Game Semantics

An elementary approach

Game semantics is a denotational semantics for programming languages which interprets a term as an interaction between itself and its context. In this tutorial introduction we give an elementary introduction to the area which should be accessible to a reader acquainted with operational semantics. A passing awareness of category theory would enhance the appreciation of game semantics but it is not mandated. Starting with a general introduction to the ideas and concepts of game semantics, we focus on a particular language (Idealised Concurrent Algol) which although expressive enjoys a reasonably simple game model. The second part of the tutorial looks at a closely related formalism, that of trace semantics. If in the case of game semantics the context is best understood syntactically, in the case of trace semantics it can be generalised to an unrestricted computational environment. In both cases, of game and trace semantics, particular emphasis is placed on compositionality, the property of larger models to be constructed mathematically out of interpretations of smaller models. To enhance motivation and to seed potential research ideas a list of known applications of game and trace semantics is discussed.

This material was first presented at:

Oregon Programming Langauges Summer School (OPLSS)

July 9-21, 2018

Dan R. Ghica
University of Birmingham

Contents

1 Introduction

1.1 Denotational semantics

Game semantics (GS) is a way to mathematically specify the behaviour of programming languages (PL); it is a so-called *denotational semantics* (DS). One way to understand denotational semantics is by contrast with *operational semantics* (OS), which is a different mathematical specification of programming languages.[1]

OS is syntactic and self-contained. It defines the PL via a set of transformation rules of the form

$$t, c \longrightarrow t', c'$$

where t, t' are terms and c, c' additional configuration information.

This relation is interpreted as follows: when program t runs, in some configuration c, after one symbolic step of computation it becomes "like program t'" in some updated configuration c'. This is a common and useful way of specifying PLs. It is flexible and elementary, and it can work as a basis for compiler development and verification.

DS is not self-contained, but it requires a "meta-language". Unlike the object language (the PL) the meta-language is supposed to be well understood or, in some sense, "more fundamental" than the PL. The meta-language is called the "semantic domain" (SD), and the DS is defined by translating PL terms into the semantic domain, *inductively*, either on the syntax or, very commonly, on the typing derivation.

$$[\![-]\!] : PL \to SD$$
$$[\![K\ t_0 \cdots t_k]\!] = f([\![t_0]\!], \ldots [\![t_k]\!]),$$

where K is a syntactic term construction with sub-terms t_i, and f a function in the semantic domain.

Example 1. *Regular expressions are a syntax for regular languages. Regular expressions can be given a syntactic, OS-like, interpretation known as Kleene Algebras, or a semantic, denotational interpretation by translating them into finite state automata.*

The connection between OS and DS is akin to, and insipired by, the connection between proof theory and model theory in logic.

A DS is a more ambitious theoretical undertaking than an OS. We need to find an appropriate semantic

[1] The comparison in this section contains significant oversimplifications for pedagogical reasons.

domain. Once we define it, we need to interpret all programming language constructs translationally, and it is not as obvious as in the case of the OS that a definition in the DS actually captures the behaviour it aims to. But once this more solid foundation is laid, working with a DS can be easier than with the OS, especially when it comes to equivalence. Consider regular languages: which semantics is more convenient for proving regular-language equivalence? Finite state automata are more convenient than syntactic manipulation in the Kleene Algebra. Or what is easier, constructing a proof in propositional logic or filling out its truth table? A DS requires more of an upfront investment, but it can pay greater dividends.

In the case of PLs the equivalence problem is mathematically challenging. Consider for example OS rules in which the configuration does not change:

$$t, c \longrightarrow t', c.$$

Obviously, program t' is a slightly evolved version of t but it should be otherwise equivalent – and it is. However, there is nothing in the OS to guarantee that this equivalence is a *congruence*, i.e. if we stick t, t' in the same program *context* $C[-]$ we still get equivalent programs, i.e.

$$t \equiv t' \Rightarrow \forall C[-].C[t] \equiv C[t'].$$

Equivalences which are not congruences are virtually useless, since they cannot be used in compiler optimisations, which always apply to sub-programs rather than whole programs. Why would an equivalence not be a congruence? This can happen in two ways, one very common and one less so, at least in reasonable PLs. The common failure of congruence is if the hole contains side-effects, which means that a *mathematical congruence* such as $x + y = y + x$ is not a programming language congruence $x + y \not\equiv y + x$ because in the presence of side-effects their order matters. The uncommon source of failure is for the language to contain primitives which "inspect" a term intensionally (LISP's *quote* operator is an example).

By contrast to OS, in DS the equivalence of two terms reduces to equality in the semantic domain, which is always a congruence. In a DS,

$$[\![C[t]]\!] = [\![t]\!] \circ [\![C]\!]$$

where $[\![C]\!]$ is the interpretation of the context and $- \circ -$ is a notion of composition specific to the DS, which should be well behaved (associative, have identity, monotonic, etc.), which means that the

equivalence below always holds:

$$[\![t]\!] = [\![t']\!] \Leftrightarrow \forall C[-].[\![t]\!] \circ [\![C]\!] = [\![t']\!] \circ [\![C]\!],$$

As an aside, this is one reason why *Category Theory* is attractive to DS as a more general framework, because properties such as the one above follow from abstract, general principles rather than from the detailed properties. This means that we can work with specifications for models, rather than detailed concrete models which can be complicated, fragile, and too specific.

1.1.1 Technical properties

Because DS is not as "obvious" as the OS, it is common for an OS to also be given, as a more basic form of PL specification. The idea is that DS provides a basis for equational reasoning whereas the OS more plausibly captures intended behaviour. With two semantics at hand, it is obviously necessary to relate them, much like in logic proof theory and model theory are related.

The following are some of the key technical properties relating the two.

Termination. It is common to use termination vs. non-termination as the ultimate observational distinction between programs, that is unit-typed closed terms. Since virtually all languages have an if-then-else construct and equality testing, termination vs. non-termination subsumes program-equality at all ground types. We say that a program t terminates in any admissible initial configuration c if and only if $t, c \longrightarrow^* v, c'$ where v is a special class of programs called "values" which need no further evaluation. Let us write this as $t \Downarrow$ and $t \Uparrow$ as its negation. If a program is of ground type and it diverges (e.g. `while true do ()`) it is common to write it as Ω, i.e. $\Omega \Uparrow$. It is common to write $[\![\Omega]\!] = \bot \neq \top$ where $\top$ is the meaning of terminating unit-type programs.

Definition 1 (Observational equivalence)**.** *We say that two terms are* observationally equivalent $t_0 \equiv t_1$ *if and only if for any context $C[-]$ such that $C[t_i]$ is a well-formed program, $C[t_0]\Downarrow$ if and only if $C[t_1]\Downarrow$.*

Soundness. Two terms that are semantically equal should also be observationally equivalent, i.e. if $[\![t_0]\!] = [\![t_1]\!]$ then $t_0 \equiv t_1$. This is a minimum expectation for a DS. The alternative is nonsense.

Adequacy. A program is semantically identified as terminating $[\![t]\!] = \top$ if and only if terminates $t \Downarrow$. This is also minimally expected of a DS, but generally more difficult to establish.

Definability. Ideally, the semantic domain should contain no "garbage", i.e. for any element τ of the semantic domain which is in some sense "finite", there exists a PL term t such that $[\![t]\!] = \tau$. Generally speaking, definability has proved to be the most challenging problem in DS. It was first famously noticed for PCF, in the case of which the domains-based denotational semantics fails definability, requiring a preemptive parallel-or operator in the PL. Trying to solve the definability problem for PCF eventually led to game semantics.

Full abstraction. When soundness, adequacy, and definability all hold then we are in a "fully abstract" situation in which the DS and the OS coincide, i.e. $[\![t_0]\!] = [\![t_1]\!]$ if and only if $t_0 \equiv t_1$. For a programming language this is a very happy place to be!

Theorem 1. *Soundness, adequacy and definability imply full abstraction.*

Proof. One direction of the full-abstraction equivalence is soundness. Let us prove that $t \equiv t'$ implies $[\![t]\!] = [\![t']\!]$. We prove this by contradiction, assuming that $[\![t]\!] \neq [\![t']\!]$. Then there must be[2] some $\tau \in SD$ such that $[\![t]\!] \circ \tau = \top \neq \bot = [\![t']\!] \circ \tau$. From definability we know that there must be some term C such that $[\![C]\!] = \tau$, so $[\![t]\!] \circ [\![C]\!] = \top \neq \bot = [\![t']\!] \circ [\![C]\!]$ which, from the compositionality of the DS means $[\![C[t]]\!] = \top \neq \bot = [\![C[t']]\!]$. Using adequacy on both sides it follows that $C[t] \Downarrow$ and $C[t] \Uparrow$, which is a contradiction. □

Coherence. DS can be defined inductively on the typing derivation rather than the syntax of the term. Some languages, such as PCF, admit (essentially) unique typing derivation, which makes a term-based interpretation well defined, so it makes sense to write $[\![t]\!]$. However, for complex type systems it is often the case that the same judgement may have multiple derivations. Yet, we would expect all these derivation to produce equal interpretations. Let us write $\Delta[t]$ as shortcut for *term t has a typing judgement with derivation* Δ, then for any Δ_i, we want that $[\![\Delta_0[t]]\!] = [\![\Delta_1[t]]\!]$, which we then just write as $[\![t]\!]$. This property is ignored suprisingly often, being just presumed. But proving it can be difficult. Coherence is another property where the categorical setting is helpful because, not coincidentally, the coherence equations in a category correspond to legal manipulations of a derivation trees which preserve its validity.

[2] The reason why depends on the SD.

1.1.2 Reading list

There are many excellent books on denotational semantics, but here are two that I have a personal affinity for as they are generally programmer-friendly:

> Tennent, Robert D. Semantics of programming languages. Vol. 199. No. 1. Hemel Hempstead: Prentice-Hall, 1991.
>
> O'Hearn, Peter, and Robert Tennent. ALGOL-like Languages. Springer Science & Business Media, 2013.

The OS-DS connection has been studied in the classic

> Plotkin, Gordon D. "LCF considered as a programming language." Theoretical computer science 5.3 (1977): 223-255.

As an entry point to categorical semantics one of the early seminar papers is

> Joachim Lambek, From lambda calculus to Cartesian closed categories, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, eds. J. P. Seldin and J. Hindley, Academic Press, 1980, pp. 376-402.

But a variety of formal and informal tutorials can be found[3].

1.2 Game semantics

Out of the quest for solving the definability problem for PCF, i.e. trying to find a semantic domain in which behaviour such as that of parallel-or is mathematically ruled out, a new kind of denotational semantics emerged, *game semantics*. What makes GS distinctive is the fact that its semantic domain abandons the sets-and-functions paradigm which dominates early denotational semantics. It is a genuinely new idea, using concrete combinatorial structures of actions called "games" and "strategies". It is perhaps easy to overlook how important this shift was, since it did away with any pretense that a programming language function is somehow like a mathematical function.

The GS approach has been extraordinarily successful, solving not just the longstanding definability problem for PCF but also providing fully abstract models for more expressive languages such as ALGOL or (various fragments of) ML. Beyond this theoretical success, the concrete formulation of GS made it more suitable for applications than sets-and-functions-style DS. In a sense to be elaborated later, we will see that GS is not only denotational but also operational. It can give the same kind of step-by-step

[3]Including blog posts such as https://golem.ph.utexas.edu/category/2006/08/cartesian_closed_categories_an_1.html

behavioural model that OS can give. The aim of this presentation is to emphasise both the elementary nature of GS and its close link to OS.

2 Game semantics, an interaction semantics

2.1 Arenas, plays, strategies

The terminology of "game semantics" guides the intuition towards the realm of game theory. Indeed, there are methodological and especially historical connections between game semantics and game theory, but they are neither direct nor immediate. The games involved are rooted in logic and reach programming languages via the Curry-Howard nexus. They are not essential for a working understanding of game semantics as a model of programming languages, so we will not describe them here. But if we were to be pushed hard to give a game-theoretic analogy, the one to keep in mind are not the quantitative games of economics but rather last-player-wins games such as Nim.

It is more helpful to think of game semantics as a more general interaction, or dialogue, between two agents, rather than a game. This interaction is asymmetric. One agent (P) represents the term and the other (O) represents an abstract and general context in which the term can operate.[4] The interaction consists of sequences of events called *moves*, which can be seen as either calls, called *questions*, or returns, called *answers*. A sequence of events, with some extra structure to be discussed later, is called a *play* and it corresponds to the sequence of interactions between the term and the context in a program run. The set of all such possible plays is called a *strategy* and it gives the interpretation of the term. The strategy of a term can be constructed inductively on its syntax, from basic strategies for the atomic elements and a suitable notion of composition.

Before we proceed, a warning. The structure of a game semantics is dictated by the evaluation strategy of the language. Call-by-name games are quite differently structured than call-by-value games. In this section and the next we will assume a call-by-name evaluation strategy. The reason is didactic, as these games are easier to present. Having understood GS in the context of CBN, learning CBV games should come easy enough.

Let us consider a trivial example, the term consisting of the constant 0. The way this term can interact with any context is via two moves: a question (q) corresponding to the event interrogating the term, and an answer (0) corresponding to the term communicating its value to the context. The sequence $q \cdot 0$ is the only possible play, therefore the strategy corresponding to the set of plays $\{q \cdot 0\}$ is the interpretation of the term 0.

Let us consider a slightly less trivial example, the identity function over natural numbers $\lambda x.x : nat \to$

[4] The names stand for 'Proponent' and 'Opponent' even though there is nothing being proposed, and there is no opposition to it. The names are historical artefacts. We might as well call them 'Popeye' and 'Olive'. Same applies to "move", "play", and "strategy".

nat. The context can call this function, but also the function will enquire about the value of x. Lets call these questions q and q'. The context can answer to q' with some value n and the term will answer to q with the same value n. Even though the answers carry the same value they are different moves, so we will write n and n' to distinguish them, where then prime is a syntactic tag. All the plays in the strategy interpreting the identity over natural numbers have shape $q \cdot q' \cdot n' \cdot n$.

Let us now define the concepts more rigorously.

Definition 2 (Arena)**.** *An* arena *is a tuple* $\langle M, Q, I, O, \vdash \rangle$ *where*

- M *is a set of* moves.
- $Q \subseteq M$ *is a set of* questions*;* $A = M \setminus Q$ *is the set of* answers.
- $O \subseteq M$ *is a set of* O-moves*;* $P = M \setminus O$ *is the set of* P-moves.
- $I \subseteq Q \cap O$ *is a set of* initial moves*;*
- $\vdash \subseteq M \times M$ *is an* enabling *relation such that if* $m \vdash n$ *then*

  (e1) $m \in Q$

  (e2) $m \in O$ *if and only if* $n \in P$

  (e3) $n \notin I$.

An arena represents the set of moves associated with a type, along with the structure discussed above (questions, answers, opponent, proponent). Additionally, the arena introduces the concept of *enabling* relation, which records the fact that certain moves are causally related to other moves. Enabling satisfies certain well-formedness conditions:

(e1) Only questions can enable other moves, which could be interpreted as *all computations happen because a call.*

(e2) P-moves enable O-moves and *vice versa.* Game semantics records behaviour *at the interface* so any action from the context enables an action of the term, and the other way around.

(e3) There is a special class of opponent questions called *initial moves.* These are the moves that start the computation, and do not need to be enabled.

The informal discussion above can be made more rigorous now.

Example 2. *Let* $\mathbf{1} = \{\star\}$*. The arena of natural numbers is* $N = \langle \mathbf{1} \uplus \mathbb{N}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1} \times \mathbb{N} \rangle$.

An equivalent and perhaps simpler formulation is

Definition 3 (Arena)**.** *An arena is a tuple* $\langle OQ, OA, PQ, PA, \vdash \rangle$ *where* $\vdash \subseteq OQ \times P \cup PQ \times O$.

We write *P-moves* as $P = PQ \cup PA$, *O-moves* as $O = OQ \cup OA$, *questions* as $Q = OQ \cup PQ$ and *answers* as $A = OA \cup PA$.

Definition 4 (Initial move)**.** *The set of* initial moves *of an arena A is $I_A = \{m \in OQ \mid \forall n \in P, m \not\vdash n\}$.*

More complex arenas can be created using product $\times$ and arrow $\to$ constructs. Let

$$inl : M_A \to M_A + M_B$$
$$inr : M_B \to M_A + M_B$$

where $+$ is the co-product of the two sets of moves. We lift the notation to relations, $R + R' \subseteq (A + A') \times (B + B')$:

$$inl(R) = \{(inl(m), inl(n)) \mid (m, n) \in R\}$$
$$inr(R') = \{(inr(m), inr(n)) \mid (m, n) \in R'\}.$$

Definition 5 (Arena product and arrow)**.** *Given arenas $A = \langle M_A, Q_A, O_A, I_A, \vdash_A \rangle$ and $B = \langle M_B, Q_B, O_B, I_B, \vdash_B \rangle$ we construct the* product arena *as*

$$A \times B = \langle M_A + M_B, Q_A + Q_B, O_A + O_B, I_A + I_B, \vdash_A + \vdash_B \rangle$$

and the arrow arena *as*

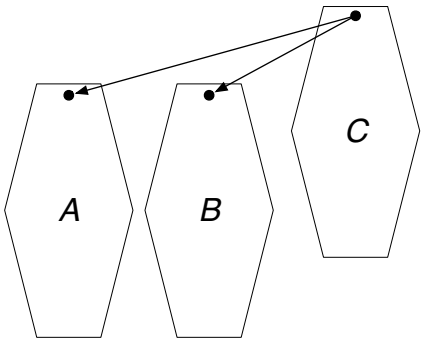
$$A \to B = \langle M_A + M_B, Q_A + Q_B, P_A + O_B, inr(I_B), \vdash_A + \vdash_B \cup inr(I_B) \times inl(I_A) \rangle.$$

If we visualise the two arenas as DAGs, with the initial moves as sources and with the enabling relation defining the edges, then the product arena is the disjoint union of the two DAGs and the arrow arena is the grafting of the A arena at the roots of the B arena, but with the O-P polarities reversed.

Since arenas will be used to interpret types we can anticipate by noting that

Theorem 2 (Currying)**.** *For any arenas A, B, C the arenas $A \times B \to C$ and $A \to B \to C$ are isomorphic.*

Proof. Both arena constructions correspond to the dag below.



