# An invitation to game semantics

Andrzej S. Murawski
Department of Computer Science
University of Warwick, UK

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

Game semantics is a flexible semantic theory that has led in recent years to an unprecedented number of full abstraction results for various programming paradigms. We present a gentle introduction to the subject, focussing on high-level ideas and examples with a view to providing a bridge to more technical literature.

## 1. INTRODUCTION

Denotational semantics aims at finding meaningful compositional interpretations (denotations) of programs, couched in a variety of mathematical universes. The quality of such interpretations can then be measured by understanding which programs are interpreted in the same way, i.e. by the same elements of the model. For example, injective interpretations will be faithful models of the syntax. In contrast to that, if the modelling objective is to characterise program behaviour then one would like the interpretations of two programs to coincide if and only if the two programs are equivalent. This criterion of modelling accuracy was introduced in the 1970s [Milner 1977], under the name *full abstraction*. It has ever since become the highest prize for the practising semanticist.

However, the quest for fully abstract models was not to be an easy one. Despite advances in domain theory, which fuelled early semantic research, the construction of fully abstract models turned out elusive, even though the techniques were ripe enough to provide many informative models for numerous complicated programming features. The efforts of the semantic community in the 1990s, focussed on the purely functional language PCF, have generated a wealth of results. Among them was the emergence of a new modelling approach, referred to as game semantics, which uses the metaphor of game playing as a foundation for building models.

## 2. GAMES

Game semantics views computation as a two-player dialogue between a program and the context (or environment) in which it was deployed. The interlocutors, or *players*, are traditionally called $O$ (Opponent) and $P$ (Proponent). The former represents the context, the latter corresponds to the program. Accordingly, a program is interpreted by a strategy for $P$ that tells $P$ how to conduct the dialogue. Game semantics is not about winning. Rather, the challenge is to design games in such a way that strategies express the *observable* behaviour of code interacting with its computational environment.

The kind of interactions that a piece of code may produce depends on its interactive potential: the more complicated the associated types the more interesting interactions we can expect. The type of free variables as well as the type of the phrase will all contribute to the shape of potential exchanges.

For example, if we simply take a constant, e.g. $\vdash 2016 : \mathsf{int}$, then one can imagine the following conversation between $O$ and $P$, but not much more.

| | |
|---|---|
| $O$ | What is the result (of evaluation)? |
| $P$ | 2016. |

Here $P$ plays according to the strategy that advocates responding with $2016$ to the initial question of the environment. The same situation will occur whenever we deal with a closed program $\vdash M : \mathsf{int}$ that evaluates to $2016$. Even though $M$ can be complicated, the only thing that the context can observe is the outcome of evaluation. Here, the interaction is admittedly quite shallow and it ends after one exchange.

This changes when we consider program phrases with more complicated types, such as $\vdash \lambda x^{\mathsf{int}}.x + 1 : \mathsf{int} \to \mathsf{int}$. Now, in addition to evaluating the term, the environment can call the function repeatedly using various arguments (call-by-value evaluation). This can be captured by a dialogue of the following kind, corresponding to the successor function. We stress that game semantics strives to capture exactly the observable behaviour. For instance, the addition operation $x + 1$ does not appear explicitly in the play.

| | |
|---|---|
| $O$ | What is the result? |
| $P$ | It's a function. |
| $O$ | What is the result if the argument is $3$? |
| $P$ | 4. |
| $O$ | What's the result for argument $5$? |
| $P$ | 6. |

More interesting exchanges are still possible for terms of type $\mathsf{int}$ provided they contain free variables, representing undefined procedures. Take, for instance, $f : \mathsf{int} \to \mathsf{int} \vdash f(f(0)) + 3$. In order for the phrase to produce a result, the context in which it is inserted must provide the missing information about $f$. Accordingly, we may expect the following conversation, in which $P$ probes the unknown parameter before returning the final value.

| | |
|---|---|
| $O$ | What is the result? |
| $P$ | What is $f(0)$? |
| $O$ | 5. |
| $P$ | What is $f(5)$? |
| $O$ | 4. |
| $P$ | 7. |

Note that the value $5$ returned by $O$ has been used in the following question. This corresponds to parameter passing: the value of $f(0)$ is passed to $f$ in order to compute $f(f(0))$.

Finally, let us consider an even more complicated example of

$$f : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \vdash \mathsf{let}\, g = f(0)\, \mathsf{in}\, \mathsf{let}\, h = f(1)\, \mathsf{in}\, g(2) + h(3) : \mathsf{int}$$

along with an associated dialogue.

| | |
|---|---|
| $O$ | What is the result? |
| $P$ | What is $f(0)$? |
| $O$ | It's a function. |
| $P$ | What is $f(1)$? |
| $O$ | It's a function. |
| $P$ | What is the result of applying the first function value to $2$? |
| $O$ | $4$. |
| $P$ | What is the result of applying the second function value to $3$? |
| $O$ | $5$. |
| $P$ | $9$. |

## 3. INSIDE A GAME MODEL

At a technical level, game semantics dialogues that capture observable aspects of computation are not expressed in English. Rather, they are expressed as sequences of abstract moves connected with pointers, called *justified sequences*. The sequences corresponding to the dialogues mentioned earlier are shown below.

