An abstract, certified account of operational game semantics

Peio Borthelle$^{1[0009-0006-7733-7439]}$, Tom Hirschowitz$^{1[1234-5678-9012]}$, Guilhem Jaber$^{2[0009-0006-7149-5073]}$, and Yannick Zakowski$^{3[0003-4585-6470]}$

[1] LAMA
[2] Nantes Université
[3] Inria Lyon

Abstract. 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.

1 Introduction

Normal form bisimulation is a technique for proving contextual equivalence of programs in various λ-calculi. Although it is generally finer than contextual equivalence, its practical value resides in the fact that it is often easier to establish on concrete examples than other such techniques, such as applicative or environmental bisimulation.

Let us briefly explain why. All three techniques proceed by defining a notion of label, an interpretation of programs as labelled transition systems (LTSs), and then comparing the interpretations of programs w.r.t. weak bisimilarity. However, the involved labels are very different. Indeed, applicative or environmental labels may contain arbitrary values, while normal form labels are restricted to so-called *ultimate patterns*. This means that they may be, e.g., tuples or elements of sum types, but cannot contain λ-abstractions. In a typed setting, in particular, all terms of functional type contained in any ultimate pattern must be variables. Normal form labels thus contain a very limited class of terms.

In order for it to be useful, normal form bisimulation should be *sound*, i.e., at least as fine as contextual equivalence. One standard method for proving soundness goes through an intermediate LTS model, with labels similar to the normal form ones, but a different interpretation, called *operational game semantics* [19, 21] (OGS). This induces a different equivalence, say *operational game bisimulation*. One then proves that normal form bisimulation is at least as fine as operational game bisimulation, which is in turn at least as fine as contextual equivalence. Although the first step is mostly straightforward, the second one is difficult, notably because it involves

- extending the OGS interpretation to suitable contexts, and
- soundly reflecting the syntactic operation of context application at the level of LTSs.

Because such soundness proofs are highly non-trivial, it seems useful to design an abstract version, covering as many existing cases as possible, and hopefully also future ones.

A few authors have started to explore this direction. Notably, Levy and Staton [22] offer a high-level categorical framework. More recently, Laird [20] proposes a unifying framework for OGS, in which he proves that operational game bisimilarity 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 for OGS, mechanised in Coq. We thus contribute to both unification and mechanisation of normal form bisimulation. Our contributions to unification are as follows.

- We introduce an abstract notion of language with evaluator, called a *language machine*, which notably covers several variants of $\overline{\lambda}\mu\tilde{\mu}$-calculus [4, 5].
- 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 called *substitution 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 (🐓)[4] to link definitions and theorems to their mechanised counterpart. The Coq development is inspired by Levy and Staton's transition systems over games [22], 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 contrasts 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 [4] (🐓) and the polarised System D (🐓) from Downen and Ariola [5].

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

- We implement (🐓) an indexed variant of the *interaction trees* library [28], which we use to define LTSs coinductively — as opposed to the more traditional, relational definition. However, in this extended abstract, we focus more on the math than on the Coq implementation, so interaction trees do not appear (see Remark 9).
- 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.

Plan Before diving into the details, let us provide a high-level overview of the technical development in §2. In §3, we explain substitution equivalence, our abstract approach to contextual equivalence, on a concrete example language. In §4, we then introduce abstract language machines, construct the OGS model of any language machine, and state our soundness result. Finally, we provide a comparison with the existing literature in §5, and conclude and give some perspectives in §6.

2 Overview

2.1 Axiomatising contextual equivalence as substitution equivalence

Soundness proofs for normal form bisimulation can be established by the following chain of inclusions

$$\begin{aligned}\text{normal form bisimulation} &\subseteq \text{OGS bisimulation}\\ &\subseteq \text{CIU equivalence}\\ &= \text{contextual equivalence.}\end{aligned}$$

Compared to contextual equivalence, CIU (*Closed Instantiation of Use*) equivalence restricts the shape of contexts that are considered. This idea of restricting contexts while keeping the same discriminating power was first explored by Milner [26]. This idea was then systematized by Mason and Talcott, who introduced CIU equivalence and proved that it coincides with contextual equivalence [23].

In this work, we focus on the middle inclusion:

$$\text{OGS bisimulation} \subseteq \text{CIU equivalence.}$$

In order to propose an abstract version of it, we start by streamlining the usual concrete presentation of languages and CIU equivalence.

Standard CIU equivalence checks that two programs, say p and q, behave the same under any closed instantiation σ of their free variables, in any closed evaluation context E of some fixed, basic type like the booleans[5]:

$$E[p[\sigma]] \cong E[q[\sigma]],$$

[5] It is a bit unfortunate that substitution $p[\sigma]$ and context application $E[p]$ have the same notation. It is, however, so common, that we stick to the usual notation.

for some sensible notion of observation of closed boolean programs.

This involves both substitution and context application, which is a bit clumsy. In order to avoid having two distinct operations in the abstract setting, we switch to a presentation that unifies them. This presentation is based on abstract machines: one evaluates *configurations* rather than programs, where a configuration consists of a pair $\langle p \mid E \rangle$ of a program p and an evaluation context E. In particular, instead of comparing programs p and q as before, we now compare configurations $\langle p \mid \alpha \rangle$ and $\langle q \mid \alpha \rangle$, where α denotes a fresh *context variable*. And now CIU equivalence is merely a matter of substitution: combining any substitution σ with $\alpha \mapsto E$, we get

$$\langle p \mid \alpha \rangle[\sigma, \alpha \mapsto E] = \langle p[\sigma] \mid E \rangle,$$

and similarly for q.

This is presented in detail in §3, but, briefly, it involves a change of typing paradigm: evaluation contexts are typed like continuations, with "negated" types. E.g., if p has type A, then α has type $\neg A$. And in a configuration $\langle p \mid E \rangle$, p and E have opposite types, e.g., $p : A$ and $E : \neg A$. We tend to use $A, B, \ldots$ to range over simple types, and τ to range over the disjoint union of simple types and formally negated simple types.

In this presentation style, evaluation contexts are written inside-out in a stack-like manner, and reduction rules "push" the evaluation context from p to E, and "pop" it when needed, e.g.,

$$\langle e_1\, e_2 \mid E \rangle \to \langle e_1 \mid (\bullet e_2); E \rangle \qquad (1)$$

$$\langle \lambda x.e \mid (\bullet a); E \rangle \to \langle e[x \mapsto a] \mid E \rangle \qquad (2)$$

$$\ldots,$$

Here, $(\bullet e_2); E$ is the inside-out analogue of the evaluation context $E[\Box\, e_2]$. Thus:

- the first rule is "searching" for the next redex in the function part of the application $e_1\, e_2$, storing the argument part in the evaluation context, while
- in the second rule, the configuration $\langle \lambda x.e \mid (\bullet a); E \rangle$ is analogous to $E[(\lambda x.e)\, a]$, so the rule is like a β-reduction in context E.

In conclusion: in the presentation based on abstract machines, CIU equivalence becomes *substitution equivalence*, which equates configurations c and d when $c[\sigma] \cong d[\sigma]$ for all closed substitutions σ, where $\cong$ denotes some suitable, yet straightforward notion of equivalence on closed configurations.

2.2 Axiomatising evaluation, a.k.a. normal forms as triples

Our next step is to analyse how OGS exploits evaluation, and then abstract over it.

To start with, let us consider a configuration c that gets stuck on a function call $f\, v$, where f is a variable and v is some value, say of the shape $\lambda x.p$. In the abstract machine presentation, this means c reduces to

$$\langle f \mid (\bullet v); E \rangle, \qquad (3)$$

for some evaluation context E. The compound evaluation context $(\bullet v); E$ means that, once f is evaluated, it should be applied to v, the result of which should be fed to E.

In such a situation, OGS splits the stuck configuration (3) into

- a *head variable*, here f,
- a first-order approximation of the environment, which we will call the *observation*[6], and
- a substitution called a *filling*.

In this case, assuming v has some functional type $A_1 \to A_2$, the observation, say o, is $\langle \Box \mid (\bullet x); \beta \rangle$, for fresh variables x and β, and the filling, say γ, is the assignment

$$[x \mapsto v, \beta \mapsto E].$$

In particular, the stuck configuration $\langle f \mid (\bullet v); E \rangle$ may be recovered as $(f.o)[\gamma]$, where

- $f.o$ denotes the result of filling the hole in o with f, i.e., $\langle f \mid (\bullet x); \beta \rangle$, and
- $X[\gamma]$ denotes capture-avoiding substitution of γ in X.

