Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 286 (2012) 191–211

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A System-Level Game Semantics

Dan R. Ghica

University of Birmingham

Nikos Tzevelekos

Queen Mary, University of London

Abstract

Game semantics is a trace-like denotational semantics for programming languages where the notion of legal observable behaviour of a term is defined combinatorially, by means of rules of a game between the term (the *Proponent*) and its context (the *Opponent*). In general, the richer the computational features a language has the less constrained the rules of the semantic game. In this paper we consider the consequences of taking this relaxation of rules to the limit, by granting the Opponent *omnipotence*, that is, permission to play any move without combinatorial restrictions. However, we impose an epistemic restriction by not granting Opponent *omniscience*, so that Proponent can have undisclosed secret moves. We introduce a basic C-like programming language and we define such a semantic model for it. We argue that the resulting semantics is an appealingly simple combination of operational and game semantics and we show how certain traces explain system-level attacks, i.e. plausible attacks that are realisable outside of the programming language itself. We also show how allowing Proponent to have secrets ensures that some desirable equivalences in the programming language are preserved.

Keywords: Game semantics, omnipotent opponent, omniscient opponent

1 Introduction

Game semantics came to prominence by solving the long-standing open problem of *full abstraction* for PCF [2,7] and it consolidated its status as a successful approach to modelling programming languages by being used in the definition of numerous other fully abstract programming language models. The approach of game semantics is to model computation as a formal interaction, called a *game*, between a term and its context. Thus, a semantic game features two players: a *Proponent* (P), representing the term, and an *Opponent* (O), representing the context. The interaction is formally described by sequences of game moves, called *plays*, and a term is modeled by a corresponding *strategy*, that is, the set of all its possible plays. To define a game semantics one needs to define what are the rules of the game and what are the abilities of the players.

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2012.08.013

For PCF games, the rules are particularly neat, corresponding to the so-called "principles of polite conversation": moves are divided into *questions* and *answers*; players must take turns; no question can be asked unless it is made possible (*enabled*) by an earlier relevant question; no answer can be given unless it is to the most recent unanswered question. The legality constraints for plays can be imposed as combinatorial conditions on sequences of moves.

Strategies also have combinatorial conditions which characterise the players rather than the game. They are uniformity conditions which stipulate that if in certain plays P makes a certain move, in other plays it will make an analogous move. The simplest condition is *determinism*, which stipulates that in any strategy if two plays are equal up to a certain P move, their subsequent P moves must also be the same. Relaxing some of the combinatorial constraints on plays and strategies elegantly leads from models of PCF to models of more expressive programming languages. For example, relaxing a condition called *innocence* leads to models of programming language with state [3], relaxing *bracketing* leads to models of programming languages with control [10], and in the absence of *alternation* we obtain languages for concurrency [6].

Contribution.

In this paper we consider the natural question of what happens if in a game semantics we remove combinatorial constraints from O's behaviour. Unlike conventional game models, our construction is asymmetric: P behaves in a way determined by the programming language and its inherent limitations, whereas O can represent plausible behaviour which may not be, however, syntactically realizable neither in the language nor in some obvious extensions. We will see that such a model is, in a technical sense, well formed and that the notion of equivalence it induces is interesting and useful.

We study such a relaxed game model using an idealized type-free C-like language. The notion of *available move* is modeled using a notion of *secret* similar to that used in models of security protocols, formally represented using *names*. This leads to a notion of Opponent which is omnipotent but not omniscient: it can make any available move in any order, but some moves can be hidden from it. This is akin to the Dolev-Yao attacker model of security.

We show how inequivalences in this semantic model capture *system-level attacks*, i.e. behaviours of the ambient system which, although not realizable in the language itself, can be nevertheless plausibly enacted within the system. Despite the ambient system allowing suprising attacks, we note that many interesting equivalences still hold. This provides evidence that questions of semantic equivalence can be formulated outside the conventional framework of a *syntactic* context.

Technically, the model is expressed in an operationalised version of game semantics like Laird's [12] and names are handled using nominal sets [5].

2 A system-level semantics

2.1 Syntax and operational semantics

We introduce a simple type-free C-like language which is just expressive enough to illustrate the basic concepts. A *program* is a list of *modules*, corresponding roughly to files in C. A *module* is a list of function or variable declarations. An exported variable or function name is globally visible, otherwise its scope is the module. In extended BNF-like notation we write:

$$\begin{array}{ll} \mathit{Prog} ::= \mathit{Mod}^* & \mathit{Hdr} ::= \texttt{export}\ \overline{x}; \texttt{import}\ \overline{x}; \\ \mathit{Mod} ::= \mathit{Hdr}\ \mathit{Dcl} & \mathit{Dcl} ::= \texttt{decl}\ x = n;\ \mathit{Dcl} \mid \texttt{decl}\ \mathit{Func};\ \mathit{Dcl} \mid \epsilon \end{array}$$

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:

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

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 $n \in \mathbb{Z}$).

$$\begin{array}{rl} \mathit{Stm} ::= & \epsilon \mid \texttt{if}\ (\mathit{Exp})\ \texttt{then}\ \{\mathit{Stm}\}\ \texttt{else}\ \{\mathit{Stm}\};\ \mathit{Stm} \\ & \mid \mathit{Exp}{=}\mathit{Exp};\ \mathit{Stm} \\ & \mid \mathit{Exp}(\mathit{Exp}^*);\ \mathit{Stm} \\ \mathit{Exp} ::= & \mathit{Exp} \star \mathit{Exp} \mid *\mathit{Exp} \mid \mathit{Exp}(\mathit{Exp}^*) \mid (\mathit{Exp}, \mathit{Exp}) \mid \texttt{new}() \mid n \mid x \end{array}$$

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 $\overline{x}$."

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

$$\begin{array}{rl} t ::= & \texttt{if}\ (\Box)\ \texttt{then}\ \{e\}\ \texttt{else}\ \{e\} \mid \Box\ op\ e \mid v\ op\ \Box \mid op'\ \Box \mid \Box\ e \mid v\ \Box \\ & \mid (\Box, e) \mid (v, \Box) \end{array}$$

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

Our semantic setting is that of nominal sets [5] (see Appendix A), constructed over the multi-sorted set of names

$$\mathcal{N} = \mathcal{N}_\lambda \uplus \mathcal{N}_\phi \uplus \mathcal{N}_\kappa$$

where each of the three components is a countably infinite set of *location names*, *function names* and *function continuation names* respectively. That is, our objects can involve finitely many elements of $\mathcal{N}$, and they come with a canonical notion of applying name permutations to them. For an object x and a permutation π, the result of applying π to x is denoted by $\pi \cdot x$. 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. For any object x involving names, we write $\nu(x)$ for its *support*, i.e. the set of all the names occurring in it.