$$\begin{tikzcd}
\star_q & 2016_a \arrow[l, bend right] \\
O & P
\end{tikzcd}$$

$$\begin{tikzcd}
\star_q & \star_a \arrow[l, bend right] & 3_{q,\star} \arrow[l, bend right] & 4_{a,\star} \arrow[l, bend right] & 5_{q,\star} \arrow[lll, bend right] & 6_{a,\star} \arrow[l, bend right] \\
O & P & O & P & O & P
\end{tikzcd}$$

$$\begin{tikzcd}
\star_q & 0_{q,f} \arrow[l, bend right] & 5_{a,f} \arrow[l, bend right] & 5_{q,f} \arrow[lll, bend right] & 4_{a,f} \arrow[l, bend right] & 7_a \arrow[lllll, bend right] \\
O & P & O & P & O & P
\end{tikzcd}$$

$$\begin{tikzcd}
\star_q & 0_{q,f} \arrow[l, bend right] & \star_{a,f} \arrow[l, bend right] & 1_{q,f} \arrow[lll, bend right] & \star_{a,f} \arrow[l, bend right] & 2_{q,f,\star} \arrow[lll, bend right] & 4_{a,f,\star} \arrow[l, bend right] & 3_{q,f,\star} \arrow[lll, bend right] & 5_{a,f,\star} \arrow[l, bend right] & 9_a \arrow[lllllllll, bend right] \\
O & P & O & P & O & P & O & P & O & P
\end{tikzcd}$$

Note that only initial moves need not have pointers. Any other move must be equipped with one ("justified") and its target cannot be chosen arbitrarily. Firstly, the targeted move must have been played earlier and, secondly, it must be related to the new move by a relation between moves called *enabling*. Thus, when defining a game in game semantics, one starts off by specifying the set of moves, their ownership ($O$ or $P$), kind (question or answer) and the enabling relation. This information is referred to as the underlying *arena*.

Arenas used in our examples are given below. The top moves are meant to belong to $O$ and then the ownership alternates between levels. The subscripts indicate which moves are questions and answers respectively. One move enables another if they are connected by an edge and the former lies one level above the latter. We use $i, j, k, l$ to range over integers, so the actual structure of the enabling relation is not exactly a tree (for example, in the second arena $\star_a$ enables both $0_{q,\star}$ and $1_{q,\star}$, each of which in turn enables $2_{a,\star}$).

$$\begin{tikzcd}
\star_q \arrow[dr, dash] & \\
& i_a
\end{tikzcd}
\qquad
\begin{tikzcd}
\star_q \arrow[dr, dash] & \\
& \star_a \arrow[d, dash] \\
& i_{q,\star} \arrow[d, dash] \\
& j_{a,\star}
\end{tikzcd}
\qquad
\begin{tikzcd}
\star_q \arrow[d, dash] \arrow[dr, dash] & \\
i_{q,f} \arrow[d, dash] & k_a \\
j_{a,f} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\star_q \arrow[d, dash] \arrow[dr, dash] & \\
i_{q,f} \arrow[d, dash] & l_a \\
\star_{a,f} \arrow[d, dash] & \\
j_{q,f,\star} \arrow[d, dash] & \\
k_{a,f,\star} &
\end{tikzcd}$$

$(\vdash \mathsf{int})$ $\quad$ $(\vdash \mathsf{int} \to \mathsf{int})$ $\quad$ $(f : \mathsf{int} \to \mathsf{int} \vdash \mathsf{int})$ $\quad$ $(f : \mathsf{int} \to \mathsf{int} \to \mathsf{int} \vdash \mathsf{int})$

After specifying an arena, a typical game model will impose further restrictions on the shape of allowable justified sequences, which will subsequently be called the *plays* of the game. The extra restrictions are needed to capture the specificities of differring programming features. In the following section we shall review the most commonly used properties and describe the corresponding computational intuitions.

Once the notion of play is established and it is known what exchanges of moves can take place, one can proceed to the concept of a strategy (for $P$). Game semantics uses strategies as denotations of terms. More concretely, a *strategy* is a set of plays which must be closed under taking prefixes and also under forming extensions using $O$-moves. The latter reflects the fact that strategies prescribe the program's (i.e. $P$'s) responses to $O$'s actions and, thus, have to be ready to react to all scenarios of play by $O$.

Models arising from game semantics can be viewed as categories, in which arenas and strategies take the role of objects and morphisms respectively. In line with the spirit of categorical semantics, arenas are used to interpret types and type constructors are interpreted by constructions on arenas. The most common type constructors such as product or sum, can be accounted for by joining up arenas corresponding to the arguments, often with the help of a few special moves. An important aspect of constructions corresponding to the formation of function spaces is the fact that in the $A_1 \Rightarrow A_2$ arena moves from both $A_1$ and $A_2$ are available but the ownership of moves from $A_1$ is reversed, while it remains the same for $A_2$.