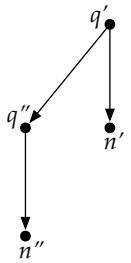
The isomorphism is a node-relabelling isomorphism induced by the associativity of the co-product. □

We also note that

Theorem 3 (Unit)**.** *The arena $I = \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ is a unit for product, i.e. for any arena A, $A \times I$, $I \times A$ are isomorphic to A.*

The isomorphism is a re-tagging of moves.

Example 3. *We talked earlier about the arena for the type $nat \to nat$. Let $inl(m) = m'$ and $inr(m) = m''$, where $'$ and $''$ are syntactic tags. The arena $Nat \to Nat$ is represented by the DAG below*



We already mentioned that for the identity all plays have the shape $q'q''n''n'$. We note that in this particular play all move occurrences are preceded by an enabling move. The move corresponding to the term returning a value n' can happen because the context initiated a play q'. The term can ask for the value of the argument q'' also because q has happened earlier. Each move occurrence is *justified* by an enabling move, according to $\vdash$, occurring earlier. The enabling relation defines the causal skeleton of the play.

Let us further consider another term in the same arena $\lambda x.x + x$. How does this term interact with its context?

1. the context initiates the computation
2. the term asks for the value of x

3. the contest returns some m
4. the term asks again for the value of x
5. the context returns some n
6. the term returns to the context $m + n$.

The reader familiar with call-by-value may be rather confused as to why the context returns first an m and then an n. This is because in call-by-name argument are thunks, and the thunks may contain side-effects, which means that repeat evaluations may yield different values.

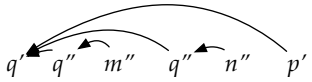
Looking at the arena, this interaction corresponds to the play $q'q''m''q''n''p'$, where $p = m + n$. The causal structure of this play is a little confusing. There are two occurrences of q'', the first one preceding both m'' and n'' and the second one only n''. It should be that the first occurrence of q'' enables m'' and the second enables n'', to reflect the proper call-and-return we might expect in a programming language. In order to do that the plays will be further instrumented with names called *pointers*. Each question has a symbolic address, and is paired with the address of the enabling move. The fully instrumented play is called a *justified sequence*

$$q'a\langle b\rangle \cdot q''b\langle c\rangle \cdot m''c\langle\_\rangle \cdot q''b\langle d\rangle \cdot n''d\langle\_\rangle \cdot p'b\langle\_\rangle,$$

noting that a is the only name without a previous binder, and is used by the initial question which needs no justification. The addresses $a, b, c, \ldots \in \mathbb{A}$ are just names, and the notation $\langle a\rangle$ means that the name a is "fresh". In general we will employ the Barendregt name convention that if two names a, b are denoted by distinct variables they are distinct $a \neq b$. Answers never justify (in these games) so we write their useless name as _ or we may omit the whole $\langle\_\rangle$ altogether.

In a justified sequence, by a *move occurrence* we mean the move along with the justifier a and, if it is the case, the pointer $\langle b\rangle$, taken as a whole. No operations on justified sequences (e.g. sub-sequence) may break this granularity.

If we were to represent the pointers graphically, the sequence above would be:



It represents not only the actions, that is the calls and returns, but also what calls correspond to what returns and even what calls are cause by other calls. In a pure language, for terms up to order three the pointers can be actually uniquely reconstructed from the sequence itself. Otherwise the justification

pointers are necessary.

The standard example in the literature for this phenomenon are the so-called "Kierstead terms"

$$\lambda f.f(\lambda x.f(\lambda \overline{y}.\underline{y})) \text{ vs. } \lambda f.f(\lambda \overline{x}.f(\lambda y.\underline{x}))$$

in arena $((Nat \to Nat) \to Nat) \to Nat$. The P-questions corresponding to red underlined variable and the enabling O-questions corresponding to the blue over-lined binder are the same, but the move occurrences which justify them are actually distinct and the plays are different, even though they consist of the same raw sequence of moves.

In general, we are concerned only with *well-justified* sequences, which we call *plays*. They represent computations which are causally sensible. Let us use $\sqsubseteq$ for sequence prefix and $\Subset$ for the sub-sequence relation.

Definition 6 (Play). *A justified sequence p in an arena A is a play when*

- *for any $p' \cdot ma \sqsubseteq p$ with $m \in M_A$ a move and a a pointer, there exists a question $q \in Q_A$ and a pointer b such that $qb\langle a\rangle \Subset p'$ and $q \vdash_A m$.*
- *if $qa\langle b\rangle \sqsubseteq p$ then $q \in I_A$, $a, b \in \mathbb{A}$.*

We denote the set of plays over arena A as $\mathbf{P}_A$ and the set of possibly ill-formed justified sequences over a set of moves M as J_M.

A *strategy* in an arena A is any set of plays which is prefix closed, closed under choices of pointer names ("equivariant"), and closed over O-moves.

Let $\pi : \mathbb{A} \to \mathbb{A}$ be bijections representing *name permutations*, and define *renaming actions* of a name permutation on a justified sequence over arena A as

$$\begin{aligned}
\pi \bullet \epsilon &= \epsilon \\
\pi \bullet (p \cdot a) &= (\pi \bullet p) \cdot (\pi(a)) && a \in \mathbb{A} \\
\pi \bullet (p \cdot \langle a\rangle) &= (\pi \bullet p) \cdot \langle \pi(a)\rangle && a \in \mathbb{A} \\
\pi \bullet (p \cdot m) &= (\pi \bullet p) \cdot m && m \in M_A.
\end{aligned}$$

Lemma 1. *If $p \in \mathbf{P}_A$ then $\pi \bullet p \in \mathbf{P}_A$, for any bijection $\pi : \mathbb{A} \to \mathbb{A}$.*

The proofs are in general elementary and will often leave them as exercise.

Definition 7 (Strategy). *A strategy over an arena $\sigma : A$ is a set of plays such that for any $p \in \sigma$*

prefix-closed *If $p' \sqsubseteq p$ then $p' \in \sigma$*

O-closed *If $p \cdot m \in \mathbf{P}_A$ for some $m \in O_A$ then $p \cdot m \in \sigma$*

equivariance *For any permutation π, $\pi \bullet p \in \sigma$.*

The conditions above have intuitive explanations. Since a strategy must capture all behaviours of a term, it makes sense for it to be prefix closed since any prefix represents a shorter interaction. The O-closure reflects the fact that a term has no control over which one of a range of possible next moves the context might chose to play. Finally, pointer equivariance is akin to an alpha-equivalence on plays, motivated by the fact that pointer names are not observable, so the choice of particular names in a play is immaterial.

Note that it is equivalently possible to present strategies as a next-move function from a play ending in an O-move to a P-move (or set of P-moves) along with the justification infrastructure, indicating the "next move" in the strategy.

$$\hat{\sigma} : \mathbf{P}_A^O \to \mathcal{P}(P_A) \times \mathbb{A}^2$$

Above, $\mathcal{P}$ is the power-set, $\mathbf{P}$ is the set of plays, P is the set of P-moves.

In this case the corresponding strategy is the smallest set σ such that

$$\epsilon \in \sigma \wedge p \in \sigma \text{ implies } p \cdot \hat{\sigma}(p) \subseteq \sigma.$$

Sometimes it is easier to present a strategy directly and some other times via the next-move function.

Definition 8. *Given a set of plays over an arena A, $\sigma \subseteq P_A$, let us write $\mathrm{strat}(\sigma)$ for the least strategy including σ.*

We will sometimes abuse the notation above by applying it to a set of sequences of moves, in the case that the pointer structure can be unambiguously reconstructed.

Example 4. *In arena Nat,*

$$\sigma = \mathrm{strat}(q \cdot 0) = \mathrm{strat}(qa\langle b\rangle \cdot 0b) = \{\epsilon, qa\langle b\rangle, qa\langle b\rangle \cdot 0b \mid a, b \in \mathbb{A}\}.$$

The next-move function is

$$\hat{\sigma}(qa\langle b\rangle) = 0a.$$

Now let us consider a variety of strategies denoting common programming language features.

2.2 Examples of strategies

2.2.1 Arithmetic

Any arithmetic operator $\circledast : nat \to nat \to nat$ is interpreted by a strategy over the arena $Nat \to Nat \to Nat$. Let us tag the moves of the first Nat arena with $-_1$, the moves of the second with $-_2$ and leave the third un-tagged (the trivial tag). Then the interpretation of the operator is in most cases

$$\sigma_{\circledast} = \text{strat}\left(\{qq_1m_1q_2n_2p \mid m, n, p \in \mathbb{N} \wedge p = m \circledast n\}\right)$$

Note that in the case of division, or any other operation with undefined values, the strategy must include those cases explicitly:

$$\sigma_{\div} = \text{strat}\left(\{qq_1m_1q_2n_np \mid m, n \neq 0, p \in \mathbb{N} \wedge p = m \div n\} \cup \{qq_1m_1q_20_2 \mid m \in \mathbb{N}\right)$$

Following the 0 O-answer, there is no way P can continue.[5]

From this point of view sequencing can be seen as a degenerate operator which evaluates then forgets the first argument, then evaluates and return the second

$$\sigma_{seq} = \text{strat}\left(\{qq_1m_1q_2n_2n \mid m, n \in \mathbb{N}\}\right)$$

Of course, sequencing commonly involves commands com which are degenerate, single-value, data types which are constructed just like the natural numbers but using a singleton set instead of $\mathbb{N}$.

The flexibility of the strategic approach also gives an easy interpretation to shortcut (lazy) arithmetic operations:

$$\sigma_{\times} = \text{strat}\left(\{qq_1m_1q_2n_2p \mid m \neq 0, n, p \in \mathbb{N} \wedge p = m \times n\} \cup \{qq_10_10\}\right)$$

This comes in handy when implementing an if-then-else operator (over natural numbers), in arena $Bool \to Nat \to Nat \to Nat$:

$$\sigma_{if} = \text{strat}\left(\{qq_1tt_1q_2n_2n \mid n \in \mathbb{N}\} \cup \{qq_1ff_1q_3n_3n \mid n \in \mathbb{N}\}\right)$$

[5] Although not common in the GS literature, we can consider this as a defeat of P in the game.

2.2.2 Non-determinism

A non-deterministic Boolean choice operator $choose\mathbb{b} : \mathbb{B}$ is interpreted by the strategy

$$\sigma_{choose\mathbb{b}} = \mathrm{strat}(\{q \cdot tt, q \cdot ff\})$$

where two P-answers are allowed. This can be extended to probabilistic choice by adding a probability distribution over the strategy.

Note that the flexibility of the strategic approach would allow the definition of computationally problematic operations such as unbounded non-determinism $choose\mathbb{n} : \mathbb{N}$,

$$\sigma_{choose\mathbb{n}} = \mathrm{strat}(\{qn \mid n \in \mathbb{N}\})$$

2.2.3 State

In order to model state we first need to find an appropriate arena to model assignable variables. In the context of call-by-name it is particularly easy to model *local* (bloc) variables ($new\ x\ in\ t$, where x is the variable name and t the term representing the variable block). It turns out that new needs not be a term-former but it can be simply a higher order language constant $new : (var \to T) \to T$ where T is some language ground type.

The type of variables var can be deconstructed following an "object oriented" approach. A variable must be readable $der : var \to nat$ and assignable $asg : var \to nat \to nat$. Since no other operations are applicable, we can simply define $var = nat \times (nat \to nat)$ which means $der = proj_1$, $asg = proj_2$ and the assignment behaves C-style, returning the assigned value. We will see later how projections are uniformly interpreted by strategies.

What is interesting is the interpretation of the new operation in arena $Var = Nat_1 \times (Nat_2 \to Nat_3)$. We used the tags directly (and informally) in the type itself in order to identify the moves.

We define this strategy using the next-move function:

$$\begin{aligned}
\hat{\sigma}_{new}(p \cdot q_3) &= q_2 \\
\hat{\sigma}_{new}(p \cdot n_2) &= n_3 \\
\hat{\sigma}_{new}(p \cdot n_3 \cdot q_1) &= n_1 \\
\hat{\sigma}_{new}(p \cdot n_1 \cdot q_1) &= n_1 \\
\hat{\sigma}_{new}(q) &= q' \\
\hat{\sigma}_{new}(p \cdot m') &= m.
\end{aligned}$$

The first two moves are just a copy-cat corresponding to assignment (a write requesting q_3 necessitates a value to be written q_2, and when the value is provided n_2 it is written n_3). The thrid and fourthe rules are the stateful behaviour, returning the most recently written n_3 or, respectively read n_1 when a read requesting question q_1 is answered to. Incidentally, this means that P has no move if an initial read request is made. The last two copy-cat rules embed the strategy for variables into $(\mathit{Var} \to T') \to T$ to interpret the binder.

2.2.4 Control

If-then-else is a very simple control operator, but more complex ones can be defined. The family of control operators is large, so let us look at a simple one, $\mathit{catch} : (\mathit{com}_1 \to \mathit{nat}_2) \to \mathit{option}\ \mathit{nat}$ where the type $\mathit{option}\ \mathit{nat}$ is interpreted in an arena constructed just like Nat but using $\mathbb{N} + 1$ instead of $\mathbb{N}$. The extra value indicates an error result (ϕ). Just like in the case of state, the construct $\mathit{catch}(\lambda x.t)$ can be sugared as $\mathit{escape}\ x\ \mathit{in}\ t$. If x is used in t then the enclosing catch returns immediately with ϕ, otherwise in returns whatever t returns.

The strategy is

$$\sigma_{catch} = \mathrm{strat}\left(\{qq_2n_2n \mid n \in \mathbb{N}\}\right) \cup \{qq_2q_1\phi\})$$

2.3 Composing strategies

In the previous section we looked at strategies interpreting selected language constants. In order to construct an interpretation of terms, denotationally, strategies need to *compose*.

The intuition of composing a strategy $\sigma : A \to B$ with a strategy $\tau : B \to C$ is to use arena B as an

interface on which in a first instance σ and τ will synchronise their moves. After that, the moves of B will be hidden, resulting in a strategy $\tau \circ \sigma : A \to C$. In order to preserved the good justification of plays all pointers that factor through hidden moves will be "extended" so that the hiding of the move will not leave them dangling.

In order to define composition some auxiliary operations are required.

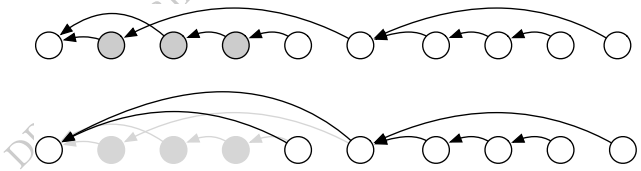
The first one is deleting move while extending the justification pointers "through" the deleted moves.

Definition 9 (Deletion)**.** *For an arena A, let $X \subseteq M_A$. For a play $p \in \mathbf{P}_A$ we define* deletion *inductively as follows, where we take $(p', \pi) = p \upharpoonright X$:*

$$\begin{aligned} \epsilon \upharpoonright X &= (\epsilon, id) \\ (p \cdot ma\langle b\rangle) \upharpoonright X &= (p' \cdot m \cdot \pi(a)\langle b\rangle, \pi) && \text{if } m \notin X \\ (p \cdot ma\langle b\rangle) \upharpoonright X &= (p', (\pi \mid b \mapsto \pi(a))) && \text{if } m \in X \end{aligned}$$

The result of a deletion is a pointer sequence along with a function $\pi : \mathbb{A} \to \mathbb{A}$ which represents the chain of pointers associated with deleted moves. Informally, by $p \upharpoonright A$ we will understand the first projection applied to the resulting pair.

For example, the removal of the greyed-out moves in the diagrammatic representation of the play below results in a sequence with reassigned pointers:



Note that in general the deletion of an arbitrary set of moves from a legal play does not result in another legal play, but in important special cases it does.

Lemma 2. *Given arenas A, B, C if $p \in \mathbf{P}_{A \times B \to C}$ then $p \upharpoonright A \in \mathbf{P}_{B \to C}$.*

The second operation is the selection of "hereditary" sub-plays, i.e. all the moves that can be reached from an initial set of moves following the justification pointers.

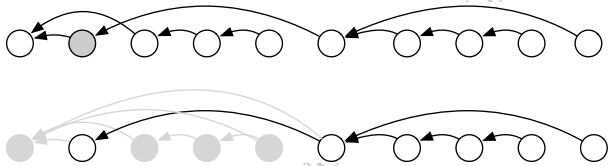
Definition 10 (Hereditary justification)**.** *Suppose $p \in \mathbf{P}_A$ and $X \subseteq \mathbb{A}$. We define the* hereditarily

justified *sequence* $p \upharpoonright X$ *recursively as below, where we take* $p \upharpoonright X = (p', X')$*:*

$$\begin{aligned} \epsilon \upharpoonright X &= (\epsilon, X) \\ (p \cdot ma\langle b\rangle) \upharpoonright X &= (p' \cdot ma\langle b\rangle, X \cup \{b\}) && \text{if } a \in X \\ (p \cdot ma\langle b\rangle) \upharpoonright X &= (p', X) && \text{if } a \notin X \end{aligned}$$

The result of a hereditary justification is a pointer sequence along with a set of names $X \subseteq \mathbb{A}$ which represents the addresses of selected questions. Informally, by $p \upharpoonright A$ we will understand the first projection applied to the resulting pair.

For example, the hereditary justification of the greyed-out moves in the diagrammatic representation of the play below results in the sequence below:



Note that in general the hereditary justification of an arbitrary set of moves from a legal play does not result in another legal play, but in important special cases it does.

Lemma 3. *Given arenas* A, B, C *if* $p \in \mathbf{P}_{A \times B \to C}$*, with* $ma\langle b\rangle \sqsubseteq p$*,* $m \in I_A$ *then* $p \upharpoonright \{b\} \in \mathbf{P}_A$*.*

Lemmas 2 and 3 are not just technical curiosities, but they have clear intuitions. The former says that if we hide all moves from A in a play over arena $A \times B \to C$, we we obtain a legal play. Moreover, the removed plays themselves are legal.

We now have the requisite operations to define the *interaction* and, finally, the *composition* of strategies.

Definition 11 (Interaction)**.** *Given sets of justified sequences* $\sigma \subseteq J_M, \tau \subseteq J_N$ *their* interaction *is defined as*

$$\sigma \underset{M,N}{\|} \tau = \{p \in J_{M \cup N} \mid p \downharpoonright (M \setminus N) \in \tau \wedge p \downharpoonright (N \setminus M) \in \sigma\}.$$

A good intuition for interaction is of two strategies synchronising their actions on the shared moves $M \cap N$.