Terminology 1. *For clarity, we will now explicitly distinguish between the two meanings of "substitution": we continue calling the operation* substitution*, while we call the arguments* assignments*.*

To fix intuition, here is a table displaying a few examples of normal forms, including the one presented previously. They are given both using standard syntax and in the abstract machine presentation. We also present how they may be split into head variable, observation, and filling:

Standard presentation	Abstract machine	Head	Observation	Filling
v	$\langle v \mid \alpha \rangle$	α	$\langle x \mid \Box \rangle$	$x \mapsto v$
$E[f\ v]$	$\langle f \mid (\bullet v); E \rangle$	f	$\langle \Box \mid (\bullet x); \alpha \rangle$	$x \mapsto v, \alpha \mapsto E$
$E[\mathbf{proj}_i\, x]$	$\langle x \mid \mathbf{proj}_i; E \rangle$	x	$\langle \Box \mid \mathbf{proj}_i; \alpha \rangle$	$\alpha \mapsto E$
$E[\texttt{if}\, x\, \texttt{then}\, e_1\, \texttt{else}\, e_2]$	$\langle x \mid (e_1, e_2); E \rangle$	x	$\langle \Box \mid (x_1, x_2); \alpha \rangle$	$x_i \mapsto e_i, \alpha \mapsto E$

- A particular case of a normal form is indeed any value v. In the abstract machine presentation, v must be fed to some context variable α, which we view as the head variable. The observation, here denoted by $\langle x \mid \Box \rangle$, means that the observed variable α is fed with some value x, while the filling associates v to x.
- The second row is the previous example.
- In the third row, $\mathbf{proj}_i; E$ is an evaluation context that takes the ith component of the running program and continues with E. The corresponding reduction rule is $\langle (e_1, e_2) \mid \mathbf{proj}_i; E \rangle \to \langle e_i \mid E \rangle$.

[6] This is sometimes called an *ultimate pattern.*

– In the fourth example, $(e_1, e_2); E$ is an evaluation context that executes e_1 or e_2 according to whether the running (boolean) program evaluates to true or false. The reduction rules are

$$\langle \mathbf{tt} \mid (e_1, e_2); E\rangle \to \langle e_1 \mid E\rangle \qquad \langle \mathbf{ff} \mid (e_1, e_2); E\rangle \to \langle e_2 \mid E\rangle. \tag{4}$$

Since our axiomatisation of evaluation is devoted to OGS, it inlines this splitting process. It goes in two steps:

– A *language* on a given, fixed set of types consists of suitably indexed sets of *values*, *configurations*, and *observations*. *Fillings* are then defined as suitable assignments from variables to values, and they are assumed to act on values, configurations, and observations – this axiomatises substitution.
– An *evaluator* consists of a suitably indexed family of partial maps from configurations to triples of a (head) variable, an observation, and a filling. Intuitively, any configuration either diverges (which is modelled by partiality), or converges to some triple (f, o, γ). In other words, we model normal forms as triples (f, o, γ).
– Conversely, a *refolding* is a map sending such normal forms back to configurations. Intuitively, this merely maps (f, o, γ) to $f.o[\gamma]$.

A *language machine* is a language equipped with an evaluator and a refolding, satisfying a few coherence axioms (Definition 13). Notably, in order to ensure soundness of OGS, we need to require that evaluation respect substitution, which roughly means that evaluating a substituted configuration $c[\gamma]$ amounts to evaluating c, and, if this converges to some normal form (f, o, δ), evaluating the substituted refolding of $(f.o[\delta])[\gamma]$.

Remark 1. Of course, this definition is informal, notably w.r.t. substitution. In particular, Definition 13 relies on the well-known machinery of substitution monoids and modules over them [8–10, 13, 14].

Remark 2. Indexing here refers to the fact that our axiomatisation is intrinsically typed and scoped. Thus, e.g., configurations are indexed over lists of types, values are indexed over *sequents*, i.e., pairs of a list of types and a type, and so on.

2.3 Substitution equivalence

The notion of language machine lets us define substitution equivalence, abstractly. In the usual presentation of λ-calculus, we were observing closed programs of some fixed, basic type like the booleans $\mathbb{B}$. Transposed to the abstract machine presentation, this amounts to observing configurations with a single free variable of type $\neg\mathbb{B}$, which models the final continuation to which the closed context returns. There are only two observations at that type, namely $\langle \mathbf{tt} \mid \square\rangle$ and $\langle \mathbf{ff} \mid \square\rangle$, for true and false, which correspond to the two expected observations.

In the abstract setting, i.e., in any given language machine, we postulate a fixed, "final" typing context $\mathbf{\Omega}$, and think of configurations with free variables in Ω as closed programs. The obvious way to observe them is to check whether they diverge, and otherwise record which observation they perform on which free variable of Ω.

Thus, two configurations c and d with free variables in Γ are *substitution equivalent* iff, for all assignments $\gamma \colon \Gamma \to \Omega$ from variables in Γ to values of the same type over Ω, $c[\gamma]$ and $d[\gamma]$ coterminate, and, if they do terminate, perform the same observation on the same variable of Ω.

2.4 Games and strategies

We now would like to construct the OGS of any language machine, but before that we need to define what we mean by games and strategies. For this, we follow Levy and Staton [22], up to slight reformulation.

We first introduce *half-games* from I to J, for any sets I and J of *Player* and *Opponent* positions, respectively. Intuitively, a half-game describes the moves available in each Player position, and the Opponent positions they lead to.

A *game* over I and J then consists of a *Player* half-game from I to J, and an *Opponent* half-game from J to I.

A *strategy* then consists of

- an I-indexed family of *active* states, where Player is to play,
- a J-indexed family of *waiting* states, where Opponent is to play,
- an *action* partial map, which, to any active state over a position $i \colon I$, either diverges, or picks a move from i and a "next" waiting state, and
- a *reaction* map, which, to any waiting state and Opponent move over a position $j \colon J$, associates a "next" active state.

2.5 Constructing the game

We saw that evaluators are viewed as either diverging, or splitting configurations into a head variable x, an observation o, and a filling γ. This splitting is used to interpret the considered configuration c as a strategy in a two-player game, where the program plays as Player and the context plays as Opponent. Let us briefly describe this game, which we call the *OGS game*.

As a first approximation, the game in question has the same Player and Opponent positions, which consist of pairs (Γ, Δ) of typing contexts:

(i) variables in Γ are thought of as defined by the currently waiting player, say W, hence unknown to the currently active player, say A,
(ii) conversely, variables in Δ are defined by A, hence unknown to W.

Accordingly, a move (by A) consists of an observation on some variable in Γ. Such an observation may introduce some fresh variables, which are modelled as a typing context Θ that we call the *context increment*. Intuitively, A holds the definitions of variables in Θ, and the next position is

$$(\Gamma', \Delta') := (\Delta + \Theta, \quad \Gamma).$$

This is consistent with *(i)–(ii)* above:

- the definition of variables in Γ' are held by the now waiting player A, while
- those of variables in Δ' are held by the now active player W.

Example 1. If $f \colon A_1 \to A_2 \in \Gamma$, then a possible move is $(f, \langle \square \mid ((\bullet x); \beta) \rangle)$, with context increment $\Theta = (x : A_1, \beta : \neg A_2)$.

Weak bisimilarity $S \approx T$ of strategies is then defined straightforwardly: either both diverge, or they both converge, play the same move, and reach weakly bisimilar strategies, coinductively.

2.6 Constructing the OGS

The crux of OGS is then to interpret the language machine as a strategy in the OGS game. This strategy, which we call the *machine strategy*, is essentially straightforward:

- an active state over (Γ, Δ) consists of a configuration c with free variables in Γ, and, for each $x \colon \tau \in \Delta$, a value of type τ with free variables in Γ, which we (continue to) call an *assignment* $\Delta \to \Gamma$;
- waiting states over (Γ, Δ) are assignments $\Gamma \to \Delta$;
- the action of any configuration c and assignment $\delta \colon \Delta \to \Gamma$ consists in evaluating c, and diverging if it does; otherwise, it evaluates to some normal form (f, o, φ), for some filling $\varphi \colon \Theta \to \Gamma$ of the context increment Θ of o; the machine strategy then
  - plays the move (f, o), and
  - picks as its next waiting state the compound assignment $[\delta, \varphi] \colon \Delta + \Theta \to \Gamma$, over the position $(\Delta + \Theta, \Gamma)$;
- the reaction of an assignment $\gamma \colon \Gamma \to \Delta$ to any move (f, o) with context increment Θ is the configuration obtained by refolding (f, o, γ), up to some technicalities that we hide under the rug for the moment. In particular, variables in the context increment Θ remain fresh for this player.

At this point, we may state the soundness property: any two configurations which give weakly bisimilar strategies are substitution equivalent.

In order to prove it, it remains to work around a few technical glitches, which we briefly describe in the coming subsections. The first amounts to taking the final typing context Ω into account at the level of games: this is easy. The second is well known to the experts: it is the "infinite chattering" problem. We deal with it in a novel way, by building acyclicity into the model. The final difficulty is new and surprising. It has to do with the fact that, in all sensible languages, repeated, non-trivial instantiation of the head variable eventually leads to some redex. In the abstract setting, we need an additional hypothesis to ensure this is indeed the case.

2.7 Final moves

It might be tempting to treat the "final" typing context Ω normally, i.e., to make it part of positions (Γ, Δ). But it would not work. Indeed, once a player makes a final move, the game must stop, in order to reflect the notion of observation that was fixed for substitution equivalence.