This change of ownership is crucial when it comes to composing strategies from arenas $A_1 \Rightarrow A_2$ and $A_2 \Rightarrow A_3$ in order to form a strategy in the arena $A_1 \Rightarrow A_3$. Then both $A_1 \Rightarrow A_2$ and $A_2 \Rightarrow A_3$ will contain moves from $A_2$, but the moves will belong to different players in the two arenas: if a move from $A_2$ belongs to $O$ in $A_1 \Rightarrow A_2$, it will belong to $P$ in $A_2 \Rightarrow A_3$, and vice versa. In contrast, ownership of moves from $A_1$ and $A_3$ in $A_1 \Rightarrow A_3$ is the same as in $A_1 \Rightarrow A_2$ and $A_2 \Rightarrow A_3$ respectively. Consequently, the composite strategy over $A_1 \Rightarrow A_3$ can be defined by appealing to the strategies involved with the caveat that, if that strategy recommends a $P$-move from $A_2$, we need to relay the response as an $O$-move to the other strategy, thus fuelling an exchange between them. If the strategies happen to interact via $A_2$-moves forever, one talks of *infinite chattering*, which corresponds to divergence. If, on the other hand, the exchange produces a move from $A_1$ or $A_3$, it is taken as part of the new composite strategy in $A_1 \Rightarrow A_3$. Because moves from $A_2$ are not present in $A_1 \Rightarrow A_3$, the exchanges in $A_2$ have to be hidden and do not feature in the composite strategy.

Following the definition of games, a typical game semantics paper will go on to discuss the kind of mathematical/categorical structure that is needed to model the programming language in question. The definition is then validated through a soundness result for closed terms, which will state that every closed term of type unit is interpreted by the strategy corresponding to skip if and only if its evaluation terminates. For many programming languages, such theorems are within reach of many other semantic paradigms and do not imply full abstraction. What really distinguishes game models at this point is *definability*: the fact that all strategies of a certain critical kind[1] are denotations of terms from the programming language in question. Intuitively, this means that the model contains no irrelevant elements ("no junk"): for any criticial strategy we can find a corresponding term. This opens up the path to full abstraction.

The most elegant results of this kind are based on exact equality of denotations (strategies). In the absence of primitives for controlling the flow of computation (jumps, continuations) in contexts, it is necessary to restrict this equality to complete plays only in order to capture the fact that the exact moment of divergence cannot be identified.

## 4. PROPERTIES OF PLAYS

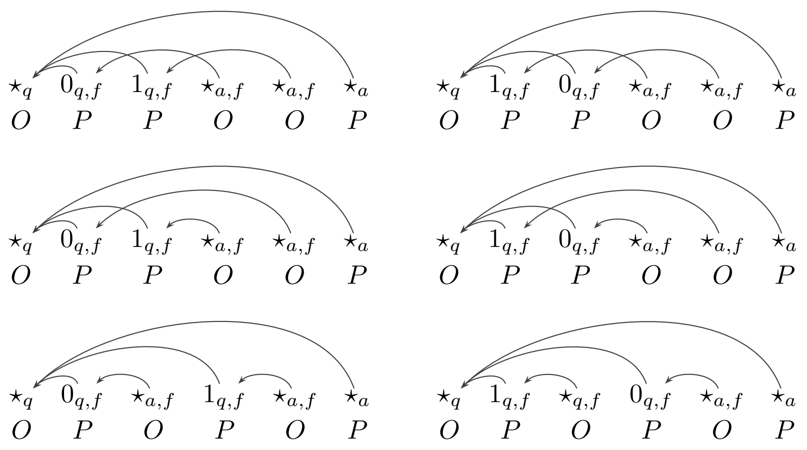
Next we survey a number of prominent combinatorial properties regarding the shape of justified sequences. Remarkably, each of them can be related to a specific programming feature.

### 4.1. Alternation

*Alternation* is the requirement that $O$ and $P$ take turns when making moves. Game semantics uses alternation to model sequential computation. It is relaxed in game models of concurrency, though. For example, the strategy corresponding to

$$f : \mathsf{int} \to \mathsf{unit} \vdash f(0) \parallel f(1) : \mathsf{unit}$$

features the following plays among others (the remaining ones are simply prefixes of those given below).



$\star_q$ $0_{q,f}$ $1_{q,f}$ $\star_{a,f}$ $\star_{a,f}$ $\star_a$
O P P O O P
$\star_q$ $1_{q,f}$ $0_{q,f}$ $\star_{a,f}$ $\star_{a,f}$ $\star_a$
O P P O O P
$\star_q$ $0_{q,f}$ $1_{q,f}$ $\star_{a,f}$ $\star_{a,f}$ $\star_a$
O P P O O P
$\star_q$ $1_{q,f}$ $0_{q,f}$ $\star_{a,f}$ $\star_{a,f}$ $\star_a$
O P P O O P
$\star_q$ $0_{q,f}$ $\star_{a,f}$ $1_{q,f}$ $\star_{a,f}$ $\star_a$
O P O P O P
$\star_q$ $1_{q,f}$ $\star_{q,f}$ $0_{q,f}$ $\star_{a,f}$ $\star_a$
O P O P O P

Note that, in the first play, the second and third moves correspond to respectively the left and right calls made inside the term. Similarly, the fourth and fifth moves are the corresponding returns. Only the last two plays satisfy alternation here. As may be expected, they correspond respectively to the terms

$$f : \mathsf{int} \to \mathsf{unit} \vdash f(0); f(1) : \mathsf{unit} \quad \text{and} \quad f : \mathsf{int} \to \mathsf{unit} \vdash f(1); f(0) : \mathsf{unit}.$$

[1]In most cases, this coincides with domain-theoretic compactness.