Observation. This definition will be used to compute the interaction of strategies $\sigma : A \to B$ and $\tau : B \to C$, but we will ignore the issue of tagging of moves as they participate in the definition of

composite arenas, and we will just assume the underlying sets of are disjoint. This is not technically correct because the arenas can be equal, case in which the tagging is essential to disambiguate the co-products. But the formalisation of tagging, de-tagging and re-tagging is tedious and may obscure the main points. We are sacrificing some formality for clarity.

Example 5. *Let* $\tau = \mathrm{strat}(qq'm'(m+1))$ *denote a function that increments its argument and* $\sigma = \mathrm{strat}(q'q''n''(2n)')$ *a function that doubles its argument,* $\sigma, \tau : Nat \to Nat$*. Their interaction* $\sigma \,||\, \tau$ *is a set of sequences of the form* $qq'q''m''(2m)'(2m+1)$*.*

Definition 12 (Iteration)**.** *Given a set of justified sequences* $\sigma \in J_M$ *its iteration on* $N \subseteq M$ *is the set of justified sequences*

$$!_N\sigma = \{p \in J_M \mid \forall ma\langle b\rangle \sqsubseteq p.m \in N \Rightarrow p \upharpoonright \{b\} \in \sigma\}$$

A good intuition of iteration is a strategy interleaving its plays. The definition says that if we select moves form an identified subset N and we trace the hereditarily justified plays, they are all in the original set. We can think of each $p \upharpoonright \{b\}$ as untangling the "thread of computation" associated with move $ma\langle b\rangle$ from the interleaved sequence.

Example 6. *In the absence of iteration a strategy can interact with another strategy only once. Let* $\tau = \mathrm{strat}(qq'm'q'n'(m+n))$ *denote a function that evaluates its argument twice and returns the sum of received values, and let* $\sigma = \mathrm{strat}(q'0')$ *be the strategy for constant 0. The interaction* $\sigma \,||\, \tau$ *cannot proceed successfully because removing the untagged moves representing the result of* τ *leaves sequences of the shape* $q'm'q'n'$ *which are not in* σ *no matter what values* m, n *take. However, the interaction with iterated 0 is* $!\sigma \,||\, \tau = qq'0'q'0'0$*.*

Composition is iterated interaction with the synchronisation moves internalised and hidden.

Definition 13 (Composition)**.** *Given strategies* $\sigma : A \to B, \tau : B \to C$ *we defined their* composition *as* $\sigma;\tau = (!_{I_B}\sigma \underset{M_{A\to B}, M_{B\to C}}{||} \tau) \downarrow M_B$*.*

Theorem 4. *Given strategies* $\sigma : A \to B, \tau : B \to C$ *their composition is also a strategy* $\sigma;\tau : A \to C$*.*

Proof. We need to show that $\sigma;\tau$ is a set of well-justified plays, and that prefix-closure, O-closure, and equivariance, hold. This is immediate because both $\downarrow$ and $\upharpoonright$ when applied to a justified sequence produce a justified sequence, directly from definition. They also preserve prefix-closure and O-closure. Equivarience holds from general principles, as studied in the theory of nominal sets. □

2.4 Categories of games

2.4.1 Associativity

Composition is well defined, but we do not know whether it also has good mathematical properties. Primarily, is composition associative? And does it have an identity at every type?

Theorem 5 (Associativity). *For any three strategies* $\sigma : A \to B, \tau : B \to C, \upsilon : C \to D$, $(\sigma;\tau);\upsilon = \sigma;(\tau;\upsilon)$.

Proof. Elaborating the definitions, the LHS is

$$(!_C((!_B\sigma \underset{AB,BC}{||} \lfloor B) \underset{ABC,CD}{||} (\nu \lfloor B)) \lfloor C \quad = (!_C((!_B\sigma \underset{AB,BC}{||} \tau) \underset{ABC,CS}{||} \nu) \lfloor BC \tag{1}$$

There are no B-moves in $(!_C((!_B\sigma \underset{AB,BC}{||} \lfloor B)$, so we can extend the scope of $\lfloor B$.

Elaborating the definitions, the RHS is

$$(!_B\sigma \underset{AB,BD}{||} (!_C\tau \underset{BC,CD}{||} \nu) \lfloor C) \lfloor B \tag{2}$$

$$= (!_B(\sigma \lfloor C) \underset{AB,BD}{||} (!_C\tau \underset{BC,CD}{||} \nu) \lfloor C) \lfloor B \tag{3}$$

$$= (!_B\sigma \underset{AB,BD}{||} (!_C\tau \underset{BC,CD}{||} \nu) \lfloor C) \lfloor BC \tag{4}$$

Eq. 3 is true because there are no C-moves in σ, so $\sigma \lfloor C = \sigma$.

Eqn. 4 is true because $\lfloor C$ distributes over concatenation.

Therefore, it is sufficient to show the expressions in Eqns. 1 and 3 are equal.

Let $p \in !_C((!_B\sigma \underset{AB,BC}{||} \tau) \underset{ABC,CS}{||} \nu$ is equivalent, by definition with

$$p \lfloor D \in !_C(!_B\sigma \underset{AB,BC}{||} \tau) \tag{5}$$

$$\wedge\, p \lfloor AB \in \nu \tag{6}$$

By elaborating the definitions:

$$\text{Prop. 5} \Leftrightarrow \forall ma\langle b\rangle.m \in I_C \Rightarrow p \lfloor D \rceil \{b\} \in !_B\sigma \underset{AB,BC}{||} \tau \tag{7}$$

$$\Leftrightarrow p \lfloor D \rceil \{b\} \lfloor C \in !_B\sigma \tag{8}$$

$$\wedge\, p \lfloor D \rceil \{b\} \lfloor A \in \tau \Leftrightarrow p \rceil \{b\} \lfloor A \in \tau \tag{9}$$

The equivalence in Eqn. 9 holds because once we restrict to moves hereditarily justified by a C-move ($m \in I_C$), the removal of D-moves has no effect since the hereditarily justified play is restricted to arenas ABC.

Elaborating the definition yet again,

$$\text{Prop. 8} \Leftrightarrow \forall nc\langle d\rangle \sqsubseteq p \lfloor D \rceil \{b\} \lfloor C.n \in I_B \tag{10}$$

$$\Rightarrow p \lfloor D \rceil \{b\} \lfloor C \rceil \{d\} \in \sigma \Leftrightarrow p \rceil \{d\} \in \sigma \tag{11}$$

Because restricting to the hereditarily justified play of a B-move ($n \in I_B$) makes the other restrictions irrelevant.

To summarise,

$$p \in !_C((!_B\sigma \underset{AB,BC}{||} \tau) \underset{ABC,CS}{||} \nu \Leftrightarrow$$

$$\forall ma\langle b\rangle.m \in I_C \Rightarrow p \rceil \{b\} \in \tau \wedge \forall nc\langle d\rangle \sqsubseteq p \rceil \{b\}.n \in I_B \Rightarrow p \rceil \{d\} \in \sigma, \tag{12}$$

This was the more difficult case.

$p \in !_B\sigma \underset{AB,BD}{||} (!_C\tau \underset{BC,CD}{||} \nu)$ is equivalent to the same conditions as in Eqn. 12 simply by elaborating the definitions, except that Prop. 9 appears as $p \lfloor A \rceil \{b\} \in \tau$, but $\rceil, \lfloor$ commute in this case. □

We show this proof in detail, because it is generally considered to be one of the most challenging proofs of game semantics. However, with the encoding of justification pointers as names the proof is an unfolding of complex definitions, but it is ultimately elementary, boiling down to some basic properties of $\lfloor$ and $\rceil$.

Composition is not only associative but also monotonic with respect to the inclusion ordering:

Theorem 6 (Monotonicity). *If $\sigma \subseteq \sigma'$ then for any τ, υ, $\sigma;\tau \subseteq \sigma';\tau$ and $\upsilon;\sigma \subseteq \upsilon;\sigma'$.*

2.4.2 Identity

A candidate for identity $\kappa_A : A_0 \to A_1$ is a strategy which immediately replicates O-moves from A_0 to A_1 and vice versa – a *copy-cat strategy*.

Definition 14 (Copy-cat)**.** *For any arena A we defined κ_A as*

$$\hat{\kappa}_A(q_1 a\langle b\rangle) = q_0 b\langle c\rangle$$
$$\hat{\kappa}_A\left(p \cdot m_i a\langle b\rangle \cdot m_{1-i} c\langle d\rangle \cdot p' \cdot n_j d\langle e\rangle\right) = n_{1-j} b\langle f\rangle$$
$$\hat{\kappa}_A\left(p \cdot m_i a\langle b\rangle \cdot m_{1-i} c\langle d\rangle \cdot p' \cdot n_j d\right) = n_{1-j} b.$$

where $i, j = 1, 2$.

A copy-cat strategy has "the same behaviour" in both components:

Lemma 4. *$\kappa_A \restriction A_0 = \kappa_A \restriction A_1$, up to a relabelling of moves.*

Theorem 7. *$\kappa_A : A_0 \to A_1$ is a strategy.*

However, κ_A is not a unit for composition. Consider the strategy $\sigma : unit_0 \to unit_1, \sigma_{bad} = \mathrm{strat}(q_1 q_0 a_1 a_0)$. When composed with κ_{unit} the resulting strategy is $\sigma_{bad}; \kappa = \mathrm{strat}\{q_1 q_0 a_1 a_0, q_1 q_0 a_0 a_1\} \neq \sigma_{bad}$. The problem is the lack of synchronisation between the plays happening in the three arenas participating in composition.

One solution is to notice that the strategy $\kappa_A^* = \kappa_A; \kappa_A$ is idempotent relative to composition, i.e. $\kappa_A^*; \kappa_A^* = \kappa_A^*$, which allows the construction of a category via the Karoubi envelope where every strategy $\sigma : A \to B$ is "saturated" $\sigma^* = \kappa_A^*; \sigma; \kappa_B^*$. Indeed these strategies now form a category.

However, by saturating the strategies we have sacrificed a great deal of expressiveness which is essential in programming language design: the ability to control the sequencing of moves! For example, the sequencing strategy $\sigma = \mathrm{strat}(q q_1 a_1 q_2 a_2 a)$ by saturation includes all plays in which q precedes q_i, a and q_i precedes a_i but any other reordering is allowed. This is no longer sequencing, but parallelism without the possibility for synchronisation. This may not be what we want.

In the sequel we will see that the solution is to restrict the shape of plays to so-called "legal plays" in which problematic sequences such as those in σ_{bad} are no longer possible, and in which some degree of control over sequencing can be expressed. One such condition is *alternation* a condition that requires that O-moves and P-moves alternate, and it is conceptually connected to sequentiality. Intuitively it means that P cannot trigger consecutive actions without O responding, and vice-versa. In σ_{bad} the moves are OPPO, which is non-alternating so that strategy is ruled out.

Subject to alternation we find plays in $unit_0 \to unit_1$ which are quite reasonable, such as $q_1 q_0 a_0 q_0 \cdots q_0 a_0 a_1$ which correspond to a function interrogating its argument sequentially a number of times, as we would find in $\lambda x.x; x; \cdots x$. But we also find plays such as $q_1 a_1 q_0 a_0$ which correspond to a rather unusual function that is described by the following set of actions: the function is called, the function returns, the function calls its argument (!), the function argument returns. This is reminiscent of a (sequential) `fork()` operator which returns before executing its argument. If this behaviour is not possible in a given language it must be ruled out too.

With these considerations in mind we are beginning to understand and appreciate the art and science of game semantics, which can be roughly summarised as

1. Start in the setting of justified sets of sequences – plays and strategies.
2. Discover combinatorial constraints on plays and strategies which rule out undesired behaviour (e.g. 'alternation').
3. Ensure that the constraints are preserved by composition.
4. Discover closure conditions on strategies which ensure strategy composition is associative and has an identity, i.e. they form a category.
5. Ensure that the closure conditions do not ruin expressiveness by introducing undesired behaviour in strategies.

In the next section we shall see such a model, for *Concurrent Idealised Algol*, a language with higher-order functions, recursion, concurrency, and local state and synchronisation primitives.

2.5 Annotated further reading list

This is one of several tutorial introductions to game semantics. For a fuller picture the reader is also recommended

> Abramsky, Samson, and Guy McCusker. "Game semantics." Computational logic. Springer, Berlin, Heidelberg, 1999.
>
> Murawski, Andrzej S., and Nikos Tzevelekos. "Nominal game semantics." Foundations and Trends in Programming Languages 2.4 (2016): 191-269.

These notes are meant to be a hands-on introduction rather than a scholarly survey. For a broader survey the reader is referred to

> Ghica, Dan R. "Applications of game semantics: From program analysis to hardware synthesis." Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on. IEEE,

2009.

The two papers that firmly put game semantics on the map are the ones that solved (independently) the definability problem for PCF, i.e. for a higher-order, pure, sequential language:

Abramsky, Samson, Radha Jagadeesan, and Pasquale Malacaria. "Full abstraction for PCF." Information and Computation 163.2 (2000): 409-470.

Hyland, J. Martin E., and C-HL Ong. "On full abstraction for PCF: I, II, and III." Information and computation 163.2 (2000): 285-408.

The second paper (Hyland-Ong) introduced the concept of 'justification pointer' which proved to be popular in subsequent development.

Both papers achieved their main result by restricting strategies to so-called *innocent* strategies, a sophisticated and somewhat mysterious version of history-freeness which generated a significant amount of research on its own:

Danos, Vincent, and Russell Harmer. "The anatomy of innocence." International Workshop on Computer Science Logic. Springer, Berlin, Heidelberg, 2001.

Harmer, Russ, and Olivier Laurent. "The anatomy of innocence revisited." International Conference on Foundations of Software Technology and Theoretical Computer Science. Springer, Berlin, Heidelberg, 2006.

Harmer, Russ, Martin Hyland, and Paul-André Mellies. "Categorical combinatorics for innocent strategies." Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on. IEEE, 2007.

The particular formalisation of justification pointers in this presentation relies on the mathematical concept of 'name' and its theory, which ensures that inductive or recursive definitions relying on names are well-defined and hold up to alpha-equivalence (i.e. name permutations). For details the reader is referred to

Pitts, Andrew M. Nominal sets: Names and symmetry in computer science. Cambridge University Press, 2013.

The original presentation of this nominal formalisation of game semantics is

Gabbay, Murdoch, and Dan Ghica. "Game semantics in the nominal model." Electronic Notes in Theoretical Computer Science 286 (2012): 173-189.

The problem of copy-cat not being an identity for composition emerged out of the game-semantic study of *linear logic*. In fact, linear logic provided an important conceptual nexus between games and programming languages. More on game semantics and linear logic can be studied in

> Blass, Andreas. "A game semantics for linear logic." Annals of Pure and Applied logic 56.1-3 (1992): 183-220.
>
> Girard, Jean-Yves. "Linear logic: its syntax and semantics." London Mathematical Society Lecture Note Series (1995): 1-42.
>
> Abramsky, Samson, and Radha Jagadeesan. "Games and full completeness for multiplicative linear logic." The Journal of Symbolic Logic 59.2 (1994): 543-574.

Note that the problem of identity is not specific to game semantics, but it is a thorny one in trace semantics of concurrent processes, both synchronous and asynchronous.

> Ghica, Dan R. "Diagrammatic reasoning for delay-insensitive asynchronous circuits." Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky. Springer, Berlin, Heidelberg, 2013. 52-68.
>
> Ghica, Dan R., and Mohamed N. Menaa. "Synchronous game semantics via round abstraction." International Conference on Foundations of Software Science and Computational Structures. Springer, Berlin, Heidelberg, 2011.

The definition of composition via 'synchronisation and hiding' is inspired by the trace semantics of CSP

> Hoare, Charles Antony Richard. "Communicating sequential processes." Communications of the ACM 21.8 (1978): 666-677.
>
> Brookes, Stephen D., Charles AR Hoare, and Andrew W. Roscoe. "A theory of communicating sequential processes." Journal of the ACM (JACM) 31.3 (1984): 560-599.
>
> Roscoe, Bill. "The theory and practice of concurrency." (1998). (`http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`)

Finally, other connections between computation and interaction, inspired by linear logic, are the encoding of the lambda calculus into process calculus and the Geometry of Interaction:

> Milner, Robin. "Functions as processes." Mathematical structures in computer science 2.2 (1992): 119-141.
>
> Mackie, Ian. "The geometry of interaction machine." Proceedings of the 22nd ACM SIGPLAN-

SIGACT symposium on Principles of programming languages. ACM, 1995.

3 Idealised Concurrent Algol

3.1 Syntax

The base types of the language are $\beta ::= exp \mid com$, i.e. integer expressions and commands. Assignable variables, and semaphores are the other ground types $\gamma ::= \beta \mid var \mid sem$. The types are $\theta ::= \gamma \mid \theta \to \theta$. The language is an applied lambda calculus with a simple type system. The judgements of the type system are $x_0 : \theta_0, \ldots, x_k : \theta_k \vdash M : \theta$ where $x_i : \theta_i$ are variable type assignments, M is a term, and θ its type. The set of variable-type assignments is denoted by Γ.

$$\frac{x : \theta \in \Gamma}{\Gamma \vdash x : \theta} \qquad \frac{\Gamma, x : \theta \vdash M : \theta'}{\Gamma \vdash \lambda x : \theta . M : \theta \to \theta'} \qquad \frac{\Gamma \vdash M : \theta \to \theta' \quad \Gamma \vdash N : \theta}{\Gamma \vdash MN : \theta'}$$

The constants of the language are:

$\underline{k} : exp$	integer constants
$skip : com$	null command
$\star : exp \to exp \to exp$	integer operations
$ifz : exp \to \theta \to \theta \to \theta$	branching on zero ifz_then_else
$seq : com \to \beta \to \beta$	sequential composition ;
$par : com \to com \to com$	parallel composition $\mid\mid$
$asg : var \to exp \to com$	assignment $:=$
$der : var \to exp$	dereferencing !
$grab, release : sem \to sem$	binary semaphore operations
$newvar : exp \to (var \to \beta) \to \beta$	variable binder $int\ x := k\ in\ M$
$newsem : exp \to (sem \to \beta) \to \beta$	semaphore binder $sem\ x := k\ in\ M$

To avoid the complications of un-initialised local variables and semaphore we give the initial value as an extra parameter.

