Game semantics for interface middleweight Java.
Murawski, AS; Tzevelekos, N

© 2017 ACM, Inc

For additional information about this publication click this link.
http://qmro.qmul.ac.uk/xmlui/handle/123456789/23397

Information about this research object was correct at the time of download; we occasionally make corrections to records, please therefore check the published record when citing. For more information contact scholarlycommunications@qmul.ac.uk
Game Semantics for Interface Middleweight Java

Andrzej S. Murawski
DIMAP and Department of Computer Science
University of Warwick

Nikos Tzevelekos
School of Electronic Engineering and Computer Science
Queen Mary, University of London

Abstract
We consider an object calculus in which open terms interact with the environment through interfaces. The calculus is intended to capture the essence of contextual interactions of Middleweight Java code. Using game semantics, we provide fully abstract models for the induced notions of contextual approximation and equivalence. These are the first denotational models of this kind.

Categories and Subject Descriptors D.3.1 [Formal Definitions and Theory]: Semantics; F.3.2 [Semantics of Programming Languages]: Denotational semantics

Keywords Full Abstraction, Game Semantics, Contextual Equivalence, Java

1. Introduction
Denotational semantics is charged with the construction of mathematical universes (denotations) that capture program behaviour. It concentrates on compositional, syntax-independent modelling with the aim of illuminating the structure of computation and facilitating reasoning about programs. Many developments in denotational semantics have been driven by the quest for full abstraction [21]: a model is fully abstract if the interpretations of two programs are the same precisely when the programs behave in the same way (i.e. are contextually equivalent). A faithful correspondence like this opens the path to a broad range of applications, such as compiler optimisation and program transformation, in which the preservation of semantics is of paramount importance.

Recent years have seen game semantics emerge as a robust denotational paradigm [4, 6, 12]. It has been used to construct the first fully abstract models for a wide spectrum of programming languages, previously out of reach of denotational semantics. Game semantics models computation as an exchange of moves between two players, representing respectively the program and its computational environment. Accordingly, a program is interpreted as a strategy in a game corresponding to its type. Intuitively, the plays that game semantics generates constitute the observable patterns that a program produces when interacting with its environment, and this is what underlies the full abstraction results. Game semantics is compositional: the strategy corresponding to a compound program phrase is obtained by canonical combinations of those corresponding to its sub-phrases. An important advance in game semantics was the development of nominal games [3, 17, 26], which underpinned full abstraction results for languages with dynamic generative behaviours, such as the $\nu$-calculus [3], higher-order concurrency [18] and ML references [24]. A distinctive feature of nominal game models is the presence of names (e.g. memory locations, references names) in game moves, often along with some abstraction of the store.

The aim of the present paper is to extend the range of the game approach towards real-life programming languages, by focussing on Java-style objects. To that end, we define an imperative object calculus, called Interface Middleweight Java (IMJ), intended to capture contextual interactions of code written in Middleweight Java (MJ) [9], as specified by interfaces with inheritance. We present both equational (contextual equivalence) and inequational (contextual approximation) full abstraction results for the language. To the best of our knowledge, these are the first denotational models of this kind.

Related Work While the operational semantics of Java has been researched extensively [7], there have been relatively few results regarding its denotational semantics. More generally, most existing models of object-oriented languages, such as [8, 15], have been based on global state and consequently could not be fully abstract.

On the other hand, contextual equivalence in Java-like languages has been studied successfully using operational approaches such as trace semantics [2, 13, 14] and environmental bisimulation [16]. The trace-based approaches are closest to ours and the three papers listed also provide characterizations of contextual equivalence. The main difference is that traces are derived operationally through a carefully designed labelled transition system and, thus, do not admit an immediate compositional description in the style of denotational semantics.

However, similarities between traces and plays in game semantics indicate a deeper correspondence between the two areas, which also manifested itself in other cases, e.g. [20] vs [19]. At the time of writing, there is no general methodology for moving smoothly between the two approaches, but we believe that there is scope for unifying the two fields in the not so distant future.

In comparison to other game models, ours has quite lightweight structure. For the most part, playing consists of calling the opponent’s methods and returning results to calls made by the opponent. In particular, there are no justification pointers between moves. This can be attributed to the fact that Java does not feature first-class higher-order functions and that methods in Java objects cannot be updated. On the other hand, the absence of pointers makes definitions of simple notions, such as well-bracketing, less direct, since the dependencies between moves are not given explicitly any
The calculus is a stripped down version of Middleweight Java (IMJ) features of the model. The developments in automata theory over infinite alphabets [10], and fresh-register automata in particular [23, 27], to account for the nominal length at most two!

\( \Delta | \Gamma \vdash a : I (a) \in \Gamma \) 
\( \Delta | \Gamma \vdash \text{skip} : \emptyset \) 
\( \Delta | \Gamma \vdash \text{null} : I \in \text{dom}(\Delta) \) 
\( \Delta | \Gamma \vdash i : \text{int} \)

more and need to be inferred from plays. The latter renders strategy composition non-standard. Because it is impossible to determine statically to which arena a move belongs, the switching conditions (cf. [6]) governing interactions become crucial for determining the strategy responsible for each move. Finally, it is worth noting that traditional copypack links are by definition excluded from our setting: a call/return move for a given object cannot be copypacked by the other player, as the move has a fixed polarity, determined by the ownership of the object. In fact, identity strategies contain plays of length at most two!

**Further Directions** In future work, we would like to look for automata-theoretic representations of fragments of our model in order to use them as a foundation for a program verification tool for Java programs. Our aim is to take advantage of the latest developments in automata theory over infinite alphabets [10], and fresh-register automata in particular [23, 27], to account for the nominal features of the model.

## 2. The language IMJ

We introduce an imperative object calculus, called Interface Middleweight Java (IMJ), in which objects are typed using interfaces. The calculus is a stripped down version of Middleweight Java (MJ), expressive enough to expose the interactions of MJ-style objects with the environment.

### Definition 1. Let Ints, Flds and Meths be sets of interface, field and method identifiers. We range over them respectively by \( I, f, m \) and variants. The types \( \theta \) of IMJ are given below, where \( \theta \) stands for a sequence \( \theta_1, \ldots, \theta_n \) of types (for any \( n \)). An interface definition \( \Theta \) is a finite set of typed fields and methods. An interface table \( \Delta \) is a finite assignment of interface definitions to interface identifiers.

### Types \( \Theta \) ::=
- \( \text{void} \) | \( \text{int} \) | \( I \)

- \( \text{Flds} \) ::=
  - \( \emptyset \) | \( \{ f : \theta, \Theta \} \) | \( \{ m : \theta \to \theta, \Theta \} \)

- \( \text{ITbls} \) ::=
  - \( \emptyset \) | \( \{ I : \Theta, \Delta \} \) | \( \{ I(I) : \Theta, \Delta \} \)

We write \( I(I') : \Theta \) for interface extension: interface \( I \) extends \( I' \) with fields and methods from \( \Theta \). We stipulate that the extension relation must not lead to circular dependencies. Moreover, each identifier \( f, m \) can appear at most once in each \( \Theta \), and each \( I \) can be defined at most once in \( \Delta \) (i.e. there is at most one element of \( \Delta \) of the form \( I : \Theta \) or \( I(I') : \Theta \)). Thus, each \( \Theta \) can be seen as a finite partial function \( \Theta : \text{Flds} \cup \text{Meths} \to \text{Types} \). We write \( \Theta(f) \) and \( \Theta.m \) for \( \Theta(m) \). Similarly, \( \Delta \) defines a partial function \( \Delta : \text{Ints} \to \text{IFlds} \) given by

\[
\Delta(I) = \begin{cases}
\Theta & \Delta(I') \in \Theta \\
\Delta(I(I')) : \Theta & \Delta(I(I')) \in \Delta \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

An interface table \( \Delta \) is well-formed if, for all interface types \( I, I' \):

- if \( I' \) appears in \( \Delta(I) \) then \( I' \in \text{dom}(\Delta) \).
- if \( (I(I') : \Theta) \in \Delta \) then \( \text{dom}(\Delta(I')) \cap \text{dom}(\Theta) = \emptyset \).

Henceforth we assume that interface tables are well-formed. Interface extensions yield a subtyping relation. Given a table \( \Delta \), we define \( \Delta \vdash \theta_1 \leq \theta_2 \) by the following rules.

\[
(I(I') : \Theta), \Delta \vdash I \leq I';
\]

\[
\Delta \vdash \theta \leq \Theta';
\]

\[
\text{dom}(\Delta) \subseteq \Theta';
\]

We might omit \( \Delta \) from subtyping judgements for economy.

### Definition 2. Let \( h \) be a countably infinite set of object names, which we range over by \( a \) and variants. IMJ terms are listed below, where we let \( x \) range over a set of variables \( \text{Vars} \), and \( i \) over \( I \). Moreover, \( \oplus \) is selected from some set of binary numeric operators. \( M \) is a method-set implementation. Again, we stipulate that each \( m \) appear in each \( M \) at most once.

\[
M ::= x | a | \text{skip} | \text{null} | i | M \oplus M | \text{let } x = M \text{ in } M | M | M \text{ if } M \text{ else } M | (I)M | \text{new}(x : I ; M) | M.f | M.f.M := M | M.m(M) \]

\[
M\text{Imps} \supseteq M \supseteq \emptyset | (m : \lambda \bar{x}.M), M
\]

The terms are typed in contexts comprising an interface table \( \Delta \) and a variable context \( \Gamma = \{ x_1 : \theta_1, \ldots, x_n : \theta_n \} \cup \{ a_1 : I_1, \ldots, a_m : I_m \} \) such that any interface in \( \Gamma \) occurs in \( \text{dom}(\Delta) \).

The typing rules are given in Figure 1.