In this example the associated arena is

$$\begin{tikzcd}
\star_q \arrow[d, dash] \arrow[dr, dash] & \\
i_{q,f} \arrow[d, dash] & \star_a \\
\star_{a,f} &
\end{tikzcd}$$

and any alternating play must begin with $\star_q$ to be followed by several segments of the shape $i_{q,f}\,\star_{a,f}$, which correspond to sequences of calls to $f$.

Another characteristic feature of concurrent game semantics is that strategies are closed with respect to unobservable rearrangements of moves: adjacent $P$-moves may be permuted, adjacent $O$-moves may be permuted and an $O$-move may go past a $P$-move (unless this is prevented by the pointer structure). The three cases correspond to the inability of programs to control environment actions and the scheduling of parallel events. *Saturation* stipulates that if a play from a strategy is subjected to a series of such rearrangements then the resultant play must also belong to the strategy. Saturation is crucial to obtaining a close match between the syntax and semantics (a definability property). Indeed, while the last two sequences can be traced back to distinct sequential computations, none of the previous four plays in isolation corresponds to a term. With saturation in place, the presence of any of the first four plays listed above necessitates the presence of all six plays. From now on we shall consider alternating plays only.

### 4.2. Well-bracketing

A justified sequence is *well-bracketed* if each answer is justified by the most recent unanswered question. The condition amounts to insisting on stack discipline between questions and answers. Intutively, it captures the fact that calls return in the same order as that in which they were made. Computationally, this corresponds to absence of programming constructs that can disturb control flow, such as continuations and exceptions. For example, consider the play given below

$$\begin{tikzcd}
\star_q & \star_{q,f} \arrow[l, bend right] & \star_{q,f,\star} \arrow[l, bend right] & \star_a \arrow[lll, bend right] \\
O & P & O & P
\end{tikzcd}$$

which is not well-bracketed, because the last answer is justified by the first question instead of the third one. The play can be used to interpret the term

$$f : (\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit} \vdash \mathsf{catch}\, x\, \mathsf{in} f(\lambda y^{\mathsf{unit}}.\mathsf{throw}\, x),$$

where $\mathsf{catch}\, x\, \mathsf{in} \cdots$ creates a local exception $x$ which, when thrown, will result in a jump out of the block.

### 4.3. Visibility

The remaining constraints to be discussed will have to do with the kind of memory available to the program. Recall that justification pointers emanating from questions have to point at earlier moves and these earlier moves must also enable the move that will be played. Intuitively, this corresponds to making calls to functions that have been generated during the current computation. However, can one assume that each of them can be accessed and called? If the analysed programming language comes equipped with a facility for recording functions (such as general references) then all intermediate functional results can indeed be remembered for future access and calls. However, if functional values cannot be stored, the program will be able to access only the functional values that are “currently in scope”, for instance, because of being bound to variables. In game semantics, the intuition behind “current scope” is captured by the concept of *view*, defined as follows.

$$\mathsf{view}(\epsilon) = \epsilon, \qquad \mathsf{view}(m) = m, \qquad \mathsf{view}(s\,\overset{\curvearrowleft}{m\;t\;n}) = \mathsf{view}(s)\,\overset{\curvearrowleft}{m\,n}.$$

The condition of *visibility* stipulates that, for any prefix $s\,\overset{\curvearrowleft}{m\,t\,n}$ of a play, $m$ must be present in $\mathsf{view}(smt)$. The following play violates visibility in its sixth move.

$$\begin{tikzcd}[row sep=tiny]
\star_q & \star_a \arrow[l, bend right] & \star_{q,\star} \arrow[l, bend right] & \star_{a,\star} \arrow[l, bend right] & \star_{q,\star} \arrow[lll, bend right] & \star_{q,\star,\star} \arrow[lll, bend right] \\
O & P & O & P & O & P
\end{tikzcd}$$

This is because, after the fifth move is played, the view (equal to $\star_q\,\overset{\curvearrowleft}{}\,\star_a\,\overset{\curvearrowleft}{}\,\star_{q,\star}$) does not contain the occurrence of $\star_{q,\star}$ pointed at by the last pointer.

The above play belongs to the strategy that denotes

$$\vdash\ \mathsf{let}\,h = \mathsf{ref}(\lambda x^{\mathsf{unit}}.x)\,\mathsf{in}\,\lambda f^{\mathsf{unit}\to\mathsf{unit}}.((!h)();h := f) : (\mathsf{unit}\to\mathsf{unit})\to\mathsf{unit}.$$

In particular, the play describes a computational scenario in which the term is called twice. As the first-order reference $h$ is initialised to the identity function, the first call returns immediately (move $\star_{a,\star}$). Note, though, that the argument of the call, which is a function of type $\mathsf{unit}\to\mathsf{unit}$, will be recorded in $h$ (thanks to $h := f$). Therefore, once the function is used for the second time, the function recorded during the first call will be called ($(!h)()$) rather than the current argument. That is why the last pointer points at the third move rather than the fifth one.

