Trace semantics for polymorphic references *

Guilhem Jaber
Université Paris Diderot

Nikos Tzevelekos
Queen Mary University of London

Abstract

We introduce a trace semantics for a call-by-value language with full polymorphism and higher-order references. This is an operational game semantics model based on a nominal interpretation of parametricity whereby polymorphic values are abstracted with special kinds of names. The use of polymorphic references leads to violations of parametricity which we counter by closely recoding the disclosure of typing information in the semantics. We prove the model sound for the full language and strengthen our result to full abstraction for a large fragment where polymorphic references obey specific inhabitation conditions.

1. Introduction

Polymorphism is a prevalent feature of modern programming languages, allowing one to use generic data structures and powerful code abstractions. Reasoning with polymorphism is both challenging and rewarding: polymorphic code is bound to have uniform behaviour under different instantiations, a property known as *Strachey parametricity* [27] and formalized by Reynolds as relational parametricity[26], which in turn provides "theorems for free" [29].

Understanding the formal semantics of polymorphism amounts to capturing the parametric behaviour of code under different instantiations. This has traditionally been hard, effectively due to the requirement for a model where instantiations from within the same model are possible. As far as the full abstraction problem is concerned, the construction of fully abstract models has so far had successes in the game semantics framework. The problem has been addressed by use of *hypergames* by Hughes [9], whereby game arenas can be seen as moves which can be opened inside enclosing arenas during a play. The model of Abramsky and Jagadeesan [1] followed a different approach, namely that of fixing a universe of moves with holes, the latter representing type variables awaiting instantiation, and constructing arenas from that given pool of moves, which is effectively closed under instantiation. While these models addressed purely functional languages, in recent years a remarkable research programme by Laird [19, 18] has extended the reach of polymorphic games to languages with higher-order state.

* Research supported by the Engineering and Physical Sciences Research Council (EP/L022478/1) and the Royal Academy of Engineering. We thank T. Cuvillier, J. Rathke and the reviewers for comments and suggestions.

LICS '16, July 05 - 08, 2016, New York, NY, USA

DOI: http://dx.doi.org/10.1145/2933575.2934509

An important aspect of previous models [9, 1, 19, 18], and of the modelled languages, is the uniformity of polymorphic behaviour. However, when we move to languages with mutable references that can extrude their scope, this property can be easily broken as we see below. Thus, the modelling of languages with ML- or Java-like references presents additional complications and, as far as we are aware, is still open. Our paper addresses precisely this problem.

The language we analyse, System ReF, includes a typed lambda calculus with products, references and polymorphism. For instance, we can examine the following type.

$$\forall\alpha.\,(\mathrm{ref}\,\alpha \times \mathrm{ref}\,\mathrm{Int}) \to \alpha$$

One may be tempted to think that any term inhabiting this type is bound to return, given input (x, y), the value stored in x. Of course, this is not necessarily the case if, for example, α is instantiated with Int and x and y happen to represent the same location. The following term would take advantage of such a coincidence,

$$\Lambda\alpha.\lambda\langle x, y\rangle^{\mathrm{ref}\,\alpha\times\mathrm{ref}\,\mathrm{Int}}.\, y := 42; !x$$

and in that case return 42 regardless of what the initial value stored in x was. Thus, in this example, the given coincidence leads to an accidental interference with the returned result. More interestingly, we can instrument our example in a way that it can *discover* such coincidences and effectively deduce that $\alpha = \mathrm{Int}$. Let us write $y{+}{+}$ below for $y := !y + 1$.

$$\begin{aligned}\Lambda\alpha.\lambda\langle x, y\rangle^{\mathrm{ref}\,\alpha\times\mathrm{ref}\,\mathrm{Int}}.\ &\mathsf{let}\ x' = \mathsf{ref}\,!x,\ y' = \mathsf{ref}\,!y\ \mathsf{in}\\ &\quad y{+}{+};\ x := x';\\ &\quad \mathsf{if}\ !y' = !y\ \mathsf{then}\ (y := 42; !x)\ \mathsf{else}\ !x\end{aligned}$$

The term above increases the value of y and then restores x to its initial value x'. It then compares the value of y with its initial one y'. If these are not the same, then x and y are different locations, so the value of x is returned. If, however, the value of y has not changed then the term has successfully discovered that x and y refer to the same location, whence 42 is returned.

The above example demonstrates that uniform polymorphic behaviour can be violated through references, as differently typed variables can be instantiated with a common reference. More than that, references can disclose type instantiation information which can then be taken advantage of by a polymorphic function. In our example above the result of this disclosure was a non-parametric return value of 42, but we can imagine scenarios where a term records the references x and y that allowed it to escape uniform behaviour, and uses them as a general-use "bridge" between values of type α and Int. In fact, such devices, called *casting functions*, shall play a central role in our semantics. More generally, our modelling approach is crafted around carefully keeping track of the type information that has been leaked from the program to its environment, and viceversa, and allowing moves to be played in accordance with that information assuming that the context (the Opponent) has the epistemic power to exploit all such leaked information.

Related work Operational techniques have been designed to study languages with both polymorphism and references. Realizability models [2, 4, 5], later refined into Kripke logical relations [3, 6], use a notion of "world as heap-invariant" to model references. Environmental bisimulations have also been designed to deal with equivalence of programs in such languages [28]. While complete, these approaches partially rely on context quantifications and in particular do not directly account for the interaction between polymorphism and references, and the kind of type disclosure that the latter brings in.

Our approach follows the line of research on trace semantics for higher-order languages [14, 15, 17, 8], which in turn can be seen as an operational reformulation of game semantics [23, 11], on one hand; and of open bisimulation techniques [20, 13, 21], on the other. In this area, Jeffrey and Rathke proposed a fully abstract trace semantics for a polymorphic variant of the pi-calculus [16], which refined a previous sound model of Pierce and Sangiorgi [25]. That work is related to ours in spirit, and it already raises the intricacies involved in combining polymorphism with name equality testing. However, the apparatus of *loc. cit.* does not lend itself to ML-like languages like System ReF, as in the latter we need stronger semantic abstractions to cater for the less expressive syntactic contexts. Overall, there seems to be a greater picture behind this work and [16, 21] which remains to be exposed.

Future directions In this work we addressed Church-style polymorphism. It would be interesting to examine whether our ideas could be adapted to deal with Curry style. In doing so, we would give a semantic reading of the *value restriction*, which ensures type safety by enforcing terms of polymorphic types to be values. This, along with the study of ML-specific restrictions like rank-1 polymorphism, would bring us closer to modelling a large fragment of ML, which can be seen as a broader goal behind this work.

Moreover, our current model sets the foundation for a sound, and complete for a large collection of types, proof methods for program equivalence. Similarly to our previous work on monomorphic languages [12, 24], we aim to explore such methods and accompany them with automated, or semi-automated, equivalence checkers.

2. System ReF

We introduce System ReF, a polymorphic call-by-value λ-calculus with higher-order references. The types of System ReF are:

$$\theta, \theta' ::= \alpha \mid \mathrm{Unit} \mid \mathrm{Int} \mid \mathrm{ref}\,\theta \mid \theta \times \theta' \mid \theta \to \theta' \mid \forall\alpha.\theta \mid \exists\alpha.\theta$$

where $\alpha \in \mathrm{TVar}$, and TVar a countably infinite set of type variables. As usual, a type is closed if all its type variables α are bound. We shall call arrow and universal types ***function types***. The syntax of values v, terms M and evaluation contexts E is given in Figure 1. We assume a countably infinite set Loc of *locations* and some standard collection of binary integer operators, which we generally denote by $\oplus$. We use the following macros: $\mathsf{let}\ x = N\ \mathsf{in}\ M$ stand for $(\lambda x.M)N$; and $N; M$ means $(\lambda x.M)N$ with x fresh in M.

The typing rules for System ReF include standard rules for functions and projections, rules for integers, and rules for polymorphism and references given in Figure 2. Typing judgments are of the form $\Delta; \Sigma; \Gamma \vdash M : \theta$, where Σ is a location context, i.e. a finite partial function from locations to *closed* types; Γ a variable context; and Δ a set of type variables containing all free type variables of Γ. Given a closed evaluation context E, we write $\Delta; \Sigma \vdash E : \theta \rightsquigarrow \theta'$ when $\Delta; \Sigma; x : \theta \vdash E[x] : \theta'$. Compared to the ML type-system, we work with *Church-style* polymorphism, where type abstractions and applications are explicit. This explains why we do not need the so-called *value restriction* [30] to accommodate references.

We next proceed with the operational semantics. Closed terms are reduced using stores containing their locations. More precisely, a ***store*** is a finite partial map $S : \mathrm{Loc} \rightharpoonup \mathrm{Val}$ from locations to values. We define the following notation for stores, which we shall also be using for general partial maps:

- The empty store is written ε. Adding a new element (l, v) to a store S is written $S \cdot [l \mapsto v]$, and is defined only if $l \notin \mathrm{dom}(S)$.
- We also define $S[l \mapsto v]$, for $l \in \mathrm{dom}(S)$, as the partial function S' which satisfies $S'(l') = S(l')$ when $l' \neq l$, and $S'(l) = v$.
- The restriction of a store S to a set of locations L is written $S_{|L}$.

We write $S : \Sigma$ just if $\cdot; \Sigma; \cdot \vdash S(l) : \theta$ for all $l \in \mathrm{dom}(S)$. Given a set L of locations and a store S, we define the image of L by S, written $S^*(L)$, as $S^*(L) = \bigcup_{j \in \omega} S^j(L)$ with $S^{j+1}(L) = S^j(L) \cup \{l \in \mathrm{Loc} \mid l \text{ contained in } S(S^j(L))\}$ and $S^0(L) = L$. S is called ***closed*** just if $\mathrm{dom}(S) = S^*(\mathrm{dom}(S))$.

$$\begin{aligned} v, u ::=\ & () \mid n \mid x \mid l \mid \lambda x^{\theta}.M \mid \Lambda\alpha.M \mid \mathsf{pack}\langle\theta, v\rangle \mid \langle v, u\rangle \\ M, N ::=\ & v \mid MN \mid M\theta \mid M \oplus N \mid \mathsf{if}\ M_1\ M_2\ M_3 \mid \langle M, N\rangle \\ & \mid \pi_1(M) \mid \pi_2(M) \mid \Omega_\theta \mid \mathsf{ref}\ M \mid !M \mid M := N \\ & \mid M = N \mid \mathsf{pack}\langle\theta, M\rangle \mid \mathsf{unpack}\ M\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ N \\ E ::=\ & \bullet \mid EM \mid E\theta \mid vE \mid E\ \mathsf{op}\ M \mid v\ \mathsf{op}\ E \mid \mathsf{if}\ E\ M\ M' \\ & \mid \mathsf{ref}\ E \mid !E \mid \langle E, M\rangle \mid \langle v, E\rangle \mid \pi_1(E) \mid \pi_2(E) \\ & \mid \mathsf{pack}\langle\theta, E\rangle \mid \mathsf{unpack}\ E\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M \end{aligned}$$

Figure 1. System ReF ($n \in \mathbb{Z}$, $l \in \mathrm{Loc}$ and $\mathsf{op} \in \{\oplus, =, :=\}$).

Definition 1. The operational semantics of System ReF involves pairs (M, S) consisting of a closed term $\Delta; \Sigma; \cdot \vdash M : \theta$ and a closed store $S : \Sigma$. Its small-step rules are given in Figure 2. We write $(M, S) \Downarrow$ when $(M, S) \to^* (v, S')$ for some value v.

Remark 2. We have equipped our language with a construct performing reference equality tests. This is in accordance with, and has the same operational semantics as, reference equality tests in ML, albeit extended to arbitrary reference types. Depending on type and type inhabitation, such tests can be encoded in ML via appropriately crafted sequences of writes and reads in examined references.

We finally introduce the notion of term equivalence we examine.

Definition 3. Let Σ be closed. Two terms $\Delta; \Sigma; \Gamma \vdash M_1, M_2 : \theta$ are ***contextually equivalent***, written $\Delta; \Sigma; \Gamma \vdash M_1 \simeq M_2 : \theta$, if for all contexts C, all $\Sigma' \supseteq \Sigma$ and all closed $S : \Sigma'$ such that $\cdot; \Sigma'; \cdot \vdash C[M_i] : \mathrm{Unit}$, we have $(C[M_1], S) \Downarrow$ iff $(C[M_2], S) \Downarrow$.

3. The Semantic Model

Our trace model is constructed within nominal sets, that is, a universe embedded with atomic objects for representing locations, type variables, functions and polymorphic values. We introduce the semantic universe next and then proceed to the operational rules defining the semantics.

3.1 Semantic Universe

We define the set of ***names*** to be:

$$\mathbb{A} = \mathrm{Loc} \uplus \mathrm{TVar} \uplus \biguplus_{\theta \in \mathrm{FT}} \mathrm{Fun}_\theta \uplus \biguplus_{\alpha \in \mathrm{TVar}} \mathrm{Pol}_\alpha$$

where θ ranges over function types and each of the components in this countable union is itself a countable set. We let $\mathrm{Fun} = \biguplus_{\theta \in \mathrm{FT}} \mathrm{Fun}_\theta$ and $\mathrm{Pol} = \biguplus_{\alpha \in \mathrm{TVar}} \mathrm{Pol}_\alpha$. We range over elements of Loc by l and variants; over TVar by α, *etc*; over Fun by f, g, *etc*; and over Pol by p, *etc*.