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 not defined on $\mathcal{X}$: $s \setminus \mathcal{X} = s \restriction (\mathsf{dom}(s) \setminus \mathcal{X})$;

(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. [1]

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:

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

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.

[1] 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.

Case $e = v$ is a value.

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

Case e is not a canonical form.

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

Fig. 1. Operational semantics

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.

2.2 System 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 Semantics (SLS)*.

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

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.

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

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

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.

$$\langle\!\langle N \mid P \vdash s \rangle\!\rangle \xrightarrow[s']{\texttt{ret}\, v,k'} \langle N \cup \nu(v,s') \mid P \cup \nu(v,s') \vdash s[s'], f, v, k\rangle$$
$$\text{if } s(k') = (f,k), \nu(v,s') \cap N \subseteq P, \lambda(\nu(v)) \subseteq \nu(s'), s \restriction \lambda(P) \sqsubseteq s'$$

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*:

(i) it can return with any value v so long as it only contains public names or fresh names (but not *private* ones);

(ii) it can update any public location with any value;

(iii) 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:

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

$$\langle N \mid P \vdash s, -, v, k\rangle \xrightarrow[s\restriction\lambda(P')]{\texttt{ret}\, v,k} \langle\!\langle N \mid P' \vdash s \rangle\!\rangle \quad \text{where } P' = Cl(s, P \cup \nu(v))$$

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.

3 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': $\quad [\![M \cdot M']\!] \;\cong_\tau\; [\![M]\!] \otimes [\![M']\!]\,.$

We call this the **principle of functional composition**.

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

(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)	$\dfrac{\mathcal{P} \xrightarrow[s]{\mathtt{call}\, f,v,k} \mathcal{S}' \qquad \mathcal{S} \xrightarrow[s]{\mathtt{call}\, f,v,k} \mathcal{P}'}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow{\tau} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi} \mathcal{P}'}$	*Cross-call* $k \in \mathcal{N}_{\mathsf{aux}} \setminus \nu(\mathcal{S})$.
(iii)	$\dfrac{\mathcal{P} \xrightarrow[s]{\mathtt{ret}\, v,k} \mathcal{S}' \qquad \mathcal{S} \xrightarrow[s]{\mathtt{ret}\, v,k} \mathcal{P}'}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow{\tau} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi} \mathcal{P}'}$	*Cross-return* $k \in \mathcal{N}_{\mathsf{aux}}$.
(iv)	$\dfrac{\mathcal{P} \xrightarrow[s]{\mathtt{call}\, f,v,k} \mathcal{S}' \qquad \mathcal{S} \xrightarrow[s]{\mathtt{call}\, f,v,k}\!\!\!\!\!\!\diagup \qquad}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow[s'[s]\restriction \Pi']{\mathtt{call}\, f,v,k} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}}$	*Program call* $\Pi' = (\Pi, s')^{P}[v,k,s]$ and $k \in \mathcal{N}_{\kappa} \setminus \nu(\mathcal{S})$.
(v)	$\dfrac{\mathcal{P} \xrightarrow[s]{\mathtt{ret}\, v,k} \mathcal{S}' \qquad \mathcal{S} \xrightarrow[s]{\mathtt{ret}\, v,k}\!\!\!\!\!\!\diagup \qquad}{\mathcal{P} \otimes^{s'}_{\Pi} \mathcal{S} \xrightarrow[s'[s]\restriction \Pi']{\mathtt{ret}\, v,k} \mathcal{S}' \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}}$	*Program return* $\Pi' = (\Pi, s')^{P}[v,s]$ and $k \in \mathcal{N}_{\kappa}$.
(vi)	$\dfrac{\mathcal{S} \xrightarrow[s]{\mathtt{call}\, f,v,k} \mathcal{P}}{\mathcal{S} \otimes^{s'}_{\Pi} \mathcal{S}' \xrightarrow[s\restriction \Pi']{\mathtt{call}\, f,v,k} \mathcal{P} \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}'}$	*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)	$\dfrac{\mathcal{S} \xrightarrow[s]{\mathtt{ret}\, v,k} \mathcal{P}}{\mathcal{S} \otimes^{s'}_{\Pi} \mathcal{S}' \xrightarrow[s\restriction \Pi']{\mathtt{ret}\, v,k} \mathcal{P} \otimes^{s'[\lambda(s)]}_{\Pi'} \mathcal{S}'}$	*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 $\nu(v, s \setminus Pr) \cap Pr = \emptyset$.

Fig. 2. Rules for semantic composition

thus given by:

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

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, [2] system stores [3] and name privacy. The latter originate from nominal game semantics [1,11] and they guarantee that the names introduced (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

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

[3] 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.

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 and prove functional composition (proof in Appendix B).

Definition 3.1 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 $\ell \in \mathcal{L}$,

(i) if $X \xrightarrow{\tau} X'$ then $R(X) = R(X')$,

(ii) if $X \to X'$ then $R(X) \to R(X')$,

(iii) if $R(X) \to Y$ and $X \not\xrightarrow{\tau}$ then $X \to X'$ with $R(X') = Y$,

(iv) if $X \xrightarrow{\ell} X'$ then $R(X) \xrightarrow{\ell} R(X')$,

(v) 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-isomoprhism $R : \mathcal{G}_1 \to \mathcal{G}_2$.

Proposition 3.2 *For all modules M, M', $[\![M]\!] \otimes [\![M']\!] \cong_\tau [\![M \cdot M']\!]$.*

4 Reasoning about SLS

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.

4.1 A system-level attack: violating secrecy

This example is inspired by a flawed security protocol which is 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 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:

(i) system calls `prot()` giving continuation k

(ii) program calls `read()` giving fresh continuation k'

(iii) system returns (from `read`) using k' and producing fresh name a_2

(iv) program returns (from `prot`) leaking local name a_1 stored in `k`

(v) system uses k' to fake a second return from `read`, using the just-learned name a_1 as a return value

(vi) 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.

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

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

Fig. 3. Secret a_0 leaks.

The critical step is (v), 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.

4.2 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 was first given in [13] and 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 [15].

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

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

where note that we orbit through in order to factor out the choice of initial private names.

Definition 4.1 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 4.2 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 $(X_1', X_2) \in \mathcal{R}$;
- $X_1 \stackrel{\ell}{\to} X_1'$ implies $X_2 \twoheadrightarrow X_2''$ with $(\pi \cdot X_2'') \stackrel{\ell}{\to} X_2'$ 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 4.3 *Bisimilarity coincides with trace equivalence.*

Proposition 4.4 *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* [14] 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.