We thus tweak the naive definitions of games and strategies given above to incorporate this idea:

(1) A game comes with a set of *final* moves.
(2) The action map of a strategy may either play a proper move as before, or play a final move.
(3) The final moves of the OGS game are observations on variables in the final typing context Ω.
(4) States of the machine strategy are modified similarly: active states over (Γ, Δ) comprise a configuration with free variables in $\Omega + \Gamma$ and an assignment $\Delta \to \Omega + \Gamma$, while waiting states are assignments $\Gamma \to \Omega + \Delta$.
(5) The action map of the machine strategy discriminates against the head variable: if it is in Γ, then a proper move is played; if it is in Ω, a final move is played.
(6) Finally, we adjust weak bisimilarity accordingly: if one strategy plays a final move, then the other should play the same, and conversely.

At the cost of some moderate additional verbosity, this builds the protocol of substitution equivalence into weak bisimilarity of OGS strategies.

2.8 Infinite chattering

Let us now deal with the second announced technical glitch: the infinite chattering problem. A symptom of it is already visible in our description of OGS. Indeed, from a pair of an active and a passive states on a given position (Γ, Δ), we would like to be able to recover a corresponding configuration with free variables in the final typing context Ω. Such a pair of states amounts to a configuration with free variables in Γ, and assignments

$$\Gamma \xrightarrow{\gamma} \Omega + \Delta \qquad\qquad \Delta \xrightarrow{\delta} \Omega + \Gamma,$$

so one might hope that substituting c with γ and δ in turn would converge to some configuration with free variables in Ω. This in fact holds for pairs of states arising from the game, but not for all pairs.

Example 2. For a silly example, consider a case where some free variable x of c is mapped by γ to some variable y in Δ, which is in turn mapped to x in Γ by δ.

Such cyclic configurations do not arise during the game, though. Crucially, in the above description, when we form the new waiting state $[\delta, \varphi] \colon \Delta + \Theta \to \Omega + \Gamma$, we know that the other assignment $\gamma \colon \Gamma \to \Omega + \Delta$ does not depend on the new variables in Θ.

This leads us to introduce a refined version of the game, in which positions record the sequence of context increments, and states of the machine strategy take them into account to build acyclicity into the model. Positions are thus sequences $\Theta_1, \ldots, \Theta_n$ of typing contexts.

The idea is that, if the current position is a sequence $\Theta_1, \ldots, \Theta_n$ of context increments, then Θ_n was introduced by the previously active, now waiting player W. Accordingly:

- A (non-final) move consists of a variable introduced by W, i.e., one in $\ldots + \Theta_{n-2} + \Theta_n$, together with an observation o on it – the first index in the sequence depending on the parity of n. The next position is of course $\Theta_1, \ldots, \Theta_n, \Theta$, where Θ denotes the context increment of o.
- An active state of the machine strategy is a configuration with free variables in $\Omega + \ldots + \Theta_{n-2} + \Theta_n$, equipped with assignments $(\ldots, \delta_{n-3}, \delta_{n-1})$ as on the left below.

$$\begin{array}{cc} \text{Active assignments} & \text{Passive assignments} \\ \Theta_{n-1} \xrightarrow{\delta_{n-1}} \Omega + \ldots + \Theta_{n-4} + \Theta_{n-2}, & \Theta_n \xrightarrow{\gamma_n} \Omega + \ldots + \Theta_{n-3} + \Theta_{n-1}, \\ \Theta_{n-3} \xrightarrow{\delta_{n-3}} \Omega + \ldots + \Theta_{n-6} + \Theta_{n-4}, & \Theta_{n-2} \xrightarrow{\gamma_{n-2}} \Omega + \ldots + \Theta_{n-5} + \Theta_{n-3}, \\ \vdots & \vdots \end{array} \tag{5}$$

- A passive state consists of complementary assignments $(\ldots, \gamma_{n-2}, \gamma_n)$ as on the right above.

This time, it is easy to recover a well-defined configuration with free variables in Ω from an active-passive pair of states, as

$$\overline{c} := c[\gamma_n][\delta_{n-1}][\gamma_{n-2}][\delta_{n-3}] \ldots \tag{6}$$

2.9 Focused redexes and eventual guardedness

Let us now come to the last technical glitch that we had to face. It surprised us, as it is rather theoretical: we know of no concrete, sensible language machine in which it arises. It may be viewed as a form of infinite chattering. Let us briefly explain it. A key lemma in virtually any OGS soundness proof roughly states the following: given any compatible pair of an active state (c, δ) and a waiting state γ, the configuration (6), obtained by iterated substitution, behaves the same as letting the strategies associated to (c, δ) and γ play against one another. This lemma is simply wrong in general, so we need an additional hypothesis.

A bit more formally, one defines a *composition* operation. The result of composition may either diverge or play a final move in Ω. And the key lemma states that the refolded configuration (6) diverges iff the composition does, and, if not, both play the same final move in Ω.

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. The composition, say $(c, \delta) \parallel \gamma$, is then computed like so:

– If the active strategy diverges or performs a final 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.

As a first step towards making this precise, we must give a bit more detail on how we handle partiality. For us, a partial map $A \to B$ is a map $A \to \mathcal{D}B$, where $\mathcal{D}$ is Capretta's *delay* monad [2], which we briefly recall in §4.2. This is a rather intensional description of partiality, in the sense that an element of $\mathcal{D}B$ consists of a sequence of "silent computation steps", denoted by τ, which is either infinite or followed by some "result" in B.

This suggests interpreting the above description as a Coq `cofixpoint` definition. However, the second clause does not satisfy Coq's guardedness criterion! Let us illustrate this:

Example 3. Consider $c = \langle x \mid (\bullet v); \alpha\rangle$ and $\gamma = [x \mapsto \lambda z.p, \alpha \mapsto E]$, for some program p and evaluation context E. Since c is a normal form, $(c, \emptyset)$ plays without any computation step the non-final move $(\bullet y); \alpha$ on x, and the second recursive clause above says that the composition $(c, \emptyset) \parallel \gamma$ *equals* the composition

$$(\quad \langle \lambda z.p \mid (\bullet y); E\rangle \quad , \quad \gamma \quad) \qquad \parallel \qquad [y \mapsto v],$$

thus making an unguarded corecursive call.

Remark 3. At this point, it is tempting to insert a dummy computation step to make the definition guarded. This does give the expected definition *up to weak bisimilarity*, but makes the proof break later on, as explained in §4.7.

To justify further, in the above example, semantically, the unguarded call seems right, because it matches the behaviour of $c[\gamma]$ as a whole, our stated goal for composition. Indeed, the second clause models communication between $(c, \emptyset)$ and γ, which is invisible in the behaviour of $c[\gamma]$.

In order to solve the issue, we need to make sure that no two strategies get stuck in a loop involving only the second clause. This is really subtle: if both players keep on exchanging non-final moves, interleaved with computation steps, then composition is well defined (and diverges); so the only problematic case is when both players exchange non-final moves indefinitely *without performing any computation step.*

How could this happen? In the machine strategy, when both players exchange a non-final move, the head variable gets instantiated. So the question becomes: in concrete cases, how could repeated instantiation of the head variable not lead to a redex? A first possibility is if the head variable always gets replaced by another variable. But this is ruled out by the refinement introduced in the previous subsection to deal with infinite chattering. Indeed, if some non-final move leads to a head variable x being instantiated by another variable y, then acyclicity tells us that y must have been introduced before x. Since this is a well-founded ordering, we are safe on this front.

However, this does not suffice in all languages!

Example 4. Consider a language involving λ-abstraction, with an exotic redex of the form

$$\langle \lambda x.p \mid \lambda y.q \rangle \to \dots$$

Then, starting from a situation with

- active state given by the configuration $\langle x_1 \mid \lambda y.q \rangle$ and empty assignment,
- and passive state given by $x_1 \mapsto \lambda x.p$,

the first move consists of the observation $\langle \square \mid y_1 \rangle$ on x_1, leading to

- as active state $(\langle \lambda x.p \mid y_1 \rangle, [x_1 \mapsto \lambda x.p])$ and
- as passive state $[y_1 \mapsto \lambda y.q]$.

We are then stuck in a loop between situations of the following forms

$$\begin{tikzcd}
(\langle x_m \mid \lambda y.q \rangle, [y_j \mapsto \lambda y.q]_{j=1,\dots,n}) \quad \| \quad [x_i \mapsto \lambda x.p]_{i=1,\dots,m} \arrow[d, bend right, "x_m.\langle \square \mid y_{n+1} \rangle"'] \\
(\langle \lambda x.p \mid y_n \rangle, [x_i \mapsto \lambda x.p]_{i=1,\dots,m}) \quad \| \quad [y_j \mapsto \lambda y.q]_{j=1,\dots,n}, \arrow[u, bend right, "y_n.\langle x_{m+1} \mid \square \rangle"']
\end{tikzcd}$$

where we ignore the fine layering of assignments for readability.