Semantic objects feature elements of $\mathbb{A}$ as atomic entities which, moreover, can be acted upon by finite permutations of $\mathbb{A}$. A ***nominal set*** [7] is a pair $(X, *)$ of a set X along with an action $(*)$ from the set of finite component-preserving computations of $\mathbb{A}$ on the set X.[1] Given some $x \in X$, the set of names featuring in x form

[1] A finite permutation $\pi : \mathbb{A} \to \mathbb{A}$ is component-preserving simply if it preserves the partition of $\mathbb{A}$, e.g. if $d \in \mathrm{Loc}$ then $\pi(d) \in \mathrm{Loc}$.

$$\frac{(l:\theta)\in\Sigma}{\Delta;\Sigma;\Gamma\vdash l:\mathrm{ref}\,\theta}\qquad \frac{\Delta;\Sigma;\Gamma\vdash M:\mathrm{ref}\,\theta}{\Delta;\Sigma;\Gamma\vdash !M:\theta}\qquad \frac{\Delta;\Sigma;\Gamma\vdash M:\mathrm{ref}\,\theta\quad \Delta;\Sigma;\Gamma\vdash N:\theta}{\Delta;\Sigma;\Gamma\vdash M:=N:\mathrm{Unit}}\qquad \frac{\Delta;\Sigma;\Gamma\vdash M:\mathrm{ref}\,\theta\quad \Delta;\Sigma;\Gamma\vdash N:\mathrm{ref}\,\theta'}{\Delta;\Sigma;\Gamma\vdash M=N:\mathrm{Int}}$$

$$\frac{\Delta,\alpha;\Sigma;\Gamma\vdash M:\theta}{\Delta;\Sigma;\Gamma\vdash \Lambda\alpha.M:\forall\alpha.\theta}\qquad \frac{\Delta;\Sigma;\Gamma\vdash M:\forall\alpha.\theta}{\Delta;\Sigma;\Gamma\vdash M\theta':\theta\{\theta'/\alpha\}}\qquad \frac{\Delta;\Sigma;\Gamma\vdash M:\theta\{\theta'/\alpha\}}{\Delta;\Sigma;\Gamma\vdash \mathsf{pack}\langle\theta',M\rangle:\exists\alpha.\theta}\qquad \frac{\Delta;\Sigma;\Gamma\vdash M:\exists\alpha.\theta\quad \Delta,\alpha;\Sigma;\Gamma,x:\theta\vdash N:\theta'}{\Delta;\Sigma;\Gamma\vdash \mathsf{unpack}\,M\,\mathsf{as}\,\langle\alpha,x\rangle\,\mathsf{in}\,N:\theta'}$$

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
(E[(\lambda x.M)v],S) & \to & (E[M\{v/x\}],S) & (E[\Omega],S) & \to & (E[\Omega],S) & (E[l=l],S) & \to & (E[1],S)\\
(E[(\Lambda\alpha.M)\theta],S) & \to & (E[M\{\theta/\alpha\}],S) & (E[\pi_i\langle v_1,v_2\rangle],S) & \to & (E[v_i],S) & (E[l=l'],S) & \to & (E[0],S)\\
(E[\mathsf{if}\ n\ M_1\ M_2],S) & \to & (E[M_i],S) & (E[\mathsf{ref}\,v],S) & \to & (E[l],S\cdot[l\mapsto v]) & (E[!l],S) & \to & (E[S(l)],S)\\
(E[l:=v],S) & \to & (E[()],S[l\mapsto v]) & \multicolumn{6}{c}{E[\mathsf{unpack}\,(\mathsf{pack}\langle\theta,v\rangle)\,\mathsf{as}\,\langle\alpha,x\rangle\,\mathsf{in}\,M]\to E[M\{\theta/\alpha\}\{v/x\}]}
\end{array}$$

Figure 2. UP: Typing rules of System ReF (excerpt). DOWN: Operational semantics (for `if`: $i=2$ if $n=0$, otherwise $i=1$).

its *support*, written $\nu(x)$, which we stipulate to be finite. Formally, $\nu(x)$ is the smallest subset of $\mathbb{A}$ such that all permutations which elementwise fix $\nu(x)$ also fix x. We shall sometimes write $\nu_{\mathrm{C}}(x)$, for $\mathrm{C}\in\{\mathrm{L},\mathrm{T},\mathrm{F},\mathrm{P}\}$ in order to select a specific kind of names from the support of x. For instance, $\nu_{\mathrm{L}}(x)=\nu(x)\cap\mathrm{Loc}$. Using the same notation, we also write $\nu_{\mathrm{T}}(\theta)$ for the free type variables of θ. We usually write $(X,*)$ simply as X, for economy.

We next introduce our basic semantic objects, which constitute the semantic representations of syntactic values.

Definition 4. We define ***abstract values*** as:

$$\mathrm{AValues}\ni v,u ::= () \mid i \mid l \mid f \mid p \mid \alpha \mid \langle u,v\rangle$$

where $i\in\mathbb{Z}$, $l\in\mathrm{Loc}$, $f\in\mathrm{Fun}$, $p\in\mathrm{Pol}$ and $\alpha\in\mathrm{TVar}$. Note we still range over abstract values by u,v (and hope no confusion arises). We similarly set ***abstract stores*** to be finite partial maps $\mathrm{Loc}\rightharpoonup\mathrm{AValues}$.

Thus, ground values (integers, () and locations) are represented by their concrete values, and for all other types but products we employ name abstractions. This abstraction is in order either because of polymorphism in the values, or simply because function code can only be examined by querying the given function. Functions are represented by functional names, and polymorphic values by polymorphic names.

The semantics of a type θ, written $[\![\theta]\!]$, consists of pairs (v,ϕ) of an abstract value v along with a function $\phi:\nu_{\mathrm{L}}(v)\to\mathcal{P}(\mathrm{Types})$, and is given as:

$$\begin{array}{lcl}
[\![\mathrm{Unit}]\!] & = & \{((),\varepsilon)\}\\
[\![\mathrm{Int}]\!] & = & \{(n,\varepsilon)\mid n\in\mathbb{Z}\}\\
[\![\mathrm{ref}\,\theta]\!] & = & \{(l,\{(l,\theta)\})\mid l\in\mathrm{Loc}\}\\
[\![\alpha]\!] & = & \{(p,\varepsilon)\mid p\in\mathrm{Pol}_\alpha\}\\
[\![\theta\to\theta']\!] & = & \{(f,\varepsilon)\mid f\in\mathrm{Fun}_{\theta\to\theta'}\}\\
[\![\forall\alpha.\theta]\!] & = & \{(f,\varepsilon)\mid f\in\mathrm{Fun}_{\forall\alpha.\theta}\}\\
[\![\exists\alpha.\theta]\!] & = & \{(\langle\alpha',v\rangle,\phi)\mid(v,\phi)\in[\![\theta\{\alpha'/\alpha\}]\!]\}\\
[\![\theta_1\times\theta_2]\!] & = & \{(\langle v_1,v_2\rangle,\phi_1\cup\phi_2)\mid(v_i,\phi_i)\in[\![\theta_i]\!]\}
\end{array}$$

The role of ϕ is to assign types to all the locations of an abstract value. As discussed in the Introduction, though, the same location can appear with several types in the execution of a given term phrase. Hence, ϕ assigns sets of types to each location instead of a unique type. More generally, a ***typing function*** is a finite map $\phi:\mathrm{Loc}\rightharpoonup\mathcal{P}(\mathrm{Types})$. The type translation is extended to typing environments by mapping each $\Delta=\{\alpha_1,\cdots,\alpha_k\}$, $\Sigma=\{l_1:\theta_1,\cdots,l_n:\theta_n\}$ and $\Gamma=\{x_1:\theta'_1,\cdots,x_k:\theta'_k\}$ to:

$$[\![\Delta,\Sigma,\Gamma]\!]=\{\,((\vec{\alpha},\vec{l},\vec{v}),\bigcup_{i=1}^{n}[l_i\mapsto\theta_i]\cup\bigcup_{j=1}^{k}\phi_j)\mid(v_j,\phi_j)\in[\![\theta'_j]\!]\,\}.$$

Extending the syntax for $\mathrm{Fun}\cup\mathrm{Pol}$ While functional and polymorphic names are not part of the syntax of System ReF, their involvement in its semantics makes it useful to introduce them as syntax as well. We hence extend the set of values of System ReF to include $\mathrm{Fun}\cup\mathrm{Pol}$, dealing with them as typed constants.

3.2 Interaction Reduction

Traces will consist of sequences of ***moves*** enriched with abstract stores and value disclosures. Moves represent the interaction between the modelled program and its enclosing context and consist of function calls and returns. Each move comes with a polarity: P for *Player* (i.e. the program produces the move), and O for *Opponent* (the context/environment). There are four kinds of moves:

PQ. *Player Questions* are moves of the form $\overline{f\langle u\rangle}$, representing a call to a functional name $f\in\mathrm{Fun}$ with argument $u\in\mathrm{AValues}$.

OQ. *Opponent Questions* are of the form $f\langle u\rangle$, with $f\in\mathrm{Fun}$ and $u\in\mathrm{AValues}$; moreover, there are *initial* opponent questions of the form $?\langle u\rangle$ ($u\in\mathrm{AValues}$).

PA. *Player Answers* are moves of the form $\langle\overline{u}\rangle$, with $u\in\mathrm{AValues}$.

OA. *Opponent Answers*, which are of the form $\langle u\rangle$ ($u\in\mathrm{AValues}$).

On the other hand, value disclosures are partial functions ρ representing the values of polymorphic names revealed in a move. Their role will be explained in the next section.

Definition 5. A ***full move*** is a triple (m,S,ρ) of a move m, a closed abstract store S and a finite map $\rho:\mathrm{Pol}\rightharpoonup\mathrm{AValues}$. A sequence of full moves is called a ***trace***.

The trace semantics is produced via a reduction relation for open terms which only reveals the steps in the computation where there is interaction: a call or return between the term and its context. More precisely, this relation is a bipartite labelled transition system between Player and Opponent configurations, where labels are full moves, and whose main components are ***evaluation stacks*** $\mathcal{E}$, defined as either:

- *passive*, which are related to Opponent configurations and are of the shape $(E^n,\theta_n\rightsquigarrow\theta'_n)::\cdots::(E^1,\theta_1\rightsquigarrow\theta'_1)$, where each E^i is an evaluation context of type $\theta_i\rightsquigarrow\theta'_i$;
- or *active*, which are related to Player configurations and are of the form $(M,\theta)::\mathcal{E}'$, i.e. they consist of a term M of type θ and a passive stack $\mathcal{E}'$.

The empty stack is written $\Diamond$.

Definition 6. A ***configuration*** is a tuple $\langle\mathcal{E},\gamma,\phi,S,\lambda\rangle$ with:

- an evaluation stack $\mathcal{E}$, a typing function ϕ for locations, and a closed store S,
- an *environment* γ mapping names to values,
- an *ownership function* $\lambda\in(\mathbb{A}\times\{O,P\})^*$ ordering played names and mapping them to the party who has introduced them;

and which satisfies the following conditions:

- the relation $\{(a,X)\mid\lambda=\lambda_1\cdot(a,X)\cdot\lambda_2\}$ is a partial function and λ has no repetition of names
- $\mathrm{dom}(\gamma)=\{a\in\mathrm{Pol}\cup\mathrm{Fun}\cup\mathrm{TVar}\mid\lambda(a)=P\}$
- $\mathrm{dom}(\phi)=\{l\in\mathrm{Loc}\cap\mathrm{dom}(\lambda)\}\subseteq\mathrm{dom}(S)$
- for all $a\in\nu(\mathcal{E},\mathrm{cod}(S),\mathrm{cod}(\gamma))\backslash\mathrm{Loc}$, $\lambda(a)=O$

where, because of the first condition above, we write $\lambda(a)=X$ if $\lambda=\lambda_1\cdot(a,X)\cdot\lambda_2$ for some λ_1,λ_2.

In addition, we include special configurations of the form $\langle \Delta; \Sigma; \Gamma \vdash M : \theta \rangle$, one for each typed term $\Delta; \Sigma; \Gamma \vdash M : \theta$.

Thus, a configuration registers syntactic and semantic information on the execution of a term necessary to produce its traces. $\mathcal{E}$ and S are syntactic objects directly connected to the operational semantics. The other components either are of semantic nature (ϕ, λ) or bridge the semantics and the syntax (γ). In γ we record the actual values that correspond to the functional and polymorphic names and type variables that the term (i.e. P) has produced. On the other hand, λ is a name-polarity function which also keeps track of the order in which names were introduced. The last condition on λ in the above definition is especially important: it stipulates that, except for location names, all the free names that appear in the term, either directly or indirectly via γ or S, must belong to O. In other words, P cannot see the abstract values that he has provided to O during the interaction.

When the evaluation of a term $E[M]$ reaches, for example, some $E[fv]$ where f is a function name provided by the context, a move asking the context to evaluate $f(v)$ will be produced. However, since v is a syntactic value and in moves we only allow semantic entities, we need a way to pass from syntactic values to abstract ones. This is achieved as follows. To each value u of type θ, we associate the set $\mathsf{AVal}(u, \theta)$ of triples (v, γ, ϕ), where each of them represents: • a corresponding abstract value v; • an environment γ instructing the related mapping of names to values; • and a typing function ϕ recording the types used for each location in the translation. It is defined as:

$$\mathsf{AVal}(u, \iota) = \{(u, \varepsilon, \varnothing)\} \quad \text{for } \iota = \mathrm{Unit} \text{ or } \mathrm{Int} \text{ and } u \in [\![\iota]\!]$$

$$\mathsf{AVal}(l, \mathrm{ref}\,\theta) = \{(l, \varepsilon, \{(l, \mathrm{ref}\,\theta)\} \mid l \in \mathrm{Loc}\}$$

$$\mathsf{AVal}(u, \alpha) = \{(p, [p \mapsto u], \varnothing) \mid p \in \mathrm{Pol}_\alpha\} \cup \{(u, \varepsilon, \varnothing) \mid u \in \mathrm{Pol}_\alpha\}$$

$$\mathsf{AVal}(u, \theta) = \{(f, [f \mapsto u], \varnothing) \mid f \in \mathrm{Fun}_\theta\} \quad \text{for } \theta \text{ functional}$$

$$\mathsf{AVal}(\langle u_1, u_2 \rangle, \theta_1 \times \theta_2)$$
$$= \{(\langle v_1, v_2 \rangle, \gamma_1 \cdot \gamma_2, \phi_1 \cup \phi_2) \mid (v_i, \gamma_i, \phi_i) \in \mathsf{AVal}(u_i, \theta_i)\}$$

$$\mathsf{AVal}(\langle \theta', u \rangle, \exists \alpha.\theta)$$
$$= \{(\langle \alpha', v \rangle, \gamma \cdot [\alpha' \mapsto \theta'], \phi) \mid (v, \gamma, \phi) \in \mathsf{AVal}(u, \theta\{\alpha'/\alpha\})\}$$

For uniformity, it makes sense to view types as values of special "universe" type $\mathcal{U}$ and set $\mathsf{AVal}(\theta, \mathcal{U}) = \{(\alpha, [\alpha \mapsto \theta], \varnothing) \mid \alpha \in \mathrm{TVar}\}$. By abuse of notation, we shall use u and variants to range over values, abstract values and types when utilising the notation presented next. Given a functional type θ and some u, we let the *argument* and *return type* of θ be:

$$\begin{array}{llll} \mathsf{arg}(\theta' \to \theta) & = \theta' & \mathsf{arg}(\forall \alpha.\theta) & = \mathcal{U} \\ \mathsf{ret}_u(\theta' \to \theta) & = \theta & \mathsf{ret}_u(\forall \alpha.\theta) & = \theta\{u/\alpha\} \end{array}$$

with the last expression above being well-defined only if u is a type.