5 Conclusion

In this paper we have developed a relaxed notion of game semantics in which the behaviour of the Opponent is defined by epistemic rather than combinatorial constraints. This has led us to two conclusions which we considered important.

First, we want to re-emphasise the fact that operational semantics can be extended in a relatively straightforward way from handling programs to handling terms, without relying on translation or interpretation. This is an idea already implicit in techniques such as *trace semantics* [8] or *environmental bisimulation* [9]. In the process, operational semantics becomes compositional. An LTS denotational interpretation of terms emerges automatically, without losing its effective presentation. Unlike previous work, however, we do not treat this extension of the operational semantics as a means to an end, e.g. studying contextual equivalence, but we treat it as important in its own right.

Secondly, and most importantly, we want to show that a meaningful and useful notion of context for the execution of terms can be constructed *outside the syntax of the language*. This has several advantages. The first one is modularity, as we can define the language and the environment in which its terms operate independently; the principle of functional composition is the consistency check that we need to satisfy for the two to be able to work together. The second one is realism, as real-life languages allow, through mechanisms such as separate compilation and foreign-function interface, programs which are syntactically heterogeneous so they cannot be characterised by the usual notion of context. The third one is simplicity, as we show how it is possible to formulate restrictions on the environment in a way which is not computational but epistemic, resembling the established Dolev-Yao characterisation of context in security. We believe this has the potential to offer a semantic foundation for the study of security properties of programs (such as information flow or tamper-proof compilation) in a way which is less dependent on the vagaries of syntax and more modular.

Relevant related work with similar aims but different philosophy has been car-

ried out in compositional compiler correctness [4]. Whereas our point of view is mainly analytic, being interested in characterising arbitrary (if not unrestricted) environments and examine operationally the behaviour of open terms in such environments, compositional compiler correctness is a primarily a synthetic concern, aiming at defining constraints on machine code which allow safe composition between code generated via compilation with code generated in arbitrary ways. We see these two approaches as two sides of the same problem and we believe a better understanding of the relation between them should be studied.

References

[1] S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *LICS*, pp 150–159, 2004.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2), 2000.

[3] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.

[4] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[5] M. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.

[6] D. R. Ghica and A. Murawski. Angelic semantics of fine-grained concurrency. *Annals of Pure and Applied Logic*, 151(2-3):89–114, 2008.

[7] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2), 2000.

[8] A. Jeffrey and J. Rathke. Java jr: Fully abstract trace semantics for a core java language. In *ESOP*, 2005.

[9] V. Koutavas, P. B. Levy, and E. Sumii. From applicative to environmental bisimulation. *Electr. Notes Theor. Comput. Sci.*, 276:215–235, 2011.

[10] J. Laird. Full abstraction for functional languages with control. In *LICS*, pp 58–67, 1997.

[11] J. Laird. A game semantics of local names and good variables. In *FoSSaCS*, pp 289–303, 2004.

[12] J. Laird. A fully abstract trace semantics for general references. In *ICALP*, pp 667–679, 2007.

[13] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL*, 1988.

[14] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.

[15] R. D. Tennent. Semantical analysis of specification logic. *Inf. Comput.*, 85(2):135–162, 1990.

A Nominal Sets

It is handy to introduce here some basic notions from the theory of nominal sets [5]. We call *nominal structure* any structure which may contain names, i.e. elements of $\mathcal{N}$, and we denote by *Perm* the set of finite permutations on $\mathcal{N}$ which are sort-preserving (i.e. if $a \in \mathcal{N}_\lambda$ then $\pi(a) \in \mathcal{N}_\lambda$, etc.). We range over permutations by π and variants. Finiteness means that each set $\{a \in \mathcal{N} \mid \pi(a) \neq a\}$ is finite. For example, $\mathtt{id} = \{(a,a) \,|\, a \in \mathcal{N}\}$ is the identity permutation. On the other hand, $(a\ \ b) = \{(a,b),(b,a)\} \cup \{(c,c) \mid c \neq a,b\}$ is the permutation which swaps a and b and fixes all other names, for all a, b of the same sort.

For each set X of nominal structures of interest, we define a function $\_\cdot\_ : \mathit{Perm} \times X \to X$ such that $\pi \cdot (\pi' \cdot x) = (\pi \circ \pi') \cdot x$ and $\mathtt{id} \cdot x = x$, for all $x \in X$ and $\pi, \pi' \in \mathit{Perm}$. X is called a *nominal set* if all its elements involve finitely many names, that is, for all $x \in X$ there is a finite set $S \subseteq \mathcal{N}$ such that $\pi \cdot x = x$ whenever $\forall a \in S.\pi(a) = a$. The minimal such set S is called the *support* of x and denoted by $\nu(x)$. For example, $\mathcal{N}$ is a nominal set with action $\pi \cdot a = \pi(a)$, and so is $\mathcal{P}_{\mathsf{fn}}(\mathcal{N})$ with action $\pi \cdot S = \{\pi(a) \mid a \in S\}$.

Also, any set of non-nominal structures is a nominal set with trivial action $\pi \cdot x = x$. More interestingly, if X, Y are nominal sets then so is $X \times Y$ with action $\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$. This extends to arbitrary products and to strings. Finally, if X, Y are nominal sets then so is the set $X \rightharpoonup_{\mathsf{fn}} Y$ with action $\pi \cdot f = \{(\pi \cdot x, \pi \cdot y) \mid (x, y) \in f\}$.

B Functional composition

We start with a lemma which stems from the definitions. We write $\kappa'(-)$ for the projection on $\mathcal{N}'_\kappa = \mathcal{N}_\kappa \uplus \mathcal{N}_{\mathsf{aux}}$, and $\kappa(-)$ for the projection on $\mathcal{N}_\kappa$.