For the operational semantics, we define the sets of term values, heap configurations and states by:

\[
\text{TVals} \supseteq v ::= \text{skip} | i | \text{null} | a
\]

\[
\text{HCnfs} \supseteq V ::= \emptyset | (f : v), V
\]

\[
\text{States} \supseteq S : A \to \text{Ints} \times (\text{HCnfs} \times \text{MImps})
\]

If \( S(a) = (I, (V, M)) \) then we write \( S(a) : I \), while \( S(a).f \) and \( S(a).m \) stand for \( V.f \) and \( M.m \) respectively, for each \( f \) and \( m \).

Given an interface table \( \Delta \) such that \( I \in \text{dom}(\Delta) \), we let the default heap configuration of type \( I \) be

\[
V_I = \{ f : v_I | \Delta(I).f = \theta \},
\]

where \( v_{\text{null}} = \text{skip}, v_{\text{int}} = 0 \) and \( v_{\text{null}} = \text{null} \). The operational semantics of IMJ is given by means of a small-step transition relation...
between terms-in-state, presented in Figure 2. The transition relation uses evaluation contexts \(E\) that are defined as follows.

\[
E := \text{let } x = _{=} \text{ in } M | \_ \oplus M | i \oplus _{=} | \_ = M | a = _{=}
\]

\[
| \_ \text{ if } _{=} \text{ then } M \_ \text{ else } M'| | (\_)\_ | \_ \_f | \_ = f := M | a.f := _{=}
\]

\[
| \_ . m(M) | a.m(v_1, \ldots, v_{i-1}, M_{i+2}, \ldots, M_n)
\]

Given \(\Delta[0] \vdash M : \text{void}\), we write \(M \downarrow\) if there exists \(S\) such that \((\emptyset, M) \rightarrow^* (S, \text{skip})\).

**Definition 3.** Given \(\Delta[\Gamma] \vdash M : \theta (i = 1, 2)\), we shall say that \(\Delta[\Gamma] \vdash M : \theta \) contextually approximate \(\Delta[\Gamma] \vdash M_2 : \theta \) if, for all \(\Delta' \supseteq \Delta\) and all contexts \(C\) such that \(\Delta[0] \vdash C[M_1]: \text{void}\), void, if \(C[M_1] \downarrow\) then \(C[M_2] \downarrow\). We then write \(\Delta[\Gamma] \vdash M_1 \equiv M_2 : \theta\). Two terms are contextually equivalent (written \(\Delta[\Gamma] \vdash M_1 \equiv M_2 : \theta\)) if they approximate each other.

For technical convenience, IMJ features the let construct, even though it is definable: given \(\Delta[\Gamma], x : \theta' \vdash M : \theta\) and \(\Delta[\Gamma] \vdash M' : \theta'\), consider \(\text{new}(x : \mathcal{I}: m : \lambda x.M).m(M')\), where \(\mathcal{I}\) is a fresh interface with a single method \(m : \theta \rightarrow \theta'\). As usual, we write \(M; M'\) for \(\text{let } x = M \in M'\), where \(x\) is not free in \(M'\).

Although IMJ does not have explicit local variables, they could easily be introduced by taking \(\text{let } x = \text{new}(y : \mathcal{I}_y)\) in \(\ldots\), where \(\mathcal{I}_y\) has a single field of type \(\theta\). In the same manner, one can define variables and methods that are private to objects, and invisible to the environment through interfaces.

**Example 1 (16).** Let \(\Delta = \{\text{Empty} : \emptyset, \text{Cell} : (\text{get} : \text{void} \rightarrow \text{Empty}, \text{set} : \text{Empty} \rightarrow \text{void}), \text{Var} : (\text{val} : \text{Empty}, \text{Var} : (\text{val} : \text{int})\}\) and consider the terms \(\Delta[0] \vdash M_1 : \text{Cell} (i = 1, 2)\) defined by:

\[
M_1 \equiv \text{let } v = \text{new}(x : \text{Var}_E)\text{ in new}(x : \text{Cell}; M_1)
\]

\[
M_2 \equiv \text{let } b = \text{new}(x : \text{Var}_I)\text{ in new}(x : \text{Cell}; M_2)
\]

with

\[
M_1 = (\text{get} : \lambda().v.\text{val}, \text{set} : \lambda y.(v.\text{val} := y))
\]

\[
M_2 = (\text{get} : \lambda().\text{if } (b.\text{val}) \text{ then } (b.\text{val} := 0; v_1.\text{val}) \text{ else } (b.\text{val} := 1; v_2.\text{val}), \text{set} : \lambda y.(v_1.\text{val} := y; v_2.\text{val} := y))
\]

We have \(\Delta[0] \vdash M_1 \equiv M_2 : \text{Cell}\). Intuitively, each of the two implementations of Cell corresponds to recording a single value of type \text{Empty} (using set) and providing access to it via get. The difference lies in the way the value is stored: a single private variable is used in \(M_1\), while two variables are used in \(M_2\). However, in the latter case the variables always hold the same value, so it does not matter which of the variables is used to return the value.

The game semantics of the two terms will turn out to consist of plays of the shape \(s^0 n^x G_0^{x} S_1 G_1^{y} \cdots S_k G_k^{y}\), where

\[
G_i = \text{call } n.\text{get}(+)E_0 \text{ ret } n.\text{get}(\text{null})E_0 \quad i = 0
\]

\[
\text{call } n.\text{get}(+)E_i \text{ ret } n.\text{get}(\text{null})E_i \quad i > 0
\]

\[
S_i = \text{call } n.\text{set}(n_i)E_i \text{ ret } n.\text{set}(+)E_i
\]

and \(\Sigma_i = \{n \mapsto (\text{Cell}, \emptyset)\} \cup \{n_j \mapsto (\text{Empty}, \emptyset) \mid 0 < j \leq i\}\). Intuitively, the plays describe all possible interactions of a Cell object. The first two moves \(s^0 n^x G_0^{x}\) correspond to object creation. After that, the \(G_i\) segments represent the environment reading the current content (initially having null value), while the \(S_i\) segments correspond to updating the content with a reference name provided by the environment. The stores \(\Sigma_i\) attached to moves consist of all names that have been introduced during the interaction so far.

It is worth noting that, because IMJ has explicit casting, a context can always guess the actual interface of an object and extract any information we may want to hide through casting.

**Example 2.** Let \(\Delta = \{\text{Empty} : \emptyset, \text{Point} : (\text{Empty} : (x : \text{int}, y : \text{int})\}\) and consider the terms \(\Delta[0] \vdash M_i : \text{Empty} (i = 1, 2)\) defined by:

\[
M_1 \equiv \text{new}(x : \text{Empty})
\]

\[
M_2 \equiv \text{let } p = \text{new}(x : \text{Point})\text{ in } p.x := 0; p.y := 1; (\text{Empty})p.
\]

In our model they will be interpreted by the strategies \(s_1 = \{\epsilon, s^0 n^x \mapsto (\text{Empty}, 0)\}\) and \(s_2 = \{\epsilon, s^0 n^x \mapsto (\text{Point}, (0, y) \mapsto 1)\}\) respectively. Using e.g. the casting context \(\Delta[\text{Cell}]\), we can see that \(\Delta[0] \vdash M_1 \not\equiv M_2 : \text{Empty}\). On the other hand, Theorem 20 will imply \(\Delta[0] \vdash M_1 \equiv M_2 : \text{Empty}\).

On the whole, IMJ is a compact calculus that strips down Middleweight Java to the essentials needed for interface-based interaction. Accordingly, we suppressed the introduction of explicit class hierarchy, as it would remain invisible to the environment anyway and any class-based internal computations can be represented using standard object encodings [1].

At the moment the calculus allows for single inheritance for interfaces only, but extending it to multiple inheritance is not problematic. The following semantic developments only rely on the assumption that \(\subseteq\) must not give rise to circularities.

3. The game model

In our discussion below we assume a fixed interface table \(\Delta\).

The game model will be constructed using mathematical objects (moves, plays, strategies) that feature names drawn from the set \(\mathcal{A}\). Although names underpin various elements of our model, we do not want to delve into the precise nature of the sets containing them. Hence, all of our definitions preserve name-invariance, i.e. our objects are (strong) nominal sets [11, 26]. Note that we do not
need the full power of the theory but mainly the basic notion of name-permutation. For an element \( x \) belonging to a (nominal) set \( X \) we write \( \nu(x) \) for its name-support, which is the set of names occurring in \( x \). Moreover, for any \( x, y \in X \), we write \( x \sim y \) if there is a permutation \( \pi \) such that \( x = \pi(y) \).

We proceed to define a category of games. The objects of our category will be arenas, which are nominal sets carrying specific type information.

**Definition 4.** An arena is a pair \( A = (M_A, \xi_A) \) where:

- \( M_A \) is a nominal set of moves;
- \( \xi_A : M_A \to (\mathbb{A} \to \text{Ints}) \) is a nominal typing function;

such that, for all \( m \in M_A \), \( \text{dom}(\xi_A(m)) = \nu(m) \).

We start by defining the following basic arenas,

\[ M_A = \{\xi \mid \xi : \nu(\xi) \in \mathbb{N} \}, \quad \mathbb{I} = \{\nu, \emptyset\}, \quad \mathbb{Z} = \{\nu, \emptyset\} \]

for all interfaces \( I \). Given arenas \( A \) and \( B \), we can form the arena \( A \times B \) by:

\[
M_{A \times B} = \{ (\nu, \emptyset) \mid \nu \in \nu(m) \cap \nu(n) \}
\]

\[
\xi_{A \times B}((\nu, \emptyset)) = \{ \xi_A(m, a) \land \xi_B(n, a) \leq \xi_A(m, a) \}
\]

Another important arena is \( #(I, \ldots, I_n) \), with:

\[
M_{#(I)} = \{ (a_1, \ldots, a_n) \mid a_i \in A^n \land a_i \text{ is distinct} \}
\]

for all \( n \in \mathbb{N} \). In particular, \( A^{#(1)} = 1 \) for any type \( \theta \), we set \( \text{Val}_\theta \) to be the set of semantic values of type \( \theta \), given by:

\[
\text{Val}_\text{oid} = M_1, \quad \text{Val}_\text{int} = M_E, \quad \text{Val}_I = M_I.
\]

For each type \( \theta \), we set \( \text{Val}_\theta = \text{Val}_\text{oid} \times \cdots \times \text{Val}_{\theta_n} \times \cdots \times \text{Val}_{\theta_m} \).