Finally, in a similar fashion that AVal allows us to move from concrete values to abstract ones, the operator AStore takes us from stores to abstractions thereof. That is, for each store S and typing function ϕ, the set $\mathsf{AStore}(S, \phi,)$ consists of triples of the form (S', γ', ϕ') where: • S' is an abstraction of S according to the type information in ϕ; • γ' is the mapping of the fresh abstract names of S' to their concrete values; • and ϕ' is the type information for any locations in the codomain of S'. The formal definition in the case where ϕ is single-valued is given as follows. We postpone the definition for general ϕ to Section 4.

$$\mathsf{AStore}(S, \phi) = \bigodot_{l \in \mathrm{dom}(S)} \{([l \mapsto v], \gamma', \phi') \mid (v, \gamma', \phi') \in \mathsf{AVal}(S(l), \phi(l))\}$$

Here $\odot$ is the pointwise concatenation of sets of triples (S, γ, ϕ), defined as $X_1 \odot X_2 = \{(S_1 \cdot S_2, \gamma_1 \cdot \gamma_2, \phi_2 \cup \phi_2) \mid (S_i, \gamma_i, \phi_i) \in X_i, i \in \{1, 2\}\}$, and $\bigodot_{i \in \varnothing} X_i = \{(\varepsilon, \varepsilon, \varnothing)\}$. A similar notion is used for producing abstract stores where only typing information (and no concrete store) is defined as follows.

$$\mathsf{S}[\![\phi]\!] = \bigodot_{l \in \mathrm{dom}(\phi)} \{([l \mapsto v], \phi') \mid (v, \phi') \in [\![\phi(l)]\!]\}$$

This is used for determining what stores can O play.

We now give the definition of our trace semantics. Note that, for syntactic objects Z and (e.g. type) environments δ, we write $Z\{\delta\}$ for the result of recursively applying δ in Z as a substitution.

Definition 7 (Trace Semantics)**.** We call ***Interaction Reduction*** the system generated by the rules in Figure 3. Given a configuration C, we let $\mathrm{Tr}(C)$ be the set of all traces produced from C. Terms are translated by setting

$$[\![\Delta; \Sigma; \Gamma \vdash M : \theta]\!] = \mathbf{comp}(\mathrm{Tr}\langle \Delta; \Sigma; \Gamma \vdash M : \theta \rangle)$$

for each typed term $\Delta; \Sigma; \Gamma \vdash M : \theta$, where $\mathbf{comp}$ selects the ***complete traces***, that is those traces where the number of answers is greater or equal to the number of questions.

In the rest of this section we explain the reduction rules and their conditions, apart from conditions P* and O* which concern type disclosure and are relegated to the next section. For the same reason, we also assume that typing functions ϕ are always single-valued and disregard any indexing with κ used in the rules (κ's are cast functions).

Internal (INT) This rule dictates that the interaction reduction includes the operational semantics of System ReF as long as internal computation steps are concerned, i.e. ones that do not involve external functions.

P-Question (PQ) This rule describes the move occurring when an external function call is reached. Thus, in order for P to provide the value (say) u and store S, he first needs to abstract it to v by hiding away all private code under fresh names. These will be the names put in λ', along with any new location names revealed in the store S' to be played. Since this is a P-move then, all names in λ' are owned by P (P1). In turn, S' is the restriction of S to public locations, again elevated to its abstraction. These abstractions result in new $\gamma' = \gamma \cdot \gamma_v \cdot \gamma_S$ and $\phi' = \phi \cup \phi_v \cup \phi_S$ (P1). Note that the λ component of a configuration enlists the *public* names of a trace, i.e. those explicitly played in moves. Hence, P3 stipulates that the locations included in the store S' are precisely the ones reachable in S from the names in λ and any names in v (put otherwise, name privacy is imposed). Finally, P2 dictates that any functional or type variable names played in the move must be fresh (as they represent abstractions of concrete values). Similarly, every polymorphic name played of type α, with α of own polarity, must be fresh. If, on the other hand, α belongs to O, then P can only play old polymorphic names of that type (P4).

P-Answer (PA) In this case, a final value is reached and returns, with similar conditions applied.

O-Question (OQ) When it is the context's turn to play, one option is for O to call one of the functions provided by P. The rule looks very similar to the P-Question, yet it differs in one important point: while O plays v and S', what is fed instead to the configuration is v where all its P polymorphic and functional names have been replaced by their actual values (i.e. $v\{\gamma\}$)[2] and the same goes for the abstract store S'. This is enforced by the use of $\widetilde{v}$ instead of v and is due to the fact that P knows the actual values of these names, and therefore they should not remain abstract to him. Another difference is the freedom to build S', which nonetheless stipulates that O cannot guess any locations from S unless the latter were already public. Finally, observe in O1 the single-played restriction on fresh polymorphic, type or function names: as each

[2] we also substitute via ρ, but this we discuss in the next section.

(INT) $\langle (M,\theta) :: \mathcal{E},\gamma,\phi,S,\lambda\rangle \longrightarrow \langle (M',\theta) :: \mathcal{E},\gamma,\phi,S',\lambda\rangle$, given $(M,S)\to(M',S')$.

(PA) $\langle (u,\theta) :: \mathcal{E},\gamma,\phi,S,\lambda\rangle \xrightarrow{\langle\overline{v}\rangle,S',\rho} \langle \mathcal{E},\gamma\cdot\gamma',\phi\cup\phi',S,\lambda\cdot\lambda'\rangle$, given $(v,\gamma_v,\phi_v)\in \mathsf{AVal}(u,\theta)_\kappa$.

(PQ) $\langle (E[f\,u],\theta) :: \mathcal{E},\gamma,\phi,S,\lambda\rangle \xrightarrow{\bar{f}\langle v\rangle,S',\rho} \langle (E,\theta'\rightsquigarrow\theta) :: \mathcal{E},\gamma\cdot\gamma',\phi\cup\phi',S,\lambda\cdot\lambda'\rangle$,
given $f\in \mathrm{Fun}_{\theta_f}$ with $\lambda(f)=O$ and $(v,\gamma_v,\phi_v)\in \mathsf{AVal}(u,\mathsf{arg}(\theta_f))_\kappa$, $\theta'=\mathsf{ret}_v(\theta_f)$.

(OA) $\langle (E,\theta'\rightsquigarrow\theta) :: \mathcal{E},\gamma,\phi,S,\lambda\rangle \xrightarrow{\langle v\rangle,S',\rho} \langle (\widetilde{E}[\widetilde{v}],\theta) :: \widetilde{\mathcal{E}},\widetilde{\gamma},\phi\cup\phi',\widetilde{S}[\widetilde{S'}],\lambda\cdot\lambda'\rangle$, given $(v,\phi_v)\in[\![\theta']\!]_\kappa$.

(OQ) $\langle \mathcal{E},\gamma,\phi,S,\lambda\rangle \xrightarrow{f\langle v\rangle,S',\rho} \langle (\widetilde{u}\,\widetilde{v},\theta) :: \widetilde{\mathcal{E}},\widetilde{\gamma},\phi\cup\phi',\widetilde{S}[\widetilde{S'}],\lambda\cdot\lambda'\rangle$
given $f\in\mathrm{Fun}_{\theta'}$ with $\lambda(f)=P$ and $(v,\phi_v)\in[\![\mathsf{arg}(\theta')]\!]_\kappa$, $\theta=\mathsf{ret}_v(\theta')$ and $\gamma(f)=u$.

(INI) $\langle \Delta;\Sigma;\Gamma\vdash M:\theta\rangle \xrightarrow{?\langle v\rangle,S',\rho} \langle (M\overrightarrow{\{\widetilde{u}/x\}},\theta),\varepsilon,\phi',\widetilde{S'},\lambda'\rangle$, given $\mathrm{dom}(\Gamma)=\{x_1,\cdots,x_n\}$, $(v,\phi_v)\in[\![\Delta,\Sigma,\Gamma]\!]$ and $v=(\vec{\alpha},\vec{l},\vec{u})$.

$\widetilde{Z}$	Above, $\widetilde{Z}=Z\{\rho\}\{\gamma\}$, if Z a term, context or stack, and $\widetilde{Z}=\{(z,\widetilde{Z(z)})\mid z\in\mathrm{dom}(Z)\}$ if Z a map into terms.
P1	$\lambda'=\{(a,P)\mid a\in\nu(v,S',\rho)\wedge a\notin\nu(\lambda)\}$, $\phi'=\phi_v\cup\phi_S\cup\phi_\rho$ and $\gamma'=\gamma_v\cdot\gamma_S\cdot\gamma_\rho$
P2	for all $f\in\nu_{\mathrm{F}}(S',v,\rho)$, $f\notin\nu(\lambda)$ and, for all $\alpha\in\nu_{\mathrm{T}}(S',v,\rho)$, $\alpha\notin\nu(\lambda)$
P3	$\nu_{\mathrm{L}}(\lambda')=S^*(\nu_{\mathrm{L}}(v,\rho,\lambda))$ and $(S',\gamma_S,\phi_S)\in\mathsf{AStore}(S_{\lvert\nu_{\mathrm{L}}(\lambda')},\phi)$
P4	for all $p\in\nu_{\mathrm{P}}(S',v,\rho)$ with $p\in\mathrm{Pol}_\alpha$, $\lambda(\alpha)=P$ iff $p\notin\nu(\lambda)$
P*	$(\rho,\gamma_\rho,\phi_\rho)\in\mathsf{AEnv}((\gamma\cdot\gamma_v\cdot\gamma_s)_{\lvert\mathrm{Pol}})_{\kappa,\kappa'}$ where $\kappa=\mathsf{Cast}(\phi)$ and $\kappa'=\mathsf{Cast}(\phi\cup\phi')$, with $\phi\cup\phi'$ valid.
O1	$\lambda'=\{(a,O)\mid a\in\nu(v,S',\rho)\wedge a\notin\nu(\lambda)\}$ $\phi'=\phi_v\cup\phi_S\cup\phi_\rho$ and each $a\in\nu(\lambda')\backslash\mathrm{Loc}$ is single-played in (v,S',ρ)
O2	for all $f\in\nu_{\mathrm{F}}(S',v,\rho)$, $f\notin\nu(\lambda)$ and for all $\alpha\in\nu_{\mathrm{T}}(S',v,\rho)$, $\alpha\notin\nu(\lambda)$
O3	S' closed, $\nu_{\mathrm{L}}(v,\rho)\subseteq\mathrm{dom}(S')=S'^*(\nu_{\mathrm{L}}(v,\rho,\lambda))$, $\mathrm{dom}(S')\cap\mathrm{dom}(S)=\nu_{\mathrm{L}}(\lambda)$ and $(S',\phi_S)\in\mathsf{S}[\![\phi']\!]$
O4	for all $p\in\nu_{\mathrm{P}}(S',v,\rho)$ with $p\in\mathrm{Pol}_\alpha$, $\lambda(\alpha)=O$ iff $p\notin\nu_{\mathrm{P}}(\lambda)$
O*	$(\rho,\phi_\rho)\in\mathsf{E}[\![\xi]\!]_{\kappa,\kappa'}$ where $\xi=\{p\in\mathrm{Pol}_\alpha\mid\lambda\cdot\lambda'(p)=O\}$, $\kappa=\mathsf{Cast}(\phi)$ and $\kappa'=\mathsf{Cast}(\phi\cup\phi')$ with $\phi\cup\phi'$ valid.

Figure 3. Interaction Reduction. Rules (PQ),(PA) satisfy conditions P1-P4 and P*, while (OQ),(OA) satisfy O1-O4 and O*. Rule (INI) satisfies O1, O3 and O* (taking $S=\varepsilon$, $\phi=\varnothing$ and $\lambda=\varepsilon$).

such introduced name has the purpose of abstracting some concrete value or type played, every such name should be distinct (and fresh).[3] This condition is implicitly imposed in P1 as well, via the domain disjointness requirements in the definition of γ'.

O-Answer (OA) On the other hand, a context can also return with a value, with similar conditions applied.

Initial move (INI) Initial moves are special O-Questions. In order for the interaction to commence, O needs to provide the context, that is, the values corresponding to the typing environment Δ,Σ,Γ.

Let us look at a couple of examples.

Example 8. Consider the term $v\equiv\Lambda\alpha.\lambda x:\alpha\times\alpha.\ \pi_1(x)$ of type $\theta=\forall\alpha.\,\alpha\times\alpha\to\alpha$. A characteristic trace of v is $?\langle\rangle\cdot\langle\bar{g}\rangle\cdot g\langle\alpha'\rangle\cdot\langle\bar{f}\rangle\cdot f\langle p_1,p_2\rangle\cdot\langle\bar{p_1}\rangle$, produced as follows (we omit empty stores and ρ's).