In order to rule out languages in which this issue arises, we state our main result under the hypothesis that a suitable binary relation $\succ$ on observations should be well-founded. Explicitly, we have $o \succ o'$ iff there exist x, o, γ, and a non-variable v such that substituting v for x in the refolding $x.o[\gamma]$ yields some normal form (without evaluating!) of the shape (x', o', γ'), i.e.,

$$(x.o[\gamma])[x \mapsto v] = x'.o'[\gamma'].$$

When $\succ$ is well-founded, we say that the considered language machine has *focused redexes.*

Assuming that the considered language machine has focused redexes, we should be able to define composition. However, Coq does not readily accept the definition sketched above, because it does not know that it is productive. In order to proceed cleanly, we introduce a relaxed fixed point operator which contents with a proof that, even though the given equations are not directly guarded in the usual sense, unfolding the definition of each unknown will reach a guard eventually. This enables us to define composition, and at last prove our main result (Theorem 8), which states that any language machine with focused redexes has a sound OGS.

3 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 boolean type and recursive functions. Terms are generated by the following grammar

$$\begin{aligned} \text{values} \ni v, w &::= x \mid \mathbf{tt} \mid \mathbf{ff} \mid \lambda^{\mathbf{rec}} f, x.\, p \\ \text{programs} \ni p, q &::= v \mid p\, q \mid \mathbf{if}(p, q_1, q_2) \end{aligned}$$

where $\lambda^{\mathbf{rec}}$ binds f and x in p, as usual. The language is typed. Types and typing contexts are generated by the following grammar,

$$A, B ::= \mathbb{B} \mid A \to B \qquad\qquad \Gamma ::= \varepsilon \mid \Gamma, x \colon A$$

with the following, standard typing rules.

$$\frac{x \colon A \in \Gamma}{\Gamma \vdash x \colon A} \qquad \frac{}{\Gamma \vdash \mathbf{tt} \colon \mathbb{B}} \qquad \frac{}{\Gamma \vdash \mathbf{ff} \colon \mathbb{B}} \qquad \frac{\Gamma, f \colon A \to B, x \colon A \vdash p \colon B}{\Gamma \vdash \lambda^{\mathbf{rec}} f, x.\, p \colon A \to B}$$

$$\frac{\Gamma \vdash p \colon A \to B \qquad \Gamma \vdash q \colon A}{\Gamma \vdash p\, q \colon B}$$

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, with straightforward typing rules.

$$\text{eval. contexts} \ni E ::= \square \mid p\, E \mid E\, v \mid \mathbf{if}(E, p, q)$$

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

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

$$\frac{}{\mathbf{if}(\mathbf{tt}, p, q) \to p} \qquad \frac{}{\mathbf{if}(\mathbf{ff}, p, q) \to q}$$

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. *We write $\Gamma \vdash \sigma \colon \Delta$ for assignments to each variable $x \colon A \in \Delta$ of a value of type A in typing context Γ.*

Definition 1. *Two programs $\varepsilon \vdash p, q$ of type $\mathbb{B}$ are deemed* observably equivalent *whenever we have $p \to^* \mathbf{tt}$ iff $q \to^* \mathbf{tt}$, and similarly with $\mathbf{ff}$.*

Definition 2. *For any context Γ and type A, two programs $\Gamma \vdash p, q \colon A$ are* CIU-equivalent*, which we denote by $p \approx_{\text{CIU}} q$, iff for all assignments $\varepsilon \vdash \sigma \colon \Gamma$, and closed evaluation contexts E of type $\mathbb{B}$ with a hole of type A, $E[p[\sigma]]$ and $E[q[\sigma]]$ are observably equivalent.*

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

$$\frac{\Gamma \vdash p : A \qquad \Gamma \vdash q : A \qquad \Gamma \vdash \pi : \neg A}{\Gamma \vdash (p, q); \pi : \neg \mathbb{B}} \qquad \frac{\Gamma \vdash p : A \qquad \Gamma \vdash \pi : \neg A}{\Gamma \vdash \langle\, p \,|\, \pi \,\rangle}$$

$$\begin{array}{rr}
\langle\, p\, q \,|\, \pi \,\rangle \to \langle\, q \,|\, p\bullet; \pi \,\rangle & \langle\, \mathbf{if}(p, q_1, q_2) \,|\, \pi \,\rangle \to \langle\, p \,|\, (q_1, q_2); \pi \,\rangle \\
\langle\, v \,|\, p\bullet; \pi \,\rangle \to \langle\, p \,|\, \bullet v; \pi \,\rangle & \langle\, \lambda^{\mathbf{rec}} f, x.\, p \,|\, \bullet v; \pi \,\rangle \to \langle\, p[f \mapsto (\lambda^{\mathbf{rec}} f, x.\, p), x \mapsto v] \,|\, \pi \,\rangle \\
\langle\, \mathbf{tt} \,|\, (q_1, q_2); \pi \,\rangle \to \langle\, q_1 \,|\, \pi \,\rangle & \langle\, \mathbf{ff} \,|\, (q_1, q_2); \pi \,\rangle \to \langle\, q_2 \,|\, \pi \,\rangle
\end{array}$$

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

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. The overall idea is to compile our simply-typed, call-by-value λ-calculus down to a slightly lower-level language, as explained in §2.1.

Let us now introduce the low-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.

$$\begin{aligned}
\text{values} \ni v, w &::= x \mid \mathbf{tt} \mid \mathbf{ff} \mid \lambda^{\mathbf{rec}} f, x.\, p \\
\text{programs} \ni p, q &::= v \mid p\, q \mid \mathbf{if}(p, q_1, q_2) \\
\text{eval. contexts} \ni \pi, \kappa &::= x \mid \bullet v; \pi \mid p\bullet; \pi \mid (p, q); \pi \\
\text{configurations} \ni c, d &::= \langle\, p \,|\, \pi \,\rangle
\end{aligned}$$

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 $\neg A$. Additional 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 displayed in the second part of Figure 1. We may now introduce substitution equivalence.

Definition 3. *For any typing context* Γ*, two configurations* $\Gamma \vdash c, d$ *are* substitution equivalent*, which we denote by* $c \approx_{\textsc{sub}} d$*, iff for all assignments* $(\alpha : \neg\mathbb{B}) \vdash \sigma : \Gamma$*,* $c[\sigma]$ *and* $d[\sigma]$ *are* observably equivalent*, in the sense that we have* $c \to^* \langle\, \mathbf{tt} \,|\, \alpha \,\rangle$ *iff* $d \to^* \langle\, \mathbf{tt} \,|\, \alpha \,\rangle$*, and similarly with* $\mathbf{ff}$*.*

The main point of this section is:

Proposition 1. *For any typing context Γ and type A of the source language, two programs $\Gamma \vdash p, q \colon A$ are* CIU*-equivalent iff, in the typing context $(\Gamma, \beta \colon \neg A)$ (with β fresh w.r.t. Γ), $\langle\, p \,|\, \beta \,\rangle$ and $\langle\, q \,|\, \beta \,\rangle$ are substitution equivalent.*

Proof. There is a bijection between assignments $(\alpha \colon \neg\mathbb{B}) \vdash \sigma \colon (\Gamma, \beta \colon \neg A)$ in the lower-level language and pairs of an assignment $\varepsilon \vdash \gamma \colon \Gamma$ and an evaluation context E of type $\mathbb{B}$ with a hole of type A in the source language. Furthermore, for such assignments and evaluation contexts, $\langle\, p \,|\, \beta \,\rangle[\sigma]$ and $E[p[\gamma]]$ are observably equivalent in the obvious sense, hence the result follows.

4 Abstract OGS

In this section, we fill the gaps left by the informal overview of §2.

4.1 An abstract account of substitution

Let us first recall (one presentation of) a standard way of abstracting over capture-avoiding substitution.

Notation 3. *We fix a set T of* types *for the whole section, and let T^* denote the set of sequences of types.*

The ambient setting for axiomatising substitution is that of families:

Definition 4. *For any sequence $X_1, \ldots, X_n$ of sets, we extend the set*

$$\widehat{X_1, \ldots, X_n} = (X_1 \to \ldots \to X_n \to \mathbf{Set})$$

to a category, by taking

$$\forall x_1 : X_1,\ \ldots,\ x_n : X_n,\ \mathcal{X}\, x_1\ \ldots\ x_n \to \mathcal{Y}\, x_1\ \ldots\ x_n$$

as hom-set $\widehat{X_1, \ldots, X_n}(\mathcal{X}, \mathcal{Y})$, for any $\mathcal{X}, \mathcal{Y}$. In particular, we call the objects of $\widehat{T^*}$ and $\widehat{T, T^*}$, unsorted *and* sorted *families, respectively.*

Example 5. A sorted family of particular interest is the one of variables (🐓), given by the proof-relevant $\in$ predicate: $(\tau \in \Gamma)$ denotes the set of indices at which τ occurs in Γ.