Let us define a store \( \Sigma \) to be a type-preserving finite partial function from names to object types and field assignments, that is, \( \Sigma : \mathbb{A} \to \text{Ints} \times (\text{Fields} \to \text{Val}) \) such that \( \Sigma \) is finite and

\[
\Sigma(a) : I \land \Delta(I), f = \theta \quad \implies \quad \Sigma(a), f = v \land \Sigma - v - \theta,
\]

where the new notation is explained below. First, assuming \( \Sigma(a) = (I', \phi) \), the judgement \( \Sigma(a) : I \) holds iff \( I = I' \) and \( \Sigma(a), f \) stands for \( \Phi(f) \). Next we define typing rules for values in store contexts:

\[
v \in \text{Val}(\Sigma) \iff \text{Val}_\theta(v) \land \Sigma(v) : I \lor v = \nu I
\]

and write \( \Sigma + v \leq \theta \) for \( \Sigma + v : \theta \lor (\Sigma + v : I \land \Delta(I) \leq \theta) \).

We let \( \text{Sto} \) be the set of all stores. We write \( \text{dom}(\Sigma) \) for the set of all \( f \) such that \( \Sigma(a), f \) is defined. We let \( \text{Sto} \) contain all stores \( \Sigma \) such that:

\[
\forall a \in \text{dom}(\Sigma), f \in \text{dom}(\Sigma(a)), \Sigma(a), f \in \{*, 0, \nu I\}
\]

and we call such a \( \Sigma \) a default store.

Given arenas \( A \) and \( B \), plays in \( AB \) will consist of sequences of moves (with store) which will be either moves from \( M_A \cup M_B \), or moves representing method calls and returns. Formally, we define:

\[
M_{AB} = M_A \cup M_B \cup \text{Calls} \cup \text{Retns}
\]

where we set \( \text{Calls} = \{ \text{call } a.m(\tilde{v}) \mid a \in A \land \tilde{v} \in \text{Val} \} \) and \( \text{Retns} = \{ \text{ret } a.m(v) \mid a \in A \land v \in \text{Val} \} \).

**Definition 5.** A legal sequence in \( AB \) is a sequence of moves from \( M_{AB} \) that adheres to the following grammar (Well-Bracketing), where \( m_A \) and \( m_B \) range over \( M_A \) and \( M_B \) respectively.

\[
\begin{align*}
L_{AB} &::= \epsilon \mid m_A(\Sigma) \mid m_B(\Sigma) \\
X &::= Y \mid \nu(a.m(\tilde{v})) X \\
Y &::= \epsilon \mid YY \mid (\text{call } a.m(\tilde{v})) Y (\text{ret } a.m(v))
\end{align*}
\]

We write \( L_{AB} \) for the set of legal sequences in \( AB \). In the last clause above, we say that \( a.m(\tilde{v}) \) justifies \( \text{ret } a.m(v) \).

To each \( s \in L_{AB} \) we assign a polarity function \( p \) from move occurrences in \( s \) to the set \( \text{Polar} = \{ O, P \} \). Polariites represent the two players in our game reading of programs: \( O \) is the Opponent and \( P \) is the Proponent in the game. The latter corresponds to the modelled program, while the former models the possible computational environments surrounding the program. Polariites are complemented via \( \overline{\text{Polar}} = \{ P \} \) and \( \overline{\text{Polar}} = \{ O \} \). In addition, the polarity function must satisfy the condition:

- For all \( m \in X \in M_S \) occurring in \( s \) we have \( p(m) = O \) and \( p(m) = P \); (O-starting)

- If \( mn \) are consecutive moves in \( s \) then \( p(n) \in p(m) \). (Alternation)

It follows that there is a unique \( p \) for each legal sequence \( s \), namely the one which assigns \( O \) precisely to those moves appearing in odd positions in \( s \).

A move-with-store in \( AB \) is a pair \( m_\Sigma \) with \( \Sigma \in \text{Sto} \) and \( m \in M_{AB} \). For each sequence \( s \) of moves-with-store we define the set of available names of \( s \) by:

\[
\text{Av}(\epsilon, \emptyset) = \emptyset, \quad \text{Av}(s.m_\Sigma) = \Sigma^* (\text{Av}(s) \cup \nu(m))
\]

where, for each \( X \subseteq \mathbb{A} \), we let \( \Sigma^*(X) = \bigcup \Sigma^*(X) \), with \( \Sigma^0(X) = X \). \( \Sigma^{i+1}(X) = \nu(\Sigma^i(X)) \).

That is, a name is available in \( s \) just if it appears inside a move in \( s \), or it can be reached from an available name through some store in \( s \). We write \( s \) for the underlying sequence of moves of \( s \) (i.e. \( \pi_1(s) \)), and let \( \mathbb{S} \) denote the prefix relation between sequences. If \( s.m_\Sigma \subseteq s \) and \( a \in \nu(m_\Sigma) \cup \nu(s) \) then we say \( a \) is introduced by \( m_\Sigma \) in \( s \). In such a case, we define the owner of the name \( a \) in \( s \), written \( o(a) \), to be \( p(m) \) (where \( p \) is the polarity associated with \( s \)). For each polarity \( X \subseteq \{ O, P \} \) we let

\[
X(s) = \{ a \in \nu(s) \mid o(a) = X \}
\]

be the set of names in \( s \) owned by \( X \).

**Definition 6.** A play in \( AB \) is a sequence of moves-with-store \( s \) such that \( s \) is a legal sequence and, moreover, for all \( s'.m_\Sigma \subseteq s \):

- It holds that \( \text{dom}(\Sigma) = \text{Av}(s'.m_\Sigma) \). (Fragility)

- If \( a \in \text{dom}(\Sigma) \) with \( \Sigma(a) : I \) then:
  - if \( m \in X \subseteq \{ A, B \} \), then \( I \leq \xi_X (m, a) \);
  - for all \( n \in \text{Sto} \), if \( a \in \text{dom}(T) \) then \( T(a) : I \);
  - if \( \Delta(I).m = \theta \to \emptyset \) then:
    - if \( m = \text{call } a.m(\tilde{v}) \) then \( \Sigma + \tilde{v} : \theta \) for some \( \tilde{v} \neq \emptyset \),
    - if \( m = \text{ret } a.m(v) \) then \( \Sigma + v : \theta \) for some \( \theta' \leq \theta \).

(Well-classing)

- If \( m = \text{call } a.m(\tilde{v}) \) then \( o(a) \in p(m) \). (Well-calling)

We write \( P_{AB} \) for the set of plays in \( AB \).

---

1. By abuse of notation, we frequently write instead “\( a \) is introduced by \( m \) in \( s \)”. Recall also that \( \nu(s) \) collects all names appearing in \( s \); in particular, \( \nu(m_1 \cdots m_i) = \nu(m_1) \cup \nu(S_2) \cup \cdots \cup \nu(m_i) \cup \nu(S_i) \).
Note above that, because of well-bracketing and alternation, if \( m = \text{ret } a.m(v) \) then well-calling implies \( o(a) = p(m) \). Thus, the frugality condition stipulates that names cannot appear in a play in unreachable parts of a store (cf. [17]). Moreover, well-calling ensures that the typing information in stores is consistent and adheres to the constraints imposed by \( \Delta \) and the underlying arenas. Finally, well-calling implements the specification that each player need only call the other player’s methods. This is because calls to each player’s own methods cannot in general be observed and so should not be accounted for in plays.

Given arenas \( A, B, C \), next we define interaction sequences, which show how plays from \( AB \) and \( BC \) can interact to produce a play in \( AC \). The sequences will rely on moves with stores, where the moves come from the set:

\[
\begin{align*}
M_{ABC} = M_A \cup M_B \cup M_C \cup \text{Calls } \cup \text{Retns }.
\end{align*}
\]

The moves will be assigned polarities from the set:

\[
\begin{align*}
\text{Pol}2 = \{O_L, P_L, O_L P_R, P_L O_R, O_R, P_R \}.
\end{align*}
\]

The index \( L \) stands for “left”, while \( R \) means “right”. The indices indicate which part of the interaction (\( A, B \) or \( C \)) a move comes from, and what polarity it has therein. We also consider an auxiliary notion of pseudo-polarities:

\[
\begin{align*}
OO = \{O_L, O_R\}, \quad PO = \{P_L, P_L O_R\}, \quad OP = \{P_R, O_L P_R\}.
\end{align*}
\]

Each polarity has an opposite pseudo-polarity determined by:

\[
\begin{align*}
O_R = O_L, P_R = PO, \quad P_R = O_L P_R, \quad O_L = OP.
\end{align*}
\]

Finally, each \( X \in \{AB, BC, AC\} \) has a designated set of polarities given by:

\[
\begin{align*}
p(AB) &= \{O_L, P_L, P_L O_R, P_L O_R\}, \\
p(BC) &= \{O_R, P_R, O_L P_R, O_L P_R\}, \\
p(AC) &= \{O_L, P_L, O_R, P_R\}.
\end{align*}
\]

Note the slight abuse of notation with \( p \) as it is also used for move polarities.