Lemma B.1 *Let $X_1 \otimes^s_\Pi X_2$ be a state in the transition graph of $[\![M]\!] \otimes [\![M']\!]$ that is reachable from the initial state. Then, if each X_i includes the triple (N_i, P_i, s_i), the following conditions hold.*

- $(N_1 \setminus P_1) \cap N_2 = N_1 \cap (N_2 \setminus P_2) = \emptyset$, $P_1 \setminus \Pi = P_2 \setminus \Pi$, $\Pi \subseteq \nu(s) \cup \kappa(P_1 \cup P_2) \subseteq P_1 \cup P_2$ *and* $\mathsf{dom}(s) = \lambda(P_1 \cup P_2)$.
- $\mathsf{dom}(\kappa'(s_1)) \cap \mathsf{dom}(\kappa'(s_2)) = \emptyset$, $(\mathsf{dom}(\kappa'(s_1)) \cup \mathsf{dom}(\kappa'(s_2))) \cap \Pi_S = \emptyset$ *and* $\kappa'(P_1 \cap P_2) \setminus \Pi_S = \kappa'(P_1 \cup P_2) \setminus \Pi \subseteq \mathcal{N}_{\mathsf{aux}}$.
- *If both X_1, X_2 are system configurations and $X_1 \otimes^s_\Pi X_2$ is preceded by a state of the form $\mathcal{P} \otimes^{s'}_{\Pi'} \mathcal{S}$ then $s \restriction P_1 \subseteq s_1$ and $s \restriction (P_2 \setminus P_1) \subseteq s_2$, and dually if preceded by $\mathcal{S} \otimes^{s'}_{\Pi'} \mathcal{P}$. Thus, in both cases, $s \restriction (P_i \setminus (P_1 \cap P_2)) \subseteq s_i$ for $i = 1, 2$.*
- *Not both X_1, X_2 are program configurations. If X_i is a program configuration then $s \restriction (P_{3-i} \setminus P_i) \subseteq s_{3-i}$.*

Semantic composition introduces a notion of private names: internal continuation names passed around between the two modules in order to synchronise their mutual function calls. As the previous lemma shows, these names remain private throughout the computation. Therefore, in checking bisimilarity for such reduction systems, special care has to be taken so that these private names cannot be captured by external system transitions. This is achieved by selecting (only) these names from the auxiliary set $\mathcal{N}_{\mathsf{aux}}$.

We define the following translation R from reachable states of $[\![M]\!] \otimes [\![M']\!]$ to configurations of $[\![M \cdot M']\!]$.

$$\begin{aligned}
&\langle\!\langle N_1 \mid P_1 \vdash s_1 \rangle\!\rangle \otimes^s_\Pi \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle \\
&\qquad\qquad \longmapsto \langle\!\langle (N_1 \cup N_2) \setminus \mathcal{N}_{\mathsf{aux}} \mid \Pi \vdash (\hat{s}_1[s'] \cup \hat{s}_2[s']) \setminus \mathcal{N}_{\mathsf{aux}} \rangle\!\rangle \\
&\langle\!\langle N_1 \mid P_1 \vdash s_1 \rangle\!\rangle \otimes^s_\Pi \langle N_2 \mid P_2 \vdash s_2, t, v, k \rangle
\end{aligned}$$

$$\longmapsto \langle (N_1 \cup N_2) \setminus \mathcal{N}_{\mathsf{aux}} \mid \Pi \vdash \hat{s}_1[\hat{s}_2] \setminus \mathcal{N}_{\mathsf{aux}}, t', v, k' \rangle$$
$$\langle N_1 \mid P_1 \vdash s_1, t, v, k \rangle \otimes^s_\Pi \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle$$
$$\longmapsto \langle (N_1 \cup N_2) \setminus \mathcal{N}_{\mathsf{aux}} \mid \Pi \vdash \hat{s}_2[\hat{s}_1] \setminus \mathcal{N}_{\mathsf{aux}}, t', v, k' \rangle$$

where $s' = s \upharpoonright (P_1 \cap P_2)$, $\hat{s}_i = s_i[k \mapsto (s_1, s_2)_\downarrow(s_i(n))]$ for all $k \in \mathsf{dom}(\kappa(s_i))$, and $(t', k') = (s_1, s_2)_\downarrow(t, k)$. The function $(s_1, s_2)_\downarrow$ fetches the full external frame stack and the external continuation searching back from (t, k), that is:

$$(s_1, s_2)_\downarrow(t, k) = \begin{cases} (t, k) & \text{if } k \notin \mathcal{N}_{\mathsf{aux}} \\ (s_1, s_2)_\downarrow(t' \circ t, k') & \text{if } k \in \mathcal{N}_{\mathsf{aux}} \text{ and } s_i(k) = (t', k') \end{cases}$$

Thus, the translation merges names from the component configurations and deletes the names in $\mathcal{N}_{\mathsf{aux}}$: these private names do not appear in $[\![M \cdot M']\!]$, as there the corresponding function calls happen without using the call-return mechanism. Note that $[\![M \cdot M']\!]$ is defined over the original $\mathcal{N}$, so it cannot capture any $k \in \mathcal{N}_{\mathsf{aux}}$. The translation also sets Π as the set of public names. Moreover, the total store is computed as follows. In system configurations we just take the union of the component stores and update them with the values of s, which contains the current values of all common public names. In program configurations we use the fact that the P-component contains more recent values than those of the S-component.

Proposition B.2 *For R defined as above and $X_1 \otimes^s_\Pi X_2$ a reachable configuration,*

1. *if $X_1 \otimes^s_\Pi X_2 \xrightarrow{\tau} X_1' \otimes^{s'}_\Pi X_2'$ then $R(X_1 \otimes^s_\Pi X_2) = R(X_1' \otimes^{s'}_\Pi X_2')$,*
2. *if $X_1 \otimes^s_\Pi X_2 \to X_1' \otimes^s_\Pi X_2'$ then $R(X_1 \otimes^s_\Pi X_2) \to R(X_1' \otimes^s_\Pi X_2')$,*
3. *if $R(X_1 \otimes^s_\Pi X_2) \to Y$ and $X_1 \otimes^s_\Pi X_2 \overset{\tau}{\not\to}$ then $X_1 \otimes^s_\Pi X_2 \to X_1' \otimes^s_\Pi X_2'$ with $Y = R(X_1' \otimes^s_\Pi X_2')$,*
4. *if $X_1 \otimes^s_\Pi X_2 \xrightarrow[s']{\alpha} X_1' \otimes^{s''}_{\Pi'} X_2'$ then $R(X_1 \otimes^s_\Pi X_2) \xrightarrow[s']{\alpha} R(X_1' \otimes^{s''}_{\Pi'} X_2')$,*
5. *if $R(X_1 \otimes^s_\Pi X_2) \xrightarrow[s']{\alpha} Y$ and $X_1 \otimes^s_\Pi X_2 \overset{\tau}{\not\to}$ then $X_1 \otimes^s_\Pi X_2 \xrightarrow[s']{\alpha} X_1' \otimes^{s''}_{\Pi'} X_2'$ with $Y = R(X_1' \otimes^{s''}_{\Pi'} X_2')$.*