To illustrate the syntax of the language (sugared or desugared) consider a term with a race condition:

Example 7. *Sugared syntax*

```
int x := 0 in
  (x := 1 + !x) || (!x := 2 * !x);
  return x
```

versus unsugared syntax

$$newvar\ 0\ (\lambda x.seq\ (par\ (asg\ x\ (1 + der\ x))\ (asg\ x\ (2 * der\ x)))\ (der\ x))$$

We will use whichever version of the syntax is appropriate in particular contexts.

3.2 Operational semantics

The execution of the language is described by a *small step* operational semantics using a transition relation

$$\Sigma \vdash M, s \longrightarrow M', s'.$$

where Σ is a set of names of memory cells and locks, s and s' are states (functions $s, s' : \Sigma \longrightarrow \mathbb{N}$, and M, M' terms. If it causes no confusion Σ, s can be elided to keep the rules simpler.

Let $f' = (f \mid x_0 \mapsto y)$ represent a function such that if $f'(x_0) = y$ and for $x \neq x_0$, $f'(x) = f(x)$.

The basic reduction rules are:

$$\underline{k}_0 \star \underline{k}_1 \longrightarrow \underline{k} \qquad k = k_0 \star k_1 \tag{13}$$

$$skip; skip \longrightarrow skip \tag{14}$$

$$skip \,||\, skip \longrightarrow skip \tag{15}$$

$$M(\lambda x.N) \longrightarrow N \qquad M \in \{newvar, newsem\}, N \in \{\underline{k}, skip\} \tag{16}$$

$$ifz\ 0\ M\ N \longrightarrow M \tag{17}$$

$$ifz\ \underline{k}\ M\ N \longrightarrow N \qquad \underline{k} \neq 0 \tag{18}$$

$$(\lambda x.M)N \longrightarrow M[N/x] \tag{19}$$

$$\Sigma, l \vdash der\ l, s \longrightarrow \underline{k}, s \qquad s(l) = \underline{k} \tag{20}$$

$$\Sigma, l \vdash asg\ l\ \underline{k}, s \longrightarrow skip, s' \qquad s' = (s \mid l \mapsto \underline{k}) \tag{21}$$

$$\Sigma, l \vdash grab\ l, s \longrightarrow skip, s' \qquad s(l) = 0 \wedge s' = (s \mid l \mapsto 1) \tag{22}$$

$$\Sigma, l \vdash release\ l, s \longrightarrow skip, s' \qquad s(l) = 1 \wedge s' = (s \mid l \mapsto 0) \tag{23}$$

$$M\ \underline{k}\ (\lambda x.M_0), s \longrightarrow M_0, s \qquad M \in \{newvar, newsem\}, M_0 \in \{skip, \underline{k}'\} \tag{24}$$

Let $f \otimes (x \mapsto y)$ be a function which extends the domain of f with a new value x, mapped to y.

Other rules serve to define the contexts in which reductions (eventually) happen:

$$\frac{\Sigma \vdash M_0, s \longrightarrow M_0', s'}{\Sigma \vdash M_0\ M_1, s \longrightarrow M_0'\ M_1, s'} \tag{25}$$

$$\frac{\Sigma \vdash M_1, s \longrightarrow M_1', s'}{\Sigma \vdash M_0\ M_1, s \longrightarrow M_0\ M_1', s'}\ (M_0 \in \{der, grab, release\}) \tag{26}$$

$$\frac{\Sigma \vdash M_1, s \longrightarrow M_1', s'}{\Sigma \vdash M_0\ M_1\ M_2, s \longrightarrow M_0\ M_1'\ M_2, s'}\ (M_0 \in \{\star, seq, asg, newvar, newsem\}) \tag{27}$$

$$\frac{\Sigma \vdash M_2, s \longrightarrow M_2', s'}{\Sigma \vdash M_0\ M_1\ M_2, s \longrightarrow M_0\ M_1\ M_2', s'}\ \left(\begin{array}{r} M_0 \in \{\star, seq, asg, newvar, newsem\}, \\ M_1 \in \Sigma \cup \{\underline{k}, skip\} \end{array}\right) \tag{28}$$

$$\frac{\Sigma \vdash M_1, s \longrightarrow M_1', s'}{\Sigma \vdash par M_1\ M_2, s \longrightarrow par M_1'\ M_2, s'} \tag{29}$$

$$\frac{\Sigma \vdash M_2, s \longrightarrow M_2', s'}{\Sigma \vdash par M_1\ M_2, s \longrightarrow par M_1\ M_2', s'} \tag{30}$$

$$\frac{\Sigma \vdash M_0, s \longrightarrow M_0', s'}{\Sigma \vdash ifz\ M_0'\ M_1\ M_2, s \longrightarrow M_0'\ M_1\ M_2, s'} \tag{31}$$

The most complex rule combines reductions with guiding the context of further reductions, for local variable and semaphore binders:

$$\frac{\Sigma, l \vdash M[l/x], s \otimes (l \mapsto k) \longrightarrow M', s' \otimes (l \mapsto k')}{\Sigma \vdash M_0\ \underline{k}\ (\lambda x.M), s \longrightarrow M_0\ \underline{k'}\ (\lambda x.M[x/l])}\ M_0 \in \{newvar, newsem\} \tag{32}$$

Some comments on the rules above:

- We have a CBN language so the function is reduced first (Rule 25) and the argument is substituted as a thunk rather than be evaluated (Rule 16).
- Unary operators evaluate their argument (Rule 26) until they reach a value, which is then processed (Rules 20, 22, 23).
- Most binary operators evaluate their first argument (Rule 27) until they reach a constant, then the second argument (Rule 28) until they reach another value, which are then reduced (Rules 14, 13, 15) ...
- ...except parallel composition, which nondeterministically chooses one of the arguments as a reduction context (Rule 29, 30).
- The branching operator evaluates the guard (Rule 31) then picks one of the branches lazily (Rules 17, 18).
- When the binders encounter a syntactically constant function such as $\lambda x.skip$ or $\lambda x.\underline{k}$ they return the constant value (Rule 24).
- Otherwise, when the binders encounter a function $\lambda x.M$ they instantiate the x to a fresh location l, extending the state with its initialiser k. The initialiser is used to capture the current value of the state for further evaluations (Rule 32).

Definition 15 (Termination). *If* $\Sigma \vdash M, s \longrightarrow^* M_0, s'$ *with* $s' \in \{\underline{k}, skip\}$ *we say that* M terminates *in state* s*, written as* $M, s \Downarrow$*. Furthermore, we write* $M \Downarrow$ *for* $M, \emptyset \Downarrow$.

Termination may fail because of 'stuck' illegal configurations such as $newsem\ 1\ (\lambda x.grab\ x)$ or $\underline{1}/\underline{0}$.

Definition 16 (Approximation). *Given* $\Gamma \vdash M_i : \theta$*,* $i = 1, 2$*, if for any context* $\mathcal{C}[-] : com.\mathcal{C}[M_1] \Downarrow$ *implies* $\mathcal{C}[M_2] \Downarrow$ *we write* $\Gamma \vdash M_1 \sqsubseteq_\theta M_2$.

Definition 17 (Equivalence). *Given* $\Gamma \vdash M_i : \theta$*,* $i = 1, 2$*, if* $\Gamma \vdash M_1 \sqsubseteq_\theta M_2$ *and* $\Gamma \vdash M_2 \sqsubseteq_\theta M_2$ *we write* $\Gamma \vdash M_1 \cong_\theta M_2$.

The contextual equivalence of two terms ensures that two terms can be substituted for each other. In a language with higher order functions and state reasoning about equivalences is a complex and subtle

undertaking.

Example 8.

$$x : var \vdash !x - !x \ncong 0$$

Consider the context

```
int v := 0 in
int z := 0 in
z := (fun x -> -) v || v := !v + 1;
return z
```

Example 9.

$$newvar\ \underline{k}\ (\lambda x : var.!x - !x) \cong 0$$

The difference between Ex. 8 and Ex. 9 is that variable x is free rather than local, allowing for hidden interference. The intuitions are clear, but a direct argument from the operational semantics and the definition of equivalence is very hard!

3.3 Game semantics

We use the arena structures and the justification sequences from the previous section. Most of the strategies involved have already been described. We only describe the only new one, parallel composition.

Let $p \mid q$ denote the set of all the possible interleavings of two justified sequences which have disjoint sets of pointer-names[6]:

$$\epsilon \mid q = \{q\}$$

$$p \mid \epsilon = \{p\}$$

$$p_0 \cdot p \mid q_0 \cdot q = p_0 \cdot (p \mid q_0 \cdot q) \cup q_0 \cdot (p_0 \cdot p \mid q)$$

with p_0, q_0 move occurrences.

$$\sigma_{par} : com_1 \to com_2 \to com$$

[6] A technical observation: The equivariance condition on strategies means that if we want to interleave two plays and the pointer-names clash we can simply pick the play which is the same except for the renaming of pointer-names so that there are no clashes. Refreshing names is a generalisation of what happens in capture-avoiding substitution in the lambda calculus, and is extensively studied in the theory of nominal sets.

$$\sigma_{par} = \mathrm{strat}(q \cdot (q_1 a_1 \mid q_2 a_2) \cdot a).$$

The question we must address now is:

> *What are the right combinatorial constraints (legality conditions) on plays and closure conditions on strategies which a) are compositional, b) allow us to express all desired behaviours, c) allow us to express only desired behaviours.*

These conditions will come from understanding the essence of ICA as an asynchronous, concurrent language in which all concurrency is "local", as introduced by the parallel composition operator. This means that all "threads" of computation must fork and join in a well-nested way. A "parent" thread may not terminate before its "child" threads have also terminated. At the level of games, this is neatly reflected by the following principle:

> *If a question is answered then all questions justified by had been answered.*

It is formally easier to break this down into two simpler properties:

Fork: Only a question that has not been answered can justify.

Join: A question can be answered only when no questions it justifies are still pending.

These can be seen as constraints on the scope of justifiers.

Definition 18 (Strict scoping)**.** *If $p \cdot ma\langle\_\rangle \cdot p' \in \mathbf{P}_A$, $m \in A_A$ we say that it is* strictly scoped *if $a \notin p'$.*

This is the legality condition for forking: if a pointer name a is used as the justifier of an answer m, then it can never be used again, so that the question which introduces it cannot justify other questions, or indeed answers.

Definition 19 (Strict nesting)**.** *If $p_1 \cdot m_1 a\langle b\rangle \cdot p_2 \cdot m_2 b\langle c\rangle \cdot p_3 \cdot m_1' b\langle\_\rangle \in \mathbf{P}_A$, $m_1' \in A_A$, we say that it is* strictly nested *if $m_2' c\langle\_\rangle \sqsubset p_3$ for some $m_2' \in A_A$.*

This formalises the legality condition for joining: if a question m_1 is answered, by some m_1', then any other question m_2 it justifies must also be answered, by some m_2'.

It can be verified that

Theorem 8. *If $\sigma : A \to B, \tau : B \to C$ are strategies consisting of strictly scoped, strictly nested plays then their composition $\sigma; \tau : A \to C$ consists of strictly scoped, strictly nested plays.*

The right closure condition for strategies is a staple of models of asynchronous concurrency: the only possible synchronisation is that P (output) can synchronise on O (input). Every other synchronisation

is impossible. For example the order in which two P-moves are issued cannot be controlled. Even if P can make the moves in a particular sequence it is no guarantee that, due to unknown delays, they will be received in the same order.

Definition 20. *A strategy of strictly scoped, strictly nested plays $\sigma : A$ is* asynchronous *if and only if for any $p \cdot p_0 \cdot p_1 \in \sigma$ with $p_0 = m_0 a_0 \langle b_0 \rangle$ and $p_1 = m_1 a_1 \langle b_1 \rangle$ and $m_0 \in P_A$ or $m_1 \in O_A$ then if $p \cdot p_1 \cdot p_0$ is strictly scoped and strictly nested then $p \cdot p_1 \cdot p_0 \in \sigma$.*

From now one when we write $\mathrm{strat}(\sigma)$ we also will mean, additionally, asynchronous-closure.

Theorem 9. *If $\sigma : A \to B, \tau : B \to C$ are asynchronous strategies then their composition $\sigma; \tau : A \to C$ is an asynchronous strategy.*

Theorem 10. *If $\sigma : A \to B$ is an asynchronous strategy then $\mathrm{strat}(\kappa_A); \sigma = \sigma = \sigma; \mathrm{strat}(\kappa_B)$*

In other words, the asynchronous-closure of the copy-cat strategy is an identity for composition.

Some of the strategies discussed above are already asynchronous. For example, σ_{par} includes all the relevant move permutations by definitions, whereas σ_{seq} does not gain any new plays by asynchronous closure.

The only strategies that are not asynchronous by construction are $\sigma_{newvar}, \sigma_{newsem}$. Using the Karoubi envelope we can "lift them" by composition with $\mathrm{strat}(\kappa_{var \to \beta})$ and $\mathrm{strat}(\kappa_{sem \to \beta})$ respectively. They are used to model *newvar* and *newsem*.

The lifted strategies for state and semaphores are rather difficult to grasp directly. The following sequence of grabs and releases is present in the unlifted strategy for semaphores (we write g, r for the questions and respectively a for the answer). We colour the moves so that we can track them through permutations, and we spell out P or O as an index:

$$\cdots g_O \cdot a_P \cdot r_O \cdot a_P \cdots \tag{33}$$

The O-moves can be arbitrarily hurried and the P-moves delayed, which means that the following is an equivalent play, which also corresponds to "grab then release"!

$$\cdots r_O \cdot g_O \cdot a_P \cdot a_P \cdots \tag{34}$$

However, since $\underline{a}_P$ can synchronise on $\overline{g}_O$ it means that the following play is impossible:

$$\cdots r_O \cdot a_P \cdot g_O \cdot a_P \cdots \tag{35}$$

Which is just what we want, since this is "release then grab", which is clearly a different way of sequencing grab and release. Also, no alternation of sequences $\cdots garagara \cdots$ will contain in its asynchronous closure sub-sequences of shape $\cdots gaga \cdots$, i.e. two successful grab operations. However, Sequence (34) contains in its asynchronous completion both Sequence (33) and (35), which means that in an unrestricted interleaving of grab and release operations one successful interleaving that can be legally synchronised can be found. But more on this in Sec. 3.4.1.

3.4 Model of the (applied) lambda calculus

In order to establish soundness we first need to establish that the category of asynchronous strategies is a model of the lambda calculus, i.e. a Cartesian Closed Category. Going via the categorical model is more economical than making a direct syntactical argument.[7]

Theorem 11 (CCC). *The category of asynchronous strategies is Cartesian Closed with*

Terminal object *is the arena I with no moves $M_I = \emptyset$*

The product *of two arenas A_1, A_2 is the arena $A_1 \times A_2$ with projections $\pi_i : A_1 \times A_2 \to A_i'$ defined by the copy-cat strategy between arenas A_i and A_i', $\pi_i = \mathrm{strat}(\kappa_{A_i})$.*

The exponential *of two arenas A, B is the arena $A \to B$ with*

- **Evaluation morphism** $ev : (A \to B) \times A' \to B'$ *with* $ev = \mathrm{strat}(\kappa_A) \cup \mathrm{strat}(\kappa_B)$
- **Transpose** *of any strategy $\sigma : A \times B \xrightarrow{\cdot} C$ is the strategy $\lambda\sigma : A \xrightarrow{\cdot} (B \to C)$ where λ is the appropriate move re-tagging operation.*

Note that the strategy for π_i, is on the arena $A_i \to A_i'$. However, since strategies are sets of sequences, any strategy on arena A may also be a strategy on B if $M_B \supseteq M_A$, and $M_{A_1 \times A_2 \to A_i'} \supseteq M_{A_i \to A_i'}$. Same for the evaluation morphism.

For the product, the product of strategies $\sigma_i : A \to A_i$, $\langle \sigma_i, \sigma_2 \rangle : A \to A_1 \times A_2$ is their union.

To understand the re-tagging operation λ let us make the tags explicit in the arena constructions ($'$ and $''$):

$$(A' \times B'')' \to C'' = (A')' \times (B'')' \to C''$$

[7] The reader with no knowledge of category theory may ignore the categorical content of this section.

versus

$$A' \to (B' \to C'')'' = A' \to (B')'' \to (C'')''$$

Modulo the tagging of the moves the two composite arenas above are the same. The re-tagging is the bijection

$$\lambda(m')' = m'$$
$$\lambda(m'')' = (m')''$$
$$\lambda(m'') = (m'')''.$$

In general we avoid diving into the details of the formalisation of the tags, but in the case of transposition it is quite important.

The interpretation of the lambda-calculus is standard in a CCC. The interpretation is inductive on the derivation of the type (which is unique). The interpretation of a judgement $\Gamma \vdash M : \theta$ is a strategy written as $[\![M]\!] : [\![\Gamma]\!] \to [\![\theta]\!]$ where $[\![\theta]\!]$ is the arena interpreting the type

$$[\![nat]\!] = Nat$$
$$[\![com]\!] = Com$$
$$[\![var]\!] = Nat \times (Nat \to Com)$$
$$[\![sem]\!] = Com \times Com$$
$$[\![\theta \to \theta']\!] = [\![\theta]\!] \to [\![\theta']\!].$$

and $[\![\Gamma]\!] = [\![x_0 : \theta_0, \ldots, x_k : \theta_k]\!] = [\![\theta_0]\!] \times \cdots \times [\![\theta_k]\!]$.

The lambda terms are interpreted by:

$$[\![\Gamma, x_k : \theta_k, \Gamma' \vdash x_k : \theta_k]\!] = \pi_k$$
$$[\![\Gamma \vdash M\ N : \theta]\!] = \big\langle [\![\Gamma \vdash M : \theta' \to \theta]\!], [\![\Gamma \vdash N : \theta']\!] \big\rangle ; ev$$
$$[\![\Gamma \vdash \lambda x : \theta . M : \theta \to \theta']\!] = \lambda [\![\Gamma, x : \theta \vdash M : \theta']\!].$$

To make it more concrete we can spell out the categorical definitions:

$$[\![\Gamma, x : \theta \vdash x_k : \theta]\!] = \mathrm{strat}(\kappa_\theta)$$
$$[\![\Gamma \vdash M\ N : \theta]\!] = \big([\![M]\!] \cup [\![N]\!]\big) ; \big(\mathrm{strat}(\kappa_\theta) \cup \mathrm{strat}(\kappa_{\theta'})\big)$$

$$[\![\Gamma \vdash \lambda x : \theta.M : \theta \to \theta']\!] = \lambda[\![M]\!].$$

For the language constants M_0, we use the strategies already defined:

$$[\![\ \ \vdash M_0 : \theta]\!] = \sigma_{M_0}.$$

This completes the definition of the game-semantic model of ICA.