Suppose \( X \in \{AB, BC, AC\} \). Consider a sequence \( s \) of moves-with-store from \( ABC \) (i.e. a sequence with elements \( m^\Sigma \) with \( m \in M_{ABC} \)) along with an assignment \( p \) of polarities from \( \text{Pol}2 \) to moves of \( s \). Let \( s^\Sigma \subseteq X \) be the subsequence of \( s \) containing those moves-with-store \( m^\Sigma \) of \( s \) for which \( p(m) \subseteq p(X) \). Additionally, we define \( s^\Sigma \subseteq X \) to be \( \gamma(s \subseteq X) \), where the function \( \gamma \) acts on moves-with-store by restricting the domains of stores to available names:

\[
\gamma(e) = e, \quad \gamma(s^\Sigma) = \gamma(s) m^\Sigma(\text{Av}(s^\Sigma)).
\]

**Definition 7.** An interaction sequence in \( ABC \) is a sequence \( s \) of moves-with-store in \( ABC \) satisfying the following conditions:

- For each \( s^\Sigma \subseteq s, \text{dom}(\Sigma) = \text{Av}(s^\Sigma), \) (Frugality)
- If \( s^\Sigma \subseteq s \) and \( a \in \text{dom}(\Sigma) \) with \( \Sigma(a) : T \) then:
  - if \( m \in M_X \), for \( X \in \{AB, BC\} \), then \( I \subseteq \chi(m,a) \);
  - for all \( n^T \in s^\Sigma \), if \( a \in \text{dom}(T) \) then \( T(a) : T \);
  - if \( \Delta(T), m = \bar{\theta} \rightarrow \theta \) then:
    - if \( m = \text{call } a.m(v) \) then \( \Sigma \vdash \bar{\theta} : \theta \bar{\theta} \) for some \( \theta \leq \bar{\theta} \),
    - if \( m = \text{ret } a.m(v) \) then \( \Sigma \vdash \bar{\theta} : \theta \) for some \( \theta \leq \bar{\theta} \). (Well-classing)
- There is a polarity function \( p \) from move occurrences in \( s \) to \( \text{Pol}2 \) such that:
  - For all \( m \in M_X \) (\( X = A, B, C \)) occurring in \( s \) we have \( p(m_A) = O_L, p(m_B) = P_L O_R \) and \( p(m_C) = P_R \);
  - If \( mn \) are consecutive moves in \( s \) then \( p(n) \subseteq p(m) \). (Alternation)

**Figure 3.** Interaction diagram for \( \text{Int}(ABC) \). The diagram specifies the alternation of polarities in interaction sequences. Transitions are labelled by move polarities, while \( OO \) is the initial state.

- If \( s^\Sigma \subseteq s \) then \( m = \text{call } a.m(v) \) implies \( o(a) \in \overline{p(m)} \). (Well-calling)
- For each \( X \in \{AB, BC, AC\} \), \( s \parallel X \in L_X \). (Projecting)
- If \( s^\Sigma \subseteq s \) and \( m = \text{ret } a.m(v) \) then there is a move \( n^\Sigma \) in \( s' \) such that, for all \( X \) such that \( p(m) \subseteq p(X) \), \( \gamma \) is the justifier of \( m \) in \( s \parallel X \). (Well-returning)
- Laird’s conditions [17]:
  - \( P(s \parallel AB) \cap P(s \parallel BC) = \emptyset \);
  - \( P(s \parallel AB) \cup P(s \parallel BC) \cap O(s \parallel AC) = \emptyset \);
- For each \( s^\Sigma \subseteq s \) ending in \( m^\Sigma \) and each \( a \in \text{dom}(T) \), if \( p(m) \subseteq PO \) and \( a \notin \bar{\nu}(s^\Sigma, \bar{\theta}, AB) \),
  - or \( p(m) \subseteq OP \) and \( a \notin \bar{\nu}(s^\Sigma, BC) \),
  - or \( p(m) \subseteq OO \) and \( a \notin \bar{\nu}(s^\Sigma, AC) \), then \( \Sigma(a) = T(a) \).

We write \( \text{Int}(ABC) \) for the set of interaction sequences in \( ABC \).

Note that, by projecting and well-returning, each return move in \( s \) has a unique justifier. Next we show that the polarities of moves inside an interaction sequence are uniquely determined by the interaction diagram of Figure 3. The diagram can be seen as an automaton accepting \( s \), for each \( s \in \text{Int}(ABC) \). The edges represent moves by their polarities, while the labels of vertices specify the polarity of the next (outgoing) move. For example, from \( OO \) we can only have a move \( m \) with \( p(m) \in \{O_L, O_R\} \), for any \( p \).

**Lemma 1.** Each \( s \in \text{Int}(ABC) \) has a unique polarity function \( p \).

**Proof.** Suppose \( s \in \text{Int}(ABC) \). We claim that the alternation, well-calling, projecting and well-returning conditions uniquely specify \( p \). Consider the interaction diagram of Figure 3, which we read as an automaton accepting \( s \), call it \( A \). The edges represent moves by their polarities, while the labels of vertices specify the polarity of the next (outgoing) move. By projecting we obtain that the first element of \( s \) is some \( m_A \) and, by alternation, its polarity is \( O_L \). Thus, \( OO \) is the initial state.

We now use induction on \( |s| \) to show that \( A \) has a unique run on \( s \). The base case is trivial, so suppose \( s = s'k \). By induction hypothesis, \( A \) has a unique run on \( s' \), which reaches some state \( X \). We do a case analysis on \( m \). If \( m \in M_A \cup M_B \cup M_C \), then there is a unique edge accepting \( m \) and, by alternation, this edge must depart from \( X \). If, on the other hand, \( m = \text{call } a.m(v) \) then the fact that \( o(a) \in \overline{p(m)} \) gives two possible edges for accepting \( m \). But observe that no combination of such edges can depart from \( X \). Finally, let \( m = \text{ret } a.m(v) \) be justified by some \( n \) in \( s' \). Then, by well-bracketing, \( n \) is the justifier of \( m \) in all projections, and hence the edge accepting \( m \) must be the opposite of the one accepting \( n \) (e.g. if \( m \) is accepted by \( O_L \) then \( n \) is accepted by \( P_L \)).

Next we show that interaction sequences project to plays. The projection of interaction sequences in \( ABC \) on \( AB, BC \) and \( AC \)
leads to the following definition of projections of polarities,

\[ \pi_{AB}(X_L) = X \quad \pi_{AB}(X_L Y_R) = X \quad \pi_{AB}(Y_R) = \text{undef.} \]

\[ \pi_{BC}(X_L) = \text{undef.} \quad \pi_{BC}(X_L Y_R) = Y \quad \pi_{BC}(Y_R) = Y \]

\[ \pi_{AC}(X_L) = X \quad \pi_{AC}(X_L Y_R) = \text{undef.} \quad \pi_{AC}(Y_R) = Y \]

where \( X, Y \in \{ O, P \} \). We can now show the following.

**Lemma 2.** Let \( s \in \text{Int}(ABC) \). Then, for each \( X \in \{ AB, BC, AC \} \) and each \( m^2 \) in \( s \), if \( p(m) \in p(X) \) then \( \pi_X(p(m)) = p_X(m) \), where \( p_X \) is the polarity function of \( s \) in \( X \).

**Proof.** We show this for \( X = AB \), the other cases are proven similarly, by induction on \( |s| \geq 0 \); the base case is trivial. For the inductive case, if \( m \) is the first move in \( s \) in polarity in \( p(AB) \) then, by projecting, \( m \in M_A \) and therefore \( p(m) = O_L \) and \( p(AB) = m \), as required. Otherwise, let \( n \) be the last move in \( s \) in polarity in \( p(AB) \) before \( m \). By IH, \( p_{AB}(n) = \pi_{AB}(p(n)) \). Now, by projecting, \( p_{AB}(m) = p_{AB}(n) \) and observe that, for all \( X \in \{ p(n), \pi_{AB}(X) = p_{AB}(p(n)) \} \), so in particular \( \pi_{AB}(p(m)) = \pi_{AB}(p(n)) = p_{AB}(n) = p_{AB}(m) \).

The following lemma formulates a taxonomy on names appearing in interaction sequences.

**Lemma 3.** Let \( s \in \text{Int}(ABC) \). Then,

1. \( \nu(s) = O(s \upharpoonright AC) \upharpoonright P(s \upharpoonright AB) \cup P(s \upharpoonright BC) \);
2. if \( s = \text{tm}^2 \) and:
   - \( p(m) \in OO \) and \( s \upharpoonright AC = t \text{m}^2 \).
   - or \( p(m) \in PO \) and \( s \upharpoonright AB = t \text{m}^2 \).
   - or \( p(m) \in OP \) and \( s \upharpoonright BC = t \text{m}^2 \).

then \( \nu(t) \cap \nu(m^2) \subseteq \nu(t) \) and, in particular, if \( m \) introduces name \( a \) in \( t \text{m}^2 \) then \( m \) introduces \( a \) in \( s \).

**Proof.** For 1, by definition of interactions we have that these sets are disjoint. It therefore suffices to show the left-to-right inclusion. Suppose that \( a \in \nu(s) \) is introduced in some \( m^2 \) in \( s \) with \( p(m) \in PO \), and let \( s \upharpoonright AB = \ldots \cdot m^2 \ldots \). If \( a \in \nu(m^2) \) then \( a \in P(s \upharpoonright AB) \), as required. Otherwise, by Laird’s last set of conditions, \( a \) is copied from the store of the move preceding \( m^2 \) in \( s \), a contradiction to its being introduced at \( m^2 \). Similarly if \( p(m) \in OP \). Finally, if \( p(m) \in OO \) then we work similarly, considering \( O(s \upharpoonright AC) \).

For 2, we show the first case, and the other cases are similar. It suffices to show that \( \nu(m^2 \cdot \nu(t)) \cap \nu(t) = \emptyset \). So suppose \( a \in \nu(m^2 \cdot \nu(t)) \), therefore \( a \in O(s \upharpoonright AC) \). But then we cannot have \( a \in \nu(t) \) as the latter, by item 1, would imply \( a \in P(s \upharpoonright AB) \cup P(s \upharpoonright BC) \).