Let us now explain what it means for sorted and unsorted families to be equipped with substitution. As is standard in the well-scoped approach, we mean substitution in the parallel sense. The basic ingredient for this is the following standard notion, which we call *assignment.*

Definition 5 (🐓). *For any typing contexts $\Gamma, \Delta \colon T^*$ and sorted family $\mathcal{X}$, an* $\mathcal{X}$-assignment *$\sigma \colon \Gamma \to_{\mathcal{X}} \Delta$, or* assignment *when $\mathcal{X}$ is clear from context, consists of an element of $\mathcal{X}\, \tau\, \Delta$, for all $x : \tau \in \Gamma$. In other words, we have*

$$- \to_{\_} - \colon T^* \to \widehat{T, T^*} \to T^* \to \mathbf{Set}$$
$$\Gamma \to_{\mathcal{X}} \Delta := \forall \tau \colon T,\ \tau \in \Gamma \to \mathcal{X}\, \tau\, \Delta.$$

In order to axiomatise substitution for both kinds of families, we introduce the following notion of *power* families:

Definition 6. *Fixing a sorted family $\mathcal{M}$, for any sorted family $\mathcal{S}$ and unsorted family $\mathcal{U}$, we define the power objects*

$$[\![\mathcal{M},\mathcal{S}]\!]_s : \widehat{T,T^*} \qquad\qquad [\![\mathcal{M},\mathcal{U}]\!] : \widehat{T^*}$$

by

$$[\![\mathcal{M},\mathcal{S}]\!]_s\,\tau\,\Gamma := \forall\Delta\colon T^*,\ (\Gamma \to_{\mathcal{M}} \Delta) \to \mathcal{S}\,\tau\,\Delta$$
$$[\![\mathcal{M},\mathcal{U}]\!]\,\Gamma := \forall\Delta\colon T^*,\ (\Gamma \to_{\mathcal{M}} \Delta) \to \mathcal{U}\,\Delta.$$

We may now axiomatise substitution, for both kinds of families:

Definition 7 (🐓). *A* substitution monoid *is a sorted family $\mathcal{M}$, equipped with morphisms*

$$\mathtt{var}\colon -\!\in\!- \to \mathcal{M} \qquad\qquad \mathtt{sub}\colon \mathcal{M} \to [\![\mathcal{M},\mathcal{M}]\!]_s,$$

subject to associativity and unitality laws.

Definition 8 (🐓). *A* substitution module *over a substitution monoid* ($\mathcal{M}$, var, sub)*, or* substitution $\mathcal{M}$-module *for short, is an unsorted family $\mathcal{U}$, equipped with a morphism*

$$\mathtt{sub}\colon \mathcal{U} \to [\![\mathcal{M},\mathcal{U}]\!],$$

subject to assocativity and unitality laws.

In both cases, the substitution morphism takes elements over any context Γ (and type τ, if relevant) and assignments $\Gamma \to_{\mathcal{M}} \Delta$, to elements over Δ: this is indeed the expected type for substitution.

4.2 Modelling divergence: the delay monad

Now that we have recalled the standard axiomatisation of substitution, let us explain more precisely how we handle divergence in Coq, which is very simple and standard: we use Capretta's *delay* monad [2].

Definition 9 (🐓). *The* delay *endomap on* **Set** *is defined* coinductively *by the following inference rules.*

$$\frac{x : X}{\eta\, x : \mathcal{D}\, X} \qquad\qquad \frac{a : \mathcal{D}\, X}{\tau; a : \mathcal{D}\, X}\ .$$

We also denote by $\mathcal{D}$ the pointwise liftings of delay to categories of families $\widehat{X_1,\ldots,X_n}$, e.g., $\widehat{T,T^*}$ and $\widehat{T^*}$.

Remark 4. The notation $\tau; a$ is meant to suggest a silent computation step. Such τs should not be confused with types, which hopefully will be easy from context.

Elements of $\mathcal{D}\,X$ are thought of as potentially diverging computations of type X: divergence is modelled by the infinite chain of τs; converging computations have the shape $\tau;\ldots;\tau;\eta\,x$.

Depending on context, we will consider elements of $\mathcal{D}\,X$ equivalent up to strong or weak bisimilarity, which we now recall.

Definition 10. *For any relation $R \subseteq X \times Y$ between sets (or families) X and Y, we let $\mathcal{D}\,R \subseteq \mathcal{D}\,X \times \mathcal{D}\,Y$ be such that $(c,d)\colon \mathcal{D}\,R$ iff c and d either both diverge, or evaluate to some x and y with $(x,y) \in R$. We write $\approx_{\mathcal{D}}$ for weak bisimilarity ( ), which we define as $\mathcal{D}\,(=)$.*

We write $\cong_{\mathcal{D}}$ for strong bisimilarity ( ), the canonical lifting of R to the delay coinductive type: either both sides loop or both converge to the same element in the same number of τ steps.

Notation 4. *We write interchangeably $(x \leftarrow u; v\,x)$ and $u \gg\!= v$ for the* bind *( ) operator of $\mathcal{D}$—evaluate u, and if it returns some x, then evaluate $v\,x$.*

4.3 Language machines and substitution equivalence

Let us now introduce our axiomatisation of evaluation and observation more formally than in the overview. We will then wrap this all up into the definition of language machines, and define substitution equivalence.

Definition 11 ( ). *An* evaluation structure *on any unsorted families $\mathcal{C}, \mathcal{N} \colon \widehat{T^*}$ consists of maps:* **eval**$\colon \mathcal{C} \to \mathcal{D}\,\mathcal{N}$ *and* **refold**$\colon \mathcal{N} \to \mathcal{C}$ *such that* **eval**$\circ$**refold**$\cong_{\mathcal{D}} \eta$.

Remark 5. By Definition 10, the equation says that evaluating the embedding **refold**$\,n$ of some normal form n yields n in zero computation steps.

Definition 12 ( ). *An* observation structure *O is a type-indexed set $O\colon T \to$* **Set** *together with a map* $\mathtt{dom} : \forall\tau\colon T,\ O\,\tau \to T^*$.

For any observation structure O, we define the unsorted family $O^\bullet$ of pointed observations *by $O^\bullet\,\Gamma := \exists\tau : T,\ (\tau \in \Gamma) \times O\,\tau$. We extend the map* $\mathtt{dom}$ *to $O^\bullet$, defining* $\mathtt{dom}^\bullet \colon \forall\Gamma, O^\bullet\,\Gamma \to T^*$ *by* $\mathtt{dom}^\bullet\,\tau\,(i,o) := \mathtt{dom}\,o$.

Thus, a pointed observation over Γ consists of a variable, together with an observation at its type.

Remark 6. In the literature, observations are sometimes called *ultimate patterns* [21], *continuation patterns* [29], *atomic values* [19], or *abstract values* [17].

Definition 13. *A* language machine *consists of*

- *a substitution monoid $\mathcal{V}$ of* values,
- *a substitution $\mathcal{V}$-module $\mathcal{C}$ of* configurations,
- *an observation structure O,*
- *an evaluation structure on $\mathcal{C}$ and $\mathcal{N}_{O,\mathcal{V}}$,*

where $\mathcal{N}_{\mathcal{O},\mathcal{V}}\,\Gamma := \exists o\colon \mathcal{O}^\bullet\,\Gamma, (\mathtt{dom}^\bullet\, o \to_{\mathcal{V}} \Gamma)$.

Evaluation and refolding are furthermore required to respect substitution *(), in the sense that, for all* $u\colon \mathcal{C}\,\Gamma$*,* $\gamma\colon \Gamma \to_{\mathcal{V}} \Delta$*,* $(x,o,\theta)\colon \mathcal{O}^\bullet\,\Gamma$*, with* $x\colon \tau \in \Gamma$ *and* $\gamma(x) = y\colon \tau \in \Delta$*, the following hold*

$$\begin{array}{rcl}\mathbf{eval}\,(u[\gamma]) & \approx_{\mathcal{D}} & n \leftarrow \mathbf{eval}\,u \ ;\ \mathbf{eval}\,((\mathbf{refold}\,n)[\gamma]) \\ x.o(\theta)[\gamma] & = & y.o(\theta[\gamma]),\end{array}$$

where we denote

– *both substitution maps and pointwise substitution by* $X[\gamma]$*, and*
– *the refolding* **refold** (x,o,θ) *of any normal form* $(x,o,\theta)\colon \mathcal{N}_{\mathcal{O},\mathcal{V}}\,\Gamma$ *by* $x.o(\theta)$*.*

Remark 7. By pointwise substitution, we mean that $\theta[\gamma](z) := \theta(z)[\gamma]$.