Proof. For 1, let $X_1 = \langle N_1 \mid P_1 \vdash s_1, t \circ f(\Box), v, k \rangle$, $X_2 = \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle$ and the τ-transition being due to an internal transition with label $(s_{\mathsf{i}}, \mathtt{call}\, f, v, k')$. Thus, $X_1' = \langle\!\langle N_1' \mid P_1' \vdash s_1' \rangle\!\rangle$, $X_2' = \langle N_2' \mid P_2' \vdash s_2', f(\Box), v, k' \rangle$, and so $R(X_1 \otimes^s_\Pi X_2) = \langle N_0 \mid \Pi \vdash s_0, t_0, v, k_0 \rangle$ and $R(X_1' \otimes^{s'}_\Pi X_2') = \langle N_0' \mid \Pi \vdash s_0', t_0', v, k_0' \rangle$. Note that $k' \in \mathcal{N}_{\mathsf{aux}}$. Moreover, $s_1' = s_1[k' \mapsto (t, k)]$ and $s_2' = s \cup (s_2 \setminus P_2)$, so $(t_0', k_0') = (s_1', s_2')_\downarrow(f(\Box), k') = (s_1', s_2')_\downarrow(t \circ f(\Box), k) = (t_0, k_0)$. Moreover, $N_0 = (N_1 \cup N_2) \setminus \mathcal{N}_{\mathsf{aux}}$ and $N_0' = (N_1' \cup N_2') \setminus \mathcal{N}_{\mathsf{aux}} = (N_1 \cup \{k'\} \cup N_2 \cup \nu(v, s_{\mathsf{i}})) \setminus \mathcal{N}_{\mathsf{aux}}$. As $\nu(v, s_{\mathsf{i}}) \subseteq N_1$ and $k' \in \mathcal{N}_{\mathsf{aux}}$, we get $N_0 = N_0'$. Finally, $s_0 = \hat{s}_2[\hat{s}_1] \setminus \mathcal{N}_{\mathsf{aux}}$ and $s_0' = \hat{s}_1'[\hat{s}_2'] \setminus \mathcal{N}_{\mathsf{aux}}$. Thus, $s_0' = \hat{s}_1[\hat{s}_2'] \setminus \mathcal{N}_{\mathsf{aux}} = \hat{s}_1[s' \cup (\hat{s}_2 \setminus \lambda(P_2))] \setminus \mathcal{N}_{\mathsf{aux}}$. Moreover, $s' = s_1 \upharpoonright \lambda(P_1')$ so $s_0' = \hat{s}_1[\hat{s}_2 \setminus \lambda(P_2)] \setminus \mathcal{N}_{\mathsf{aux}}$. But now note that $\mathsf{dom}(s_2 \setminus \lambda(P_2)) \cap \mathsf{dom}(s_1) = \emptyset$: by the previous lemma, $\mathsf{dom}(s_1)$ and $\mathsf{dom}(s_2)$ share no continuation names, and if a is a location name in $\mathsf{dom}(s_2) \setminus P_2$ then $a \notin N_1$. Thus, $s_0 = s_0'$. Similarly if the τ-transition is due to an internal return.

Item 2 is straightforward. For 3, the only interesting issue is establishing that if $X_1 \otimes^s_{\Pi} X_2$ is in such a form that a τ-transition needs to take place then the latter is possible. This follows directly from the definitions and the conditions of the previous lemma. In the following cases we consider call transitions; cases with return transitions are treated in a similar manner.

For 4, let $X_1 = \langle\!\langle N_1 \mid P_1 \vdash s_1 \rangle\!\rangle$, $X_2 = \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle$, $\alpha = (s', \mathtt{call}\, f, v, k)$ and suppose the transition is due to X_1 reducing to $X_1' = \langle N_1' \mid P_1' \vdash s_1', f(\Box), v, k\rangle$ with label $(s_{\mathsf{i}}, \mathtt{call}\, f, v, k)$. We have $\Pi' = \Pi \cup \nu(v, k, s_{\mathsf{i}} \setminus Pr)$, $Pr = (N_1 \cup N_2) \setminus \Pi$, $s' = s_{\mathsf{i}} \restriction \Pi'$ and $\nu(v, s_{\mathsf{i}} \setminus Pr) \cap Pr = \emptyset$. Let $R(X_1 \otimes^s_{\Pi} X_2) = \langle\!\langle N_0 \mid \Pi \vdash s_0 \rangle\!\rangle$. As $k \notin \mathsf{dom}(s_1)$ and $k \notin \nu(X_2) \setminus \Pi_S$, by previous lemma we obtain $k \notin \mathsf{dom}(s_0)$, so the latter reduces to $\langle N_0' \mid P \vdash s_0', f(\Box), v, k\rangle$ with transition $(s''', \mathtt{call}\, f, v, k)$, for any appropriate s'''. In fact, if $\nu(v, s') \cap N_0 \subseteq \Pi$ then we can choose $s''' = s'$. Indeed, $(\nu(v, s') \cap N_0) \setminus \Pi \subseteq \nu(v, s') \cap (N_0 \setminus \Pi) \subseteq \nu(v, s') \cap Pr = \nu(v, s_{\mathsf{i}} \restriction \Pi') \cap Pr = \nu(v, s_{\mathsf{i}} \setminus Pr) \cap Pr = \emptyset$. Let $R(X_1' \otimes^{s''}_{\Pi'} X_2) = \langle\!\langle N_0'' \mid \Pi' \vdash s_0'' \rangle\!\rangle$. We can see that $N_0' = N_0''$. Also, $P = \Pi \cup \{k\} \cup \nu(v, s')$ while $\Pi' = \Pi \cup \nu(v, k, s_{\mathsf{i}} \setminus Pr) = \Pi \cup \nu(v, k, s')$. Moreover, $s_0' = s' \cup (s_0 \setminus \lambda(\Pi)) = s' \cup ((\hat{s}_1[s_{12}] \cup \hat{s}_2[s_{12}]) \setminus (\mathcal{N}_{\mathsf{aux}} \cup \lambda(\Pi)))$ with $s_{12} = s \restriction (P_1 \cap P_2)$, and $s_0'' = \hat{s}_2[\hat{s}_1'] \setminus \mathcal{N}_{\mathsf{aux}} = \hat{s}_2[s_{\mathsf{i}} \cup (\hat{s}_1 \setminus \lambda(P_1))] \setminus \mathcal{N}_{\mathsf{aux}}$. Moreover, s_0' and s_0'' agree on the domain of s' and on continuation names. Also, if location name $a \in N_0' \setminus N_0$ then $a \in \nu(v, s')$ and thus $a \in \mathsf{dom}(s')$. Thus, we need to show that s_0', s_0'' agree on location names a from $N_0 \setminus \Pi$. If $a \in N_1 \setminus P_1$ then $s_0'(a) = s_1(a) = s_0''(a)$, and similarly if in $N_2 \setminus P_2$ using the fact that $(N_2 \setminus P_2) \cap N_1 = \emptyset$. Finally, if $a \in P_1 \setminus \Pi = P_2 \setminus \Pi$ then $s_0'(a) = s(a) = s_{\mathsf{i}}(a) = s_0''(a)$, by restrictions on s_{i}.