**Proposition 4.** For all \( s \in \text{Int}(ABC) \), the projections \( s \upharpoonright AB, s \upharpoonright BC \) and \( s \upharpoonright AC \) are plays in \( AB, BC \) and \( AC \) respectively.

**Proof.** By frugality of \( f \) and application of \( \gamma \), all projections satisfy frugality. Moreover, well-classing is preserved by projections. For well-calling, let \( m = \text{call} \text{a.m}(s) \) be a move in \( s \) and let \( n^2 \) be the move introducing \( a \) in \( s \). Suppose \( p(m) \in P(AB) \) and let us assume \( p_{AB}(m) = O \). We need to show that \( o_{AB}(m) = P \). By \( p_{AB}(m) = O \) we obtain that \( p(m) \in \{ O_L, O_L, P_O \} \). and, by well-calling of \( s \), we have that \( o_X(a) = O \). Thus, \( p(m) \in PO \) and, by Lemma 3, \( a \) introduces \( a \) in \( s \). By definition, \( o_{AB}(m) = P \). Then we obtain \( p(m) \in OO \cup OP \) and therefore, by Lemma 3, \( a \in P(s \upharpoonright BC) \cup O(s \upharpoonright AC) \). Thus, by the same lemma, \( a \notin P(s \upharpoonright AB) \) and hence \( o_{AB}(a) = O \). The cases for the other projections are shown similarly.

In our setting programs will be represented by strategies between arenas. We shall introduce them next after a few auxiliary definitions. Intuitively, strategies capture the observable computational patterns produced by a program.

Let us define the following notion of subtyping between stores. For \( \Sigma, \Sigma' \in \text{Str} \), \( \Sigma \subseteq \Sigma' \) holds if, for all names \( a \),

\[ \Sigma'(a) : T' \Rightarrow \Sigma(a) \leq T' \land \forall f \in \text{dom}(\Sigma(a)), \Sigma(a).f = \Sigma'(a).f \]

In particular, if \( a \) is in the domain of \( \Sigma' \), \( \Sigma' \) may contain more information about \( a \) because of assigning to \( a \) a larger interface. Accordingly, for plays \( s, s' \in P_{AB} \), we say that \( s \) is an O-extension of \( s' \) if \( s \) and \( s' \) agree on their underlying sequences, while their stores may differ due to subtyping related to O-names. Where such subtyping leads to \( s \) having stores with more fields than those in \( s' \), \( P \) is assumed to copy the values of those fields. Formally, \( s \leq_O s' \) is defined by the rules:

\[ s \leq_O s' \quad s \subseteq \Sigma \quad \Sigma' \subseteq s \leq_O \Sigma' \quad T \subseteq T' \]

where \( \Sigma \) extends \( \Sigma' \) by \( T \) if:

- for all \( a \in \text{dom}(\Sigma) \land \text{dom}(\Sigma') \), \( \Sigma(a) = T(a) \);
- for all \( a \) and \( f \in \text{dom}(\Sigma(a)) \land \text{dom}(\Sigma'(a)) \), \( \Sigma'(a).f = T(a).f \).

The utility of O-extension is to express semantically the fact that the environment of a program may use up-casting to inject in its objects additional fields (and methods) not accessible to the program.

**Definition 8.** A strategy \( \sigma \) in \( AB \) is a non-empty set of even-length plays from \( P_{AB} \) satisfying the conditions:

- if \( sm^2 \cdot n^2 \in \sigma \) then \( s \in \sigma \). (Even-prefix closure)
- if \( sm^2, sn^2 \in \sigma \) then \( sm^2 \sim sn^2 \). (Determinacy)
- if \( s \in \sigma \) and \( s \sim t \) then \( t \in \sigma \). (Equivalence)
- if \( s \land \sigma \) and \( t \land s \in \sigma \). (O-extension)

We write \( \sigma : A \to B \). When \( \sigma \) is a strategy in \( AB \). If \( \sigma : A \to B \) and \( \tau : B \to C \), we define their composition \( \sigma \tau \) by:

\[ \sigma \tau = \{ s \upharpoonright AC \mid s \in \sigma \land \tau \}

where \( \sigma \land \tau = \{ s \in \text{Int}(ABC) \mid s \upharpoonright AB \in \sigma \land s \upharpoonright BC \in \tau \} \).

In definitions of strategies we may often leave the presence of the empty sequence implicit, as the latter is a member of every strategy. For example, for each arena \( A \), we define the strategy:

\[ \mathbf{id}_A : A \to A = \{ m^2 \in m^2 \in P_{AA} \}

The next series of lemmata allow us to show that strategy composition is well defined.

**Lemma 5.** If \( sm^2 \cdot sn^2 \in \sigma \land \tau \) with \( p(m) \notin OO \) then \( sm^2 \sim sn^2 \). Hence, if \( s_1 m^2, s_2 n^2 \in \sigma \land \tau \) then \( s_1 m^2 \sim s_2 n^2 \).

**Proof.** For the latter part, if \( s_1 = \cdot s_2 \) then, since \( \pi \cdot (s_2 n^2) \in \sigma \land \tau \), by former part of the claim we have \( s_1 m^2 \sim \pi \cdot (s_2 n^2) \) so \( s_1 m^2 \sim s_2 n^2 \).

Now, for the former part, suppose WLOG that \( p(m) \in PO \). Then, by the interaction diagram, we also have \( p(n) \in PO \). As \( sm^2, sn^2 \in AB \in \sigma \), by determinacy of \( \sigma \) we get \( s^m n^2 \sim s^m n^2 \).

Recall that, for any nominal set \( X \) and \( x, y \in X \), we write \( x \sim y \) just if there is a permutation \( \pi \) such that \( \pi(x) = y \).
with \( s^t \) \( m^t = \text{sm}^t \ |_\gamma AB \) and \( s^t \) \( n^t = \text{sn}^t \ |_\gamma AB \). We therefore have \((s^t, m^t) \sim (s^t, n^t)\) and, trivially, \((s, s^t) \sim (s, s^t)\).
Moreover, by Lemma 3, \( \nu (m^t) \cap \nu (s) \subseteq \nu (s^t) \) and \( \nu (n^t) \cap \nu (s) \subseteq \nu (s^t) \) hence, by Strong Support Lemma [26], \( \text{sm}^t \sim \text{sn}^t \). By Laird’s last set of conditions, the remaining values of \( \Sigma, T \) are determined by the last store in \( s \), hence \( \text{sm}^t \sim \text{sn}^t \). 

Lemma 6. If \( s_1, s_2 \in \sigma \| \tau \) and \( \text{moves in with polarities in } p(AC) \) and \( s_1 \mid \gamma AC \) then \( s_1 \sim s_2 \).

Proof. By induction on \( |\gamma AC| \geq 0 \). The base case is encompassed in \( s_1 = s_1^t m^t \) with \( p(m) \in \text{OO}, i = 1, 2 \), where note that by IH \( m \) will have the same polarity in \( s_1, s_2 \). Then, by IH we get \( s_1^t = \pi \cdot s_2^t \), for some \( \pi \). Let \( s_1^t m^t = s_1 \gamma AC \), for \( i = 1, 2 \), so in particular \( s_1^t = \pi \cdot s_2^t \) and therefore \((s_1^t, s_1^t) \sim (s_2^t, s_2^t)\). Moreover, by hypothesis, we trivially have \((m^t, s_1^t) \sim (m^t, s_2^t)\) and hence, by Lemma 3 and Strong Support Lemma [26], we obtain \( s_1^t m^t \sim s_2^t m^t \) which implies \( s_1 \sim s_2 \). By Laird’s conditions. Suppose now \( s_1 = s_1^t m^t, i = 1, 2 \), with \( p(m) \in P(AC) \) \( \setminus \text{OO} \) and the last move in \( s_1^t \) being the last move in \( s_1^t \) having polarity in \( p(AC) \). By IH, \( s_1 \sim s_2 \). Then, by consecutive applications of Lemma 5, we obtain \( s_1 \sim s_2 \). 

Proposition 7. If \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) then \( \sigma ; \tau : A \rightarrow C \).

Proof. We show that \( \sigma \cdot \tau \) is a strategy. Every-prefix closure and equivariance are clear. Moreover, since each \( s \in \sigma \| \tau \) has even-length projections in \( AB \) and \( BC \), we can show that its projection in \( AC \) is even-length too. For \( O \)-extension, if \( s \in \sigma \| \tau \) is even-length \( s = u \mid_\gamma AB \) and \( v \mid_\gamma AC \) and \( u \leq \nu (\sigma) \), we can construct \( v \mid_\gamma BC \) such that \( t = v \mid_\gamma BC \) and \( v \leq \nu (\tau) \). This is used for constructing parts of \( \tau \) in an analogous way as for \( BC \). Finally, for 

The above result shows that strategies are closed under composition. We can prove that composition is associative and, consequently, obtain a category of games.

Proposition 8. For all \( \rho : A \rightarrow B, \sigma : B \rightarrow C \) and \( \tau : C \rightarrow D \), \((\rho ; \sigma) ; \tau = \rho (\sigma ; \tau)\).

Definition 9. Given a class table \( \Delta \), we define the category \( \mathcal{G}_\Delta \) having arenas as objects and strategies as morphisms. Identity morphisms are given by \( \text{id}_\Delta \), for each arena \( \Delta \).

Note that neutrality of identity strategies easily follows from the definitions and, hence, \( \mathcal{G}_\Delta \) is well defined. In the sequel, when \( \Delta \) can be inferred from the context, we shall write \( \mathcal{G}_\Delta \) simply as \( \mathcal{G} \).