### 4.4. Innocence

While visibility characterizes the absence of storage for higher-order values, innocence is a property corresponding to absolute lack of storage. A strategy is called *innocent* if and only if $P$'s actions depend solely on the current view. Observe that this strengthens visibility: not only must the pointer be directed into the view, but the view is the only information available to $P$. The play given below violates innocence.

$$\begin{tikzcd}[row sep=tiny]
\star_q & \star_q \arrow[l, bend right] & \star_{q,\star} \arrow[l, bend right] & 1_{q,\star} \arrow[l, bend right] & \star_{q,\star} \arrow[lll, bend right] & 2_{q,\star} \arrow[l, bend right] \\
O & P & O & P & O & P
\end{tikzcd}$$

It is taken from the strategy that interprets the term

$$\vdash\ \mathsf{let}\,y = \mathsf{ref}(0)\,\mathsf{in}\,\lambda x^{\mathsf{unit}}.(y := !y+1); !y : \mathsf{unit}\to\mathsf{int},$$

which returns the number of times the function has been called.

The play above violates innocence in the sixth move: after the fifth move the view is the same as after the third one, but the respective following $P$ moves are different ($1_{q,\star}$ and $2_{q,\star}$ respectively). So, the strategy fails to behave uniformly with respect to views.

Let us finish this section with an example of an innocent strategy corresponding to the successor function $\vdash\ \lambda x^{\mathsf{int}}.x{+}1 : \mathsf{int}\to\mathsf{int}$. After $O$ plays $\star_q$ at the start, the strategy will respond with $\star_a$. Afterwards, whenever $O$ plays $i_{q,\star}$, $P$ will reply with $(i+1)_{a,\star}$. Thus the strategy is completely determined by plays of the form

$$\begin{tikzcd}[row sep=tiny]
\star_q & \star_a \arrow[l, bend right] & i_{q,\star} \arrow[l, bend right] & (i+1)_{a,\star} \arrow[l, bend right] \\
O & P & O & P
\end{tikzcd}$$

where $i \in \mathbb{Z}$. Here is a typical play belonging to the strategy.

$$\begin{tikzcd}[column sep=small, row sep=tiny]
\star_q & \star_a \arrow[l, bend right] & 0_{q,\star} \arrow[l, bend right] & 1_{a,\star} \arrow[l, bend right] & 5_{q,\star} \arrow[lll, bend right] & 6_{a,\star} \arrow[l, bend right] & 2_{q,\star} \arrow[lllll, bend right] & 3_{a,\star} \arrow[l, bend right] \\
O & P & O & P & O & P & O & P
\end{tikzcd}$$

## 5. NOMINAL GAME SEMANTICS

Nominal game semantics is a recent branch of game semantics that provides faithful models of generative effects, such as objects, references or exceptions found in ML- and Java-like languages or the $\pi$-calculus. In particular, it uses a countably infinite set of *names* to account for addresses of resources and the infinite cardinality can then be used to model freshness: the generation of a fresh resource, which can be a new memory cell, a new object or a new exception. The names are embedded in moves and also feature in stores that are carried by moves in the game. Intuitively, the stores correspond to the observable part of program memory. All artifacts in nominal game semantics are closed under name-permutation. This reflects the fact that the concrete nature of names is irrelevant: we assume the set of names lacks structure and names can only be compared for equality. Thus, from a mathematical point of view, plays and strategies in nominal game semantics will be *nominal sets* [Gabbay and Pitts 2002; Pitts 2013], a topic explored in the previous issue of the Semantics Column [Pitts 2016].

In order to illustrate the spirit of nominal game semantics, we consider two simple terms that generate reference cells for storing integers, along with representative plays. The first term creates only one reference at the very beginning, which it returns in response to every call. In contrast, the second term generates a new reference each time it is called. Note that moves can carry a store: once a new name has been introduced into play, it is added to the domain of the store.

$\vdash\ \mathsf{let}\, n = \mathsf{ref}(0)\, \mathsf{in}\, (\lambda x^{\mathsf{unit}}.n) : \mathsf{unit} \to \mathsf{int\, ref}$

$$\begin{tikzcd}[column sep=small, row sep=tiny]
\star_q & \star_a \arrow[l, bend right] & \star_{q,\star} \arrow[l, bend right] & n_{a,\star}{}^{(n,0)} \arrow[l, bend right] & \star_{q,\star}{}^{(n,5)} \arrow[lll, bend right] & n_{a,\star}{}^{(n,5)} \arrow[l, bend right] & \star_{q,\star}{}^{(n,12)} \arrow[lllll, bend right] & n_{a,\star}{}^{(n,12)} \arrow[l, bend right] \\
O & P & O & P & O & P & O & P
\end{tikzcd}$$

$\vdash\ \lambda x^{\mathsf{unit}}.\mathsf{ref}(0) : \mathsf{unit} \to \mathsf{int\, ref}$