3.4.1 Examples

For a closed program of command type the only observable behaviour is termination vs. non-termination. It is useful to look at a non-terminating program.

Example 10 ($\Omega : com$)**.**

$$\Omega = newsem\ 1\ (\lambda x.grab\ x)$$

The body of the command is $[\![\lambda x.grab\ x]\!] = \mathrm{strat}(q''g'a'a'') : sem' \to com''$*. This needs to compose to* $\sigma_{newsem,1} : (sem' \to com'') \to com$ *which has plays of the form* $qq''r'a'g'a' \cdots$*. The only plays that can be in the interaction sequence is* qq'' *since* g' *and* r' *do not match.*

Therefore, $[\![\Omega]\!] = \mathrm{strat}(q)$*, i.e. a strategy which does not respond to the initial O-move.*

Example 11 (Test)**.** *A useful function is* $test : exp' \to com''$*,* $test = \lambda x.ifz\ x\ skip\ \Omega$*,* $[\![test]\!] = \mathrm{strat}(q''q'0'a, q''q'n')$ *where* $n \neq 0$*.*

Example 12 (Synchronisation)**.**

$$[\![newsem\ 1\ (\lambda x.grab\ x\ ||\ release\ x)]\!] = \mathrm{strat}(qa) = [\![skip]\!].$$

The strategy for $grab\ x\ ||\ release\ x$ *has all the interleavings of* qa *and* ra*, which includes* $raga$ *which will compose successfully with* $newsem\ 1$*. It is interesting to note that it also has interleavings such as* $raga$ *which do not compose successfully. However, since strategies are prefix-closed these failed compositions do not contribute any interesting behaviour to the strategies, as they are prefixes of successful compositions.*

Example 13 (Choice)**.**

$$[\![newvar1\ (\lambda x.(x := 1\ ||\ x := 0); !x)]\!] = \mathrm{strat}(q0, q1)$$

As one might expect, concurrency and race conditions can be used to implement non-deterministic behaviour.

Example 14 (Angelic choice)**.**

$$[\![test(newvar1\ (\lambda x.(x := 1 \,||\, x := 0); !x))]\!] = \mathrm{strat}(q0) = [\![0]\!].$$

The same situation from Ex. 12 arises here, except the result is not maybe what one would expect. The term being tested is 0 or 1, non-deterministically. If it produces 0 then we terminate, otherwise we get stuck. Because a 'stuck' play is a sub-play of a successful play it contributes nothing to the strategy, which ends up being equal to that for the constant 0.

While this is clearly indicative of a weakness in the model, it is consistent with the behaviour described by our operational termination predicate, $test(newvar1\ (\lambda x.(x := 1 \,||\, x := 0); !x)) \Downarrow$*. We will discuss this in the next section.*

3.5 Soundness, Adequacy, and Full Abstraction

Ex. 14 suggests that incomplete plays do not matter, and indeed this intuition is correct.

There are two strategies for com, the stuck strategy $\sigma_\bot = \mathrm{strat}(q)$ and the un-stuck strategy $\top = \mathrm{strat}(qa)$. We can order strategies using the following test:

Definition 21 (Preorder)**.** *For any strategies* $\sigma_i : A$*,*

$$\sigma_1 \leq \sigma_2 \text{ if and only if } \forall \sigma : A \to com.\ \text{if } \sigma_1; \sigma = \top \text{ then } \sigma_2; \sigma = \top.$$

Lemma 5. $\leq$ *is a preorder.*

The relation is $\leq$ is a pre-order because it is not symmetric, i.e. $\sigma_1 \leq \sigma_2$ and $\sigma_2 \leq \sigma_1$ does not imply $\sigma_1 = \sigma_2$. This means that there are strategies which are distinct but the difference cannot be observed by testing it by composition! This is generally not good, because rather then comparing strategies directly we need to quotient them by the equivalence $\sigma_1 \sim \sigma_2 = \sigma_1 \leq \sigma_2 \wedge \sigma_2 \leq \sigma_1$. In some sense this undermines our aim to replace an equivalence (contextual equality) by an equality, for ease of reasoning. The quotient is not quite as bad as forcing observational equivalence to be a congruence by quantifying over all contexts, but can we do better? It turns out that in ICA we can!

Let us denote by $comp(\sigma)$ the set of *complete* plays in the strategy σ, that is, those plays in which the opening question is answered, so that by strict nesting all questions are answered.

Theorem 12 (Characterisation)**.** *Let* $\sigma_1, \sigma_2 : A$ *be strategies.* $\sigma_1 \leq \sigma_2$ *if and only if* $comp(\sigma_1) \subseteq comp(\sigma_2)$*. Consequently,* $\sigma_1 \sim \sigma_2$ *if and only if* $comp(\sigma_1) = comp(\sigma_2)$*.*

This means that our game model truly is a denotational model, since we can use semantic equality as we wanted all along.

Let us write $\lambda\overline{x} = \lambda x_0 \cdots \lambda x_k$. Let us interpret a state s as a strategy $[\![s]\!]$ consisting of suitably initialised stateful variable plays and correctly synchronised locks, depending on the location type, so that for any configuration M, s $[\![M]\!]; [\![s]\!] = [\![\overline{new}(\lambda\overline{x}.M)]\!]$ for the appropriately chosen and initialised binders $newvar, newsem$.

The main soundness result is

Theorem 13 (Soundness). *For any term* $\Sigma \vdash M : \beta$*,*

- *If* $\Sigma \vdash M, s \longrightarrow M', s'$ *then* $[\![\lambda\overline{x}.M]\!]; [\![s]\!] \subseteq [\![\lambda\overline{x}.M']\!]; [\![s]\!]'$.
- *If* $\Sigma \vdash M, s \longrightarrow^* M', s'$ *then* $[\![\lambda\overline{x}.M]\!]; [\![s]\!] \subseteq [\![\lambda\overline{x}.M']\!]; [\![s]\!]'$.
- *If* $\Sigma \vdash M, s \Downarrow$ *then* $[\![\lambda\overline{x}.M]\!]; [\![s]\!] = \top$.

Unlike the previous results which have direct elementary proofs, the soundness result requires some intermediate steps. The details are spelled out in Props. 29-31 in *loc. cit.*. The main proof is by induction on the derivation of the $M, s \longrightarrow M', s'$ relation, noting that the only observable actions in $[\![M]\!]$ are state actions (read/write, grab/release) and that any derivation correspond either to no change in these observable actions (the "pure" operations) or precisely one question-answer pair of moves (the "stateful" operations).

The same idea is used in the proof of adequacy:

Theorem 14 (Adequacy). *For any* $\Sigma \vdash M : \beta$ *and state* $s \in States(\sigma)$ *if* $[\![\lambda\overline{x}.M]\!]; [\![\sigma]\!] = \top$ *then* $\Sigma \vdash M, s \Downarrow$.

The difference between adequacy and soundness is that the induction proceeds on the types, using a logical relation. The details are in Props. 34-36 in *loc. cit.*.

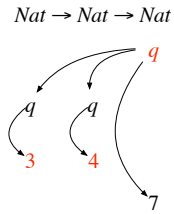
Theorem 15 (Inequational soundness). *For any* $\Gamma \vdash M_i : \theta$*, if* $[\![\Gamma \vdash M_i : \theta]\!] \subseteq [\![\Gamma \vdash M_2 : \theta]\!]$ *then* $\Gamma \vdash M_1 \subseteq_\theta M_2$

This follows from soundness (Thm. 13), monotonicity of composition (Thm. 6) and adequacy (Thm. 14) following a general argument.

Sound and adequate models for ICA can be obtained using other denotational techniques such as functor categories. What sets game models apart is their *definability* property, i.e. there is no "garbage" in the model. Every strategy corresponds to a term. For any play $p : [\![\theta]\!]$ will define an algorithm Θ such that $\Theta(p)$ is a term of type θ such that $[\![\Theta(p) = \mathrm{strat}(p)]\!]$.

The idea of the algorithm is as described below. It will be illustrated with the play $qq_1 3_1 q_2 4_2 7 \in \mathbf{P}_{Nat_1 \to Nat_2 \to Nat}$ as a running example. This is quite obviously the complete play of $[\![\lambda x.\lambda y.x + y]\!]$ when applied to $3, 4$.

- First only consider the justification structure of a play, ignoring the sequencing (O-moves in red):



- Interpret any Q-Q justification as a function call (or thunk evaluation)
- Interpret multiple Q-moves with the same justification as running in parallel
- Interpret any PA-moves as constants.

  At this stage a candidate working solution is the term

$$\lambda x_1 \lambda x_2.(x_1 \,||\, x_2); 7$$

  noting that it doesn't quite type check. Note that the only interesting interaction, from a definability point of view, is the evaluation of x_i and the interleaving of the two plays on x_i. We can force that using an auxiliary variable:

$$\lambda x_1 \lambda x_2.newvar\ 0\ (\lambda y.$$
$$(y := x_1 \,||\, y := x_2); 7)$$

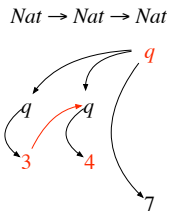
- The next step is to ensure that the only O-moves are 3_1 and 4_2. We can do that using the $test$ function introduced in Sec. 3.4.1, blocking all plays starting with moves we don't want. The candidate term would be something like:

$$\lambda \overline{x}.newvar\ \overline{y} := 0\ in$$
$$(y := x_1; test(x_1 = 3) \,||\, y := x_2; test(x_2 = 4)); 7)$$

except that this would introduce new observable x_i. So we use more auxiliary local variables to cache the value of x:

$$\lambda\overline{x}.newvar\ \overline{y} := 0\ in$$
$$(y_0 := x_1; test(!y_0 = 3)\ ||\ y_1 := x_2; test(!y_1 = 4)); 7$$

- The final step is to enforce the desired schedule. We remember that only P-moves can synchronise on O-moves, so these are the only synchronisations we even attempt. Also, we only need to identify those synchronisations which are not already implied by justification structure or the legal-play constraints (fork/join). We represent them as a red arrow in our running example:



We do that by using local semaphores, grabbed at the point of the O-move and released at the point of the P-move:

$$\lambda\overline{x}.newvar\ \overline{y} := 0\ in.newsem\ \overline{z} := 0\ in$$
$$(y_0 := x_1; grab(z_0); test(!y_0 = 3)\ ||\ release(z_0); y_1 := x_2; test(!y_1 = 4)); 7$$

The detailed algorithm is a little more intricate than that, but the basic idea is just as above. Create un-scheduled, parallel plays, then restrict using testing and synchronisation. It is worth noting that the "angelic" notion of termination, also known as "may-testing" we employ significantly simplifies the task of re-constructing a desired play.

Using the choice operator introduced in Sec. 3.4.1 we can then show that

Theorem 16 (Definability). *For any finite strategy $\sigma : [\![\theta]\!]$ there exists a closed term $M : \theta$ such that $[\![M]\!] = \sigma$.*

The main result equating operational and game-semantic (in)equational reasoning

Theorem 17 (Full abstraction). *For any* $\Gamma \vdash M, N : \theta$*,*

$$\Gamma \vdash M_1 \sqsubseteq_\theta M_2 \textit{ if and only if } [\![\Gamma \vdash M_1 : \theta]\!] \subseteq [\![\Gamma \vdash M_2 : \theta]\!].$$

The left-to-right direction is inequational soundness (Thm. 15)

The right-to-left direction is an immediate and general consequence of definability (Thm. 16) and computational adequacy (Thm. 14).

3.5.1 Examples

Now we can finally address equations such as the ones in Ex. 8 by computing the strategies of the two sides

$$x : var \vdash !x - !x \cong 0$$

$[\![0]\!] = \mathrm{strat}(q0)$ but $[\![!x - !x]\!] = \mathrm{strat}(qq'm'q'n'p)$ where $p = m - n$. The two are obviously distinct.

More subtle equivalences such as

$$p : com' \to com'' \vdash newvar\ x := 0\ in\ p(x := !x + 1); test(!x = 0) \cong_{com} p(\Omega)$$

can also be proved by computing the strategies and noticing that $[\![LHS]\!] = [\![RHS]\!] = \mathrm{strat}(qq''a''a)$ i.e. only those p's which do not evaluate their argument can terminate successfully. The reasons of failure on LHS and RHS are different, but the overall behaviour is the same.

3.6 Annotated further reading list

The main reference for this section is the journal paper

> Ghica, Dan R., and Andrzej S. Murawski. "Angelic semantics of fine-grained concurrency." Annals of Pure and Applied Logic 151.2-3 (2008): 89-114.

A more concise version is that published in a conference proceedings:

> Ghica, Dan R., and Andrzej S. Murawski. "Angelic semantics of fine-grained concurrency." International Conference on Foundations of Software Science and Computation Structures. Springer, Berlin, Heidelberg, 2004.

The asynchronous closure conditions were introduced in the context of game semantics by

> Laird, James. "A game semantics of idealized CSP." Electronic Notes in Theoretical Computer Science 45 (2001): 232-257.

and in the general setting of asynchronous processes by

> Jifeng, He, Mark B. Josephs, and C. A. R. Hoare. "A theory of synchrony and asynchrony." Programming Concepts and Methods. Elsevier, 1990.

The soundness and adequacy proof uses a technique quite well illustrated in

> Nygaard, Mikkel, and Glynn Winskel. "HOPLA—a higher-order process language." International Conference on Concurrency Theory. Springer, Berlin, Heidelberg, 2002.

For going beyond may-testing in the context of game semantics, by distinguishing between various kinds of termination:

> Harmer, Russell, and Guy McCusker. "A fully abstract game semantics for finite nondeterminism." LICS. IEEE, 1999.

Non-termination distinguishing between deadlocks and livelocks, in trace semantics is discussed in the context of CSP:

> Brookes, Stephen D., Charles AR Hoare, and Andrew W. Roscoe. "A theory of communicating sequential processes." Journal of the ACM (JACM) 31.3 (1984): 560-599.

In pure languages an "intrinsic quotient" is required in order to compare strategies – this appears in the loc. cit. papers on the full abstraction for PCF.

In this introduction we have not discussed recursion. Because strategy composition is monotonic w.r.t. inclusion, the category of games is "CPO-enriched" and contains, therefore, a standard model of recursion.

There is a wealth of fully abstract game semantic models in the literature, for any conceivable language feature or combination theoreof. The first one, after the original PCF, was that of "Idealised Algol"

> Abramsky, Samson, and Guy McCusker. "Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions." Algol-like languages. Birkhäuser, Boston, MA, 1997. 297-329.

Other notable models:

Abramsky, Samson, Kohei Honda, and Guy McCusker. "A fully abstract game semantics for general references." Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on. IEEE, 1998.

Laird, James. "A fully abstract game semantics of local exceptions." LICS. IEEE, 2001.

Danos, Vincent, and Russell S. Harmer. "Probabilistic game semantics." ACM Transactions on Computational Logic (TOCL) 3.3 (2002): 359-382.

Honda, Kohei, and Nobuko Yoshida. "Game-theoretic analysis of call-by-value computation." Theoretical Computer Science 221.1-2 (1999): 393-456.

Abramsky, Samson, Dan R. Ghica, Andrzej S. Murawski, C-HL Ong, and Ian David Bede Stark. "Nominal games and full abstraction for the nu-calculus." In LICS, pp. 150-159. IEEE, 2004.

Finally, there is a long tradition of denotational semantics of Algol-like languages

O'Hearn, Peter, and Robert Tennent. Algol-like Languages. Springer Science & Business Media, 2013.

3.7 Applications of game semantics

The concrete presentation of GS makes it more immediately applicable than the conventional sets-and-functions DSs. The first paper that noticed that GS models can be concretely presented was:

Ghica, Dan R., and Guy McCusker. "Reasoning about Idealized Algol using regular languages." International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 2000.

Model checking. The paper above opened the door to algorithmic reasoning about program equivalence, which has been pursued both practically in terms of building tools

Abramsky, S., Ghica, D. R., Murawski, A. S., & Ong, C. H. L. (2004, March). Applying game semantics to compositional software modeling and verification. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 421-435). Springer, Berlin, Heidelberg.

and studying more advanced PL fragments for decidability and complexity properties, e.g.

Ong, C. H. (2002). Observational equivalence of 3rd-order Idealized Algol is decidable. In Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on (pp. 245-256). IEEE.

Hopkins, D., Murawski, A. S., & Ong, C. H. L. (2011, July). A fragment of ML decidable by visibly pushdown automata. In International Colloquium on Automata, Languages, and Programming (pp. 149-161). Springer, Berlin, Heidelberg.

Games-based model checking is very similar to conventional model checking, and the usual tricks of lazy model construction or counterexample-guided refinement apply:

Bakewell, A., & Ghica, D. R. (2008, March). On-the-fly techniques for game-based software model checking. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 78-92). Springer, Berlin, Heidelberg.

Dimovski, A., Ghica, D. R., & Lazić, R. (2006, March). A counterexample-guided refinement tool for open procedural programs. In International SPIN Workshop on Model Checking of Software (pp. 288-292). Springer, Berlin, Heidelberg.

Note that the game-semantic paradigm allows the development of new abstraction methods for model checking:

Ghica, Dan R., and Adam Bakewell. "Clipping: A semantics-directed syntactic approximation." In 2009 24th Annual IEEE Symposium on Logic In Computer Science, pp. 189-198. IEEE, 2009.

High-level synthesis. The same paper (ICALP 2000) suggested that indeed via game-models certain terms can be represented as finite-state machines, which can be in turn compiled into digital circuits, such as FPGAs, a process called *The Geometry of Synthesis* (GOS):

Ghica, Dan R. "Geometry of synthesis: a structured approach to VLSI design." In ACM SIGPLAN Notices, vol. 42, no. 1, pp. 363-375. ACM, 2007.

Ghica, Dan R., and Alex Smith. "Geometry of Synthesis II: From games to delay-insensitive circuits." Electronic Notes in Theoretical Computer Science 265 (2010): 301-324.

Ghica, Dan R., and Alex Smith. "Geometry of synthesis III: resource management through type inference." In ACM SIGPLAN Notices, vol. 46, no. 1, pp. 345-356. ACM, 2011.

Ghica, Dan R., Alex Smith, and Satnam Singh. "Geometry of synthesis IV: compiling affine

recursion into static hardware." In ACM SIGPLAN Notices, vol. 46, no. 9, pp. 221-233. ACM, 2011.