As a final note, for class tables \( \Delta \subseteq \Delta' \), we can define a functor \( \Delta / \Delta' : \mathcal{G}_\Delta \rightarrow \mathcal{G}_{\Delta'} \) which acts as the identity map on arenas, and sends each \( \sigma : A \rightarrow B \) of \( \mathcal{G}_\Delta \) to:

\[(\Delta / \Delta')(\sigma) = \{ s \in P_{\Delta}^\Delta \mid \exists \delta \in \sigma, s \leq \delta \} \]

where \( P_{\Delta}^\Delta \) refers to plays in \( \mathcal{G}_{\Delta'} \). In the other direction, we can define a strategy transformation:

\[(\Delta' / \Delta')(\sigma) = \sigma \cap P_{\Delta}^\Delta \]

which satisfies \( \Delta' / (\Delta / \Delta')(\sigma) = \sigma \). 

4. Soundness
Here we introduce constructions that will allow us to build a model of IMJ. We begin by defining a special class of strategies. A strategy \( \sigma : A \rightarrow B \) is called evaluated if there is a function \( f_\sigma : M_A \rightarrow M_B \) such that:

\[\{ m \bar{X} m_B \in P_{AB} \mid m_B = f_\sigma (m_A) \} .\]

Note that equivariance of \( \sigma \) implies that, for all \( m_A \in M_A \) and permutations \( \pi \), it holds that \( \pi \cdot f_\sigma (m_A) = f_\sigma (\pi \cdot m_A) \). Thus, in particular, \( f_\sigma (m_A) \subseteq f_\sigma (m_A) \).

Recall that, for arenas \( A \) and \( B \), we can construct a product arena \( A \times B \). We can also define projection strategies:

\[\pi_1 : A \times B \rightarrow A = \{ (m_A, m_B) \mid m_A \in \{ m \bar{X} m_B \} \} \]

and, analogously, \( \pi_2 : A \times B \rightarrow B \). Note that the projections are evaluated. Moreover, for each object \( A \),

\[\{ m \bar{X} m_A \in P_{A} \} \]

is the unique evaluated strategy of type \( A \rightarrow 1 \).

Given strategies \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) with \( \tau \) evaluated, we define:

\[(\sigma, \tau) : A \rightarrow B \times C = \{ m_\Sigma [s \mid (m_\Sigma, f_\sigma (m_A))] \mid m_\Sigma \in \sigma \} \]

where we write \( s[m \mid m_B] \) for the sequence obtained from \( s \) by replacing any occurrences of \( m_B \) in it by \( m' \) (note that there can be at most one occurrence of \( m_B \) in \( s \)).

The above structure yields products for evaluated strategies.

Lemma 9. Evaluated strategies form a wide subcategory of \( \mathcal{G} \) which has finite products, given by the above constructions.

Moreover, for all \( \sigma : A \rightarrow B \) and \( \tau : B \rightarrow C \) with \( \tau \) evaluated, \( \langle \sigma, \tau \rangle, \pi_1 = \sigma \) and \( \langle \sigma, \pi_2 \rangle = \langle \sigma, \id_\Delta \rangle \), \( \langle \pi_2, \pi_2 \rangle \).

Using the above result, we can extend pairings to general \( \sigma : A \rightarrow B \) and \( \tau : A \rightarrow C \) by:

\[(\sigma, \tau) = A ((\sigma ; \id_\Delta), B \times A ((\pi_2, \tau), s)) \times C \rightarrow B \times C \]

where \( \cong \) is the isomorphism \( \langle \pi_2, \pi_2 \rangle \). The above represents a notion of left-pairing of \( \sigma \) and \( \tau \), where the effects of \( \sigma \) precede those of \( \tau \). We can also define a left-tensor between strategies:

\[\times \tau = A \times B ((\pi_1, \pi_2), s) \rightarrow A' \times B' \]

for any \( \sigma : A \rightarrow A' \) and \( \tau : B \rightarrow B' \).

Lemma 10. Let \( \tau' : A' \rightarrow A, \sigma : A \rightarrow B_1, \tau : A \rightarrow B_2, \sigma_1 : B_1 \rightarrow C_1 \) and \( \sigma_2 : B_2 \rightarrow C_2 \), with \( \tau \) evaluated. Then \( \tau' (\sigma, \tau), \langle \sigma_1, \pi_2, \sigma_2 \rangle = \langle \tau', (\sigma, \sigma), \sigma_1, \tau', \tau, \sigma_2 \rangle \).

Proof. The result follows from the simpler statements:

\[(\sigma, \id_\Delta) = (\tau, \sigma, \tau), \langle \sigma, \id_\Delta \rangle, (\sigma', \pi_2, \sigma_2) = (\sigma, \id_\Delta, \sigma', \id_\Delta)\]

for all appropriately typed \( \sigma, \sigma', \tau \), with \( \tau \) evaluated, and Lemma 9. 

An immediate consequence of the above is:

\[A ((\sigma, \tau), B_1 \times B_2 ((\pi_2, \sigma_2), s), C_1 \times C_2) = A ((\sigma, \pi_1, \sigma_2), s) \rightarrow C_1 \times C_2 \]

More generally, Lemma 10 provides us with naturality conditions similar to those present in Freyd categories [25] or, equivalently, categories with monadic products [22].

We also introduce the following weak notion of coproduct:

Given strategies \( \sigma, \tau : A \rightarrow B \), we define:

\[\{ \sigma, \tau \} : \mathbb{Z} \rightarrow A \rightarrow B = \{ (1, m_\Sigma), m_\Sigma \} \]

and

\[\{ (0, m_\Sigma) \mid m_\Sigma \in \tau \} \]
Setting $i : 1 \rightarrow Z = \{ * \mid i \}$, for each $i \in Z$, we can follow the showing.

**Lemma 11.** For all strategies $\sigma' : A' \rightarrow A$ and $\sigma, \tau : A \rightarrow B$,
- $\langle \emptyset, 0, i \rangle ; [\sigma, \tau] = \tau$ and $\langle \emptyset, 1, i \rangle ; [\sigma, \tau] = \sigma$;
- if $\sigma'$ is evaluated then $\langle 1, m, \sigma' \rangle; [\sigma, \tau] = [\sigma'; \sigma, \sigma']$.

Method definitions in IMJ induce a form of exponentiation:

$$\phi_{m_1, m_2}^{(\lambda) \gamma \Sigma} \vdash \phi_{\gamma \Sigma}$$

the modelling of which requires some extra semantic machinery. Traditionally, in call-by-value game models, exponentiation leads to ‘effectless’ strategies, corresponding to higher-order value terms. In our case, higher-order values are methods, manifesting themselves via the objects they may inhabit. Hence, exponentiation necessarily passes through generation of fresh object names containing these values. These considerations give rise to two classes of strategies introduced below.

We say that an even-length play $s \in P_{AB}$ is total if it is either empty or $s = m_A^\Sigma m_B^{\Sigma}_s s$ and:

- $T \in St_{\Sigma} \text{ and } \nu(m_B) \cup \nu(\Sigma) \subseteq \nu(m_A)$,
- if $s = s' \cdot m^\Sigma_B s''$ and $a \in dom(\Sigma), \nu(\gamma(m^\Sigma_B m_{\Sigma} s', s''))$, for
  $\Sigma_0 \in St_{\Sigma}$ such that $\gamma(m^\Sigma_A m_{\Sigma} s', s') \in P_{AB}$, then $a \notin \nu(n)$ and $T'(a) = \Sigma'(a)$.

We write $P^B_{AB}$ for the set of total plays in $AB$. Thus, in total plays, the initial move $m_A$ is immediately followed by a move $m_B$, and the initial store $\Sigma$ is invisible to $P$ in the sense that $P$ cannot use its names nor their values. A strategy $\phi : A \rightarrow B$ is called single-threaded if it consists of total plays and satisfies the conditions:

- for all $m_A^\Sigma \in P_{AB}$ there is $m_B^\Sigma m_B^{\Sigma}_s s' \in \phi$;
- if $m_A^\Sigma m_B^{\Sigma}_s s' \in \phi$, then $m_B^\Sigma m_B^{\Sigma}_{\Sigma_0} s\in \phi$, for $\Sigma_0 \in St_{\Sigma}$;
- if $m_A^\Sigma m_B^{\Sigma}_s^s' s' \in \phi$ and $a \in \nu(T)$ then $\phi = s$.

Thus, single-threaded strategies reply to every initial move $m_A^\Sigma$ with a move $m_B^\Sigma$ which depends only on $m_A$ (i.e. $P$ does not read before playing). Moreover, $m_B^\Sigma$ does not change the values of $\Sigma$ ($P$ does not write) and may introduce some fresh objects, albeit with default values. Finally, plays of single-threaded strategies consist of just one thread, where a thread is a total play in which there can be at most one call to names introduced by its second move.

Conversely, given a total play starting with $m_A^\Sigma m_B^{\Sigma}_s s'$, we can extract its threads by tracing back for each move in $s$ the method call of the object $a \in (\nu(T))$ it is related to. Formally, for each total play $s = m_A^\Sigma m_B^{\Sigma}_s \cdot s$ with $s \neq 0$, the thread move of $s$, written $\text{thr}(s)$, is given by induction:

- $\text{thr}(s) = a \cdot m(\mathcal{V})^\Sigma$, if $p(m) = P$;
- if $m_B^\Sigma m_B^{\Sigma}_s s' \in \phi$ and $a \in \nu(T)$;
- if $m_B^\Sigma m_B^{\Sigma}_s s' \in \phi$, then $a \in P(s) \nu(\nu(T))$ and $n$ introduces $a$;
- if $m_B^\Sigma m_B^{\Sigma}_s s' \in \phi$, then $p(m) = O$ and $n$ justifies $m$.