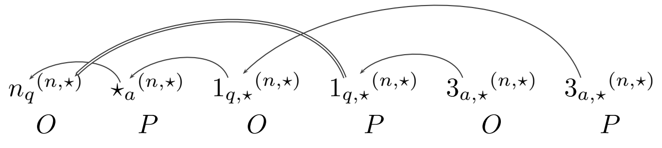
$$\begin{tikzcd}[column sep=small, row sep=tiny]
\star_q & \star_a \arrow[l, bend right] & \star_{q,\star} \arrow[l, bend right] & (n_1)_{a,\star}{}^{(n_1,0)} \arrow[l, bend right] & \star_{q,\star}{}^{(n_1,5)} \arrow[lll, bend right] & (n_2)_{a,\star}{}^{(n_1,5)(n_2,0)} \arrow[l, bend right] & \star_{q,\star}{}^{(n_1,12)(n_2,7)} \arrow[lllll, bend right] & (n_3)_{a,\star}{}^{(n_1,12)(n_2,7)(n_3,0)} \arrow[l, bend right] \\
O & P & O & P & O & P & O & P
\end{tikzcd}$$

Observe that, in the two plays, the initial value of each cell is $0$ (corresponding to $\mathsf{ref}(0)$) and that $P$ never modifies the content of the cells afterwards. However, once the corresponding names become part of play, $O$ is free to modify them as the names are now available to the environment.

The above approach can also be applied to higher-order storage, i.e. computational scenarios in which functions can be stored. However, one cannot simply reveal the exact values that are being stored, because they can only be observed to the extent to which they are going to be used during the computation and cannot be readily compared with other higher-order values. Accordingly, we shall use $\star$ to represent every higher-order value and allow plays to explore stored values by playing moves with a special pointer to the store rather than to another move. We shall use double lines when drawing such pointers in figures. For example, the term

$$n : (\mathsf{int} \to \mathsf{int})\ \mathsf{ref} \vdash\ !n : \mathsf{int} \to \mathsf{int}.$$

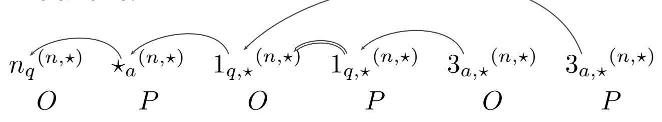
is modelled, among others, by the play given below.


$n_q^{(n,\star)}$
$\star_a^{(n,\star)}$
$1_{q,\star}^{(n,\star)}$
$1_{q,\star}^{(n,\star)}$
$3_{a,\star}^{(n,\star)}$
$3_{a,\star}^{(n,\star)}$
$O$ $P$ $O$ $P$ $O$ $P$

While the third move is of a similar kind to our previous examples and corresponds to calling the evaluated term on argument 1, the fourth move could be viewed as forwarding the call to the value stored initially in the reference cell corresponding to $n$. Note that the content of $n$ can change throughout the computation, so $P$ could also point at stores associated with other moves, as in our next example. Here, the term is

$$n : (\mathsf{int} \to \mathsf{int})\ \mathsf{ref} \vdash \lambda h^{\mathsf{int}}.(!n)h : \mathsf{int} \to \mathsf{int}$$

and a call to the term must trigger a call to the *latest* value stored in the reference rather than the initial one.


$n_q^{(n,\star)}$
$\star_a^{(n,\star)}$
$1_{q,\star}^{(n,\star)}$
$1_{q,\star}^{(n,\star)}$
$3_{a,\star}^{(n,\star)}$
$3_{a,\star}^{(n,\star)}$
$O$ $P$ $O$ $P$ $O$ $P$

This is reflected by the target of the pointer out of the fourth move, which points at the store of the preceding move.

## 6. DIRECTIONS IN GAME SEMANTICS

The style of modelling sketched in our article is known as *HO/N-games* or *pointer games*. The first papers propounding the approach were written to solve the full abstraction problem for the purely functional language PCF [Hyland and Ong 2000; Nickau 1994]. Another solution to the problem was obtained at the same time using AJM-games [Abramsky et al. 2000], which do not have pointers and rely on possibly nested numerical indices to compensate for their absence.

The two ways of modelling subsequently led to a wealth of full abstraction results for a whole variety of programming features, e.g. state [Abramsky and McCusker 1997b], control constructs [Laird 1997], call-by-value evaluation [Abramsky and McCusker 1997a; Honda and Yoshida 1999][2], general references [Abramsky et al. 1998], nondetermism [Malacaria and Hankin 1999], probabilistic computation [Danos and Harmer 2000], exceptions [Laird 2001], concurrency [Ghica and Murawski 2008] and polymorphism [Hughes 1997; Laird 2013]. A tutorial overview of HO/N-game semantics can be found in [Abramsky and McCusker 1998] and both approaches are discussed in [Hyland 1997].