$\langle\cdot;\cdot;\cdot\vdash v:\theta\rangle \xrightarrow{?\langle\rangle} \langle(v,\theta),\varepsilon,\varnothing,\varepsilon,\varepsilon\rangle$ $(\theta=\forall\alpha.\,\alpha\times\alpha\to\alpha)$
$\xrightarrow{\langle\bar{g}\rangle} \langle\Diamond,\gamma_1,\varnothing,\varepsilon,\lambda_1\rangle$ $(\gamma_1=[g\mapsto v],\lambda_1=(g,P))$
$\xrightarrow{g\langle\alpha'\rangle} \langle(v\,\alpha',\theta'),\gamma_1,\varnothing,\varepsilon,\lambda_2\rangle$ $(\theta'=\alpha'\times\alpha'\to\alpha',\lambda_2=\lambda_1\cdot(\alpha',O))$
$\to\langle(v',\theta'),\gamma_1,\varnothing,\varepsilon,\lambda_2\rangle$ $(v'\equiv\lambda x{:}\alpha'\times\alpha'.\,\pi_1(x))$
$\xrightarrow{\langle\bar{f}\rangle} \langle\Diamond,\gamma_2,\varnothing,\varepsilon,\lambda_3\rangle$ $(\gamma_2=\gamma_1\cdot[f\mapsto v'],\lambda_3=\lambda_2\cdot(f,P))$
$\xrightarrow{f\langle p_1,p_2\rangle} \langle(v'\langle p_1,p_2\rangle,\alpha'),\gamma_2,\varnothing,\varepsilon,\lambda_4\rangle$ $(\lambda_4=\lambda_3\cdot(p_1,O)(p_2,O))$
$\to^*\langle(p_1,\alpha'),\gamma_2,\varnothing,\varepsilon,\lambda_4\rangle \xrightarrow{\langle\bar{p_1}\rangle}\langle\Diamond,\gamma_2,\varnothing,\varepsilon,\lambda_4\rangle$

Informally, after the initial move is played, the term is already evaluated to a function of type $\forall\alpha.\,\alpha\times\alpha\to\alpha$ and so P plays the move $\langle\bar{g}\rangle$ with $g\in\mathrm{Fun}_{\forall\alpha.\,\alpha\times\alpha\to\alpha}$. At that point, the environment (O) may wish to interrogate g, supplying a type variable α' which is an abstraction of any type instantiation the environment may have chosen. Such a question would be of the form $g\langle\alpha'\rangle$. To the latter, P replies with a functional name f, via the move $\langle\bar{f}\rangle$, of type $(\alpha'\times\alpha')\to\alpha'$. Next, O decides to also interrogate f, say on input $\langle 4,2\rangle$. This translates to the move $f\langle p_1,p_2\rangle$, where now $p_1\mapsto 4$ and $p_2\mapsto 2$ for O. The trace concludes with P replying $\langle\bar{p_1}\rangle$, which is the return value of the first projection on $\langle p_1,p_2\rangle$.

Example 9. Let us take $v\equiv\lambda x:(\forall\alpha.\,\alpha\to\alpha).\,x\,\mathrm{Int}\,3+x\,\mathrm{Int}\,5$ of type $\theta=(\forall\alpha.\alpha\to\alpha)\to\mathrm{Int}$. A characteristic trace of v is $?\langle\rangle\langle\bar{f}\rangle\cdot f\langle g\rangle\cdot\bar{g}\langle\alpha_1\rangle\cdot\langle g_1\rangle\cdot\bar{g_1}\langle p_1\rangle\cdot\langle p_1\rangle\cdot\bar{g}\langle\alpha_2\rangle\cdot\langle g_2\rangle\cdot\bar{g_2}\langle p_2\rangle\cdot\langle p_2\rangle\cdot\langle\bar{8}\rangle$ and can be produced by the following interaction.

$\langle\cdot;\cdot;\cdot\vdash v:\theta\rangle \xrightarrow{?\langle\rangle} \langle(v,\theta),\varepsilon,\varnothing,\varepsilon,\varepsilon\rangle$
$\xrightarrow{\langle\bar{f}\rangle} \langle\Diamond,\gamma_1,\varnothing,\varepsilon,\lambda_1\rangle$ $(\gamma_1=[f\mapsto v],\lambda_1=(f,P))$
$\xrightarrow{f\langle g\rangle} \langle(vg,\mathrm{Int}),\gamma_1,\varnothing,\varepsilon,\lambda_2\rangle$ $(\lambda_2=\lambda_1\cdot(g,O))$
$\to\langle(g\,\mathrm{Int}\,3+g\,\mathrm{Int}\,5,\mathrm{Int}),\gamma_1,\varnothing,\varepsilon,\lambda_2\rangle$ $(\gamma_2=\gamma_1\cdot[\alpha_1\mapsto\mathrm{Int}])$
$\xrightarrow{\bar{g}\langle\alpha_1\rangle} \langle(\bullet\,3+g\,\mathrm{Int}\,5,\alpha_1\to\alpha_1\rightsquigarrow\mathrm{Int}),\gamma_2,\varnothing,\varepsilon,\lambda_3\rangle$ $(\lambda_3=\lambda_2\cdot(\alpha_1,P))$
$\xrightarrow{\langle g_1\rangle} \langle(g_1\,3+g\,\mathrm{Int}\,5,\mathrm{Int}),\gamma_2,\varnothing,\varepsilon,\lambda_4\rangle$ $(\lambda_4=\lambda_3\cdot(g_1,O))$
$\xrightarrow{\bar{g_1}\langle p_1\rangle} \langle(\bullet+g\,\mathrm{Int}\,5,\alpha_1\rightsquigarrow\mathrm{Int}),\gamma_3,\varnothing,\varepsilon,\lambda_5\rangle$ $(\lambda_5=\lambda_4\cdot(p_1,P))$
$\xrightarrow{\langle p_1\rangle} \langle(3+g\,\mathrm{Int}\,5,\mathrm{Int}),\gamma_3,\varnothing,\varepsilon,\lambda_5\rangle$ $(\gamma_3=\gamma_2\cdot[p_1\mapsto 3])$
$\xrightarrow{\bar{g}\langle\alpha_2\rangle} \langle(3+\bullet\,5,\alpha_2\to\alpha_2\rightsquigarrow\mathrm{Int}),\gamma_4,\varnothing,\varepsilon,\lambda_6\rangle$ $(\lambda_6=\lambda_5\cdot(\alpha_2,P))$
$\xrightarrow{\langle g_2\rangle} \langle(3+g_2\,5,\mathrm{Int}),\gamma_4,\varnothing,\varepsilon,\lambda_7\rangle$ $(\gamma_4=\gamma_3\cdot[\alpha_2\mapsto\mathrm{Int}])$
$\xrightarrow{\bar{g_2}\langle p_2\rangle} \langle(3+\bullet,\alpha_2\rightsquigarrow\mathrm{Int}),\gamma_5,\varnothing,\varepsilon,\lambda_8\rangle$ $(\lambda_7=\lambda_6\cdot(g_2,O))$
$\xrightarrow{\langle p_2\rangle} \langle(3+5,\mathrm{Int}),\gamma_5,\varnothing,\varepsilon,\lambda_8\rangle$ $(\gamma_5=\gamma_4\cdot[p_2\mapsto 5],\lambda_8=\lambda_7\cdot(p_2,P))$
$\to\langle(8,\mathrm{Int}),\gamma_5,\varnothing,\varepsilon,\lambda_8\rangle \xrightarrow{\langle\bar{8}\rangle}\langle\Diamond,\gamma_5,\varnothing,\varepsilon,\lambda_8\rangle$

Notice that p_1,p_2 are of different type, respectively α_1 and α_2. As an exercise, we invite the reader to verify that the term $v'\equiv\lambda x:(\forall\alpha.\,\alpha\to\alpha).\,\mathsf{let}\ h=x\,\mathrm{Int}\ \mathsf{in}\ h\,3+h\,5$ of the same type θ produces the trace $?\langle\rangle\langle\bar{f}\rangle\cdot f\langle g\rangle\cdot\bar{g}\langle\alpha'\rangle\cdot\langle g'\rangle\cdot\bar{g'}\langle p_1\rangle\cdot\langle p_1\rangle\cdot\bar{g'}\langle p_2\rangle\cdot\langle p_1\rangle\cdot\langle\bar{6}\rangle$. The latter behaviour can be triggered by a context which uses local state to record polymorphic values of older calls:

$$C\equiv\bullet\big(\Lambda\alpha.\,\mathsf{let}\,y=\mathsf{ref}\,(\lambda\_.\Omega_\alpha)\,\mathsf{in}\,\mathsf{let}\,z=\mathsf{ref}\,0\,\mathsf{in}\ \lambda x:\alpha.\,\mathsf{if}\,(!z)\,(!y())\,(z:=!z+1;y:=(\lambda\_.x);x)\big)$$

[3] Formally, a move (m,S,ρ) is said to *single-play* a name $a\in\mathbb{A}\backslash\mathrm{Loc}$ if m is equal to $\bar{f}\langle v\rangle$, $f\langle v\rangle$, $\langle\bar{v}\rangle$ or $\langle v\rangle$ (for some f) with $a\in\nu(v,S,\mathrm{cod}(\rho))$ and there is only one occurrence of a in (v,S,ρ).

4. Type Disclosure, Casts and *-Conditions

As already discussed in the Introduction, the existence of references can be used to the advantage of a program in order to break parametricity. This is done by discovering variables of different reference types which, upon execution, end up with the same concrete location. Once such an *aliased pair* has been identified, of type say $\mathrm{ref}\,\theta_1, \mathrm{ref}\,\theta_2$, then a casting function between θ_1 and θ_2 is readily available. For instance, if the two variables are $x_i : \mathrm{ref}\,\theta_i$, here is a casting function from θ_1 to θ_2:

$$\mathsf{cast}_1 \;\equiv\; \lambda z_1 : \theta_1.\; x_1 := z_1;\; !x_2 : \theta_1 \to \theta_2$$

Clearly, if the same location l flows in x_1 and x_2 then we obtain $\mathsf{cast}_1\{l/x_1, l/x_2\}$ which casts indeed as designed. The reader may wonder under what circumstances can the same location be passed to variables of different types. This can be achieved, for instance, by a context:

$$C \equiv \mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ (\Lambda\alpha.\, \lambda y_1 : \mathrm{ref}\,\alpha.\, \lambda y_2 : \mathrm{ref}\,\mathrm{Int}.\, \bullet)\,\mathrm{Int}\,x\,x$$

whereby $\theta_1 = \alpha$ and $\theta_2 = \mathrm{Int}$.

These considerations bring about *type disclosure*, which we examine next in detail. We conclude the prelude to this section with some interesting equivalence examples/non-examples, left as a quiz for the reader.

Example 10. Suppose $f : (\mathrm{ref}\,\mathrm{Int} \times \mathrm{ref}\,\mathrm{Int}) \to \mathrm{Unit}$, $g : \forall\alpha.\, \mathrm{ref}\,\alpha \to \mathrm{ref}\,\alpha$ and $h : \forall\alpha, \alpha'.(\mathrm{ref}\,(\alpha' \to \alpha) \times \mathrm{ref}\,(\alpha' \to \mathrm{Int}) \times \alpha) \to \alpha$.

1. $\mathsf{let}\,x, y = \mathrm{ref}\,0\,\mathsf{in}\,f(x, y); \mathsf{let}\,u = g\,\mathrm{Int}\,x\ \mathsf{in\ if}\ (u = y)\ 1\ 2$
   $\cong?\ \mathsf{let}\,x, y = \mathrm{ref}\,0\,\mathsf{in}\,f(x, y); \mathsf{let}\,u = g\,\mathrm{Int}\,x\ \mathsf{in\ if}\ (u = y)\ 3\ 2$
2. $\mathsf{let}\,x = \mathrm{ref}\,(\lambda y.1)\,\mathsf{in\ let}\,u = h\,\mathrm{Int}\,\mathrm{Int}\,(x, x, 0)\,\mathsf{in\ if}\ u\ 1\ 2$
   $\cong?\ \mathsf{let}\,x = \mathrm{ref}\,(\lambda y.1)\,\mathsf{in\ let}\,u = h\,\mathrm{Int}\,\mathrm{Int}\,(x, x, 0)\,\mathsf{in\ if}\ u\ 3\ 2$

4.1 Type disclosure and casts

Type disclosure is the result of the same location appearing in several positions in the code, each expecting some different type. In such cases, we need to associate in our semantics a set of types to each location, employing the non-unicity of typing functions ϕ. In order to restrict the behaviour of O in the interaction to plausible computations, we shall impose some validity conditions to ϕ: after all, not all types can be instantiations of the same type variable (for instance, $\phi(l) = \{\mathrm{ref}\,\mathrm{Int}, \mathrm{ref}\,\mathrm{Unit}\}$ is not allowed).

Validity is also dependent on precedence of type variables in the trace: a recent type variable cannot be instantiating one which has appeared before it in the trace. We define a partial relation $\leq_\Phi$ on types, indexed by an *ordered* set Φ of type variables, as:

$$\frac{}{\theta \leq_\Phi \theta} \qquad \frac{\theta_1 \leq_\Phi \theta_2 \leq_\Phi \theta_3}{\theta_1 \leq_\Phi \theta_3} \qquad \frac{\nu_{\mathrm{T}}(\theta) <_\Phi \alpha}{\theta \leq_\Phi \alpha} \qquad \frac{\theta \leq_\Phi \theta'}{\mathrm{ref}\,\theta \leq_\Phi \mathrm{ref}\,\theta'}$$

$$\frac{\theta_1 \leq_\Phi \theta_1'\ \ \theta_2 \leq_\Phi \theta_2'}{\theta_1 \times \theta_2 \leq_\Phi \theta_1' \times \theta_2'} \qquad \frac{\theta \leq_\Phi \theta'\ \ \alpha \notin \Phi}{Q\alpha.\theta \leq_\Phi Q\alpha.\theta'} \qquad \frac{\theta_1 \leq_\Phi \theta_1'\ \ \theta_2 \leq_\Phi \theta_2'}{\theta_1 \to \theta_2 \leq_\Phi \theta_1' \to \theta_2'}$$

for $Q = \exists, \forall$ and with $\nu_{\mathrm{T}}(\theta) <_\Phi \alpha$ meaning that all $\alpha' \in \nu_{\mathrm{T}}(\theta)$ are before α in Φ. Let us fix some Φ for the next definition.