Now let $X_1 = \langle N_1 \mid P_1 \vdash s_1, t \circ f(\Box), v, k\rangle$, $X_2 = \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle$, $\alpha = \mathtt{call}\, f, v, k'$ and suppose the transition is due to X_1 reducing to $X_1' = \langle\!\langle N_1' \mid P_1' \vdash s_1' \rangle\!\rangle$ with label $(s_{\mathsf{i}}, \mathtt{call}\, f, v, k')$. We have $(\Pi', s'') = (\Pi, s)[v, s_{\mathsf{i}}]$ and $s' = s'' \restriction \Pi'$. We can assume, by definition, that $(s_1, s_2)_K(t \circ f(\Box), k) = (t_0 \circ f(\Box), k_0)$, so $R(X_1 \otimes^s_{\Pi} X_2) = \langle N_0 \mid \Pi \vdash s_0, t_0 \circ f(\Box), v, k_0\rangle$. As f is not defined in either of the modules and k' is completely fresh, the latter reduces to $\langle\!\langle N_0' \mid P \vdash s_0' \rangle\!\rangle$ with transition $(s''', \mathtt{call}\, f, v, k')$. Let $R(X_1' \otimes^{s''}_{\Pi'} X_2) = \langle\!\langle N_0'' \mid \Pi' \vdash s_0'' \rangle\!\rangle$. It is easy to see that $N_0' = N_0''$. Moreover, $s_0' = s_0[k' \mapsto (t_0, k_0)]$ and $s_0'' = (\hat{s}_1'[s_{12}''] \cup \hat{s}_2[s_{12}'']) \setminus \mathcal{N}_{\mathsf{aux}}$ where $s_{12}'' = s'' \restriction (P_1' \cap P_2)$. Note that $s_0''(k') = \hat{s}_1'(k') = (s_1', s_2)_{\downarrow}(t, k) = (t_0, k_0)$. Also, s_0' and s_0'' agree on all other continuation names. Thus, in order to establish that $s_0' = s_0''$, it suffices to show that $s_2[s_1]$ and $s_1[s_{12}''] \cup s_2[s_{12}'']$ agree on locations. From the previous lemma, s'' agrees with s_1 on locations in P_1' and with s_2 on locations in $P_2 \setminus P_1'$, and so $s'' \subseteq s_2[s_1]$. Thus, $\lambda(s_1[s_{12}''] \cup s_2[s_{12}'']) = \lambda(s_1 \cup (s_2 \setminus P_1')) = \lambda(s_2[s_1])$.

For public names, we have $P = Cl(s_0, \Pi \cup \nu(v)) \cup \{k'\} = Cl(s_0', \Pi \cup \nu(v, k'))$ while $\Pi' = Cl(s'', \Pi \cup \nu(v, k'))$. As $\kappa(P) = \kappa(\Pi) \cup \{k'\} = \kappa(\Pi')$, we can focus on location names. We have $s'' \subseteq s_0'$ and, moreover, $\mathsf{dom}(s'') = \lambda(P_1' \cup P_2) \supseteq \lambda(\Pi \cup \nu(v, k'))$, thus $P = \Pi'$. Finally, $s' = s'''$ follows from the fact that these are restrictions of the final stores to the final sets of public location names.

For 5, let $X_1 = \langle\!\langle N_1 \mid P_1 \vdash s_1 \rangle\!\rangle$, $X_2 = \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle$, $R(X_1 \otimes^s_{\Pi} X_2) = \langle\!\langle N_0 \mid \Pi \vdash s_0 \rangle\!\rangle$ and $\alpha = (s', \mathtt{call}\, f, v, k)$. We have that f is defined in $M \cdot M'$ so WLOG assume that it is defined in M. Then, X_1 reduces to $X_1' = \langle N_1' \mid P_1' \vdash s_1', f(\Box), v, k\rangle$ with $(s_{\mathsf{i}}, \mathtt{call}\, f, v, k)$, $s_{\mathsf{i}} = s' \cup (s \setminus \Pi)$, if the relevant conditions for S-P calls are

satisfied.

If $k \in \mathsf{dom}(s_1)$ then, by lemma, $k \notin \Pi_S$. By assumption, $k \in \Pi$ so $k \notin \kappa(P_1 \cup P_2) \setminus \Pi$ and thus, by lemma, $k \notin \kappa(P_1 \cap P_2) \setminus \Pi_S$ so $k \notin P_2$. But the latter would imply $k \in \mathsf{dom}(s_0)$, which is disallowed by definition. Thus, $k \notin \mathsf{dom}(s_1)$.

Moreover, if $a \in \nu(v, s_{\mathsf{i}}) \cap (N_1 \setminus P_1) = \nu(v, s') \cap (N_1 \setminus P_1)$ then $a \in \nu(v, s') \cap (N_0 \setminus P_1)$ and $a \notin P_2$, so $a \in \nu(v, s') \cap (N_0 \setminus (P_1 \cup P_2)) \subseteq \nu(v, s') \cap (N_0 \setminus \Pi)$, thus contradicting the conditions for the transition α. We still need to check that $s_1 \restriction \lambda(P_1) \sqsubseteq s_{\mathsf{i}} = s' \cup (s \setminus \Pi)$. Given that $s_0 \restriction \lambda(\Pi) = (s_1[s_{12}] \cup s_2[s_{12}]) \restriction \lambda(\Pi) \sqsubseteq s'$, the condition follows from the previous lemma. We therefore obtain a transition from $X_1 \otimes^{s}_{\Pi} X_2$ to $X'_1 \otimes^{s''}_{\Pi'} X_2$; the relevant side-conditions are shown to be satisfied similarly as above. Finally, working as in 4, we obtain $R(X'_1 \otimes^{s''}_{\Pi'} X_2) = Y$ and $s' = s'''$.