To date, GOS remains the only practical method that can compile higher-order, recursive languages with effects into FPGA fabrics.

Heterogeneous compilation. Moreover, the fact that the same denotational model is used as an intermediate composition protocol allows the seamless composition of software, independent of syntax and the underlying architecture. This argument is made initially in

Ghica, Dan R. "Function interface models for hardware compilation." In Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on, pp. 131-142. IEEE, 2011.

A games-based compiler for the heterogeneous (CPU and FPGA) Zinq architecture, supporting two high-level languages (C and Verity[8].) is described in

Fleming, Shane T., Ivan Beretta, David B. Thomas, George A. Constantinides, and Dan R. Ghica. "PushPush: Seamless integration of hardware and software objects via function calls over AXI." In Field Programmable Logic and Applications (FPL), 2015 25th International Conference on, pp. 1-8. IEEE, 2015.

Fleming, Shane, David Thomas, George Constantinides, and Dan Ghica. "System-level linking of synthesised hardware and compiled software using a higher-order type system." In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 214-217. ACM, 2015.

Thomas, David B., Shane T. Fleming, George A. Constantinides, and Dan R. Ghica. "Transparent linking of compiled software and synthesized hardware." In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pp. 1084-1089. EDA Consortium, 2015.

[8] A modernised Algol

4 Trace semantics, another interaction semantics

The game semantic model of the previous section was based on the so-called "principles of polite conversation", in which P, and most importantly O, behave according to pre-determined rules. What determines the rules? As we have seen, they come from an analysis of the operational semantics, which is a *syntax-directed* specification of the behaviour of programs.

However, defining semantics from the syntax (only) may be too great an idealisation from a practical point of view: most real languages are not syntactically-closed. Real languages can interact with non-syntactic contexts in several ways. The first one is via separate compilation and linking. When modules are linked, the linker assumes that the object code behaves *as if* it is compiled in the same language, but that is not the same thing as being actually compiled in the same language. The journey from source code to object code is not necessarily reversible due to optimisations in the back end of a compiler. The second one is via *foreign function interfaces* (FFI), which allow code to link explicitly against code written in a different language, usually C taken as a lowest common denominator. The third one is via *extrinsic functions*, which are additions to the language not implemented in the language itself, conceptually very similar to FFI but more tightly integrated syntactically. Finally, the fourth one is via *system calls* which make binary code interact with other binary code at run-time. We will refer to all of the above, generically, as "FFI."

In all these situation we can use the game semantics of the language as a specification for defining a syntax-independent interaction protocol between a language and an FFI semantically. However, what if the environment does not conform to the specification? What if O plays impolitely? Or, indeed, what if O plays maliciously?

Consider the code below, in a generic C-like syntax:

```
int read ();
int foo () {
  int s = new ();
  int k = new ();
  int x = read ();
  if (*x == *k) then return *s else return *k; }
```

We have local variables, s holding a "secret location" and k holding a "private location". We use the non-local, system-provided, function *read* to obtain a name from the system, which cannot be that stored at s or k. A value is read into x using untrusted system call *read*. Can the secrecy of s be violated by

making the name stored into it public, i.e. "leaking" it? Even if the test `*x == *k` should always fail because, isn't that right?, the contents of k can never be guessed?

The point of attack is the procedure *read* and it involves the misbehaviour of O. It may not be syntactically realisable but it is computationally feasible. Perhaps the simplest attack would be realised by cloning the configuration by running the program in a virtual machine, pausing at the point of *read*, running the first copy to obtain k, then feeding k into the second copy to finally obtain s.

Therefore in this section we will make a radical change of perspective, with O being treated as an *adversary* rather than as a partner. In the tradition of security analysis we will only constrain O by its knowledge rather than by its actions, the omnipotent but not omniscient Dolev-Yao model. This adversary will collect all the information at its disposal and use it arbitrarily. Our only weapon is to hide information from it, to have secrets.

The unomniscience of O means that the language is not entirely chaotic! For example, the following three programs are equivalent:

```
export f; import g; int f() {int x; g(); return x;}
```

```
export f; import g; int x; int f() {g(); return x;}
```

```
export f; import g; decl f() {g(); return 0;}
```

In both first and second case, the untrusted, external function g cannot access the local variable or the private module variable x, which ensures that its default value 0 is reliably returned by f.

An important observation must be made here emphatically. The equivalence above must not be interpreted as being preserved by all C compilers but rather that it is possible for a C compiler to be implemented such that this equivalence is preserved. Indeed, a tamper-proof compiler in which variables are "hidden" via memory layout randomisation can make x arbitrarily difficult to guess, at the expense of using more memory. In contrast, conventional C compilers will not preserve this equivalence if they use simple deterministic stack allocation, since g can reach through the stack.

We will illustrate this approach with a simple C-like language, noting that this approach can be extended to any concrete version of C.

$$\begin{aligned} \mathit{Prog} &::= \mathit{Mod}^* \\ \mathit{Hdr} &::= \texttt{export}\ \overline{x}; \texttt{import}\ \overline{x}; \end{aligned}$$

$$Mod \ ::= Hdr \ Dcl$$

$$Dcl \ ::= \texttt{decl}\ x = n;\ Dcl \mid \texttt{decl}\ Func;\ Dcl \mid \epsilon$$

A program is a list of modules, consisting of a header and a list of function declarations. The header *Hdr* is a list of names exported and imported by the program, with x an identifier (or list of identifiers $\overline{x}$) taken from an infinite set $\mathcal{N}$ of *names*, and $n \in \mathbb{Z}$.

As in C, functions are available only in global scope and in uncurried form:

$$Func ::= x(\overline{x})\{\texttt{local}\ \overline{x};\ Stm\ \texttt{return}\ Exp;\}$$

A function has a name and a list of arguments. In the body of the function we have a list of local variable declarations followed by a list of statements terminated by a return statement. We define statements and expressions as follows (with $n \in \mathbb{Z}$).

$$Stm ::= \epsilon \mid \texttt{if}(Exp)\texttt{then}\{Stm\}\texttt{else}\{Stm\}; Stm \mid Exp{=}Exp; Stm \mid Exp(Exp^*); Stm$$

$$Exp ::= Exp \star Exp \mid *Exp \mid Exp(Exp^*) \mid (Exp, Exp) \mid \texttt{new}() \mid n \mid x$$

Statements are branching, assignment and function call. For simplicity, iteration is not included as we allow recursive calls. Expressions are arithmetic and logical operators, variable dereferencing ($*$), pairing, variable allocation and integer and variable constants. A function call can be either an expression or a statement. Because the language is type-free the distinction between statement and expression is arbitrary and only used for convenience.

If $\texttt{decl}\ f(\overline{x})\{e\}$ is a declaration in module M we define $f \mathbin{@} M = e[\overline{x}]$, interpreted as "the definition of f in M is e, with arguments $\overline{x}$."

We will give a Felleisen-style operational semantics for this language.

A *frame* is given by the grammar below, with $op \in \{=, \star, ;\}$, $op' \in \{*, -\}$.

$$t ::= \texttt{if}\ (\square)\ \texttt{then}\ \{e\}\ \texttt{else}\ \{e\} \mid \square\ op\ e \mid v\ op\ \square \mid op'\ \square \mid \square\ e \mid v\ \square \mid (\square, e) \mid (v, \square)$$

We denote the "hole" of the frame by $\square$. We denote by $\mathcal{F}s$ the set of lists of frames, the *frame stacks*. By v we denote *values*, defined below.

Let $\mathcal{N} = \mathcal{N}_\lambda \uplus \mathcal{N}_\phi \uplus \mathcal{N}_\kappa$ be a many-sorted set of names, where each of the three components is a

countably infinite set of *location names*, *function names* and *function continuation names* respectively. We range over names by a, b, etc. Specifically for function names we may use f, etc.; and for continuation names k, etc. For each set of names $\mathcal{X}$ we write $\lambda(\mathcal{X})$, $\phi(\mathcal{X})$ and $\kappa(\mathcal{X})$ for its restriction to location, function and continuation names respectively.

Definition 22 (Store)**.** *A store is defined as a pair of partial functions with finite domain:*

$$s \in Sto = (\mathcal{N}_\lambda \rightharpoonup_{\mathsf{fn}} (\mathbb{Z} \uplus \mathcal{N}_\lambda \uplus \mathcal{N}_\phi)) \times (\mathcal{N}_\kappa \rightharpoonup_{\mathsf{fn}} \mathcal{F}s \times \mathcal{N}_\kappa).$$

The first component of the store assigns integer values (data), other locations (pointers) or function names (pointers to functions) to locations. The second stores *continuations*, used by the system to resume a suspended function call.

We write $\lambda(s)$, $\kappa(s)$ for the two projections of a store s. By abuse of notation, we may write $s(a)$ instead of $\lambda(s)(a)$ or $\kappa(s)(a)$. Since names are sorted, this is unambiguous. The support $\nu(s)$ of s is the set of names appearing in its domain or value set. For all stores s, s' and set of names $\mathcal{X}$, we use the notations:

(restrict-to) the sub-store of s defined on $\mathcal{X}$: $s \restriction \mathcal{X} = \{(a, y) \in s \mid a \in \mathcal{X}\}$;

(restrict-from) the sub-store of s that is not defined on $\mathcal{X}$: $s \setminus \mathcal{X} = s \restriction (\mathsf{dom}(s) \backslash \mathcal{X})$;

(update) change the values in s: $s[a \mapsto x] = \{(a, x)\} \cup (s \setminus \{a\})$ and, more generally: $s[s'] = s' \cup (s \setminus \mathsf{dom}(s'))$;

(extension) $s \sqsubseteq s'$ if $\mathsf{dom}(s) \subseteq \mathsf{dom}(s')$;

(closure) $Cl(s, \mathcal{X})$ is the least set of names containing $\mathcal{X}$ and all names reachable from $\mathcal{X}$ through s in a transitively closed manner, i.e. $\mathcal{X} \subseteq Cl(s, \mathcal{X})$ and if $(a, y) \in s$ with $a \in Cl(s, \mathcal{X})$ then $\nu(y) \in Cl(s, \mathcal{X})$.

We define a *value* to be a name, an integer, or a tuple of values: $v ::= () \mid a \mid n \mid (v, v)$. The value $()$ is the unit for the tuple operation. Tupling is associative and for simplicity we identify tuples up to associativity and unit isomorphisms, so $(v, (v, v)) = ((v, v), v) = (v, v, v)$ and $(v, ()) = v$, etc.

We give a semantics for the language using a frame-stack abstract machine. It is convenient to take *identifiers* to be *names*, as it gives a simple way to handle pointers to functions in a way much like that of the C language. The *Program configurations* of the abstract machine are of the form:

$$\langle N \mid P \vdash s, t, e, k \rangle \in \mathcal{N} \times \mathcal{N} \times Sto \times \mathcal{F}s \times Exp \times \mathcal{N}_\kappa$$

Case $e = v$ is a value.

$$\langle N \mid P \vdash s, t \circ (\texttt{if}\ (\Box)\ \texttt{then}\ \{e_1\}\ \texttt{else}\ \{e_2\}), v, k\rangle \longrightarrow \langle N \mid P \vdash s, t, e_1, k\rangle, \text{ if } v \in \mathbb{Z} \setminus \{0\}$$
$$\langle N \mid P \vdash s, t \circ (\texttt{if}\ (\Box)\ \texttt{then}\ \{e_1\}\ \texttt{else}\ \{e_2\}), v, k\rangle \longrightarrow \langle N \mid P \vdash s, t, e_2, k\rangle, \text{ if } v = 0$$
$$\langle N \mid P \vdash s, t \circ (\Box\ op\ e), v, k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ (v\ op\ \Box), e, k\rangle \text{ for } op \in \{=, \star, ;\}$$
$$\langle N \mid P \vdash s, t \circ (v \star \Box), v', k\rangle \longrightarrow \langle N \mid P \vdash s, t, v'', k\rangle, \text{ and } v'' = v \star v'$$
$$\langle N \mid P \vdash s, t \circ (v; \Box), v', k\rangle \longrightarrow \langle N \mid P \vdash s, t, v', k\rangle$$
$$\langle N \mid P \vdash s, t \circ (a = \Box), v, k\rangle \longrightarrow \langle N \mid P \vdash s[a \mapsto v], t, (), k\rangle$$
$$\langle N \mid P \vdash s, t \circ (*\Box), v, k\rangle \longrightarrow \langle N \mid P \vdash s, t, s(v), k\rangle$$
$$\langle N \mid P \vdash s, t \circ (\Box; e), \texttt{local}\ x, k\rangle \longrightarrow \langle N \cup \{a\} \mid P \vdash s[a \mapsto 0], t, e[a/x], k\rangle, \text{ if } a \notin N$$
$$\langle N \mid P \vdash s, t \circ (\Box(e)), v, k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ (v(\Box)), e, k\rangle$$
$$\langle N \mid P \vdash s, t \circ ((\Box, e)), v, k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ ((v, \Box)), e, k\rangle$$
$$\langle N \mid P \vdash s, t \circ ((v, \Box)), v', k\rangle \longrightarrow \langle N \mid P \vdash s, t, (v, v'), k\rangle$$
$$\langle N \mid P \vdash s, t \circ (f(\Box)), v', k\rangle \longrightarrow \langle N \mid P \vdash s, t, e[v'/\overline{x}], k\rangle, \text{ if } f \mathbin{@} M = e[\overline{x}] \qquad \text{(F)}$$

Case e is not a value.

$$\langle N \mid P \vdash s, t, \texttt{if}\ (e)\ \texttt{then}\ \{e_1\}\ \texttt{else}\ \{e_2\}, k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ (\texttt{if}\ (\Box)\ \texttt{then}\ \{e_1\}\ \texttt{else}\ \{e_2\}), e, k\rangle$$
$$\langle N \mid P \vdash s, t, e\ op\ e', k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ (\Box\ op\ e'), e, k\rangle, \text{if } op \in \{=, \star, ;\}$$
$$\langle N \mid P \vdash s, t, *e, k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ (*\Box), e, k\rangle$$
$$\langle N \mid P \vdash s, t, \texttt{return}(e), k\rangle \longrightarrow \langle N \mid P \vdash s, t, e, k\rangle$$
$$\langle N \mid P \vdash s, t, \texttt{new}(), k\rangle \longrightarrow \langle N \cup \{a\} \mid P \vdash s[a \mapsto 0], t, a, k\rangle, \text{if } a \in \mathcal{N}_\lambda \setminus N$$
$$\langle N \mid P \vdash s, t, e(e'), k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ (\Box(e')), e, k\rangle$$
$$\langle N \mid P \vdash s, t, (e, e'), k\rangle \longrightarrow \langle N \mid P \vdash s, t \circ ((\Box, e')), e, k\rangle$$

Figure 1: Operational semantics

N is a set of *used names*; $P \subseteq N$ is the set of *public names*; s is the *program state*; t is a list of frames called the *frame stack*; e is the expression being evaluated; and k is a *continuation name*, which for now will stay unchanged.

The transitions of the abstract machine are a relation on the set of configurations. They are defined by case analysis on the structure of e then t in a standard fashion, as in Fig. 1. Branching is as in C, identifying non-zero values with true and zero with false. Binary operators are evaluated left-to-right, also as in C. Arithmetic and logic operators ($\star$) have the obvious evaluation. Dereferencing is given the usual evaluation, with a note that in order for the rule to apply it is implied that v is a location and $s(v)$ is defined. Local-variable allocation extends the domain of s with a fresh secret name. Local variables are created fresh, locally for the scope of a function body. The `new()` operator allocates a secret and fresh location name, initialises it to zero and returns its location. The return statement is used as a syntactic marker for an end of function but it has no semantic role.

Structural rules, such as function application and tuples are as usual in call-by-value languages, i.e. left-to-right. Function call also has a standard evaluation. The body of the function replaces the function call

and its formal arguments $\overline{x}$ are substituted by the tuple of arguments v' in point-wise fashion. Finally, non-canonical forms also have standard left-to-right evaluations.

4.1 Trace semantics

The conventional function-call rule (F) is only applicable if there is a function definition in the module. If the name used for the call is not the name of a known function then the normal operational semantics rules no longer apply. We now extend our semantics so that calls and returns of locally undefined functions become a mechanism for interaction between the program and the ambient system. We call the resulting semantics the *System Level (Trace) Semantics (SLS)*. The name emphasises the intuition that now O represents a (run-time) system rather than a (syntactic) context.

Given a module M we will write as $[\![M]\!]$ the transition system defining its SLS. Its states are $\mathcal{S}[\![M]\!] = \mathit{Sys}[\![M]\!] \cup \mathit{Prog}[\![M]\!]$, where $\mathit{Prog}[\![M]\!]$ is the set of abstract-machine configurations of the previous section and $\mathit{Sys}[\![M]\!]$ is the set of system configurations, which are of the form:

$$\langle\!\langle N \mid P \vdash s \rangle\!\rangle \ \in \ \mathcal{N} \times \mathcal{N} \times \mathit{Sto}.$$

The SLS is defined at the level of modules, that is programs with missing functions, similarly to what is usually deemed a *compilation unit* in most programming languages. The transition relation $\delta[\![M]\!]$ of the SLS operates on a set of labels $\mathcal{L} \uplus \{\epsilon\}$ and is of the following type.

$$\delta[\![M]\!] \ \subseteq (\mathit{Prog}[\![M]\!] \times \{\epsilon\} \times \mathit{Prog}[\![M]\!]) \cup (\mathit{Prog}[\![M]\!] \times \mathcal{L} \times \mathit{Sys}[\![M]\!]) \cup (\mathit{Sys}[\![M]\!] \times \mathcal{L} \times \mathit{Prog}[\![M]\!])$$

$$\begin{aligned} \mathcal{L} \ = \ & \{(s, \texttt{call}\, f, v, k) \mid s \in \mathit{Sto},\ \kappa(s) = \emptyset,\ f \in \mathcal{N}_\lambda,\ k \in \mathcal{N}_\kappa,\ v \text{ a value}\} \\ & \cup \{(s, \texttt{ret}\, v, k) \mid s \in \mathit{Sto},\ \kappa(s) = \emptyset,\ k \in \mathcal{N}_\kappa,\ v \text{ a value}\} \end{aligned}$$

Thus, at the system level, program and system configurations may call and return functions in alternation, in very much the same way that P and O make moves in game semantics. We write $X \xrightarrow[s]{\alpha} X'$ for $(X, (s, \alpha), X') \in \delta[\![M]\!]$, and $X \to X'$ for $(X, \epsilon, X') \in \delta[\![M]\!]$.