Notation 5. *In the sequel, we often treat refolding* **refold**$\colon \mathcal{N}_{\mathcal{O},\mathcal{V}} \to \mathcal{C}$ *as an implicit coercion. We also extend the notation* $x.o(\gamma)$ *to arbitrary values, writing* $v.o(\gamma)$ *for* $\mathbf{refold}(x,o,\gamma)[x \mapsto v]$*, for fresh* x*.*

Example 6. The lower-level language of §3 forms a language machine. We take T to consist of types A and negated types $\neg A$ (i.e., it is the disjoint union of two copies of simple types); $\mathcal{C}\,\Gamma$ consists of configurations $\Gamma \vdash \langle p \mid \pi\rangle$; $\mathcal{V}\,A\,\Gamma$ consists of values $\Gamma \vdash v : A$, and $\mathcal{V}\,\neg A\,\Gamma$ consists of all contexts $\Gamma \vdash \pi : \neg A$. Before defining $\mathcal{O}$, we introduce the family $\mathcal{U}\colon \widehat{T, T^*}$ of *ultimate values*, which is the subfamily $\mathcal{U}\,\tau \subseteq \exists\Gamma, \mathcal{V}\,\tau\,\Gamma$ defined inductively by the following linear type system,

$$\frac{}{x : A \to B \vdash_{\mathcal{U}} x : A \to B} \qquad \frac{}{\vdash_{\mathcal{U}} \mathbf{tt} : \mathbb{B}} \qquad \frac{}{\vdash_{\mathcal{U}} \mathbf{ff} : \mathbb{B}}$$

$$\frac{\Gamma \vdash_{\mathcal{U}} v : A}{\Gamma, x : \neg B \vdash_{\mathcal{U}} (\bullet v); x : \neg(A \to B)} \qquad \frac{}{x : \neg\mathbb{B} \vdash_{\mathcal{U}} x : \neg\mathbb{B}}$$

where we write $\Gamma \vdash_{\mathcal{U}} v : \tau$ for $(\Gamma, v)\colon \mathcal{U}\,\tau$. In words, at each τ (a simple or negated simple type), $x : \tau \vdash x : \tau$ is an ultimate value; we have $(\vdash \mathbf{tt}), (\vdash \mathbf{ff})\colon \mathcal{U}\,\mathbb{B}$, and so on. We then define $\mathcal{O}$ with similar notation:

$$\frac{\Gamma \vdash_{\mathcal{U}} \pi : \neg A}{\Gamma \vdash_{\mathcal{O}} \langle \square \mid \pi\rangle : A} \qquad \frac{\Gamma \vdash_{\mathcal{U}} v : A}{\Gamma \vdash_{\mathcal{O}} \langle v \mid \square\rangle : \neg A}\ .$$

Let us now define substitution equivalence, for any language machine $\mathcal{M} = (\mathcal{V}, \mathcal{C}, \mathcal{O}, \mathbf{eval}, \mathbf{refold})$.

Definition 14 (). *We define* $\mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\colon \mathcal{C} \to \mathcal{O}^\bullet$ *at any* Γ *to be the composite*

$$\mathcal{C}\,\Gamma \xrightarrow{\mathbf{eval}} \mathcal{D}(\mathcal{N}_{\mathcal{O},\mathcal{V}}\,\Gamma) \xrightarrow{\mathcal{D}\,\pi_1} \mathcal{D}(\mathcal{O}^\bullet\,\Gamma).$$

Definition 15 (). *For any fixed* final *typing context* $\Omega\colon T^*$*, two configurations* $u, v\colon \mathcal{C}\,\Gamma$ *are* substitution equivalent at Ω*, written* $u \approx_{\text{SUB}} w$*, iff:*

$$\forall \gamma\colon \Gamma \to_{\mathcal{V}} \Omega,\ \mathtt{eval}^{\mathrm{o}}\,(u[\gamma]) \approx_{\mathcal{D}} \mathtt{eval}^{\mathrm{o}}\,(w[\gamma]).$$

4.4 Games and strategies

We now turn to making precise the contents of §2.4. 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-game", and then define games as pairs thereof.

Definition 16. *A* half-game *() over sets I and J consists of an I-indexed family of* moves $\texttt{move}\colon I \to \textbf{Set}$*, and a* next *map in* $\forall i\colon I, \texttt{move}\, i \to J$*. We denote by* $\texttt{HGame}\, I\, J$ *the set of half-games over I and J.*

A game *() over sets I and J consists of a* client *half-game in* $\texttt{HGame}\, I\, J$*, a* server *half-game in* $\texttt{HGame}\, J\, I$*, and a set of* final moves*. We denote by* $\texttt{Game}\, I\, J$ *the set of games over I and J.*

Notation 6. *We denote by* $\texttt{move}_{\mathcal{H}}$ *and* $\texttt{next}_{\mathcal{H}}$ *the components of any half-game $\mathcal{H}$, and by* $\texttt{client}_{\mathcal{G}}$*,* $\texttt{server}_{\mathcal{G}}$*, and* $\texttt{final}_{\mathcal{G}}$ *the components of any game $\mathcal{G}$.*

We will need the following notion of dual game.

Definition 17. *The dual* $\mathcal{G}^{\perp}\colon \texttt{Game}\, J\, I$ *of a game* $\mathcal{G}\colon \texttt{Game}\, I\, J$ *is defined by swapping the client and server:* $\texttt{client}_{\mathcal{G}^{\perp}} := \texttt{server}_{\mathcal{G}}$*,* $\texttt{server}_{\mathcal{G}^{\perp}} := \texttt{client}_{\mathcal{G}}$ *and* $\texttt{final}_{\mathcal{G}^{\perp}} := \texttt{final}_{\mathcal{G}}$*.*

Now that games are defined, we turn to defining strategies in a game. We proceed coalgebraically, for which we need the following "derived" functors.

Definition 18 (). *Given any half-game* $\mathcal{H} : \texttt{HGame}\, I\, J$*, we define two functors* $\widehat{J} \to \widehat{I}$*, the* action *functor* $[\![\mathcal{H}]\!]^{+}$ *and the* reaction *functor* $[\![\mathcal{H}]\!]^{-}$*:*

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

We may now define strategies.

Definition 19. *Given a game* $\mathcal{G}\colon \texttt{Game}\, I\, J$*, a* strategy *for $\mathcal{G}$ consists of families* $\mathcal{S}^{+}\colon \widehat{I}$ *and* $\mathcal{S}^{-}\colon \widehat{J}$ *of* active *and* waiting *states, respectively, together with*

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

which we respectively call the action *and* reaction *morphisms.*

We denote by $\texttt{Strat}_{\mathcal{G}}$ *the set of strategies for $\mathcal{G}$.*

Notation 7. *We denote by* $\texttt{play}_{\mathcal{S}}$ *and* $\texttt{coplay}_{\mathcal{S}}$ *the action and reaction morphisms of any strategy $\mathcal{S}$.*

Remark 8. The occurrence of the (family lifting 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.

Remark 9. In the code, this is where indexed interaction trees come in. We define strategies (🐓) not as coalgebras, as here, but as interaction trees, i.e., elements of the final coalgebra $\nu A.(\mathtt{final}_{\mathcal{G}} + A + [\![\mathtt{client}_{\mathcal{G}}]\!]^{+}([\![\mathtt{server}_{\mathcal{G}}]\!]^{-} A))$.

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 20. *Given two strategies $\mathcal{S}, \mathcal{T} : \mathtt{Strat}_{\mathcal{G}}$, a* weak bisimulation $\alpha : \mathcal{S} \approx_{\mathcal{G}} \mathcal{T}$ *is a pair of an I-indexed relation $\alpha^{+} \subseteq \mathcal{S}^{+} \times \mathcal{T}^{+}$ between active states and a J-indexed relation $\alpha^{-} \subseteq \mathcal{S}^{-} \times \mathcal{T}^{-}$ between waiting states, such that*

$$\mathtt{play}_{\mathcal{S}} \approx\!\![\alpha^{+} \to \mathcal{D}(\mathtt{Eq}_{\mathtt{final}_{\mathcal{G}}} + [\![\mathtt{client}_{\mathcal{G}}]\!]^{+}\, \alpha^{-})]\!\!\approx \mathtt{play}_{\mathcal{T}} \text{ and}$$
$$\mathtt{coplay}_{\mathcal{S}} \approx\!\![\alpha^{-} \to [\![\mathtt{server}_{\mathcal{G}}]\!]^{-}\, \alpha^{+}]\!\!\approx \mathtt{coplay}_{\mathcal{T}},$$

where $u \approx\!\![R \to S]\!\!\approx v$ 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.*

4.5 The OGS game

Let us now define the OGS game corresponding to any language machine. For §4.5–4.7, we fix a set T of types, a language machine $\mathcal{M} = (\mathcal{V}, \mathcal{C}, \mathcal{O}, \mathbf{eval}, \mathbf{refold})$, and a typing context $\Omega : T^{*}$.

Definition 21 (🐓). *An* interleaved context *is a list of contexts. We denote by $T^{**} = (T^{*})^{*}$ 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 22 (🐓). *We define two* collapsing *functions $\downarrow^{+}, \downarrow^{-} : T^{**} \to T^{*}$ as follows:*

$$\downarrow^{+}\emptyset := \emptyset \qquad \downarrow^{-}\emptyset := \emptyset$$
$$\downarrow^{+}(\Phi, \Gamma) := \downarrow^{-}\Phi + \Gamma \qquad \downarrow^{-}(\Phi, \Gamma) := \downarrow^{+}\Phi.$$

Remark 10. 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.