If $s = s' \cdot n^T s''$ with $|s'| \geq 1$, we set $\text{thr}(s') = \text{thr}(s' n^T)$. Then, the current thread of $s$ is the subsequence of $s$ containing only moves with the same threader move as $s$, that is, if $\text{thr}(s) = m^\Sigma$ and $s = m_A^\Sigma m_B^{\Sigma}_s m^\Sigma T s'$ then:

$$[s] = m_A^\Sigma m_B^{\Sigma}_s m^\Sigma T (s' \mid m^\Sigma)$$

where the restriction retains only those moves $n^T s'$ of $s'$ such that $\text{thr}(n^T) = m^\Sigma$. We extend this to the case of $|s| \geq 2$ by setting $[s] = s$. Finally, we call a total play $s \in P_{AB}$ thread-independent if for all $s' m^\Sigma_B \subseteq \nu_{even} s$ with $|s'| > 2$:

- if $\gamma(s') \subseteq \nu(\Sigma') \cap \nu(s'') \subseteq \nu(s'')$;
- if $s'$ ends in some $n^T$ and $a \in \nu(\Sigma) \nu(\gamma(s') \subseteq \nu(s''))$ then $\Sigma'(a) = \Sigma'(a)$.

We write $P^B_{AB}$ for the set of thread-independent plays in $AB$.

We can now define strategies which occur as interlavings of single-threaded ones. Let $\phi : A \rightarrow B$ be a single-threaded strategy. We define:

$$\phi^+ = \{ s \in P_{AB} \mid \forall s' \subseteq \nu_{even} s, \gamma(s') \in \phi \}.$$

**Lemma 12.** $\phi^+$ is a strategy, for each single-threaded $\phi$.

**Proof.** Equivariance. Even-prefix closure and $O$-extension follow from the corresponding conditions on $\phi$. For determinacy, if $sm^\Sigma s \in \phi^+$ with $|s| > 0$ then, using determinacy of $\phi$ and the fact that $P$-moves do not change the current thread, nor do they modify or use names from other threads, we can show that $sm^\Sigma s \in sm^\Sigma s$.

We say that a strategy $\sigma$ is thread-independent if $\sigma = \sigma^+$ for some single-threaded strategy $\tau$. Thus, thread-independent strategies do not depend on initial stores and behave in each of their threads in an independent manner. Note in particular that evaluated strategies are thread-independent (and single-threaded).

**Lemma 13.** For $\sigma : A \rightarrow B$ and $\tau : A \rightarrow C$ be strategies with $\tau$ thread-independent. Then, $\langle \sigma, \tau \rangle ; \pi_1 = \sigma$ and $\langle \sigma, \tau \rangle ; \pi_2 = \tau$.

**Proof.** The former claim is straightforward. For the latter, we observe that the initial effects of $\alpha$ and $\tau$ commute: on initial move $m_A^\Sigma$, $\tau$ does not read the store updates that $\sigma$ includes in its response $m_B^\Sigma$, while $\sigma$ cannot access the names created by $\tau$ in its second move $m_C^\Sigma$.

It is worth noting that the above lemma does not suffice for obtaining categorical products. Allowing thread-independent strategies to create fresh names in their second move breaks universality of pairings. Considering, for example, the strategy:

$$\sigma : 1 \rightarrow I \times I = \{ * \rightarrow a, b \} \in \{ \Sigma \mid St_{\Sigma} \}$$

we can see that $\sigma \notin \{ \Sigma; \pi_1, \pi_2 \}$, as the right-hand-side contains plays of the form $a. (a, b)^T$ with $a \neq b$. We can now define an appropriate notion of exponential for our games. Let us assume a translation assigning an arena $[\theta]$ to each type sequence $\theta$. Moreover, let $I$ be an interface such that

$$\Delta(\theta) \mid \text{Metth} = \{ m_1 : \theta_1 \rightarrow \theta_1, \ldots, m_n : \theta_n \rightarrow \theta_n \}$$

where $\theta_1 = \theta_{i_1}, \ldots, \theta_{i_ \ell}$, for each $i$. For any arena $A$, given single-threaded strategies $\phi_1, \ldots, \phi_n : A \rightarrow I$ such that, for each $i$, if $m_A^\Sigma \subseteq \nu_{even} \phi_i$ then $a \notin \nu(\Sigma) \nu(\langle \phi_1, \ldots, \phi_n \rangle : A \rightarrow I = \bigcup_{i=1}^n \phi_i)$.

Note that the $a$ above is fresh for each $m_A^\Sigma$ (i.e. $a \notin \nu(m_A^\Sigma)$).
Let now \( \sigma_1, \ldots, \sigma_n \) be strategies with \( \sigma_i : A \times \{\theta_i\} \to \{\theta_i\} \). For each \( i \), we define the single-threaded strategy \( \Lambda(\sigma_i) : A \to I : \)
\[
\Lambda(\sigma_i) = \{ m_{\Delta A}^{\Sigma m, \Sigma^T T} \text{call } a.m.(v)^{\Sigma^T} s \in P_{\Delta A} | \gamma((a.m.(v)^{\Sigma^T} s) \in \sigma_i \}
\cup \{ m_{\Delta A}^{\Sigma m, \Sigma^T T} \text{call } a.m.(v)^{\Sigma^T} s \text{ ret } a.m.(v)^{\Sigma^T} s' \in P_{\Delta A} | \\
\gamma((m_{\Delta A}^{\Sigma m, \Sigma^T T} \text{call } a.m.(v)^{\Sigma^T} s \text{ ret } a.m.(v)^{\Sigma^T} s') \in \sigma_i) \cup \{ m_{\Delta A}^{\Sigma m, \Sigma^T T} \in P_{\Delta A} | \\
(\forall s \in P_{\Delta A} | (\forall v^T \Sigma^T T) s \in \sigma_i) \}
\]
where \( a \notin v(\Sigma, \Sigma^T T) \) and \( T(a) : I \). By definition, \( \Lambda(\sigma_i) \) is single-threaded. Therefore, setting
\[
\Lambda(\sigma_1, \ldots, \sigma_n) = \langle (\Lambda(\sigma_1), \ldots, \Lambda(\sigma_n)) \rangle : A \to I,
\]
we obtain a thread-independent strategy implementing a simultaneous currying of \( \sigma_1, \ldots, \sigma_n \). In particular, given translations \( [M] \) for each method in a method-set implementation \( M \), we can construct:
\[
[M] : [\Gamma] \to I = \Lambda([M_1], \ldots, [M_n]).
\]
Finally, we define evaluation strategies \( \text{ev}_{m_1} : I \times \{\theta_i\} \to \{\theta_i\} \) by (taking even-length prefixes of):
\[
\text{ev}_{m_1} = \{(a, \Sigma \Sigma^T \text{call } a.m.(v)^{\Sigma^T} \text{ ret } a.m.(v)^{\Sigma^T} s \in P_{\Delta A} | \Sigma(a) \leq I \}
\]
where \( A_i = \{I \times \{\theta_i\} \} \). We can now show the following natural mapping from groups of strategies in \( A \times \{\theta_i\} \to \{\theta_i\} \) to thread-independent ones in \( A \to I \).

**Lemma 14.** Let \( \sigma_1, \ldots, \sigma_n \) be as above, and let \( \tau : A' \to A \) be evaluated. Then,
- \( \Lambda(\sigma_1, \ldots, \sigma_n) \times \text{id : ev}_{m_1} \times \{\theta_i\} \) by (taking even-length prefixes of):

Apart from dealing with exponentials, in order to complete our translation we need also to address the appearance of \( x : I \) in the rule:
\[
\Gamma, x : I, \Delta \vdash M : \Theta
\]
Recall that
\[
[M] : [\Gamma] \times I \to I
\]
is obtained using exponentiation. Thus, the second move of \([M] \) will appear in the right-hand-side \( I \) above and will be a fresh name \( b \) which will serve as a handle to the methods of \( M \); in order to invoke \( m : \lambda e. M \) on input \( \overline{e} \), the Opponent would have to call \( b.m(\overline{e}) \). The remaining challenge is to merge the two occurrences of \( I \) in (1). We achieve this as follows. Let us assume a well-formed extension \( \Delta' \) of \( \Delta \):
\[
\Delta' = ([I'] : ([f'] : [I]), \Delta)
\]
that is, \( I' \) contains a single field \( f' \) of type \( I \). We next define the strategy \( \kappa_{\Delta} : I \to I' \times I \) of \( G_{\Delta} \):
\[
\kappa_{\Delta} = \{ (a,b,\sigma_0) \Sigma m.b.m.(v)^{\Sigma^T} \text{call } a.m.(v)^{\Sigma^T} \text{ ret } a.m.(v)^{\Sigma^T} \}
\]
where \( m \in \text{dom}(\Delta(I)) \), \( b = \Sigma(\sigma_0)f' \), and \( \Sigma_0 \in \text{St}(\sigma_0) \) is such that \( \Sigma_0(a) : I \) and \( \Sigma_0(a) : I' \). We let \([\text{new}(x : I; M)] \) be the strategy:
\[
\begin{align*}
[\Gamma] \times I \times I \times I \to I \times I &\langle \text{id} \times [\text{dom}(\Delta)], \text{id} \times [\text{dom}(\Delta)], \text{id} \times [\text{dom}(\Delta)], \text{id} \times [\text{dom}(\Delta)] \rangle \times I \\
\text{as}_{\text{new}} : I \times \{\theta\} \to \{\theta\} &\{ (a, v)^{\Sigma^T} \in P_{\text{new}(x: I; M)} \}
\end{align*}
\]
for each field \( f \). Thus, object creation involves creating a pair of names \((a',a)\) with \( a : I \) and \( a' : I' \), where \( a \) is the name of the object we want to return. The name \( a' \) serves as a store where the handle of the method implementations, that is, the name created by the second move of \([M] \), will be passed. The strategy \( \kappa_{\Sigma^T} \), upon receiving a request \( \text{call } a.m.(v)^{\Sigma^T} \), simply forwards it to the respective method of \( a' f' \) and, once it receives a return value, copies it back as the return value of the original call.

Let \( \#(\overline{I}) : I \to \#(\overline{I}) = \{a^T \Sigma^T | a, s \text{ distinct} \} \), for each sequence of interfaces \( \overline{I} \). The latter has a right inverse \( \#(\overline{I})^{-1} : \#(\overline{I}) \to \overline{I} \) with the same plays. We can now define the semantic translation of terms.