In transferring control between Program and System the continuation pointers ensure that upon return the right execution context can be recovered. We impose several hygiene conditions on how continuations can be used. We distinguish between P- and S-continuation names. The former are created by the Program and stored for subsequent use, when a function returns. The latter are created by the

System and are not stored. The reason for this distinction is both technical and intuitive. Technically it will simplify proving that composition is well-defined. Mixing S and P continuations would not create any interesting behaviour: if P receives a continuation it does not know then the abstract machine of P cannot evaluate it, which can be interpreted as a crash. But S always has ample opportunities to crash the execution, so allowing it seems uninteresting. However, this is in some sense a design decision and an alternative semantics, with slightly different properties, can be allowed to mix S and P continuations in a promiscuous way.

The first new rule, called **Program-to-System call** is:

$$\langle N \mid P \vdash s, t \circ (f(\square)), v, k\rangle \xrightarrow[s\restriction\lambda(P')]{\texttt{call}\, f,v,k'} \langle\!\langle N \cup \{k'\} \mid P' \cup \{k'\} \vdash s[k' \mapsto (t,k)]\rangle\!\rangle$$

if $f \mathbin{@} M$ not defined, $k' \notin N, P' = Cl(s, P \cup \nu(v))$

When a non-local function is called, control is transferred to the system. In game semantics this corresponds to a *Proponent question*, and is an observable action. Following it, all the names that can be transitively reached from public names in the store also become public, so it gives both control and information to the System. Its observability is marked by a label on the transition arrow, which includes: a tag `call`, indicating that a function is called, the name of the function (f), its arguments (v) and a fresh continuation (k'), which stores the code pointer; the transition also marks that part of the store which is observable because it uses publicly known names.

The counterpart rule is the **System-to-Program return**, corresponding to a return from a non-local function.

$$\langle\!\langle N \mid P \vdash s\rangle\!\rangle \xrightarrow[s']{\texttt{ret}\, v,k'} \langle N \cup \nu(v,s') \mid P \cup \nu(v,s') \vdash s[s'], f, v, k\rangle$$

if $s(k') = (f,k), \nu(v,s') \cap N \subseteq P, \lambda(\nu(v)) \subseteq \nu(s'), s \restriction \lambda(P) \sqsubseteq s'$

This is akin to the game-semantic *Opponent answer*. Operationally it corresponds to S returning from a function. Note here that the only constraints on what S can do in this situation are *epistemic,* i.e. determined by what it *knows*:

1. it can return with any value v so long as it only contains public names or fresh names (but not *private* ones);
2. it can update any public location with any value;
3. it can return to any (public) continuation k'.

However, the part of the store which is private (i.e. with domain in $N \setminus P$) cannot be modified by S. So S has no restrictions over what it can do with known names and to known names, but it cannot guess

private names. Therefore it cannot do anything with or to names it does not know. The restriction on the continuation are just hygienic, as explained earlier.

There are two converse transfer rules **System-to-Program call** and **Program-to-System return**, corresponding to the program returning and the system initiating a function call:

$\langle\!\langle N \mid P \vdash s \rangle\!\rangle \xrightarrow[s']{\texttt{call}\, f,v,k} \langle N \cup \{k\} \cup \nu(v,s') \mid P \cup \{k\} \cup \nu(v,s') \vdash s[s'], f(\square), v, k \rangle$ if $f@M$ defined, $k \notin \mathsf{dom}(s), \nu(f,v,s') \cap N \subseteq P, \lambda(\nu(v)) \subseteq \nu(s'), s{\restriction}\lambda(P) \sqsubseteq s'$
$\langle N \mid P \vdash s, -, v, k \rangle \xrightarrow[s{\restriction}\lambda(P')]{\texttt{ret}\, v,k} \langle\!\langle N \mid P' \vdash s \rangle\!\rangle \quad$ where $P' = Cl(s, P \cup \nu(v))$

In the case of the S-P call it is S which calls a publicly-named function from the module. As in the case of the return, the only constraint is that the function f, arguments v and the state update s' only involve public or fresh names. The hygiene conditions on the continuations impose that no continuation names are stored, for reasons already explained. Finally, the P-S return represents the action of the program yielding a final result to the system following a function call. The names used in constructing the return value are disclosed and the public part of the store is observed. In analogy with game semantics the function return is a *Proponent answer* while the system call is an *Opponent question*.

The *initial configuration* of the SLS for module M is $S^0_M = \langle\!\langle N_0 \mid P_0 \vdash s_0 \rangle\!\rangle$. It contains a store s_0 where all variables are initialised to the value specified in the declaration. The set N_0 contains all the exported and imported names, all declared variables and functions. The set P_0 contains all exported and imported names. When M is not clear from the context, we may write P^0_M for P_0, etc.

4.2 Compositionality

The SLS of a module M gives us an interpretation $[\![M]\!]$ which is modular and effective (i.e. it can be executed) so no consideration of the context is required in formulating properties of modules based on their SLS. Technically, we can reason about SLS using standard tools for transition systems such as trace equivalence, bisimulation or Hennessy-Milner logic.

We first show that the SLS is consistent by proving a *compositionality* property. SLS interpretations of modules can be composed semantically in a way that is consistent with syntactic composition. Syntactic composition for modules is concatenation with renaming of un-exported function and variable names to prevent clashes, which we will denote by using $- \cdot -$. In particular, we show that we can define a semantic SLS composition $\otimes$ so that, for an appropriate notion of isomorphism in the presence of τ-transitions ($\cong_\tau$), the following holds.

For any modules M, M': $[\![M \cdot M']\!] \cong_\tau [\![M]\!] \otimes [\![M']\!]$.

We call this the **principle of functional composition**. Its interpretation is as a "no cheating" condition: the operational semantics must disclose enough information to the system to allow the system to functionally compose separate modules. The process by functional composition is realised in practice is *linking*. For example, it can be easily seen that if P does not disclose the current continuation to S in an P-S call then S is unable to return from the call.

Let $\mathcal{P}$ range over program configurations, and $\mathcal{S}$ over system configurations. Moreover, assume an extended set of continuation names $\mathcal{N}'_\kappa = \mathcal{N}_\kappa \uplus \mathcal{N}_{\mathsf{aux}}$, where $\mathcal{N}_{\mathsf{aux}}$ is a countably infinite set of fresh auxiliary names. We define semantic composition of modules inductively as in Fig. 2 (all rules have symmetric, omitted counterparts). We use an extra component Π containing those names which have been communicated between either module and the outside system, and we use an auxiliary store s containing values of locations only. The latter records the last known values of location names that are not private to a single module. Continuation names in each Π are assigned Program/System polarities (we write $k \in \Pi_P$ / $k \in \Pi_S$), thus specifying whether a continuation name was introduced by either of the modules or from the outside system. Cross calls and returns are assigned τ-labels and are marked by auxiliary continuation names. We also use the following notations for updates of Π when an interaction with the outside system is made, where we write Pr for the set of private names $\nu(\mathcal{S}, \mathcal{S}') \setminus \Pi$.

- $(\Pi, s')^P[v, k, s] = Cl(s'[s], \nu(v) \cup \Pi) \cup \{k\}$, and assign P polarity to k;
- $(\Pi, s')^S[v, k, s] = \Pi \cup \nu(v, s \setminus Pr) \cup \{k\}$, and assign S polarity to k.

The notations apply also to the case when no continuation name k is included in the update (just disregard k). The semantic composition of modules M and M' is thus given by:

$$[\![M]\!] \otimes [\![M']\!] = [\![M]\!] \otimes_{\Pi_0}^{s_0 \cup s'_0} [\![M']\!]$$

where s_0 is the store assigning initial values to all initial public locations of $[\![M]\!]$, and similarly for s'_0, and Π_0 contains all exported and imported names.

The rules of Fig. 2 feature side-conditions on choice of continuation names,[9] system stores[10] and name privacy. The latter originate from nominal game semantics and they guarantee that the names intro-

[9] That is, auxiliary names are used precisely for cross calls and returns.

[10] These stipulate (rules (vi)-(vii)) that the store produced in each outside system transition must: (a) be defined on all public names (i.e. names in Π), and (b) agree with the old one on all other names. The latter safeguards against the system breaking name privacy.

(i)	$\dfrac{\mathcal{P} \longrightarrow \mathcal{P}'}{\mathcal{P} \otimes^{s}_{\Pi} \mathcal{S} \longrightarrow \mathcal{P}' \otimes^{s}_{\Pi} \mathcal{S}}$	*Internal move* $\nu(\mathcal{P}') \cap \nu(\mathcal{S}) \subseteq \nu(\mathcal{P})$.
(ii)	$\dfrac{\mathcal{P} \xrightarrow[s]{\texttt{call}\, f,v,k} \mathcal{S}' \qquad \mathcal{S} \xrightarrow[s]{\texttt{call}\, f,v,k} \mathcal{P}'}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow{\tau} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi} \mathcal{P}'}$	*Cross-call* $k \in \mathcal{N}_{\mathsf{aux}} \setminus \nu(\mathcal{S})$.
(iii)	$\dfrac{\mathcal{P} \xrightarrow[s]{\texttt{ret}\, v,k} \mathcal{S}' \qquad \mathcal{S} \xrightarrow[s]{\texttt{ret}\, v,k} \mathcal{P}'}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow{\tau} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi} \mathcal{P}'}$	*Cross-return* $k \in \mathcal{N}_{\mathsf{aux}}$.
(iv)	$\dfrac{\mathcal{P} \xrightarrow[s]{\texttt{call}\, f,v,k} \mathcal{S}' \qquad \mathcal{S} \not\xrightarrow[s]{\texttt{call}\, f,v,k}}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow[s'[s]\restriction \Pi']{\texttt{call}\, f,v,k} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}}$	*Program call* $\Pi' = (\Pi, s')^{P}[v, k, s]$ and $k \in \mathcal{N}_{\kappa} \setminus \nu(\mathcal{S})$.
(v)	$\dfrac{\mathcal{P} \xrightarrow[s]{\texttt{ret}\, v,k} \mathcal{S}' \qquad \mathcal{S} \not\xrightarrow[s]{\texttt{ret}\, v,k}}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow[s'[s]\restriction \Pi']{\texttt{ret}\, v,k} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}}$	*Program return* $\Pi' = (\Pi, s')^{P}[v, s]$ and $k \in \mathcal{N}_{\kappa}$.
(vi)	$\dfrac{\mathcal{S} \xrightarrow[s]{\texttt{call}\, f,v,k} \mathcal{P}}{\mathcal{S} \otimes^{s'}_{\Pi} \mathcal{S}' \xrightarrow[s\restriction \Pi']{\texttt{call}\, f,v,k} \mathcal{P} \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}'}$	*System call* $k \in \mathcal{N}_{\kappa} \setminus (\nu(\mathcal{S}') \setminus \Pi_S)$, $\Pi' = (\Pi, s')^{S}[v, k, s]$, $\lambda(\Pi) \subseteq \mathsf{dom}(s)$, $s' \setminus \Pi \subseteq s$ and $\nu(v, s \setminus Pr) \cap Pr = \emptyset$.
(vii)	$\dfrac{\mathcal{S} \xrightarrow[s]{\texttt{ret}\, v,k} \mathcal{P}}{\mathcal{S} \otimes^{s'}_{\Pi} \mathcal{S}' \xrightarrow[s\restriction \Pi']{\texttt{ret}\, v,k} \mathcal{P} \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}'}$	*System return* $k \in \mathcal{N}_{\kappa} \setminus \nu(\mathcal{S}')$, $\Pi' = (\Pi, s')^{S}[v, s]$, $\lambda(\Pi) \subseteq \mathsf{dom}(s)$, $s' \setminus \Pi \subseteq s$ and $\nu(v, s \setminus Pr) \cap Pr = \emptyset$.

Figure 2: Rules for semantic composition

duced (freshly) by M and M' do not overlap (rule (i)), and that the names introduced by the system in the composite module do not overlap with any of the names introduced by M or M' (rules (vi)-(vii)). They safeguard against incorrect name flow during composition.

Let us call the four participants in the composite SLS *Program A*, *System A*, *Program B*, *System B*. Whenever we use X, Y as Program or System names they can be either A or B, but different. Whenever we say *Agent* we mean Program or System. A state of the composite system is a pair (Agent X, Agent Y) noting that they cannot be both Programs. The composite transition rules reflect the following intuitions.

- Rule (i): If Program X makes an internal (operational) transition System Y is not affected.
- Rules (ii)-(iii): If Program X makes a system transition to System X and System Y can match the transition going to Program Y then the composite system makes an internal (τ) transition. This is the most important rule and it is akin to game semantic composition via "synchronisation and hiding". It signifies M making a call (or return) to (from) a function present in M'.

- Rules (iv)-(v): If Program X makes a system transition that cannot be matched by System Y then it is a system transition in the composite system, a non-local call or return.
- Rules (vi)-(vii): From a composite system configuration (both entities are in a system configuration) either Program X or Program Y can become active via a call or return from the system.

We can now formalise functional composition.

Definition 23. *Let $\mathcal{G}_1, \mathcal{G}_2$ be LTSs corresponding to a semantic composite SLS and an ordinary SLS respectively. A function R from states of $\mathcal{G}_1$ to configurations of $\mathcal{G}_2$ is called a* τ-isomorphism *if it maps the initial state of $\mathcal{G}_1$ to the initial configuration of $\mathcal{G}_2$ and, moreover, for all states X of $\mathcal{G}_1$ and $\ell \in \mathcal{L}$,*

1. *if $X \xrightarrow{\tau} X'$ then $R(X) = R(X')$,*
2. *if $X \to X'$ then $R(X) \to R(X')$,*
3. *if $R(X) \to Y$ and $X \not\xrightarrow{\tau}$ then $X \to X'$ with $R(X') = Y$,*
4. *if $X \xrightarrow{\ell} X'$ then $R(X) \xrightarrow{\ell} R(X')$,*
5. *if $R(X) \xrightarrow{\ell} Y$ and $X \not\xrightarrow{\tau}$ then $X \xrightarrow{\ell} X'$ with $R(X') = Y$.*

We write $\mathcal{G}_1 \cong_\tau \mathcal{G}_2$ if there is a τ-isomorphism $R : \mathcal{G}_1 \to \mathcal{G}_2$.

Theorem 18. *For all modules M, M', $[\![M]\!] \otimes [\![M']\!] \cong_\tau [\![M \cdot M']\!]$.*

The principle of functional composition holds and thus linking modules is possible.

As a final comment, the compositional extension of an operational semantics to a system-level (trace) semantics is something that should always be attempted. A conventional, closed-system operational semantics has no support for FFI, an essential feature of any realistic language. The SLS makes an informational rather than simply behavioural specification of the language, as in what names of things are public and what names of things are to be kept secret. The preservation of secrecy is the mechanism by which equational properties can still hold. If all names are public the System can break any equivalence. But the secrecy mechanism will always come at a cost, because names must be hidden in one way or another (encryption, randomisation). The SLS makes this important trade-off explicit and specifies it mathematically.

The epistemically-constrained system-level semantics gives a security-flavoured semantics for the programming language which is reflected by its logical properties and by the notion of equivalence it gives rise to.

We will see that certain properties of traces in the SLS of a module correspond to "secrecy violations", i.e. undesirable disclosures of names that are meant to stay secret. In such traces it is reasonable to refer to

the System as an *attacker* and consider its actions an attack. We will see that although the attack cannot be realised within the given language it can be enacted in a realistic system by system-level actions.

We will also see that certain equivalences that are known to hold in conventional semantics still hold in a system-level model. This means that even in the presence of an omnipotent attacker, unconstrained by a prescribed set of language constructs, the epistemic restrictions can prevent certain observations, not only by the programming context but by any ambient computational system. This is a very powerful notion of equivalence which embodies *tamper-resistance* for a module.

Note that we chose these examples to illustrate the conceptual interest of the SLS-induced properties rather than as an illustration of the mathematical power of SLS-based reasoning techniques. For this reason, the examples are as simple and clear as possible.

4.3 A system-level attack: violating secrecy

Let us now formally revisit the earlier example, inspired by a flawed security protocol informally described as follows.

> Consider a secret, a locally generated key and an item of data read from the environment. If the local key and the input data are equal then output the secret, otherwise output the local key.

In a conventional process-calculus syntax the protocol can be written as

$$\nu s \nu k.\mathsf{in}(a).\mathsf{if}\ k{=}a\ \mathsf{then}\ \mathsf{out}(s)\ \mathsf{else}\ \mathsf{out}(k).$$

It is true that the secret s is not leaked because the local k cannot be known as it is disclosed only at the very end. This can be proved using bisimulation-based techniques for anonymity. Let us consider an implementation of the protocol:

```
export prot;
import read;
decl prot( ) {
  local s, k, x; s = new(); k = new(); x = read();
  if (*x == *k) then *s else *k}
```

We have local variables `s` holding the "secret location" and `k` holding the "private location". We use the non-local, system-provided, function `read` to obtain a name from the system, which cannot be that

$$\langle\!\langle N_0 \mid P_0 \vdash \emptyset \rangle\!\rangle \xrightarrow[\emptyset]{\texttt{call prot}\,(),k} \langle N_0, k \mid P_0, k \vdash \emptyset, -, E, k\rangle$$
$$\longrightarrow^* \langle N_1, k, a_0, a_1 \mid P_0, k \vdash (\texttt{s} \mapsto a_0, p \mapsto a_1, \texttt{x} \mapsto 0),$$
$$(\Box; \texttt{if}(*x == *p)\ \texttt{then}\ *s\ \texttt{else}\ *p) \circ (\texttt{x} = \Box) \circ (\texttt{read}(\Box)), (), k\rangle$$
$$\xrightarrow[\emptyset]{\texttt{call read}\,(),k'} \langle\!\langle N_1, k, k', a_0, a_1 \mid P_1 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto 0, k' \mapsto (t, k))\rangle\!\rangle$$
$$\xrightarrow[\emptyset]{\texttt{ret}\ a_2,k'} \langle N_2 \mid P_1, a_2 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto 0, k' \mapsto (t, k)), t, a_2, k\rangle$$
$$\longrightarrow^* \langle N_2 \mid P_1, a_2 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto a_2, k' \mapsto (t, k)), -, a_1, k\rangle$$
$$\xrightarrow[\emptyset]{\texttt{ret}\ a_1,k} \langle\!\langle N_2 \mid P_2, a_2, a_1 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto a_2, k' \mapsto (t, k))\rangle\!\rangle$$
$$\xrightarrow[\emptyset]{\texttt{ret}\ a_1,k'} \langle N_2 \mid P_1, a_2, a_1 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto a_2, k' \mapsto (t, k)), t, a_1, k\rangle$$
$$\longrightarrow^* \langle N_2 \mid P_1, a_2, a_1 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto a_1, k' \mapsto (t, k)), -, a_0, k\rangle$$
$$\xrightarrow[\emptyset]{\texttt{ret}\ a_0,k} \langle\!\langle N_2 \mid P_2, a_2, a_1, a_0 \vdash (\texttt{s} \mapsto a_0, \texttt{k} \mapsto a_1, \texttt{x} \mapsto a_2, k' \mapsto (t, k))\rangle\!\rangle.$$

Above, $t = (\Box; \texttt{if}(*x == *k)\ \texttt{then}\ *s\ \texttt{else}\ *k) \circ (\texttt{x} = \Box)$, $N_0 = P_0 = \{\texttt{prot}, \texttt{read}\}$, $N_1 = N_0 \cup \{\texttt{s}, \texttt{k}, \texttt{x}\}$, $N_2 = N_1 \cup \{k, k', a_0, a_1, a_2\}$ and $P_1 = P_0 \cup \{k, k'\}$.