Nominal game semantics [Laird 2008; Abramsky et al. 2004; Tzevelekos 2008] was introduced about 10 years after the original models of PCF and made it possible to eliminate a number of imperfections in modelling reference types, known as the bad-variable problem [Abramsky and McCusker 1997b; Abramsky et al. 1998]. In particular, it opened up the way to faithful models of name-based programming abstractions

[2]All of our examples of arenas and plays are couched in the call-by-value framework from [Honda and Yoshida 1999].

such as threads [Laird 2006], references [Murawski and Tzevelekos 2009; 2011a] and objects [Murawski and Tzevelekos 2014]. A tutorial account of nominal game semantics can be found in [Murawski and Tzevelekos 2016].

We conclude our article with a concise overview of some recent directions within game semantics.

*Operational game semantics.* Research into understanding the operational flavour of game semantics has revealed numerous connections to executions of abstract machines [Danos et al. 1996; Curien and Herbelin 1998]. There also exist full abstraction results obtained using operational (rather than denotational) techniques, e.g. [Jeffrey and Rathke 1999; Laird 2007; Lassen and Levy 2008]. These are based on ingenious instrumentations of labelled transition systems that capture equivalence through traces. In some cases, e.g. [Laird 2007] and [Murawski and Tzevelekos 2011b], it has been shown that the traces and game semantics coincide [Jaber 2015]. However, it is still not clear how to transfer results between the two methodologies seamlessly and, most interestingly, how to derive compositional definitions from non-compositional operational descriptions. A first step to investigate such connections was recently made in [Levy and Staton 2014].

*Algorithmic game semantics.* The concrete nature of game semantics makes it an appealing framework for deriving program analyses (see e.g. [Malacaria and Hankin 1999]). Since plays can be viewed as words over a suitably chosen alphabet, one can use automata-theoretic techniques to represent game models and scrutinise them in an automated fashion [Ghica and McCusker 2003]. This creates scope for applying game semantics as a compositional foundation for a wide range of verification tasks [Abramsky et al. 2004]. For nominal games (see e.g. [Murawski and Tzevelekos 2012; Murawski et al. 2015]), one can then tap into the vast literature on automata theory over infinite alphabets [Neven et al. 2004; Segoufin 2006]. Intuitions from game semantics have also played a crucial role in the first proof of decidability of MSO (monadic second-order logic) over trees generated by higher-order recursion schemes [Ong 2006].

*Concurrent games.* Although concurrent computation can be expressed in game semantics via interleaving, it would be highly desirable to establish a dedicated framework for modelling concurrency directly, in the spirit of true concurrency. This goal has been pursued in the early days of the field in the context of linear logic [Abramsky and Melliès 1999]. More recent developments include asynchronous games [Melliès 2006; Melliès and Mimram 2007] and concurrent games [Rideau and Winskel 2011; Castellan et al. 2015]. In particular, the latter framework uses event structures as the underlying theory. The success of partial-order methods in verification provides another, more practical, motivation for developing games in this direction.

## REFERENCES

S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, and I. D. B. Stark. 2004. Nominal Games and Full Abstraction for the Nu-Calculus. In *Proceedings of LICS*. IEEE Computer Society Press, 150–159.

S. Abramsky, K. Honda, and G. McCusker. 1998. Fully Abstract Game Semantics for General References. In *Proceedings of IEEE Symposium on Logic in Computer Science*. Computer Society Press, 334–344.

S. Abramsky, R. Jagadeesan, and P. Malacaria. 2000. Full Abstraction for PCF. *Information and Computation* 163 (2000), 409–470.

S. Abramsky and G. McCusker. 1997a. Call-by-value games. In *Proceedings of CSL (Lecture Notes in Computer Science)*, Vol. 1414. Springer-Verlag, 1–17.

S. Abramsky and G. McCusker. 1997b. Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions. In *Algol-like languages*, P. W. O'Hearn and R. D. Tennent (Eds.). Birkhaüser, 297–329.

S. Abramsky and G. McCusker. 1998. Game semantics. In *Logic and Computation*, H. Schwichtenberg and U. Berger (Eds.). Springer-Verlag. Proceedings of the 1997 Marktoberdorf Summer School.

S. Abramsky and P.-A. Melliès. 1999. Concurrent Games and Full Completeness. In *Proceedings, Fourteenth IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 431–442.

S. Castellan, P. Clairambault, and G. Winskel. 2015. The Parallel Intensionally Fully Abstract Games Model of PCF. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*. 232–243.

P.-L. Curien and H. Herbelin. 1998. Computing with abstract Böhm trees. In *Proceedings of Third Fuji International Symposium on Functional and Logic Programming, Kyoto, April 1998*. World Scientific, 20–39.

V. Danos and R. Harmer. 2000. Probabilistic game semantics. In *Proceedings of the IEEE Symposium on Logic in Computer Science*. Computer Science Society, 204–213.

V. Danos, H. Herbelin, and L. Regnier. 1996. Game semantics and abstract machines. In *Proceedings of 11th Annual IEEE Symposium on Logic in Computer Science*. Computer Society Press.

M. J. Gabbay and A. M. Pitts. 2002. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing* 13 (2002), 341–363.

D. R. Ghica and G. McCusker. 2003. The Regular Language Semantics of Second-Order Idealized Algol. *Theoretical Computer Science* 309 (2003), 469–502.