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

Definition 23 (🐓). *We define the* OGS half-game $\mathtt{HOGS} : \mathtt{HGame}\, T^{**}\, T^{**}$ *by:*

$$\mathtt{move}_{\mathtt{HOGS}}\, \Phi := \mathcal{O}^{\bullet} \downarrow^{+}\Phi \qquad \text{and} \qquad \mathtt{next}_{\mathtt{HOGS}}\, \Phi\, m := (\Phi,\ \mathtt{dom}^{\bullet}\, m).$$

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

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

4.6 The machine strategy

Let us now define the machine strategy for the given language machine $\mathcal{M}$ and final typing context Ω, fixed at the beginning of §4.5.

Definition 24 (🐓). *Let* $\mathtt{Env}^+$ *and* $\mathtt{Env}^-$ *denote the* T^{**}*-indexed families of* active*, resp.* waiting*,* interleaved assignments *defined inductively as follows.*

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

Remark 11. This is merely a formal version of (5). Beware, though, 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 25 (🐓). *The* collapsing *functions for interleaved assignments are defined by mutual induction as follows, for all* $\Phi \colon T^{**}$*,*

$$\begin{aligned}
&\downharpoonleft^+ \colon \mathtt{Env}^+\ \Phi \to \downharpoonleft^- \Phi \to_{\mathcal{V}} (\Omega + \downharpoonleft^+ \Phi) &\qquad& \downharpoonleft^- \colon \mathtt{Env}^-\ \Phi \to \downharpoonleft^+ \Phi \to_{\mathcal{V}} (\Omega + \downharpoonleft^- \Phi)\\
&\qquad \downharpoonleft^+ \varepsilon := \mathtt{elim}_{\emptyset} && \qquad \downharpoonleft^- \varepsilon := \mathtt{elim}_{\emptyset}\\
&\quad \downharpoonleft^+ (e, \cdot) := (\downharpoonleft^- e)[\mathtt{wkn}] && \quad \downharpoonleft^- (e, \gamma) := [\downharpoonleft^+ e,\ \gamma],
\end{aligned}$$

where $\mathtt{wkn}$ *denotes the obvious weakening: we have* $\Phi = (\Phi', \Gamma)$ *for some* Φ' *and* Γ*, and the result is* $\downharpoonleft^+ \Phi' \xrightarrow{\downharpoonleft^- e}_{\mathcal{V}} \Omega + \downharpoonleft^- \Phi' \xrightarrow{\mathtt{wkn}}_{\mathcal{V}} \Omega + \downharpoonleft^- \Phi' + \Gamma$.

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

Definition 26 (🐓). *The machine strategy* $\widetilde{\mathcal{M}} \colon \mathtt{Strat}_{\mathtt{OGS}}$ *is defined as follows:*

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

$$\begin{aligned}
&\mathtt{play}_{\widetilde{\mathcal{M}}} \colon \widetilde{\mathcal{M}}^+ \to \mathcal{D}\, (\mathcal{O}^{\bullet}\, \Omega + [\![\mathtt{OGS}]\!]^+\, \widetilde{\mathcal{M}}^-)\\
&\mathtt{play}_{\widetilde{\mathcal{M}}}\ \Phi\, (c, e) := \left(x.o(\gamma) \leftarrow \mathbf{eval}\, c\, ; \begin{cases} \eta\, (\mathtt{inl}\, (x, o)) & \text{if } x \in \Omega \\ \eta\, (\mathtt{inr}\, ((x, o), (e, \gamma))) & \text{if } x \in \downharpoonleft^+ \Phi \end{cases} \right) ;
\end{aligned}$$

- *the reaction morphism is defined by*

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

where δ_o *denotes the obvious assignment* $\mathtt{dom}\, o \to_{\mathcal{V}} \Omega + \downharpoonleft^- \Phi + \mathtt{dom}\, o$*.*

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

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

where w is the obvious weakening $\Gamma \to_{\mathcal{V}} \Omega + \Gamma$, and $(\Gamma,)$ is the singleton sequence.

4.7 Soundness

We may now state the main result, recalling Notation 5. We introduce the following technical, yet mild conditions on the considered language machine:

Definition 27. *A substitution monoid $\mathcal{V}$ is* clear-cut *iff its unit $(-\in-) \to \mathcal{V}$ is injective (where $\in$ denotes the variables family of Example 5), and has decidable image whose complement is furthermore stable under renaming. For a clear-cut $\mathcal{V}$, we let $\mathcal{V}^{\setminus\in}$ denote the subfamily of non-variable elements.*

Definition 28. *Assuming $\mathcal{V}$ is clear-cut, we define the binary relation $\succ$ on $\exists\tau : T,\, O\,\tau$ by*

$$(\tau, o) \succ (\tau', o') \qquad \textit{iff} \qquad \exists v : \mathcal{V}^{\setminus\in},\, \gamma,\, x,\, \delta,\, \textbf{eval}\,(v.o(\gamma)) = \eta\,(x.o'(\delta)).$$

A language machine has focused redexes *iff $\succ$ is well-founded.*

Theorem 8 (🐓). *For any language machine with focused redexes and clear-cut values, weak bisimilarity of induced OGS strategies is sound w.r.t. substitution equivalence, i.e., for any pair of configurations u and w, weak bisimilarity of induced strategies entails substitution equivalence:*

$$\forall c, d,\ [\![c]\!]^+ \approx^+_{OGS} [\![d]\!]^+ \to c \approx_{\text{SUB}} d.$$

Remark 12. Let us unfold notations a bit: $\approx^+_{\text{OGS}}$ denotes the positive component of weak bisimilarity between strategies (Definition 20) in the OGS game (Definition 23), and $\approx_{\text{SUB}}$ denotes substitution equivalence (Definition 15).

The rest of this section is devoted to sketching the proof. We first describe the overall structure, and then focus on the main difficulty.

We start by defining a composition operation

$$- \parallel - : \quad \forall\Phi, \quad C\,(\Omega + {\downarrow^+}\Phi) \times \texttt{Env}^+\,\Phi \quad\to\quad \texttt{Env}^-\,\Phi \quad\to\quad \mathcal{D}\,(O^\bullet\,\Omega).$$

We expect this operation to satisfy the following properties.

Definition 29.

1. *Composition is* adequate *(🐓) iff, for all $c : C\,\Gamma$ and $\gamma : \Gamma \to_{\mathcal{M}} \Omega$, we have $\texttt{eval}^{\circ}_{\mathcal{M}}\,(c[\gamma]) \approx_{\mathcal{D}} [\![c]\!]^+ \parallel [\![\gamma]\!]^-$.*
2. *Weak bisimilarity is a* congruence *(🐓) for composition iff, for any $s_1 \approx^+_{\text{OGS}} s_2$ and $t_1 \approx^-_{\text{OGS}} t_2$, we have $s_1 \parallel t_1 \approx_{\mathcal{D}} s_2 \parallel t_2$.*

Remark 13. The adequacy equation lives in $\mathcal{D}\,(O^\bullet\,\Omega)$ (recalling Definition 14).

Let us readily show that soundness follows from congruence and adequacy.

Proposition 2. *If any adequate composition for which weak bisimilarity is a congruence exists, then OGS is sound.*

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

$$\begin{aligned}
\mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\,(c_1[\gamma]) &\approx_{\mathcal{D}} [\![c_1]\!]^+ \parallel [\![\gamma]\!]^- && \text{(by adequacy)}\\
&\approx_{\mathcal{D}} [\![c_2]\!]^+ \parallel [\![\gamma]\!]^- && \text{(by congruence of weak bisimilarity)}\\
&\approx_{\mathcal{D}} \mathtt{eval}^{\mathrm{o}}_{\mathcal{M}}\,(c_2[\gamma]) && \text{(by adequacy again),}
\end{aligned}$$

hence $c_1 \approx_{\textsc{sub}} c_2$, as desired.

It thus remains to define a congruent and adequate composition. The plan for this is to take the fixed point of an equation, in the following sense.

Definition 30 (). *An* equation *consists of a set X of* variables*, a set Y of* constants*, and a* definition *function $X \to \mathcal{D}\,(Y + X)$.*

Since we are interested in weak bisimilarity, it seems easier to try and construct *weak fixed points* of equations $f \colon X \to \mathcal{D}\,(Y + X)$, that is, maps $p \colon X \to \mathcal{D}\,(Y)$ such that $p\,x \approx_{\mathcal{D}} f\,x \gg\!= [\eta, p]$ for all $x \colon X$. This may be done by safely guarding all occurrences of "variables" in X by a τ:

Definition 31 (). *Given an equation $f \colon X \to \mathcal{D}\,(Y + X)$, the* iteration *of f is a map $f^\dagger \colon X \to \mathcal{D}\,Y$ given coinductively by:*

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

Proposition 3. *The iteration of any equation is a weak fixed point.*

Using this technique, we may define a composition operation for which weak bisimilarity is a congruence. We will see that adequacy is more problematic.