Figure 3: Secret a_0 leaks.

stored at `s` or `k`. A value is read into `x` using untrusted system call `read()`. Can the secrecy of `s` be violated by making the name stored into it public? Unlike in the process-calculus model, the answer is "yes".

The initial configuration is $\langle\!\langle$ `prot`, `read` $\mid$ `prot`, `read` $\vdash \emptyset\rangle\!\rangle$. We denote the body of `prot` by E. The transition corresponding to the secret being leaked is shown in Fig. 3. The labelled transitions are the interactions between the program and the system and are interpreted as follows:

1. system calls `prot()` giving continuation k
2. program calls `read()` giving fresh continuation k'
3. system returns (from `read`) using k' and producing fresh name a_2
4. program returns (from `prot`) leaking local name a_1 stored in `k`
5. system uses k' to fake a second return from `read`, using the just-learned name a_1 as a return value
6. with a_1 the program now returns the secret a_0 stored in s to the environment.

Values of a_2 are omitted as they do not affect the transitions.

The critical step is (5), where the system is using a continuation in a presumably illegal, or at least unexpected, way. This attack can be implemented in several ways:

- If the attacker has access to more expressive control commands such as `callcc` then the continuation can simply be replayed.
- If the attacker has low-level access to memory it can clone (copy and store) the continuation k', i.e. the memory pointed at by the name k'. In order to execute the attack it is not required to have an understanding of the actual machine code or byte-code, as the continuation is treated as a black box. This means that the attack cannot be prevented by any techniques reliant on obfuscation of the instruction or address space, such as randomisation.
- If the attacker has access to `fork`-like concurrency primitives then it can exploit them for the attack because such primitives duplicate the thread of execution, creating copies of all memory segments. Note that the behaviour of the conventional Unix `fork` is richer than what we consider in our system model, but it can be readily accommodated in our framework by a configuration-cloning system transition:

$$\langle\!\langle N \mid P \vdash s \rangle\!\rangle \to \langle\!\langle N + N \mid P + P \vdash s[N/\mathrm{inl}(N)] \cup s[N/\mathrm{inr}(N)] \rangle\!\rangle$$

- The attacker's ability to clone the configuration can lead to attacks which are purely systemic, for example executing the program into a virtual machine, pausing execution, cloning the state of the machine, then playing the two copies against each other.

4.4 Equivalence

Functional Compositionality gives an internal consistency check for the semantics. This already shows that our language is "well behaved" from a system-level point of view. In this section we want to further emphasise this point. We can do that by proving that there are nontrivial equivalences which hold. There are many such equivalences we can show, but we will choose a simple but important one, because it embodies a principle of locality for state.

This deceptively simple example establishes the fact that a *local* variable cannot be interfered with by a *non-local* function. This was an interesting example because it highlighted a significant shortcoming of *global state* models of imperative programming. Although not pointed out at the time, functor-category models of state developed roughly at the same time gave a mathematically clean solution for this equivalence, which followed directly from the type structure of the programming language.

We compare SLSs be examining their traces. Formally, the set of traces of module M is given by

$$T(M) = \{(\pi \cdot w) \in \mathcal{L}^* \mid S^0_M \xrightarrow{w} \!\!\!\!\rightarrow\; X,\, \forall a \in P^0_M.\, \pi(a) = a.\}$$

Definition 24. *Let M_1, M_2 be modules with common public names. We say that M_1 and M_2 are* trace equivalent, *written $M_1 \cong M_2$, if $T(M_1) = T(M_2)$.*

The above extends to modules with $P^0_{M_1} \neq P^0_{M_2}$ by explicitly filling in the missing public names on each side. We next introduce a handy notion of bisimilarity which precisely captures trace equivalence. For each configuration X, let us write $P(X)$ for the set of public names of X

Definition 25. *Let $\mathcal{R}$ be a relation between configurations. $\mathcal{R}$ is a* simulation *if, whenever $(X_1, X_2) \in \mathcal{R}$, we have $P(X_1) = P(X_2)$ and also:*

- *$X_1 \to X_1'$ implies $(X_1', X_2) \in \mathcal{R}$;*
- *$X_1 \xrightarrow{\ell} X_1'$ implies $X_2 \twoheadrightarrow X_2''$ with $(\pi \cdot X_2'') \xrightarrow{\ell} X_2'$ and $(X_1', X_2') \in \mathcal{R}$, for some name permutation π such that $\pi(a) = a$ for all $a \in P(X_1)$.*

$\mathcal{R}$ is a bisimulation *if it and its inverse are simulations. Modules M_1 and M_2 are* bisimilar, *written $M_1 \sim M_2$, if there is a bisimulation $\mathcal{R}$ such that $(S^0_{M_1}, S^0_{M_2}) \in \mathcal{R}$.*

Lemma 6. *Bisimilarity coincides with trace equivalence.*

Proposition 1. *Trace equivalence is a congruence for module composition $- \cdot -$.*

It is straightforward to check that the following three programs have bisimilar SLS transition systems:

```
export f; import g; decl f() {local x; g(); return *x;}
export f; import g; decl x; decl f() {g(); return *x;}
export f; import g; decl f() {g(); return 0;}
```

Intuitively, the reason is that in the first two programs `f`-local (module-local, respectively) variable `x` is never visible to non-local function `g`, and will keep its initial value, which it `0`. The bisimulation relation is straightforward as the three LTSs are equal modulo silent transitions and permutation of private names for `x`.

Other equivalences, for example in the style of *parametricity* also hold, with simple proofs of equivalence via bisimulation:

```
export inc, get;
decl x;
decl inc(){x = (*x + 1) % 3;}
decl get() {return *x;}
```

```
export inc, get;
decl x;
decl inc() {x = (*x - 1) % 3;}
decl get() {return -*x;}
```

These two programs, or rather libraries, implement a modulo-3 counter as an abstract data structure, using private hidden state `x`. The environment can increment the counter (`inc`) or read its value (`get`) but nothing else. The first implementation counts up, and the second counts down.

4.5 Annotated further reading

The material presented in this section is closely based on

> Ghica, Dan R., and Nikos Tzevelekos. "A system-level game semantics." Electronic Notes in Theoretical Computer Science 286 (2012): 191-211.

The paper was used in proving the correctness of separate compilation such as

> Stewart, Gordon, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. "Compositional compcert." ACM SIGPLAN Notices 50, no. 1 (2015): 275-287.

Some of the papers that introduce the key ideas of trace semantics for programming languages are:

> Jeffrey, Alan, and Julian Rathke. "A fully abstract may testing semantics for concurrent objects." Theoretical Computer Science 338, no. 1-3 (2005): 17-63.

> Lassen, Soren B., and Paul Blain Levy. "Typed normal form bisimulation." International Workshop on Computer Science Logic. Springer, Berlin, Heidelberg, 2007.

> Laird, James. "A fully abstract trace semantics for general references." International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 2007.

A remarkable language for which the emphasis is on security properties via its trace semantics is presented in

> Jagadeesan, Radha, Corin Pitcher, Julian Rathke, and James Riely. "Local memory via layout randomization." In Computer Security Foundations Symposium (CSF), 2011 IEEE 24th, pp. 161-174. IEEE, 2011.

Finally, a formal connection between game semantics and trace semantics has been studied in

> Jaber, Guilhem. "Operational nominal game semantics." In International Conference on

Foundations of Software Science and Computation Structures, pp. 264-278. Springer, Berlin, Heidelberg, 2015.

5 References

1. Abramsky, S., Ghica, D. R., Murawski, A. S., & Ong, C. H. L. (2004, March). Applying game semantics to compositional software modeling and verification. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 421-435). Springer, Berlin, Heidelberg.

2. Abramsky, Samson, Dan R. Ghica, Andrzej S. Murawski, C-HL Ong, and Ian David Bede Stark. "Nominal games and full abstraction for the nu-calculus." In null, pp. 150-159. IEEE, 2004.

3. Abramsky, Samson, Kohei Honda, and Guy McCusker. "A fully abstract game semantics for general references." Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on. IEEE, 1998.

4. Abramsky, Samson, and Radha Jagadeesan. "Games and full completeness for multiplicative linear logic." The Journal of Symbolic Logic 59.2 (1994): 543-574.

5. Abramsky, Samson, Radha Jagadeesan, and Pasquale Malacaria. "Full abstraction for PCF." Information and Computation 163.2 (2000): 409-470.

6. Abramsky, Samson, and Guy McCusker. "Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions." Algol-like languages. Birkhäuser, Boston, MA, 1997. 297-329.

7. Abramsky, Samson, and Guy McCusker. "Game semantics." Computational logic. Springer, Berlin, Heidelberg, 1999.

8. Bakewell, A., & Ghica, D. R. (2008, March). On-the-fly techniques for game-based software model checking. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 78-92). Springer, Berlin, Heidelberg.

9. Blass, Andreas. "A game semantics for linear logic." Annals of Pure and Applied logic 56.1-3 (1992): 183-220.

10. Brookes, Stephen D., Charles AR Hoare, and Andrew W. Roscoe. "A theory of communicating sequential processes." Journal of the ACM (JACM) 31.3 (1984): 560-599.

11. Brookes, Stephen D., Charles AR Hoare, and Andrew W. Roscoe. "A theory of communicating sequential processes." Journal of the ACM (JACM) 31.3 (1984): 560-599.

12. Danos, Vincent, and Russell Harmer. "The anatomy of innocence." International Workshop on

Computer Science Logic. Springer, Berlin, Heidelberg, 2001.

13. Danos, Vincent, and Russell S. Harmer. "Probabilistic game semantics." ACM Transactions on Computational Logic (TOCL) 3.3 (2002): 359-382.

14. Dimovski, A., Ghica, D. R., & Lazić, R. (2006, March). A counterexample-guided refinement tool for open procedural programs. In International SPIN Workshop on Model Checking of Software (pp. 288-292). Springer, Berlin, Heidelberg.

15. Fleming, Shane T., Ivan Beretta, David B. Thomas, George A. Constantinides, and Dan R. Ghica. "PushPush: Seamless integration of hardware and software objects via function calls over AXI." In Field Programmable Logic and Applications (FPL), 2015 25th International Conference on, pp. 1-8. IEEE, 2015.

16. Gabbay, Murdoch, and Dan Ghica. "Game semantics in the nominal model." Electronic Notes in Theoretical Computer Science 286 (2012): 173-189.

17. Ghica, Dan R. "Geometry of synthesis: a structured approach to VLSI design." In ACM SIGPLAN Notices, vol. 42, no. 1, pp. 363-375. ACM, 2007.

18. Ghica, Dan R. "Applications of game semantics: From program analysis to hardware synthesis." Logic In Computer Science, 2009. LICS'09. 24th Annual IEEE Symposium on. IEEE, 2009.

19. Ghica, Dan R. "Function interface models for hardware compilation." In Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on, pp. 131-142. IEEE, 2011.

20. Ghica, Dan R. "Diagrammatic reasoning for delay-insensitive asynchronous circuits." Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky. Springer, Berlin, Heidelberg, 2013. 52-68.

21. Ghica, Dan R., and Adam Bakewell. "Clipping: A semantics-directed syntactic approximation." In 2009 24th Annual IEEE Symposium on Logic In Computer Science, pp. 189-198. IEEE, 2009.

22. Ghica, Dan R., and Mohamed N. Menaa. "Synchronous game semantics via round abstraction." International Conference on Foundations of Software Science and Computational Structures. Springer, Berlin, Heidelberg, 2011.

23. Ghica, Dan R., and Guy McCusker. "Reasoning about Idealized Algol using regular languages." International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 2000.

24. Ghica, Dan R., and Andrzej S. Murawski. "Angelic semantics of fine-grained concurrency." International Conference on Foundations of Software Science and Computation Structures. Springer, Berlin, Heidelberg, 2004.

25. Ghica, Dan R., and Andrzej S. Murawski. "Angelic semantics of fine-grained concurrency." Annals of Pure and Applied Logic 151.2-3 (2008): 89-114.

26. Ghica, Dan R., and Alex Smith. "Geometry of Synthesis II: From games to delay-insensitive circuits." Electronic Notes in Theoretical Computer Science 265 (2010): 301-324.

27. Ghica, Dan R., and Alex Smith. "Geometry of synthesis III: resource management through type inference." In ACM SIGPLAN Notices, vol. 46, no. 1, pp. 345-356. ACM, 2011.

28. Ghica, Dan R., Alex Smith, and Satnam Singh. "Geometry of synthesis IV: compiling affine recursion into static hardware." In ACM SIGPLAN Notices, vol. 46, no. 9, pp. 221-233. ACM, 2011.

29. Ghica, Dan R., and Nikos Tzevelekos. "A system-level game semantics." Electronic Notes in Theoretical Computer Science 286 (2012): 191-211.

30. Girard, Jean-Yves. "Linear logic: its syntax and semantics." London Mathematical Society Lecture Note Series (1995): 1-42.

31. Harmer, Russ, Martin Hyland, and Paul-André Mellies. "Categorical combinatorics for innocent strategies." Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on. IEEE, 2007.

32. Harmer, Russell, and Guy McCusker. "A fully abstract game semantics for finite nondeterminism." LICS. IEEE, 1999.

33. Harmer, Russ, and Olivier Laurent. "The anatomy of innocence revisited." International Conference on Foundations of Software Technology and Theoretical Computer Science. Springer, Berlin, Heidelberg, 2006.

34. Hoare, Charles Antony Richard. "Communicating sequential processes." Communications of the ACM 21.8 (1978): 666-677.

35. Honda, Kohei, and Nobuko Yoshida. "Game-theoretic analysis of call-by-value computation." Theoretical Computer Science 221.1-2 (1999): 393-456.

36. Hopkins, D., Murawski, A. S., & Ong, C. H. L. (2011, July). A fragment of ML decidable by visibly pushdown automata. In International Colloquium on Automata, Languages, and Programming (pp. 149-161). Springer, Berlin, Heidelberg.

37. Hyland, J. Martin E., and C-HL Ong. "On full abstraction for PCF: I, II, and III." Information and computation 163.2 (2000): 285-408.

38. Jagadeesan, Radha, Corin Pitcher, Julian Rathke, and James Riely. "Local memory via layout randomization." In Computer Security Foundations Symposium (CSF), 2011 IEEE 24th, pp. 161-174. IEEE, 2011.

39. Jaber, Guilhem. "Operational nominal game semantics." In International Conference on Foundations of Software Science and Computation Structures, pp. 264-278. Springer, Berlin, Heidelberg, 2015.

40. Jeffrey, Alan, and Julian Rathke. "A fully abstract may testing semantics for concurrent objects." Theoretical Computer Science 338, no. 1-3 (2005): 17-63.

41. Jifeng, He, Mark B. Josephs, and C. A. R. Hoare. "A theory of synchrony and asynchrony." Programming Concepts and Methods. Elsevier, 1990.

42. Joachim Lambek, From lambda calculus to Cartesian closed categories, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, eds. J. P. Seldin and J. Hindley, Academic Press, 1980, pp. 376-402.

43. Laird, James. "A fully abstract game semantics of local exceptions." LICS. IEEE, 2001.

44. Laird, James. "A game semantics of idealized CSP." Electronic Notes in Theoretical Computer Science 45 (2001): 232-257.

45. Laird, James. "A fully abstract trace semantics for general references." International Colloquium on Automata, Languages, and Programming. Springer, Berlin, Heidelberg, 2007.

46. Lassen, Soren B., and Paul Blain Levy. "Typed normal form bisimulation." International Workshop on Computer Science Logic. Springer, Berlin, Heidelberg, 2007.

47. Mackie, Ian. "The geometry of interaction machine." Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1995.

48. Milner, Robin. "Functions as processes." Mathematical structures in computer science 2.2 (1992): 119-141.

49. Murawski, Andrzej S., and Nikos Tzevelekos. "Nominal game semantics." Foundations and Trends in Programming Languages 2.4 (2016): 191-269.

50. Nygaard, Mikkel, and Glynn Winskel. "HOPLA—a higher-order process language." International

Conference on Concurrency Theory. Springer, Berlin, Heidelberg, 2002.

51. O'Hearn, Peter, and Robert Tennent. ALGOL-like Languages. Springer Science & Business Media, 2013.

52. Ong, C. H. (2002). Observational equivalence of 3rd-order Idealized Algol is decidable. In Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on (pp. 245-256). IEEE.

53. Pitts, Andrew M. Nominal sets: Names and symmetry in computer science. Cambridge University Press, 2013.

54. Plotkin, Gordon D. "LCF considered as a programming language." Theoretical computer science 5.3 (1977): 223-255.

55. Roscoe, Bill. "The theory and practice of concurrency." (1998). (`http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`)

56. Stewart, Gordon, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. "Compositional compcert." ACM SIGPLAN Notices 50, no. 1 (2015): 275-287.

57. Tennent, Robert D. Semantics of programming languages. Vol. 199. No. 1. Hemel Hempstead: Prentice-Hall, 1991.

58. Thomas, David B., Shane T. Fleming, George A. Constantinides, and Dan R. Ghica. "Transparent linking of compiled software and synthesized hardware." In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pp. 1084-1089. EDA Consortium, 2015.