Now let $X_1 = \langle N_1 \mid P_1 \vdash s_1, t \circ f(\Box), v, k \rangle$, $X_2 = \langle\!\langle N_2 \mid P_2 \vdash s_2 \rangle\!\rangle$, $R(X_1 \otimes^{s}_{\Pi} X_2) = \langle N_0 \mid \Pi \vdash s_0, t_0 \circ f(\Box), v, k_0 \rangle$ and $\alpha = \mathtt{call}\, f, v, k'$. By hypothesis, k' is fresh and therefore X_1 reduces to $X'_1 = \langle\!\langle N'_1 \mid P'_1 \vdash s'_1 \rangle\!\rangle$ with $(s_{\mathsf{i}}, \mathtt{call}\, f, v, k')$, and thus $X_1 \otimes^{s}_{\Pi} X_2$ reduces to $X'_1 \otimes^{s''}_{\Pi'} X_2$ with $(s''', \mathtt{call}\, f, v, k')$. Working as in 4, $R(X'_1 \otimes^{s''}_{\Pi'} X_2) = Y$ and $s' = s'''$. □

C Equivalence

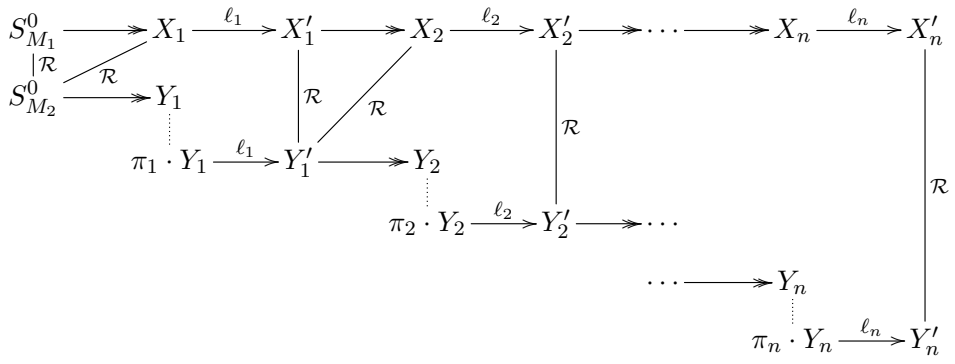
We first define a notion of accepted traces for arbitrary configurations by setting $T(X) = \{(\pi \cdot w) \in \mathcal{L}^* \mid X \xrightarrow{w} \!\!\!\!\rightarrow\, X', \forall a \in P(X).\, \pi(a) = a\}$, where $P(X)$ is the set of public names of X. Note that $w \in T(X)$ implies $\pi \cdot w \in T(X)$ for any π that fixes all names in $P(X)$ and, moreover, since $X \xrightarrow{\ell/\epsilon} X' \implies (\pi \cdot X) \xrightarrow{\pi \cdot \ell/\epsilon} (\pi \cdot X')$, that $T(X) = \{w \in \mathcal{L}^* \mid (\pi \cdot X) \xrightarrow{w} \!\!\!\!\rightarrow\, X', \forall a \in P(X).\, \pi(a) = a\}$.

Lemma C.1 *Let M_1, M_2 be modules with common initial public names. Then, $M_1 \sim M_2 \iff M_1 \cong M_2$.*

Proof. ($\Rightarrow$) Let $\mathcal{R}$ be a bisimulation witnessing $M_1 \sim M_2$. For each transition sequence

$$S^0_{M_1} \twoheadrightarrow X_1 \xrightarrow{\ell_1} X'_1 \twoheadrightarrow X_2 \xrightarrow{\ell_2} X'_2 \cdots \twoheadrightarrow X_n \xrightarrow{\ell_n} X'_n,$$

$\mathcal{R}$ yields a diagram as below,



and hence filling in the gaps we obtain:

$$\pi_1^n \cdot S^0_{M_2} \twoheadrightarrow \pi_1^n \cdot Y_1 \xrightarrow{\pi_2^n \cdot \ell_1} \pi_2^n \cdot Y_1' \twoheadrightarrow \pi_2^n \cdot Y_2 \xrightarrow{\pi_3^n \cdot \ell_2} \pi_3^n \cdot Y_2' \cdots \twoheadrightarrow \pi_n^n \cdot Y_n \xrightarrow{\ell_n} Y_n'$$

where $\pi_i^n = \pi_n \circ \cdots \circ \pi_i$. By definition, each π_i fixes all names in $P(X_i)$ so, in particular, $\pi_i \cdot \ell_j = \ell_j$ for all $j < i$, thus $\pi_1^n \cdot S^0_{M_2} \xrightarrow{\ell_1 \ldots \ell_n} Y_n'$.
Hence, $w \in T(M_1)$ implies $w \in T(M_2)$. The other inclusion is shown similarly.

Conversely, suppose $T(M_1) = T(M_2)$ and let us define the relation:

$$\mathcal{R} = \{(X_1, X_2) \mid S^0_{M_i} \xrightarrow{w_i} X_i,\ P(X_1) = P(X_2),\ T(X_1) = T(X_2)\}$$

We claim that $\mathcal{R}$ is a bisimulation. By symmetry, it suffices to show it is a simulation. Observe that all related configurations have common public names and, moreover, that $\mathcal{R}$ is closed under ϵ-transitions (by determinacy of internal transitions up to choice of fresh names). Now let $(X_1, X_2) \in \mathcal{R}$ with $X_1 \xrightarrow{\ell} X_1'$. Since $\ell \in T(X_1) = T(X_2)$, there is a configuration X_2'' and a permutation π fixing all elements of $P = P(X_1) = P(X_2) = P(X_2'')$ such that $\pi \cdot X_2 \twoheadrightarrow X_2'' \xrightarrow{\ell} X_2'$. Thus, $X_2 \twoheadrightarrow \pi^{-1} \cdot X_2''$ and $\pi \cdot (\pi^{-1} \cdot X_2'') \xrightarrow{\ell} X_2'$. Also, $P' = P(X_1') = P(X_2') = P \cup \nu(\ell)$.
Take now any $X_1' \xrightarrow{w} X_1''$. We have $\ell w \in T(X_1) = T(X_2)$ so, for some π'' fixing all names in P, $\pi'' \cdot X_2 \twoheadrightarrow \tilde{X}_2'' \xrightarrow{\ell} \tilde{X}_2 \xrightarrow{w} \tilde{X}_2'$. In particular, $X_2 \twoheadrightarrow \pi''^{-1} \cdot \tilde{X}_2''$ and hence, since internal transitions are deterministic up to choice of fresh (private) names, $X_2'' = \tilde{\pi} \cdot \tilde{X}_2''$ for some $\tilde{\pi}$ fixing all names in P. We thus obtain:

$$X_2'' \xrightarrow{\ell} X_2', \quad X_2'' \xrightarrow{\tilde{\pi} \cdot \ell} \tilde{\pi} \cdot \tilde{X}_2 . \qquad (*)$$