Definition 11. A typing function ϕ is said to be ***valid*** if for all $l \in \mathrm{dom}(\phi)$ there exists a type θ_0 such that $\theta_0 \leq_\Phi \theta$ for all $\theta \in \phi(l)$.

In the sequel we will be using a very specific set Φ, which we shall be leaving implicit. For any configuration C with components λ and ϕ, we say that ϕ is valid if it is so with respect to the ordered set Φ_λ of type variables obtained from λ: $\Phi_\lambda = \pi_1(\lambda) \restriction \mathrm{TVar}$.

As type instantiations are noticed during an interaction, the two parties can start forming cast functions to move between types. We introduce the notion of ***cast relations*** κ, which are simply relations over types. The fact that $(\theta, \theta') \in \kappa$ means that we can cast values of type θ to θ'.

Casts yield other casts. For example, a cast from $\theta_1 \times \theta_2$ to $\theta_1' \times \theta_2'$ yields subcasts from θ_1 to θ_1', and from θ_2 to θ_2'.[4] We formalise this as follows. Given a cast relation κ, we define its closure $\bar{\kappa}$ by:

$$\frac{(\theta, \theta') \in \kappa}{(\theta, \theta') \in \bar{\kappa}} \qquad \frac{}{(\theta, \theta) \in \bar{\kappa}} \qquad \frac{(\theta, \theta'') \in \bar{\kappa}\ \ (\theta'', \theta') \in \bar{\kappa}}{(\theta, \theta') \in \bar{\kappa}} \qquad \frac{(\mathrm{ref}\,\theta, \mathrm{ref}\,\theta') \in \bar{\kappa}}{(\theta, \theta') \in \bar{\kappa}}$$

$$\frac{(\theta_1, \theta_1') \in \bar{\kappa}\ \ (\theta_2, \theta_2') \in \bar{\kappa}}{(\theta_1 \times \theta_2, \theta_1' \times \theta_2') \in \bar{\kappa}} \qquad \frac{(\theta_1 \times \theta_2, \theta_1' \times \theta_2') \in \bar{\kappa}}{(\theta_1, \theta_1') \in \bar{\kappa}} \qquad \frac{(\theta_1 \times \theta_2, \theta_1' \times \theta_2') \in \bar{\kappa}}{(\theta_2, \theta_2') \in \bar{\kappa}}$$

$$\frac{(\theta_1', \theta_1) \in \bar{\kappa}\ \ (\theta_2, \theta_2') \in \bar{\kappa}}{(\theta_1 \to \theta_2, \theta_1' \to \theta_2') \in \bar{\kappa}} \qquad \frac{(\theta_1' \to \theta_2, \theta_1 \to \theta_2') \in \bar{\kappa}}{(\theta_1, \theta_1') \in \bar{\kappa}} \qquad \frac{(\theta_1 \to \theta_2, \theta_1' \to \theta_2') \in \bar{\kappa}}{(\theta_2, \theta_2') \in \bar{\kappa}}$$

$$\frac{(\theta, \theta') \in \bar{\kappa}\ \ \alpha \notin \nu(\kappa)}{(Q\alpha.\theta, Q\alpha.\theta') \in \bar{\kappa}} \qquad \frac{(Q\alpha.\theta, Q\alpha.\theta') \in \bar{\kappa}\ \ \chi(\alpha, \theta, \theta')}{(\theta\{\theta_0/\alpha\}, \theta'\{\theta_0/\alpha\}) \in \bar{\kappa}}\ (*)$$

for $Q = \exists, \forall$, where $\chi(\alpha, \theta, \theta')$ means that α does not appear in the scope of a ref constructor in θ, θ'. Notice that all the rules are going in both directions, but the one on ref types. Indeed, being able to cast from θ to θ' does not imply we can cast from $\mathrm{ref}\,\theta$ to $\mathrm{ref}\,\theta'$. This observation allows us to see that the terms of Example 10 (1) are equivalent despite the type disclosure (cf. Section 4.3).

We can now define the cast relation $\mathsf{Cast}(\phi)$ related to a typing function ϕ. We can show that, for any valid typing function ϕ, $\mathsf{Cast}(\phi)$ is a valid cast relation.

Definition 12. Given a typing function ϕ, its associated cast relation $\mathsf{Cast}(\phi)$ is the closure of $\{(\theta, \theta') \mid \exists l.\, \theta, \theta' \in \phi(l)\}$.

Given a cast relation κ and a type θ, we let

$$\min(\kappa(\theta)) = \{\theta' \in \kappa(\theta) \mid \forall \theta'' \in \kappa(\theta).\, \theta'' \leq \theta' \implies \theta'' = \theta'\}$$

be the set of minimal types of $\kappa(\theta)$. Because the closure rules above are not reversible on ref types, this set is not in general a singleton (e.g. $\min(X) = X$ for $X = \{\mathrm{ref}\,(\alpha \times \mathrm{Int}), \mathrm{ref}\,(\mathrm{Int} \times \alpha')\}$). This means that a type θ can have several minimal types in its cast class, and each of them needs to be taken in to account when computing abstract values to be played in a move. Hence, minimal types are central to the (full) definitions of AVal, AStore, etc.

4.2 The starred conditions

We next look at the use of environments ρ and the conditions O* and P* which govern type disclosure in the interaction reduction.

Each move (m, S, ρ) played in an interaction has the potential to reveal type information. Looking at the reduction rules, in particular, we see that such a move can enlarge the current typing function ϕ to a (valid) superset $\phi \cup \phi'$: this is due to the fact that locations l which up until now had types $\phi(l)$ are put in positions which expect types $\theta \notin \phi(l)$ (e.g. in return position of some $f \in \mathrm{Fun}_{\theta' \to \theta}$). This leads to a corresponding increase in the cast capabilities to $\kappa' = \mathsf{Cast}(\phi \cup \phi')$. Cast capabilities, though, may reveal the values behind polymorphic names: for instance, if we are able to form a cast from α to Int, we can go back to an old $p \in \mathbb{A}_\alpha$, cast it as an integer and read its value. This decoding capability is the reason behind the presence of ρ in the move: ρ contains all those polymorphic names p whose value is being revealed (indirectly, via casts) through the current move, along with the revealed values.

The way polymorphic values are revealed is governed by conditions P* and O*. The former stipulates that, given the old cast relation κ, the new casting κ' is the one we obtain via the updated typing function $\phi \cup \phi'$. Moreover, as explained above, each concrete value $\gamma(p)$ of a polymorphic name p needs to be partially revealed. The degree to which the codomain of $\gamma_{|\mathrm{Pol}}$ will be revealed is determined by the function AEnv. That is, $\mathsf{AEnv}(\gamma_{|\mathrm{Pol}})_{\kappa,\kappa'}$ comprises a new abstract environment $(\rho, \gamma_\rho, \phi_\rho)$ for these newly revealed values, that is moreover unique up to permutation of fresh names. The

[4] Assuming θ_1 and θ_2 are inhabited types.

first component (ρ) is the map from polymorphic names to their revealed values. The other two components record the locations types (ϕ_ρ) and value abstraction (γ_ρ) occurring via this disclosure. In the case of O*, a similar disclosure occurs, only that this time there is no γ to guide the revealed values; rather, O supplies the disclosure in a non-deterministic fashion.

The definition of AEnv and its O-counterpart are given below,

$$\mathsf{AEnv}(\gamma)_{\kappa,\kappa'} = \bigodot_{\substack{p\in\mathrm{dom}(\gamma)\\ \text{s.t. } X_p\neq\varnothing}} \{([p\mapsto v],\gamma_v,\phi_v) \mid \phi_v = \bigcup_{\theta\in X_p}\phi_\theta \\ \wedge\, \forall\theta\in X_p.\,(v,\gamma_v,\phi_\theta)\in\mathsf{AVal}(\gamma(p),\theta)\}$$

$$\mathsf{E}[\![\xi]\!]_{\kappa,\kappa'} = \bigodot_{p\in\xi \text{ s.t. } X_p\neq\varnothing} \{([p\mapsto v],\phi_v) \mid \phi_v = \bigcup_{\theta\in X_p}\phi_\theta \\ \wedge\, \forall\theta\in X_p.\,(v,\phi_\theta)\in[\![\theta]\!]\}$$

with $\mathrm{dom}(\gamma),\xi\subseteq\mathrm{Pol}$ and $X_p = \min\kappa'(\alpha)\backslash\min\kappa(\alpha)$ for $p\in\mathrm{Pol}_\alpha$. Thus, for each $p\in\mathrm{Pol}_\alpha$ in the domain of γ such that, going from κ to κ', there is a new type disclosure on the type of p (i.e. such that $X_p\neq\varnothing$), to compute the disclosure happening on $\gamma(p)$ we look at all the newly disclosed types $\theta\in X_p$ and for each of them select an abstract environment from $\mathsf{AVal}(\gamma(p),\theta)$. If we can pick these environments so that they all agree in their value component v, we can reveal that p maps to v. Note that X_p determines how much of $\gamma(p)$ is revealed: for instance, $X_p=\{\alpha'\}$ with α' another type variable, then v will simply be another polymorphic name p'. On the other hand, $\mathsf{E}[\![\xi]\!]_{\kappa,\kappa'}$ is more liberal in choosing the common revealed value p, as it scans through each $[\![\theta]\!]$ instead of $\mathsf{AVal}(\gamma(p),\theta)$. In a similar vein, we get:

$$\mathsf{AVal}(u,\theta)_\kappa = \{(v,\gamma,\phi) \mid \phi = \bigcup_{\theta'\in X}\phi_{\theta'} \\ \wedge\forall\theta'\in X.\,(v,\gamma,\phi_{\theta'})\in\mathsf{AVal}(u,\theta')\}$$

$$\mathsf{AStore}(S,\phi) = \bigodot_{l\in\mathrm{dom}(S)} \{([l\mapsto v],\gamma_v,\phi_v) \mid \phi_v = \bigcup_{\theta\in X_l}\phi_\theta \\ \wedge\forall\theta\in X_l.\,(v,\gamma_v,\phi_\theta)\in\mathsf{AVal}(S(l),\theta)\}$$

$$[\![v]\!]_\kappa = \{(v,\phi) \mid \phi = \bigcup_{\theta'\in X}\phi_{\theta'}\wedge\forall\theta'\in X.\,(v,\phi_{\theta'})\in[\![\theta']\!]\}$$

$$\mathsf{S}[\![\phi]\!] = \bigodot_{l\in\mathrm{dom}(\phi)} \{([l\mapsto v],\phi_v) \mid \phi_v = \bigcup_{\theta\in X_l}\phi_\theta\wedge\forall\theta\in X_l.\,(v,\phi_\theta)\in[\![\theta]\!]\}$$

with $X=\min(\kappa(\theta))$ and $X_l=\bigcup_{\theta\in\phi(l)}\min(\mathsf{Cast}(\phi)(\theta))$.

While there is some circularity between the different new components in condition P*, we can always pick them in a nominally deterministic way. We conclude this section with a couple of examples demonstrating type disclosure.

4.3 Examples

We first look at a term that uses type disclosure to cast between two of its inputs, similarly to the initial examples of the paper. Let us set $\theta = \mathrm{ref}\,\alpha\times\mathrm{ref}\,\mathrm{Int}\times\alpha$ and $v\equiv\Lambda\alpha.\lambda\langle x,y,z\rangle^\theta.M$ with $M\equiv\mathsf{if}\ x=y\ \mathsf{then}\ (y:=42;!x)\ \mathsf{else}\ z$. A characteristic trace of v is the following (e.g. for $S=[l\mapsto 9],\rho=[p\mapsto 7]$),

$$\begin{aligned}
&\langle\cdot;\cdot;\cdot\vdash v:\theta\rangle \xrightarrow{?\langle\rangle} \langle(v,\theta),\varepsilon,\varnothing,\varepsilon,\varepsilon\rangle\\
&\xrightarrow{\langle\bar f\rangle} \langle\Diamond,\gamma_1,\varnothing,\varepsilon,\lambda_1\rangle && (\gamma_1=[f\mapsto v],\lambda_1=(f,P))\\
&\xrightarrow{f\langle\alpha\rangle} \langle(v\alpha,\theta\to\alpha),\gamma_1,\varnothing,\varepsilon,\lambda_2\rangle && (\lambda_2=\lambda_1\cdot(\alpha,O))\\
&\xrightarrow{\langle\bar g\rangle} \langle\diamond,\gamma_2,\varnothing,\varepsilon,\lambda_2\rangle && (\gamma_2=\gamma_1\cdot[g\mapsto\lambda\langle x,y,z\rangle^\theta.M])\\
&\xrightarrow{g\langle l,l,p\rangle,S,\rho} \langle(M',\alpha),\gamma_2,\phi_1,S,\lambda_3\rangle && (\phi_1=(l,\mathrm{Int}),(l,\alpha))\\
&\xrightarrow{\langle\bar 42\rangle,S} \langle\Diamond,\gamma_2,\phi_1,S,\lambda_3\rangle && (\lambda_3=\lambda_2\cdot(l,O)\cdot(p,O))
\end{aligned}$$

where $M'\equiv M\{l/x,y\}\{p/z\}\{\rho\}\equiv\mathsf{if}\ l=l\ \mathsf{then}\ (l:=42;!l)\ \mathsf{else}\ 7$.