**Definition 10.** The semantic translation is given as follows.
- Contexts \( \Gamma = \{ x_1 : \Theta_1, \ldots, x_n : \Theta_n \} \) are translated into arenas by
\[
[\Gamma] = \{[\theta_1] \times \cdots \times [\theta_n] \times \#(\Theta_1, \ldots, \Theta_m).
\]
where \( \{\text{void}\} = 1, \{\text{int}\} = \mathbb{Z} \) and \( [I] = I \).
- Terms are translated as in Figure 4 (top part).

In order to prove that the semantics is sound, we will also need to interpret terms inside state contexts. Let \( \Gamma \vdash M : \Theta \), with \( \Gamma = \Theta_1, \Gamma_2 \), where \( \Theta_1 \) contains only variables and \( \text{dom}(\Theta_2) = \text{dom}(S) \). A term-in-state-context \((S, M)\) is translated into the strategy:
\[
[S]_1 \vdash (S, M) = \{[\Gamma_1] \times [\Gamma_1], [I], [\Gamma_1] \times [\Gamma_1]) \times I \times \{\text{id} \times \#(\overline{I}) \}
\]
\[
\text{as}_{\text{new}} : I \times \{\theta\} \to \{\theta\} = \{ (a, v)^{\Sigma^T} \in P_{\text{new}(x: I; M)} \}
\]
Recall that
\[
[M] : \Gamma \times I \to I
\]
and \( \text{as}_{\text{new}} \) is the assignment strategy:
\[
\text{as}_{\text{new}} : I \times \{\theta\} \to \{\theta\} = \{ (a, v)^{\Sigma^T} \in P_{\text{new}(x: I; M)} \}
\]
4Note that \( x \) may appear free in \( M \); it stands for the keyword this of Java.
\[
\text{as}_{\text{new}} : I \times \{\theta\} \to \{\theta\} = \{ (a, v)^{\Sigma^T} \in P_{\text{new}(x: I; M)} \}
\]
5Here we omit wrapping \([M] \) inside \( \Delta' \), as well as wrapping the whole \([\text{new}(x: I; M)] \) in \( \Delta' / \Delta \), for conciseness.
commute (we write $\sigma$).

- $[\Gamma \vdash x_i : \theta_i] = [\Gamma] \xrightarrow{x_i \mapsto \theta_i}$;
- $[\Gamma \vdash \text{skip} : \text{void}] = [\Gamma] \xrightarrow{\bot}$;
- $[\Gamma \vdash \text{int} : \mathbb{Z}] = [\Gamma] \xrightarrow{|i|} \mathbb{Z}$;
- $[\Gamma \vdash \text{let} x = M' \in M : \theta] = [\Gamma] \xrightarrow{\text{let}(M', \theta)} [\Gamma] \times [\theta] [\Gamma']$;
- $[\Gamma \vdash (I)M : \mathcal{I}] = [\Gamma] \xrightarrow{\text{step}_{\mathcal{I}} I} [\Gamma] \times \mathbb{Z}$, where $\text{step}_{\mathcal{I}} I = I \mapsto I$.

**Figure 4.** The semantic translation of IMJ.

that the copy of $\mathcal{M}_i$ differs from the original just in the handle name (the one returned in the codomain of $[\mathcal{M}_i]$), but the latter is hidden via composition with $\mathcal{E}_m$. The latter diagram stipulates that if we create $\vec{a}$ with methods $\vec{M}$, then calling $a_i$ on $m$ is the same as calling $\mathcal{M}_i$ on $m$. The latter holds because of the way that $\vec{a}_m$ manipulates calls inside the interaction, by delegating calls to methods of $a_i$ to $\mathcal{M}_i$.

**Proposition 16 (Computational Soundness).** For all $M : \text{void}$, if $M \not\Downarrow$ then $[M] = \{ \ast, +, \ast \}$ (i.e. $[M] = [\text{skip}]$).

**Proof.** This directly follows from Correctness.

**Proposition 17 (Computational Adequacy).** For all $M : \text{void}$, if $[M] = \{ \ast, +, \ast \}$ then $M \not\Downarrow$.

**Proof.** Suppose, for the sake of contradiction, that $[M] = \{ \ast, +, \ast \}$ and $M \Downarrow$. We notice that, by definition of the translation for blocking constructs (castings and conditionals may block) and due to Correctness, if $M \Downarrow$ were due to some reduction step being blocked then the semantics would also block. Thus, $M \Downarrow$ must be due to divergence. Now, the reduction relation restricted to all rules but METHODCL is strongly normalising, as each transition decreases the size of the term. Hence, if $M$ diverges then it must involve infinitely many METHODCL reductions and our argument below shows that the latter would imply $[M] = \{ \ast \}$.

For any term $\Gamma \vdash N : \theta$ and $a \in A \setminus \text{dom}(\Gamma)$, construct $\Gamma_a \vdash N_a$, where $\Gamma_a = \Gamma \cup \{ a : \text{Var}_i \}$, by recursively replacing each subterm of $N$ of the shape $N'.m(N)$ with $a.f := (a.f + 1); N'.m(N)$.
Var_I is an interface with a sole field f : int. Observe that each s ∈ [I + N] induces some s′ ∈ [Γ_a + N_a] such that a appears in s′ only in stores (and in a single place in the initial move) and O never changes the value of a.f, while P never decreases the value of a.f. We write [Γ_a + N_a] for the subset of [Γ + N] containing precisely these plays. Then, take M_0 to be the term let x = new(x : Var_I) in (M_a[x/α]; x.f), where x a fresh variable. Because s ∈ [M_0], we get s ∈ [M_0] for some j ∈ Z. Consider now the infinite reduction sequence of (θ, M). It must have infinitely many METHOD steps, so suppose (θ, M) —→ (S, M′) contains j + 1 such steps. Then, we obtain (θ, M) —→ (S_a, (M′)_a; a.f), with S_a(a).f = j + 1. By Correctness, we have that s ∈ [S_a, (M′)_a; a.f] = [S_a; f(x #)]; [(M′)_a; a.f]. Since in [(M′)_a]_{o}, the value of a cannot decrease, and its initial value is j + 1 (as stipulated by S_a), we reach a contradiction.

5. Full Abstraction

Recall that, given plays s, s′, we call s an O-extensions of s′ (written s ≤_O s′) if s, s′ are identical except the type information regarding O-names present in stores: the types of O-names in s may be subtypes of those in s′. Then we define s ≤_p s′ for the dual notion involving P-names, i.e., s ≤_p s′ if s′ is a complete play (together with other plays implied by O-closure) originates (from a term.

Next we establish a definability result stating that any complete play (including the unfolding play) in a global variable. It should be clear that the recursive and O-closure makes this possible. This will be used to enforce the intended interlacing of threads after their composition (in the style of his setting of thread divergence).

Suppose P(s) = p_i, and p_i : I_p. Let Δ′ = Δ ⊕ (I_p → f_i; I_p → f_i). Consider s′ = (s, q)^{Σ^p_{\Gamma → I_p}} m_0^{Σ^p_{\Gamma → I_p}} m_1^{Σ^p_{\Gamma → I_p}} m_2^{Σ^p_{\Gamma → I_p}}, where Σ_q = Σ_q ⊕ {n ↦ (I_p, pull)} and Σ_i′ = Σ_i ⊕ {n ↦ (I_p, pull)}(p_i ↦ (I_p, pull)) (p_i ↦ (I_p, pull)) (p_i ↦ (I_p, pull)) (p_i ↦ (I_p, undefined)), p_i ∈ P(s).

Observe that s′ ∈ P_{Δ′}^p. The second transformation consists in storing the unfolding play in a global variable. Now we shall decompose m_1^{Σ^p_{\Gamma → I_p}} into threads. Recall that each of them is a subsequence of s′′ of the form

call p.m(u)_{Σ^p_{\Gamma → I_p}} t ret p.m(v)_{Σ^p_{\Gamma → I_p}}

where the segment t moves contents of the form call o or ret o for some o ∈ O(s). We would now like to invoke the IH for each thread but, since a thread is not a play, we do so for the related called play (h, n, q, u)^{Σ^p_{\Gamma → I_p}} (h, n, q, u)^{Σ^p_{\Gamma → I_p}} (h, n, q, u)^{Σ^p_{\Gamma → I_p}} (h, n, q, u)^{Σ^p_{\Gamma → I_p}}. Let us call the resultant term M_p,m,w,Σ. Next we combine terms related to the same p : I_p into an object definition

M_p := new(x : I_p, m : Λ)(w.cased(x, Σ_n)[M_p,m,w,Σ_n])

The case statement, which can be implemented in IMJ using nested if’s, is needed to recognize instances of w and Σ_n that really occur in threads related to p. In such cases the corresponding term M_p,m,w,Σ_n will be run. Otherwise, the statement leads to divergence. The term M for s can now be obtained by taking

let x_n = new(x : I_p) in
let x_h = new(x : History) in
let p_i = M_p, in
assert(q^Σ^p_{\Gamma → I_p}) ; x_n.f = x_p ; m(make(Σ^n_{\Gamma → I_p}) play(m_0))

where x_p = M_p_i represents a series of bindings (one for each P-name p_i ∈ P(s)), assert(h, n, q)^Σ^p_{\Gamma → I_p} is a conditional that converges if and only if the initial values of free Ω identifiers as well as values accessible through them are consistent with q and Σ_n respectively, make(Σ^n_{\Gamma → I_p}) is a sequence of assignments that set values to those specified in Σ^n_{\Γ → I_p} (up-casts need to be performed to ensure typability) and play(m_0) is i, null or, if m_0 is a name,
The proof uses the following play transformation. Given \( (\tau, \Sigma, n, s, \theta, \Delta') \) test, which is a term of the form \( (\tau, \Sigma, n, s, \theta, \Delta') \ Regents of FMCO, 2004.