Definition 32 (). *The* composition equation *is:*

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

Let naive composition *be the iteration of* `comp-eqn`*.*

Proposition 4. *Weak bisimilarity is a congruence for naive composition.*

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

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 composition), the prime reasoning scheme is uniqueness of fixed points. Indeed, assuming the composition equation has a unique fixed point, and that substituting-then-evaluating-then-observing is one, then both must agree. However, general equations do not have unique *weak* fixed points, so we have to apply uniqueness of *strong* fixed points, i.e., fixed points w.r.t. strong bisimilarity. But there is a further complication: while all equations admit weak fixed points, given by iteration as we saw, this is not the case for strong fixed points. Coq's basic cofixpoint feature enables the construction of strong fixed points for *guarded* equations, in the following sense.

Definition 33 (). *An element* $u\colon \mathcal{D}\,(Y+X)$ *is* guarded *if it is not of the form* $\eta\,(\mathtt{inr}\,x)$. *An equation* $e\colon X \to \mathcal{D}\,(Y+X)$ *is* guarded *if for all* x, $e\,x$ *is guarded.*

However, `comp-eqn` is *not* guarded in general, as explained in Example 3, which may lead the definition to be ill-founded in some languages, as sketched in Example 4. This is where the focused redexes hypothesis comes in: it allows us to show that `comp-eqn` is *eventually guarded*, in the following sense.

Definition 34. *An equation* $e\colon X \to \mathcal{D}(Y+X)$ *is* eventually guarded *if for all* x, *there exists an* $n\colon \mathbb{N}$ *such that* $e^n\,x$ *is guarded, where by definition*

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

Proposition 5 (,). *All eventually guarded equations admit a unique strong fixed point, which is pointwise weakly bisimilar to any weak fixed point.*

Proof (sketch). 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 fixed point of e as the guarded fixed point of $e'\,x := e^{n_x}\,x$.

Proposition 6 (). *If* $\mathcal{M}$ *has focused redexes and clear-cut values, then* `comp-eqn` *is eventually guarded, and thus admits a strong fixed point, say* $- \parallel_g -$.

We may now conclude our soundness proof.

Proposition 7 (). *If the language machine* $\mathcal{M}$ *has focused redexes, then composition is adequate.*

Proof (sketch). The idea is to show that the map $(c,\gamma) \mapsto \mathtt{eval}^{\text{o}}\,(c[\gamma])$ is something like a strong fixed point of `comp-eqn`. This does not quite type check, however, so we need to generalize the two arguments (c,γ) to pairs of active and passive machine strategy states. Following (6), we define `z-e-obs` by:

$$\begin{array}{ccl} \exists\Phi, \widetilde{\mathcal{M}}^{+}\,\Phi \times \widetilde{\mathcal{M}}^{-}\,\Phi & & \to \mathcal{D}\,(\mathcal{O}^{\bullet}\,\Omega) \\ (\ (c,(\ldots,\delta_{n-3},\delta_{n-1}))\ \ ,\ \ (\ldots,\gamma_{n-2},\gamma_n)\ ) & & \mapsto \mathtt{eval}^{\text{o}}\,(c[\gamma_n][\delta_{n-1}][\gamma_{n-2}][\delta_{n-3}]\ldots). \end{array}$$

We have $\mathtt{z\text{-}e\text{-}obs}\,[\![c]\!]^{+}\,[\![\gamma]\!]^{-} = \mathtt{eval}^{\text{o}}\,(c[\gamma])$, and, as desired, `z-e-obs` is a strong fixed point of `comp-eqn` (). At last, we have, for any $c\colon \mathcal{C}\,\Gamma$ and $\gamma\colon \Gamma \to \Delta$:

$$\begin{array}{rcll} [\![c]\!]^{+} \parallel [\![\gamma]\!]^{-} & \approx_{\mathcal{D}} & [\![c]\!]^{+} \parallel_g [\![\gamma]\!]^{-} & \text{by Proposition 5} \\ & \cong_{\mathcal{D}} & \mathtt{z\text{-}e\text{-}obs}\,[\![c]\!]^{+}\,[\![\gamma]\!]^{-} & \text{by Proposition 5 again} \\ & = & \mathtt{eval}^{\text{o}}\,(c[\gamma]) & \text{as we just saw.} \end{array}$$

5 Related work

Beyond the already discussed, closely related work [20, 22, 28], let us mention recent work on the structure needed to collapse the composition of an active and a waiting state into a language machine configuration [18,20]. Unlike in our work is that acyclicity is not built into the OGS game, but proved after the fact.

The issue of infinite chattering was also studied in game semantics [1, 3, 16] for showing that total strategies are closed under composition. This notion of infinite chattering thus differs from the one studied here, which allows 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.

Finally, there is a lot of work on unique solutions of (co)recursive equations. Deducing bisimilarity of two LTSs 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 [27], which is still explored today [6]. Let us also mention the category-theoretic work on monads with iteration operators [7, 11, 24, 25], which recently culminated in a widely unifying approach [12], that features an abstract notion of guardedness. Links with the interaction trees library [28] have been established, showing that the `itree` datatype, considered up to weak bisimilarity, forms a coinductive resumption monad, i.e., it computes cofree coalgebras for a functor of the form $A \mapsto T(X + \Sigma(A))$. The resulting monad is furthermore complete Elgot (i.e., it admits potentially non-unique solutions of all equations), and iterative (i.e., it admits unique solutions of "guarded" equations).

6 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. We have demonstrated the expressiveness of our framework by instantiating it on a variety of simply-typed λ-calculi with control effects – although only one is treated here for lack of space.

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, i.e., 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. Semantics of interaction: an introduction to game semantics. *Semantics and Logics of Computation*, 14(1), 1997.
2. Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005.
3. Pierre Clairambault and Russ Harmer. Totality in arena games. *Ann. Pure Appl. Log.*, 161(5):673–689, 2010.
4. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In Martin Odersky and Philip Wadler, editors, *Proc. 5th International Conference on Functional Programming*, pages 233–243. ACM, 2000.
5. Paul Downen and Zena M. Ariola. Compiling with classical connectives. *Log. Methods Comput. Sci.*, 16(3), 2020.
6. Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Divergence and unique solution of equations. *Log. Methods Comput. Sci.*, 15(3), 2019.
7. Calvin C. Elgot. Monadic computation and iterative algebraic theories. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 175–230. Elsevier, 1975.
8. Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proc. 14th Symposium on Logic in Computer Science*. IEEE, 1999.
9. Marcelo Fiore and Dmitrij Szamozvancev. Formal metatheory of second-order abstract syntax. *Proc. ACM Program. Lang.*, 6(POPL):1–29, 2022.
10. Marcelo P. Fiore. Second-order and dependently-sorted abstract syntax. In *Proc. 23rd Symposium on Logic in Computer Science*, pages 57–68. IEEE, 2008.
11. Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Julian Jakob. Unguarded recursion on coinductive resumptions. *Log. Methods Comput. Sci.*, 14(3), 2018.
12. Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg. Unifying guarded and unguarded iteration. In Javier Esparza and Andrzej S. Murawski, editors, *Proc. 20th International Conference on Foundations of Software Science and Computation Structures*, volume 10203 of *Lecture Notes in Computer Science*, pages 517–533, 2017.
13. André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In *Proc. 14th International Workshop on Logic, Language, Information and Computation*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007.
14. André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Information and Computation*, 208(5):545–564, 2010.
15. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
16. Martin Hyland. Game semantics. *Semantics and logics of computation*, 14:131, 1997.
17. Guilhem Jaber and Andrzej S. Murawski. Complete trace models of state and control. In Nobuko Yoshida, editor, *Proc. 30th European Symposium on Programming*, volume 12648 of *Lecture Notes in Computer Science*, pages 348–374. Springer, 2021.
18. Guilhem Jaber and Andrzej S. Murawski. Compositional relational reasoning via operational game semantics. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 1–13. IEEE, 2021.
19. James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Proc. 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.

20. James Laird. A Curry-style semantics of interaction: From untyped to second-order lazy $\lambda\mu$-calculus. In Jean Goubault-Larrecq and Barbara König, editors, *Proc. 23rd International Conference on Foundations of Software Science and Computation Structures*, volume 12077 of *Lecture Notes in Computer Science*, pages 422–441. Springer, 2020.
21. Søren B. Lassen and Paul Blain Levy. Typed normal form bisimulation. In Jacques Duparc and Thomas A. Henzinger, editors, *Proc. 21st International Workshop on Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2007.
22. Paul Blain Levy and Sam Staton. Transition systems over games. In *Proc. 29th Symposium on Logic in Computer Science*, pages 64:1–64:10. ACM, 2014.
23. Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *J. Funct. Program.*, 1(3):287–327, 1991.
24. Stefan Milius. Completely iterative algebras and completely iterative monads. *Inf. Comput.*, 196(1):1–41, 2005.
25. Stefan Milius and Tadeusz Litak. Guard your daggers and traces: Properties of guarded (co-)recursion. *Fundam. Informaticae*, 150(3-4):407–449, 2017.
26. Robin Milner. Fully abstract models of typed *lambda*-calculi. *Theor. Comput. Sci.*, 4(1):1–22, 1977.
27. Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
28. Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.
29. Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, USA, 2009. AAI3358066.