Now, going back to Example 10, let $f:(\mathrm{ref}\,\mathrm{Int}\times\mathrm{ref}\,\mathrm{Int})\to\mathrm{Unit}$, $g:\forall\alpha.\ \mathrm{ref}\,\alpha\to\mathrm{ref}\,\alpha$ and $M\equiv\mathsf{let}\,x,y=\mathrm{ref}\,0\,\mathsf{in}\,f\langle x,y\rangle;\mathsf{let}\,u=g\,\mathrm{Int}\,x\,\mathsf{in}\,\mathsf{if}\ (u=y)\ 1\ 2$ and $N\equiv\mathsf{if}\ (u=l')\ 1\ 2$. Then, taking $\gamma=[\alpha\mapsto\mathrm{Int}]$, M can produce characteristic traces of two kinds:

$$\begin{aligned}
&\langle\cdot;\cdot;\Gamma\vdash M:\mathrm{Int}\rangle \xrightarrow{?\langle f,g\rangle} \langle(M,\mathrm{Int}),\varepsilon,\varnothing,\varepsilon,\lambda_1\rangle && (\lambda_1=(f,O)\cdot(g,O))\\
&\to^* \langle(f(l,l');\mathsf{let}\,u=g\,\mathrm{Int}\,l\ \mathsf{in}\,N,\mathrm{Int}),\varepsilon,\varnothing,S_1,\lambda_1\rangle && (S_1=[l\mapsto 0,l'\mapsto 0])\\
&\xrightarrow{\bar f\langle l,l'\rangle,S_1} \langle\bullet;\mathsf{let}\,u=g\,\mathrm{Int}\,l\ \mathsf{in}\,N,\varepsilon,\phi_1,S_1,\lambda_2\rangle && (\lambda_2=\lambda_1\cdot(l,P)\cdot(l',P))
\end{aligned}$$

$$\begin{aligned}
&\xrightarrow{\langle()\rangle,S_2} \langle(();\mathsf{let}\,u=g\,\mathrm{Int}\,l\ \mathsf{in}\,N,\mathrm{Int}),\varepsilon,\phi_1,S_2,\lambda_2\rangle && (\phi_1=(l,\mathrm{Int}),(l',\mathrm{Int}))\\
&\to\xrightarrow{\bar g\langle\alpha\rangle,S_2} \langle(\mathsf{let}\,u=\bullet\,l\ \mathsf{in}\,N,\mathrm{Int}),\gamma,\phi_1,S_2,\lambda_3\rangle && (\lambda_3=\lambda_2\cdot(\alpha,P))\\
&\xrightarrow{\langle h\rangle,S_3} \langle(\mathsf{let}\,u=h\,l\ \mathsf{in}\,N,\mathrm{Int}),\gamma,\phi_1,S_3,\lambda_4\rangle && (\lambda_4=\lambda_3\cdot(h,O))\\
&\xrightarrow{\bar h\langle l\rangle,S_3} \langle(\mathsf{let}\,u=\bullet\ \mathsf{in}\,N,\mathrm{Int}),\gamma,\phi_2,S_3,\lambda_4\rangle && (\phi_2=\phi_1,(l,\alpha))\\
&[1]\xrightarrow{\langle l\rangle,S_4} \langle(\mathsf{let}\,u=l\,\mathsf{in}\,N,\mathrm{Int}),\gamma,\phi_2,S_4,\lambda_4\rangle \to^* \xrightarrow{\langle\bar 2\rangle,S_4} \langle\Diamond,\gamma,\phi_2,S_4,\lambda_4\rangle\\
&[2]\xrightarrow{\langle l''\rangle,S_4} \langle(\mathsf{let}\,u=l''\,\mathsf{in}\,N,\mathrm{Int}),\gamma,\phi_3,S_4,\lambda_5\rangle \to^* \xrightarrow{\langle\bar 2\rangle,S_4} \langle\Diamond,\gamma,\phi_3,S_4,\lambda_5\rangle
\end{aligned}$$

according to choices [1] and [2] for O's last move. In particular, O can either return the $l:\mathrm{ref}\,\alpha$ he received, or create a new $l'':\mathrm{ref}\,\alpha$ and return it. Due to ϕ_2, O can cast from Int to α and put arbitrary values in l,l''. However, as $\mathsf{Cast}(\phi_2)(\mathrm{ref}\,\alpha)=\{\mathrm{ref}\,\alpha\}$, he has no cast from ref Int to ref α and hence cannot return l'.

5. Soundness

We show that our model is sound, i.e. equality of term denotations implies contextual equivalence. In fact, we prove a stronger result (Theorem 24), whereby equality is replaced by a larger equivalence relation which rules out some over-distinguishing O behaviours.

5.1 Valid configurations

To reason on the interaction reduction, we prove it preserves some invariants which we collect in the notion of *valid configuration*.

An obvious invariant we want to preserve is that elements of the evaluation stack are well-typed. However, due to the fact that locations do not always have a unique type, and the ensuing casting capabilities that arise, we cannot use the standard typing system defined in Section 2. We thus need to generalise it by allowing location contexts to be multi-valued, i.e. use valid typing functions ϕ (instead of Σ), together with the new typing rule:

$$\frac{\Delta;\phi;\Gamma\vdash_e M:\theta \quad (\theta,\theta')\in\mathsf{Cast}(\phi)}{\Delta;\phi;\Gamma\vdash_e M:\theta'}$$

We write $S:_e\phi$ if $\forall l\in\mathrm{dom}(S).\exists\theta\in\phi(l).\ \nu_{\mathrm{T}}(\phi);\phi;\cdot\vdash_e S(l):\theta$.

The extended type system still satisfies a safety property, on which rely in order to show our model sound.

Lemma 13. *Given $\Delta;\phi;\cdot\vdash_e M:\theta$ and $S:_e\phi$ such that for all $p\in\nu(M,S)\cap\mathrm{Pol}_\alpha,\min(\mathsf{Cast}(\phi)(\alpha))=\{\alpha\}$ either (M,S) diverges or there exists (M',S') irreducible such that:*

- $(M,S)\to^*(M',S')$,
- *M' is either equal to a value v or to a callback $E[f\ v]$,*
- *there exists ϕ' disjoint from ϕ such that $\Delta;\phi\cup\phi';\cdot\vdash_e M':\theta$ and $S':_e\phi\cup\phi'$.*

Using this extended system, we can type evaluation stacks of configurations. A passive evaluation stack $(E_n,\theta_n\rightsquigarrow\theta'_n)::\ldots::(E_1,\theta_1\rightsquigarrow\theta'_1)$ is said to be *well-typed* w.r.t. a typing function ϕ and a type environment $\delta:\mathrm{TVar}\rightharpoonup\mathrm{Types}$ if, for all $1\le i\le n$, $\Delta;\phi\vdash_e E_i:\theta_i\{\delta\}\rightsquigarrow\theta'_i\{\delta\}$. An active evaluation stack $(M,\theta,\Phi)::\mathcal{E}$ is well-typed for ϕ,δ if $\Delta;\phi;\cdot\vdash_e M:\theta\{\delta\}$ and $\mathcal{E}$ is well-typed for ϕ,δ. We can now specify which configurations are valid.

Definition 14. We call $\langle\mathcal{E},\gamma,\phi,S,\lambda\rangle$ a ***valid configuration*** if:

- $\mathrm{dom}(\gamma)=\{a\in\mathrm{Pol}\cup\mathrm{Fun}\cup\mathrm{TVar}\mid\lambda(a)=P\}$,
- $\mathrm{dom}(\phi)=\mathrm{dom}(\lambda)\cap\mathrm{Loc}\subseteq\mathrm{dom}(S)$,
- for all $a\in\nu(\mathcal{E},\mathrm{cod}(S),\mathrm{cod}(\gamma))\backslash\mathrm{Loc}$, $\lambda(a)=O$,
- there exists ϕ' disjoint of ϕ s.t. $S:_e\phi\cup\phi'$,
- $\mathcal{E}$ is well-typed for $\phi\cup\phi',\gamma_{|\mathrm{TVar}}$
- for all $p\in\nu(\mathcal{E},S,\mathrm{cod}(\gamma))\cap\mathrm{Pol}_\alpha$, $\min(\mathsf{Cast}(\phi)(\alpha))=\{\alpha\}$.

We write $C\xRightarrow{m,S,\rho}C'$ when $C\to^*C''\xrightarrow{m,S,\rho}C'$ for some configuration C''. Validity of configurations is preserved as follows.

Lemma 15. *If $C\xRightarrow{m,S,\rho}C'$ and C is valid then so is C'.*

5.2 Composite reduction

The main ingredient in the soundness argument is a refinement of the LTS introduced previously which will eventually allow us to compose term denotations, in a way akin to composition in game semantics: each term in the composition becomes the Opponent for the other term. More concretely, in the composite LTS the behaviour of Opponent is fully specified by expanding the configurations with an extra evaluation stack, environment and store.

The new LTS is called ***composite interaction reduction***. It works on *composite configurations* $\langle \mathcal{E}_P, \mathcal{E}_O, \gamma_P, \gamma_O, \phi, S_P, S_O \rangle$, where:

- $\mathcal{E}_P, \mathcal{E}_O$ are evaluation stacks (one passive and one active); γ_P, γ_O are environments; and S_P, S_O are stores;
- ϕ is a common typing function for locations.

The rules of the composite reduction are in effect the P-rules of the ordinary interaction reduction, plus dual forms thereof fleshing out the O-rules.

A trace t is said to be *generated* by a composite configuration C if it can be written as a sequence $(m_1, S_1, \rho_1) \cdots (m_n, S_n, \rho_n)$ of full moves such that $C \xrightarrow{m_1, S_1, \rho_1} C_1 \xrightarrow{m_2, S_2, \rho_2} \cdots \xrightarrow{m_n, S_n, \rho_n} C_n$, in which case we write $C \overset{t}{\Longrightarrow} C_n$. We say that a composite configuration C *terminates* with the trace t, written $C \Downarrow_t$, if there exists a store S such that $C \xrightarrow{t \cdot ((()), S, \epsilon)} \langle \Diamond, \Diamond, \gamma'_P, \gamma'_O, \phi'_P, S'_P, S'_O \rangle$.

We now define how to merge configurations C_P, C_O into a composite one. For each $X \in \{O, P\}$ we write $X^\perp$ for its dual ($\{X, X^\perp\} = \{O, P\}$), and extend this to $\lambda^\perp = (\_^\perp) \circ \lambda$.

Definition 16. Given a pair of environments (γ_P, γ_O) from $\mathbb{A} \backslash \mathrm{Loc}$ to values, we say these are *compatible* when:

- $\mathrm{dom}(\gamma_P) \cap \mathrm{dom}(\gamma_O) = \varnothing$,
- for all $a \in \mathrm{dom}(\gamma_X)$ $(X \in \{P, O\})$, $\nu(\gamma_X(a)) \backslash \mathrm{Loc} \subseteq \mathrm{dom}(\gamma_{X^\perp})$,
- setting $\gamma^0 = \gamma_P \cdot \gamma_O$, and $\gamma^i = \{(a, v\{\gamma\}) \mid (a, v) \in \gamma^{i-1}\}$ $(i > 0)$, there is an integer n such that $\nu(\mathrm{cod}(\gamma^n)) \backslash \mathrm{Loc} = \varnothing$;

and write $(\gamma_P \cdot \gamma_O)^*$ for the environment from $\mathbb{A} \backslash \mathrm{Loc}$ to Val defined as γ^n, for the least n satisfying the latter condition above.

A pair of valid configurations (C_P, C_O) are called ***compatible*** if, given $C_X = \langle \mathcal{E}_X, \gamma_X, \phi_X, S_X, \lambda_X \rangle$ (for $X \in \{P, O\}$):

- $\phi_P = \phi_O$ and $\lambda_P = \lambda_O^\perp$,
- (γ_P, γ_O) are compatible and $\mathrm{dom}(\gamma_P \cdot \gamma_O) = \mathrm{dom}(\lambda_P) \backslash \mathrm{Loc}$,
- $\mathrm{dom}(S_P) \cap \mathrm{dom}(S_O) = \mathrm{dom}(\lambda_P) \cap \mathrm{Loc}$,
- the merge $\langle \mathcal{E}_P, \mathcal{E}_O, \gamma_P, \gamma_O, \phi_P, S_P, S_O \rangle$ of C_P and C_O is valid.

We write $C_P \between C_O$ for $\langle \mathcal{E}_P, \mathcal{E}_O, \gamma_P, \gamma_O, \phi_P, S_P, S_O \rangle$.

We can merge the (well-typed) evaluation stacks $(\mathcal{E}_P, \mathcal{E}_O)$ of compatible configurations by the following operation:

$$\begin{aligned} \Diamond \| (E, \theta \rightsquigarrow \theta') = E \qquad ((M, \theta) :: \mathcal{E}_P) \| \mathcal{E}_O &= (\mathcal{E}_P \| \mathcal{E}_O)[M] \\ ((E, \theta \rightsquigarrow \theta') :: \mathcal{E}_P) \| ((M, \theta) :: \mathcal{E}_O) &= (\mathcal{E}_P \| \mathcal{E}_O)[E[M]] \\ ((E, \theta \rightsquigarrow \theta') :: \mathcal{E}_P) \| ((E', \theta' \rightsquigarrow \theta'') :: \mathcal{E}_O) &= (\mathcal{E}_P \| \mathcal{E}_O)[E'[E]] \end{aligned}$$

and obtain a correspondence with the operational semantics.

Lemma 17. *Given $C = \langle \mathcal{E}_P, \mathcal{E}_O, \gamma_P, \gamma_O, \phi, S_P, S_O \rangle$ a valid composite configuration and $\gamma = \gamma_P \cdot \gamma_O$, there exists a complete trace t such that $C \Downarrow_t$ iff $(\mathcal{E}_P \| \mathcal{E}_O\{\gamma^*\}, S_P\{\gamma^*\}) \to^* ((), S')$ for some S'.*

On the other hand, there is a semantic way to compare Player and Opponent configurations, by checking that the traces they generate are compatible. Given a trace t, let us write $t^\perp$ for its dual obtained by switching the polarity of each move in t (e.g. each $\bar{f}(v\rangle$ is changed to $f(v\rangle$, and so on).