D. R. Ghica and A. S. Murawski. 2008. Angelic Semantics of Fine-Grained Concurrency. *Annals of Pure and Applied Logic* 151(2-3) (2008), 89–114.

K. Honda and N. Yoshida. 1999. Game-theoretic analysis of call-by-value computation. *Theoretical Computer Science* 221, 1–2 (1999), 393–456.

D. H. D. Hughes. 1997. Games and definability for System F. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science*. IEEE Computer Science Society, 76–86.

J. M. E. Hyland. 1997. Game semantics. In *Semantics and Logics of Computation*, A. Pitts and P. Dybjer (Eds.). Cambridge Univ. Press, 131–182.

J. M. E. Hyland and C.-H. L. Ong. 2000. On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. *Information and Computation* 163(2) (2000), 285–408.

G. Jaber. 2015. Operational Nominal Game Semantics. In *Proceedings of FOSSACS*. 264–278.

A. Jeffrey and J. Rathke. 1999. Towards a Theory of Bisimulation for Local Names. In *Proceedings of LICS*. 56–66.

J. Laird. 1997. Full Abstraction for Functional Languages with Control. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science*. 58–67.

J. Laird. 2001. A fully abstract games semantics of local exceptions. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 105–114.

J. Laird. 2006. Game Semantics for Higher-Order Concurrency. In *FSTTCS (Lecture Notes in Computer Science)*, Vol. 4337. 417–428.

J. Laird. 2007. A Fully Abstract Trace Semantics for General References. In *Proceedings of ICALP*. Lecture Notes in Computer Science, Vol. 4596. Springer, 667–679.

J. Laird. 2008. A game semantics of names and pointers. *Annals of Pure and Applied Logic* 151 (2008), 151–169.

J. Laird. 2013. Game semantics for a polymorphic programming language. *J. ACM* 60, 4 (2013), 29.

S. B. Lassen and P. B. Levy. 2008. Typed Normal Form Bisimulation for Parametric Polymorphism. In *Proceedings of LICS*. IEEE Computer Society, 341–352.

P. B. Levy and S. Staton. 2014. Transition systems over games. In *Proceedings of CSL-LICS*. 64:1–64:10.

P. Malacaria and C. Hankin. 1999. Non-deterministic games and program analysis: an application to security. In *Proceedings of LICS*. IEEE, 443–452.

Paul-André Melliès. 2006. Asynchronous games 2: The true concurrency of innocence. *Theor. Comput. Sci.* 358, 2-3 (2006), 200–228.

P.-A. Melliès and S. Mimram. 2007. Asynchronous Games: Innocence Without Alternation. In *Proceedings of CONCUR'07 (Lecture Notes in Computer Science)*, Vol. 4703. Springer, 395–411.

R. Milner. 1977. Fully Abstract Models of Typed Lambda-Calculi. *Theoretical Computer Science* 4, 1 (1977), 1–22.

A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. 2015. Game Semantic Analysis of Equivalence in IMJ. In *Proceedings of ATVA'15 (Lecture Notes in Computer Science)*, Vol. 9364. Springer, 411–428.

A. S. Murawski and N. Tzevelekos. 2009. Full Abstraction for Reduced ML. In *Proceedings of FOSSACS*. Lecture Notes in Computer Science, Vol. 5504. Springer-Verlag, 32–47.

A. S. Murawski and N. Tzevelekos. 2011a. Algorithmic nominal game semantics. In *Proceedings of ESOP*. Lecture Notes in Computer Science, Vol. 6602. Springer-Verlag, 419–438.

A. S. Murawski and N. Tzevelekos. 2011b. Game Semantics for Good General References. In *Proceedings of LICS*. IEEE Computer Society Press, 75–84.

A. S. Murawski and N. Tzevelekos. 2012. Algorithmic games for full ground references. In *Proceedings of ICALP*. Lecture Notes in Computer Science, Vol. 7392. Springer, 312–324.

A. S. Murawski and N. Tzevelekos. 2014. Game Semantics for Interface Middleweight Java. In *POPL*. 517–528.

A. S. Murawski and N. Tzevelekos. 2016. Nominal Game Semantics. *Foundations and Trends in Programming Languages* 2, 4 (2016), 191–269.

F. Neven, T. Schwentick, and V. Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 5, 3 (2004), 403–435.

H. Nickau. 1994. Hereditarily sequential functionals. In *Proceedings of the Symposium of Logical Foundations of Computer Science*. Springer-Verlag. LNCS.

C.-H. L. Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *Proceedings of LICS*. Computer Society Press, 81–90.

A. M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science, Vol. 57. Cambridge University Press.

A. M. Pitts. 2016. Nominal Techniques. *ACM SIGLOG News* 3, 1 (Jan. 2016), 57–72.

S. Rideau and G. Winskel. 2011. Concurrent Strategies. In *Proceedings of LICS'11*. IEEE Computer Society, 409–418.

L. Segoufin. 2006. Automata and Logics for Words and Trees over an Infinite Alphabet. In *Proceedings of CSL (Lecture Notes in Computer Science)*, Vol. 4207. Springer, 41–57.

N. Tzevelekos. 2008. Nominal Game Semantics. (2008). D.Phil. thesis, Oxford University Computing Laboratory.