Suppose X_2'' is a P configuration. Then, by determinacy, $\ell = \tilde{\pi} \cdot \ell$ and $X_2' = \tilde{\pi} \cdot \tilde{X}_2$. Recall that $\tilde{\pi}$ fixes all names in P. Moreover, the closure conditions on P-to-S transitions stipulate that all names in $P' \setminus P$ are reachable from $P \cup \nu(v)$ through the store s, where v, s the value and store components of ℓ respectively. This implies that $\tilde{\pi}$ fixes all names in P'. Hence, from $\tilde{X}_2 \xrightarrow{w} \tilde{X}_2'$ we obtain $w \in T(X_2')$.
On the other hand, if X_2'' is an S configuration then let $a_1, \cdots, a_N$ be an enumeration of $\nu(X_2'')$. We define permutations $\pi_0, \pi_1, ..., \pi_N$ by:

$$\pi_0 = \mathtt{id}, \quad \pi_{i+1} = (a_{i+1}\ \ (\pi_i \circ \tilde{\pi})(a_{i+1})) \circ \pi_i .$$

We claim that, for each $0 \le i \le N$ and $1 \le j \le i$, we have

$$\pi_i \cdot \tilde{\pi} \cdot a_j = a_j , \quad \forall a \in \nu(\ell) \backslash P.\, \pi_i \cdot \tilde{\pi} \cdot a = \tilde{\pi} \cdot a , \quad \forall a \in P.\, \pi_i \cdot a = a .$$

We do induction on i; the case of $i = 0$ is clear. For the inductive step, if $\pi_i \cdot \tilde{\pi} \cdot a_{i+1} = a_{i+1}$ then $\pi_{i+1} = \pi_i$, and $\pi_{i+1} \cdot \tilde{\pi} \cdot a_{i+1} = \pi_i \cdot \tilde{\pi} \cdot a_{i+1} = a_{i+1}$. Moreover, by IH, $\pi_{i+1} \cdot \tilde{\pi} \cdot a_j = a_j$ for all $1 \le j \le i$, and $\pi_{i+1} \cdot \tilde{\pi} \cdot a = \tilde{\pi} \cdot a$ for all $a \in \nu(\ell) \setminus P$, and $\pi_{i+1} \cdot a = a$ for all $a \in P$. If $\pi_i \cdot \tilde{\pi} \cdot a_{i+1} = a'_{i+1} \neq a_{i+1}$ then, by construction, $\pi_{i+1} \cdot \tilde{\pi} \cdot a_{i+1} = a_{i+1}$. Moreover, for each $1 \le j \le i$, by IH, $\pi_{i+1} \cdot \tilde{\pi} \cdot a_j = (a_{i+1}\ a'_{i+1}) \cdot a_j$, and the latter equals a_j since $a_{i+1} \neq a_j$ implies $a'_{i+1} \neq \pi_i \cdot \tilde{\pi} \cdot a_j = a_j$. For any

$a \in \nu(\ell) \setminus P$, $\pi_{i+1} \cdot \tilde{\pi} \cdot a = (a_{i+1}\ \ a'_{i+1}) \cdot \pi_i \cdot \tilde{\pi} \cdot a = (a_{i+1}\ \ a'_{i+1}) \cdot \tilde{\pi} \cdot a$, by IH. Now, $a \neq a_{i+1}$ since $\nu(\ell) \cap \nu(X''_2) \subseteq P$, hence $\tilde{\pi} \cdot a = \pi_i \cdot \tilde{\pi} \cdot a \neq a'_{i+1}$. Moreover, $\nu(\tilde{\pi} \cdot \ell) \cap \nu(X''_2) \subseteq P$ implies $\tilde{\pi} \cdot a \neq a_{i+1}$, so $\pi_{i+1} \cdot \tilde{\pi} \cdot a = \tilde{\pi} \cdot a$. Finally, for any $a \in P$ we have $\pi_{i+1} \cdot a = (a_{i+1}\ \ a'_{i+1}) \cdot \pi_i \cdot a = (a_{i+1}\ \ a'_{i+1}) \cdot a$. If $a = a_{i+1}$ then $(a_{i+1}\ \ a'_{i+1}) \cdot a = a'_{i+1} = \pi_i \cdot \tilde{\pi} \cdot a = \pi_i \cdot a = a$, by IH and the fact that $\tilde{\pi}$ fixes all $a \in P$. If $a = a'_{i+1}$ then $(a_{i+1}\ \ a'_{i+1}) \cdot a = a_{i+1} = (\pi_i \circ \tilde{\pi})^{-1} \cdot a = a$.
Setting $\hat{\pi} = \pi_N \circ \tilde{\pi}$, for each $1 \leq j \leq N$ we thus have $\hat{\pi} \cdot a_j = a_j$ and $\hat{\pi} \cdot \ell = \tilde{\pi} \cdot \ell$. So, $\hat{\pi} \cdot X''_2 = X''_2$ and therefore $X''_2 \xrightarrow{\hat{\pi} \cdot \ell} \hat{\pi} \cdot X'_2$, that is, $X''_2 \xrightarrow{\tilde{\pi} \cdot \ell} \hat{\pi} \cdot X'_2$. Hence, by $(*)$ and determinacy, $\hat{\pi} \cdot X'_2 = \tilde{\pi} \cdot \tilde{X}_2$, so $X'_2 = \hat{\pi}^{-1} \cdot \tilde{\pi} \cdot \tilde{X}_2$. But observe that $\hat{\pi}^{-1} \circ \tilde{\pi}$ fixes all names in $P' = P \cup \nu(\ell)$ and therefore $w \in T(X'_2)$.
The above shows that $T(X'_1) \subseteq T(X'_2)$ and similarly we show $T(X'_2) \subseteq T(X'_1)$. Hence, $(X'_1, X'_2)\mathcal{R}$ and $\mathcal{R}$ is a simulation. □

Proposition C.2 *Trace equivalence is a congruence for module composition* $- \cdot -$.

Proof. Let M_1, M_2 be modules with common public names, and M a third module. By the previous lemma, it suffices to show that $M_1 \sim M_2$ implies $(M_1 \cdot M) \sim (M_2 \cdot M)$. So let us assume $M_1 \sim M_2$ with $\mathcal{R}$ a witnessing bisimulation and let $R_i : [\![M_i]\!] \otimes [\![M]\!] \to [\![M_i \cdot M]\!]$ be a τ-isomorphism, for $i = 1, 2$. We define the following relation between configurations of $[\![M_1 \cdot M]\!]$ and $[\![M_2 \cdot M]\!]$.

$$\mathcal{R}' = \{(R_1(X_1 \otimes^s_{\Pi} Y), R_2(X_2 \otimes^s_{\Pi} Y)) \mid X_i \otimes^s_{\Pi} Y \text{ reachable}, (X_1, X_2) \in \mathcal{R}\}$$

Using Proposition B.2 we can show that $\mathcal{R}'$ is a bisimulation. □