Definition 18. Let C_P and C_O be two configurations. We write $C_P | C_O \downarrow_t$ when there exists a complete trace t and a store S such that $t \in [\![C_P]\!]$ and $t^\perp \cdot (\langle () \rangle, S, \epsilon) \in [\![C_O]\!]$.

We therefore have the following correspondence between semantic and syntactic composition.

Theorem 19. *For all pairs of compatible configurations C_P and C_O, $C_P | C_O \downarrow_T$ iff $C_P \between C_O \Downarrow_T$.*

5.3 Soundness result

We need two final pieces of machinery for soundness. The first one is so-called *ciu-equivalence*, which allows one to characterise contextual equivalence by restricting focus to evaluation contexts.

Definition 20. Let Σ be a location context. Two terms $\Delta; \Sigma; \Gamma \vdash M_1, M_2 : \theta$ are ciu-equivalent, written $\Delta; \Sigma; \Gamma \vdash M_1 \simeq_{ciu} M_2 : \theta$, when for all typing substitutions $\cdot; \cdot; \cdot \vdash \delta : \Delta$, location contexts $\Sigma' \supseteq \Sigma$, closed stores $S : \Sigma'$, value substitutions $\cdot; \Sigma'; \cdot \vdash \gamma : \Gamma\{\delta\}$ and evaluation contexts $\cdot; \Sigma' \vdash E : \theta\{\delta\} \rightsquigarrow \theta'$, we have $(E[M_1\{\gamma\}\{\delta\}], S) \Downarrow$ iff $(E[M_2\{\gamma\}\{\delta\}], S) \Downarrow$.

Theorem 21. $\Delta; \Sigma; \Gamma \vdash M_1 \simeq M_2 : \theta$ *iff* $\Delta; \Sigma; \Gamma \vdash M_1 \simeq_{ciu} M_2 : \theta$.

As mentioned at the beginning of this section, we introduce an equivalence on term denotations which includes equality. The motivation for this is so as to prune out some distinctions that the model makes between behaviours that are in fact indistinguishable. More precisely, our model abstracts away any actual values provided by Opponent for polymorphic inputs by names in Pol. Moreover, when P plays back one of those names, O is in position to determine precisely which actual value is P returning in reality (as all polymorphic names introduced by O must be distinct). This discipline is based on the assumption that O can always instrument the values he provides to P so that he can later distinguish between them. It is a valid assumption, apart from the case when later in the trace there is some value disclosure for those polymorphic names which forbids O to implement such instrumentations.

To remove this extra intensionality from the model, we introduce an equivalence of traces which blurs out such distinctions:

- we first substitute in every P-move all the *O polymorphic names* whose value have been disclosed by their disclosed value;
- we then enforce the freshness of *P polymorphic names* played in P moves, which may be broken because of these substitutions.

The latter step is implemented via a name-refreshing procedure, defined as follows. Given traces t, t', we say that t' is a *P-refreshing* of t, written $t \rightsquigarrow t'$, if $t = t_1 \cdot (m, S, \rho) \cdot t_2$, $t' = t_1 \cdot t'_2$, with m a P-move, and there are polymorphic names p, p' such that:

- $p \in \nu(t_1) \cap \nu(m, S, \mathrm{cod}(\rho))$ is introduced in a P-move of t_1,
- $p' \notin \nu(t)$ and t'_2 is $(m, S, \rho) \cdot t_2$ where we first replace a single occurrence of p in $(m, S, \mathrm{cod}(\rho))$ by p', and then replace any $[p \mapsto v]$ in the resulting subtrace by $[p \mapsto v] \cdot [p' \mapsto v]$.

P-refreshing is bound to terminate in the traces we examine. We write $\mathcal{F}(t)$ for the set of all t' such that $t \rightsquigarrow^* t'$ and $t' \not\rightsquigarrow$.

Definition 22. Two traces t_1, t_2 are said to be equivalent, written $t_1 \sim t_2$, if $\mathcal{F}(\overline{t_1}^\varepsilon) = \mathcal{F}(\overline{t_2}^\varepsilon)$, where $\overline{t}^{\rho_1 \cdots \rho_n}$ is defined as:

$$\overline{t \cdot (m, S, \rho)}^{\rho_1 \cdots \rho_n} = \begin{cases} \overline{t}^{\rho_1 \cdots \rho_n} \cdot ((m, S, \rho)\{\rho_1\} \cdots \{\rho_n\}) & \text{if } m \text{ a P-move} \\ \overline{t}^{\rho_1 \cdots \rho_n \rho} \cdot (m, S, \rho) & \text{otherwise} \end{cases}$$

We extend equivalence to sets of traces in an elementwise fashion.

Lemma 23. *Let t_1 be a trace such that $t_1^\perp \in \mathrm{Tr}(C)$ with C a valid configuration. Then for all $t_2 \sim t_1$ we have $t_2^\perp \in \mathrm{Tr}(C)$.*

We can now prove the main theorem of this section.

Theorem 24 (Soundness)**.** *For all terms $\Delta; \Sigma; \Gamma \vdash M_1, M_2 : \theta$, $[\![M_1]\!] \sim [\![M_2]\!]$ implies $M_1 \cong M_2$.*

Proof. Suppose $[\![M_1]\!] \sim [\![M_2]\!]$. Using Theorem 21, we prove that $M_1 \simeq_{ciu} M_2$. Let us take $\delta, \Sigma' \supseteq \Sigma, S, \gamma$ and E as in Definition 20, and suppose that $(E[M_1\{\gamma\}\{\delta\}], S) \Downarrow$.

Take $(\vec{\alpha}, \vec{l}, \vec{u}) \in [\![\Delta, \Sigma, \Gamma]\!]$ and write $C_{P,1}$ for the P-configuration $\langle (M_1\{\overrightarrow{\widetilde{u}/x}\}, \theta), \epsilon, \phi, S, \lambda \rangle$, so $\langle \Delta; \Sigma; \Gamma \vdash M_1 : \theta \rangle \xrightarrow{?\langle \vec{\alpha}, \vec{l}, \vec{u} \rangle), S', \rho} C_{P,1}$.

Let $C_O = \langle (E, \theta \rightsquigarrow \theta'), \gamma' \cdot \delta, \phi, S, \lambda^\perp \rangle$ where $\gamma' = \{(u_i, v_i) \mid \gamma(x_i) = v_i\}$. From Lemma 17, there exists a complete trace t such that $C_{P,1} \twoheadrightarrow C_O \Downarrow_t$. Then, from Theorem 19, $C_{P,1}|C_O \downarrow_t$, so that $t \in \mathrm{Tr}(C_{P,1})$ and $t^\perp \in \mathrm{Tr}(C_O)$. Writing $C_{P,2}$ for the Player configuration $\langle (M_2\{\overrightarrow{\widetilde{u}/x}\}, \theta), \epsilon, \phi, S, \lambda \rangle$, from the hypothesis of the theorem, there exists a complete trace $t' \sim t$ such that $t' \in \mathrm{Tr}(C_{P,2})$. From Proposition 23, $t'^\perp \in \mathrm{Tr}(C_O)$, so that $C_{P,2}|C_O \downarrow'_t$, and using Theorem 19 (in the other direction), we get that $C_{P,2} \twoheadrightarrow C_O \Downarrow'_t$. Finally, using Lemma 17, we get that $(E[M_2\{\gamma\}\{\delta\}], S) \Downarrow$. □

6. Completeness

While sound, our model fails to be fully abstract as it overestimates the power of O: the way cast relations (Cast) are computed overapproximates the casts that can be implemented by the context in practice, as inhabitation constraints are not taken into account. For instance, a cast from $\theta \to \theta_1$ to $\theta \to \theta_2$ does not yield one from θ_1 to θ_2 unless a value of type θ is available. In this section we restrict our attention to a fragment of System ReF, called System ReF*, carved in such a way that the above problem cannot be manifested. We then prove our model fully abstract for terms in System ReF*.

System ReF* is defined by means of restricting the types allowed at the type interface of a term. In particular, we pose the following restrictions affecting the types which can appear under a ref constructor. First, we do not allow any binders $\forall, \exists$ to appear in the scope of a ref and, moreover, any type variable α inside a $\mathrm{ref}\,\theta$ must be *reachably inhabited*: in order for a value of type $\mathrm{ref}\,\theta$ to be played in a trace, a value of type α must have been played before.

Both these restrictions are captured by the following type predicate $\mathrm{good}_\Upsilon(\theta)$, which determines whether a type θ is in the defined fragment, assuming that the type variables in Υ are inhabited.

$$\begin{array}{lcl}
\mathrm{good}_\Upsilon(\mathrm{ref}\,\theta) & = & \mathrm{good}_\Upsilon(\theta) \wedge \nu_{\mathrm{T}}(\theta) \subseteq \Upsilon \wedge \theta \text{ is quantifier-free} \\
\mathrm{good}_\Upsilon(\theta \to \theta') & = & \mathrm{good}_\Upsilon(\theta) \wedge \mathrm{good}_{\Upsilon \cup \mathrm{gtv}(\theta)}(\theta') \\
\mathrm{good}_\Upsilon(\forall \alpha.\theta) & = & \mathrm{good}_\Upsilon(\theta) \\
\mathrm{good}_\Upsilon(\theta \times \theta') & = & \mathrm{good}_\Upsilon(\theta) \wedge \mathrm{good}_\Upsilon(\theta') \\
\mathrm{good}_\Upsilon(\exists \alpha.\theta) & = & \mathrm{good}_{\Upsilon \cup \{\alpha\}}(\theta) \\
\mathrm{good}_\Upsilon(\theta) & = & \mathbf{true} \quad \text{otherwise}
\end{array}$$

Above, $\mathrm{gtv}(\theta)$ returns the type variables at the ground level of θ:

$$\begin{array}{lllll}
\mathrm{gtv}(\alpha) & = & \{\alpha\} & \mathrm{gtv}(\theta \times \theta') & = \mathrm{gtv}(\theta) \cup \mathrm{gtv}(\theta') \\
\mathrm{gtv}(\exists \alpha.\theta) & = & \mathrm{gtv}(\theta) \backslash \{\alpha\} & \mathrm{gtv}(\mathrm{ref}\,\theta) & = \mathrm{gtv}(\theta)
\end{array}$$

and $\mathrm{gtv}(\theta) = \varnothing$ otherwise. We extend goodness to type interfaces by setting, given $\Sigma = \{l_1 : \theta_1, \cdots, l_n : \theta_n\}, \Gamma = \{x_1 : \theta'_1, \cdots, x_m : \theta'_m\}$:

$$\mathrm{good}(\Delta; \Sigma; \Gamma \vdash \theta) = \mathrm{good}_\varnothing((\mathrm{ref}\,\theta_1 \times \cdots \times \mathrm{ref}\,\theta_n \times \theta'_1 \times \cdots \times \theta'_m) \to \theta)$$

Definition 25. We let System ReF* contain all terms $\Delta; \Sigma; \Gamma \vdash M : \theta$ such that $\mathrm{good}(\Delta; \Sigma; \Gamma \vdash \theta)$ holds.

Example 26. The terms form Example 10 (2) are not in System ReF*, as α' is not inhabited. The two terms are then equivalent, because Opponent cannot cast α to Int, lacking a value of type α' to do so. Our model, however, does not capture this equivalence.

Moreover, we call an initial configuration $\langle \Delta; \Sigma; \Gamma \vdash M : \theta \rangle$ ***good*** just if its interface is, while a valid configuration $\langle \mathcal{E}, \gamma, \phi, S, \lambda \rangle$ is good just if, taking $X_\lambda = \{\alpha \mid \nu(\lambda) \cap \mathrm{Pol}_\alpha \neq \varnothing\}$, $\nu_{\mathrm{T}}(\phi) \subseteq X_\lambda$ and $\mathrm{good}_{X_\lambda}(\theta)$ hold, for all $\theta \in \mathrm{cod}(\phi) \cup \{\theta \mid \nu(\lambda) \cap \mathrm{Fun}_\theta \neq \varnothing\}$. We can then check that goodness is preserved under reduction.

Working in this restricted fragment, we can always implement all possible casts anticipated from the cast closure construction of Section 4. More specifically, a ***cast-term*** from θ to θ' based on *aliased pairs* $(\theta_1, \theta'_1), \ldots, (\theta_n, \theta'_n)$ and *inhabited variables* $\alpha_1, \cdots, \alpha_m$ is a term $\mathbf{cast}_{\theta \to \theta'}$ such that:

- $\Delta; ; \overrightarrow{x_i : \mathrm{ref}\,\theta_i}, \overrightarrow{y_i : \mathrm{ref}\,\theta'_i}, \overrightarrow{z_j : \alpha_j} \vdash \mathbf{cast}_{\theta \to \theta'} : \theta \to \theta'$
- for any $\Sigma = \{\overrightarrow{l_i : \theta_i}\}$, $p_j \in \mathrm{Pol}_{\alpha_j}$, $S : \Sigma$ and $\Delta; \Sigma'; \vdash v : \theta$, $((\mathbf{cast}_{\theta \to \theta'} \overrightarrow{\{l_i/x_i, y_i\}} \overrightarrow{\{p_j/z_j\}})v, S) \twoheadrightarrow^* (v', S \cdot S')$ with $v \cong v'$,

with S' disjoint of S. Recall now $\mathsf{Cast}(\phi)$ from Definition 12 and define its restriction $\mathsf{Cast}^\circ(\phi)$, the closure of $\{(\theta, \theta') \mid \exists l.\, \mathrm{ref}\,\theta, \mathrm{ref}\,\theta' \in \phi(l)\}$ using all cast closure rules from Section 4 apart from $(*)$.

Lemma 27. *Let ϕ be a valid typing function with $\vec{\alpha}$ all free type variables in ϕ. Then, for all $(\theta, \theta') \in \mathsf{Cast}^\circ(\phi)$ there is a cast-term $\mathbf{cast}_{\theta \to \theta'}$ based on pairs $\{(\theta'', \theta''') \mid \exists l.\, \theta'', \theta''' \in \phi(l)\}$ and $\vec{\alpha}$.*

The $(*)$ rule, though useful for soundness, has no clear way to be implemented with cast-terms, hence the reason for aiming at its exclusion. The restriction we pose on System ReF* in that quantifiers cannot appear under a ref constructor renders the rule indeed redundant. Each ϕ produced in the model contains no types with quantifiers, so that the $(*)$ rule can be eliminated.

