

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 (both *action* and *reaction*). In the Coq mechanisation, instead of working with such coalgebras, we opt to work with *elements of the final coalgebra*, that is, with a coinductive type. In doing so, we introduce an indexed variant of interaction trees ([34],) which has been instrumental to precisely represent strategies. Again, both representations are extensionally isomorphic. The coinductive representation is much easier to program with in a proof assistant as it eludes the two carrier families. But as even in the mechanisation a couple constructions are more easily dealt with by introducing coalgebras anyway, we have eliminated any mention of the final coalgebra in the paper.

Axioms and executability. All our results are admit free, relying only on the Coq's standard library's statement of axiom K. This axiom, while probably not strictly required, has been of great use to ease dependent pattern matching on indexed families, which we have used liberally, following the correct-by-construction approach to certified development.

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

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 [12, 23, 24]; (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 [18, 21]. 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 [23], 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 waiting state into a language machine configuration was studied in [21, 24]. 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 [24] 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 [26]. 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, 17] 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.

Deducing bisimilarity of two LTS from the fact that they satisfy the same recursive equation, and that this equation admits a unique solution (up-to bisimilarity), is a standard technique in process calculi, introduced by Hoare [15] and Milner [32]. We would be interested in understanding possible connections with our result on uniqueness of solution for eventually guarded equations introduced here, exploring recent results on the uniqueness of solution for equations whose infinite unfolding never produces a divergence [9].

Introduced by Elgot [10], monads with iteration operators and their equational properties have been studied widely with varied requirements on equations verified by the iteration and on guardedness criteria [13, 29, 30]. There has been recent work on unifying the theory of guarded and unguarded iteration, provinding an axiomatisation of abstract guardedness criteria [14]. But we could not find any mention of eventual guardedness, waranting further study. In this context the construction of interaction tree [34] appears as an instance of the coinductive generalized resumption monad [13, 33] which is known to have unique strong fixed points of guarded equations and also arbitrary weak fixed points.

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] Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. 2019. Divergence and unique solution of equations. *Log. Methods Comput. Sci.* 15, 3 (2019). https://doi.org/10.23638/LMCS-15(3:12)2019

[10] Calvin C. Elgot. 1975. Monadic Computation And Iterative Algebraic Theories. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 175–230. https://doi.org/10.1016/S0049-237X(08)71949-9

[11] 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

[12] 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

[13] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Julian Jakob. 2018. Unguarded Recursion on Coinductive Resumptions. *Log. Methods Comput. Sci.* 14, 3 (2018). https://doi.org/10.23638/LMCS-14(3:10)2018

[14] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg. 2017. Unifying Guarded and Unguarded Iteration. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10203)*, Javier Esparza and Andrzej S. Murawski (Eds.). 517–533. https://doi.org/10.1007/978-3-662-54458-7_30

[15] C. A. R. Hoare. 1985. *Communicating Sequential Processes*. Prentice-Hall.

[16] 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

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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] Stefan Milius. 2005. Completely iterative algebras and completely iterative monads. *Inf. Comput.* 196, 1 (2005), 1–41. https://doi.org/10.1016/J.IC.2004.05.003

[30] Stefan Milius and Tadeusz Litak. 2017. Guard Your Daggers and Traces: Properties of Guarded (Co-)recursion. *Fundam. Informaticae* 150, 3-4 (2017), 407–449. https://doi.org/10.3233/FI-2017-1475

[31] 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

[32] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

[33] Maciej Piróg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. In *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014 (Electronic Notes in Theoretical Computer Science, Vol. 308)*, Bart Jacobs, Alexandra Silva, and Sam Staton (Eds.). Elsevier, 273–288. https://doi.org/10.1016/J.ENTCS.2014.10.015

[34] 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

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