The proof of full abstraction is based on a definability result: we show that every complete trace produced by a good P-configuration C_P can be accepted by an appropriately designed O-configuration C_O. In addition, the given trace is all C_O can accept up to nominal and trace equivalence. The technique follows e.g. [17], albeit expanded to the polymorphic setting. Note that the absence of generic types [22] in our language, because of type disclosure, rules out the option of reducing the problem to that for the monomorphic setting.

Theorem 28 (Definability)**.** *Let C_P be a good configuration and t a complete trace in $\mathrm{Tr}(C_P)$ with final store S. There exists a valid configuration C_O compatible with C_P such that $\mathrm{Tr}(C_O) = \{\pi \star t' \mid (\forall a \in \nu(t) \backslash \nu(C_O).\, \pi(a) = a) \wedge \exists t'' \sim t \cdot (\langle (\bar{()}) \rangle, S, \varnothing).\, t' \sqsubseteq t''^\perp\}$.*

We present the main ingredients of the definability argument. We argue by induction on the length of t. Suppose $C_P = \langle \mathcal{E}_P, \gamma_P, \phi_P, S_P, \lambda \rangle$, let A_0 be the set of all the names that appear in t and C_P. To determine the types behind the O-type-variables in A_0, we define a mapping δ by collecting all type constraints we can derive from the trace t about O-type-variables, mapping to Int when no such constraints exist . We number P-moves in t in decreasing order, that is, the head move of t has index $\|t\| = (t+1)/2$, and let $\Theta_{\|t\|}$ recursively include all function, reference and variable types that appear in $\phi_P\{\delta\}$ and $\lambda\{\delta\}$. At the i-th P-move of t, this set is updated to $\Theta_i = \{\theta^i_1, \theta^i_2, \cdots, \theta^i_{ts(i)}\}$ by including all the types disclosed in intermediate moves.

We use a counter cnt to determine the position we are in t and inductively construct $C_O = \langle \mathcal{E}_O, \gamma_O, \phi_O, S_O, \lambda^\perp \rangle$ with the additional assumptions that:

- $\mathcal{E}_O = (E_n, \eta_n \rightsquigarrow \eta'_n, \Phi_n) :: \cdots :: (E_1, \eta_1 \rightsquigarrow \eta'_1, \Phi_1)$, n is determined from t and $E_i \equiv (\lambda z.\, !r_i(!\mathsf{cnt})z)\bullet$, for each i;
- γ_O obeys δ (i.e. $\gamma_{O|\mathrm{TVar}} \subseteq \delta$) and, moreover, assigns values to each function or pointer name belonging to O by referring to purpose-specific private references in S_O:
  - for each f of arrow type, $\gamma_O(f) = \lambda z.\, !q_f(!\mathsf{cnt})z$
  - for each g of universal type, $\gamma_O(g) = \Lambda\alpha.\, !q'_g(!\mathsf{cnt})\alpha$
  - for each pointer name p of type β,
    - if $\gamma_O(\beta)$ an arrow type, $\gamma_O(p) = \lambda z.\, !q_p(!\mathsf{cnt})z$
    - if $\gamma_O(\beta)$ a universal type, $\gamma_O(p) = \Lambda\alpha.\, !q'_p(!\mathsf{cnt})\alpha$
    - if $\gamma_O(\beta)$ an existential type, $\gamma_O(p) = \langle \alpha', v \rangle$ and v recursively follows the same discipline
    - if $\gamma_O(\beta)$ a product type, $\gamma_O(p) = \langle v_1, v_2 \rangle$ and v_1, v_2 recursively follow the same discipline
    - if $\gamma_O(\beta) = \mathrm{Int}/\mathrm{ref}\,\theta$ and the value of p gets disclosed in t, $\gamma_O(p)$ is the revealed value; otherwise, $\gamma_O(p)$ is a unique integer/location representing p
    - if $\gamma_O(\beta) = \alpha'$, $\gamma_O(p)$ is some polymorphic name respecting the type disclosures in t;
- $\mathrm{dom}(S_O)$ contains $Q_F \uplus Q'_F \uplus Q_P \uplus Q'_P \uplus \{r_1, \cdots, r_n, l_1, \cdots, l_k\} \uplus \{\mathsf{cnt}\} \uplus \{\ell_1, \cdots, \ell_{ts(\|t\|)}\} \uplus \{\mathsf{getval}_i \mid i \in [1, \|t\|]\}$;

where Q_F contains a unique location q_f for each function name f in $\mathrm{dom}(\gamma_O)$, Q'_F contains the q'_g's, Q_P the q_p's, and Q'_P the q'_p's.

The main engine behind the construction is the use of references to record values played, continuations, functions, and generally all history of t so that O can refer to it in order to: decide to accept each expected move by P, and play the corresponding expected move themselves. Looking at the domain of S_O, the l_i's are the shared locations between C_P and C_O, while cnt is an integer counter that counts the remaining P-moves in t. We set $S_O(\mathsf{cnt}) = \|t\|$. $L = \{\ell_1, \cdots, \ell_{ts(\|t\|)}\}$ is a set of private auxiliary locations which we shall use in order to cast between known types and types obtained by opening existential packages.

The role of the getval's is to us to store all names that appear in the trace. For each i, getval_i is a location of type:

$$\exists \vec{\alpha}. \left((\mathrm{Int} \to \theta^i_1) \times \cdots \times (\mathrm{Int} \to \theta^i_{ts(i)})\right) \times \left((\mathrm{Unit} \to \mathrm{ref}\, \theta^i_1) \times \cdots \times (\mathrm{Unit} \to \mathrm{ref}\, \theta^i_{ts(i)})\right)$$

where $\vec{\alpha}$ is the sequence of all free type variables in Θ_i. Thus, the value of getval_i is an existential package whose first component contains enumerations of all values of type θ^i_j, for each i, j. These is enough to represent all the available values at each point in the trace. The second component inside the package stored in getval_i contains a single reference for each type and we shall assign to it a special role, namely of holding a private reference from the set L.

To see how the above work, let $t = (m_1, S_1, \rho_1) \cdot (m_2, S_2, \rho_2) \cdot t'$ and suppose m_1 is a question $\overline{f}\langle v \rangle$, introducing fresh type variables $\beta_1, \cdots, \beta_\iota$ (via values of existential type). We encode acceptance of these first two moves in q_f, by setting $S_O(q_f)(\|t\|)$ to be:

$$\begin{aligned}
&\mathsf{unpack}\ !\mathsf{getval}_{\|t\|}\ \mathsf{as}\ \langle \vec{\alpha}', \langle z', h\rangle\rangle\ \mathsf{in}\\
&\mathsf{let}\ z = \mathsf{castPk}\langle z', h\rangle\ \mathsf{in}\\
&\lambda x_0.\ \mathsf{unpack}\ N_1\ \mathsf{as}\ \langle \beta_1, x_1\rangle\ \mathsf{in} \cdots \mathsf{unpack}\ N_\iota\ \mathsf{as}\ \langle \beta_\iota, x_\iota\rangle\ \mathsf{in}\\
&\qquad \mathsf{let}\ \mathsf{val} = \mathrm{ref}\, \langle z, \lambda_.\Omega, \cdots, \lambda_.\Omega\rangle\ \mathsf{in}\\
&\qquad \mathsf{cnt}--; \mathsf{Fshvals};\ \mathsf{Chkvals};\ \mathsf{Newvals};\ \mathsf{Setstor};\ \mathsf{Play} \qquad (*)
\end{aligned}$$

Since the type of $!\mathsf{getval}_{\|t\|}$ is fully existentially quantified, when we (statically) unpack $!\mathsf{getval}_{\|t\|}$ and get $\vec{\alpha}', z'$, the $\vec{\alpha}'$ are distinct from the type variables $\vec{\alpha}$ in $\Theta_{\|t\|}$ and, consequently, each component $z'_i : \mathrm{Int} \to \theta'_i$ of z' is not of the expected type $\mathrm{Int} \to \theta_i$. However, when the unpack will actually happen this mismatch will be resolved. For visible types (in the game-theoretic *view* sense [10]), we need this mismatch to also be resolved statically, as we would like to be able to relate the values in z' with x_0, any open variables, or the return value of $!q_f$. Hence, we employ the castPk function which casts values of type θ'_i to θ_i in z', using the locations in L (each of type θ_i) and their representations in h.

Each term N_i is selected in such a way so that, using val and $x_0, x_1, \cdots, x_{i-1}$, it captures the precise position within (m_1, S_1, ρ_1) which introduces the type variable β_i. Note that, here and below, in order to access the values of ρ_1 we make use of the **cast** terms of Lemma 27. We then create the location val to contain the old value stores (z), extended with an empty store for each β_i $(\lambda\_.\Omega)$. Also:

□ $\mathsf{Fshvals}$ detects the positions inside (m_1, S_1, ρ_1) that introduce fresh names and updates val by adding them as new values in their corresponding types. This yields an updated store S'_O.

□ $\mathsf{Chkvals}$ checks that x_0, the public part of S'_O and the values revealed by type disclosure are the ones expected, that is, v, S_1 and ρ_1 respectively. For these comparisons to be implemented, it suffices to focus on variable types only: the rest are either integers/references (can always be checked), or units/functions (no need to check them). Variable types belonging to P cannot be checked (P always plays fresh names for them), so we skip them. Values of variable types α belonging to O will appear e.g. in x_0 with their instantiated types $\delta(\alpha)$. In this case, we are in position to distinguish between function names: these are functions provided by O as polymorphic values so O can pre-instrument so that when calling them they each produce a unique observable effect.

□ $\mathsf{Newvals}$ creates all the fresh locations of (m_2, S_2) and stores them in the corresponding index of val. Moreover, for each name f' of arrow type in (m_2, S_2), $\mathsf{Newvals}$ includes a code portion creating a reference $q_{f'}$ to store a function which takes as an argument the value of the counter specifying the current move, and returns a function following the expected behaviour (and that stipulated by the store obtained for t' by the inductive hypothesis). Similarly for names of universal types. Finally, for each polymorphic O-name p in (m_2, S_2) of type α, $\mathsf{Newvals}$ includes code creating q_p and adding a function in val according to the type $\gamma_O(\alpha)$ (e.g. if an arrow type then we add $\lambda z.\, !q_p(!\mathsf{cnt})z$, where q_p encapsulates an effect which allows its recognition in the future).

□ $\mathsf{Setstor}$ updates the store in such a way that all the values of S_2 are set, while Play is defined by case analysis on m_2.

Using Definability, we can now prove the main theorem.

Theorem 29 (Completeness)**.** *Given* System ReF* *terms* $\Delta; \Sigma; \Gamma \vdash M_1, M_2 : \theta$*, if* $M_1 \cong M_2$ *then* $[\![M_1]\!] \sim [\![M_2]\!]$.

References

[1] S. Abramsky and R. Jagadeesan. A game semantics for generic polymorphism. *APAL*, 133(1-3), 2005.
[2] A. Ahmed. *Semantics of types for mutable state*. PhD thesis, Princeton University, Princeton, NJ, USA, 2004.
[3] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.
[4] A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.
[5] L. Birkedal, K. Støvring, and J. Thamsborg. Realisability semantics of parametric polymorphism, general references and recursive types. *MSCS*, 20(04), 2010.
[6] D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *JFP*, 22, 2012.
[7] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal aspects of computing*, 13(3-5), 2002.
[8] D. R. Ghica and N. Tzevelekos. A system-level game semantics. *ENTCS*, 286, 2012.
[9] D. J. D. Hughes. *Hypergame semantics: full completeness for System F*. PhD thesis, University of Oxford, 2000.
[10] J. M. E. Hyland and C.-H. L. Ong. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2), 2000.
[11] G. Jaber. Operational nominal game semantics. In *FOSSACS*, 2015.
[12] G. Jaber and N. Tabareau. Kripke open bisimulation - a marriage of game semantics and operational techniques. In *APLAS*, 2015.
[13] R. Jagadeesan, C. Pitcher and J. Riely. Open bisimulation for aspects. In *AOSD*, 2007.
[14] A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. In *LICS*, 1999.
[15] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core java language. In *ESOP*, 2005.
[16] A. Jeffrey and J. Rathke. Full abstraction for polymorphic pi-calculus. *TCS*, 390(2-3), 2008.
[17] J. Laird. A fully abstract trace semantics for general references. In *ICALP*, 2007.
[18] J. Laird. Game semantics for call-by-value polymorphism. In *ICALP*, 2010.
[19] J. Laird. Game semantics for a polymorphic programming language. *J. ACM*, 60(4), 2013.
[20] S. Lassen. Eager normal form bisimulation. In *LICS*, 2005.
[21] S. B. Lassen and P. B. Levy. Typed normal form bisimulation for parametric polymorphism. In *LICS*, 2008.
[22] G. Longo, K. Milsted, and S. Soloviev. The genericity theorem and parametricity in the polymorphic λ-calculus. *TCS*, 121(1&2), 1993.
[23] P. Levy and S. Staton. Transition systems over games. In *LICS*, 2014.
[24] A. S. Murawski, S. J. Ramsay, and N. Tzevelekos. A contextual equivalence checker for IMJ*. In *ATVA*, 2015.
[25] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *J. ACM*, 47(3), 2000.
[26] J. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.
[27] C. Strachey. Fundamental concepts in programming languages. Reprint. *Higher-Order and Symbolic Computation*, 13(1/2), 2000.
[28] E. Sumii. A complete characterization of observational equivalence in polymorphic λ-calculus with general references. In *CSL*, 2009.
[29] P. Wadler. Theorems for free! In *FPCA*, 1989.
[30] A. K. Wright. Simple imperative polymorphism. *LASC*, 8(4), 1995.