Trace Semantics for Polymorphic References
Jaber, G; TZEVELEKOS, NP; Logic in Computer Science (LICS)

Copyright 2016 held by Owner/Author. Publication Rights Licensed to ACM. Copyright © 2016 ACM
This is a pre-copyedited, author-produced version of an article accepted for publication in LICS '16 Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science following peer review. The version of record is available http://dl.acm.org/citation.cfm?id=2934509

For additional information about this publication click this link.
http://qmro.qmul.ac.uk/xmlui/handle/123456789/13081

Information about this research object was correct at the time of download; we occasionally make corrections to records, please therefore check the published record when citing. For more information contact scholarlycommunications@qmul.ac.uk
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 recording 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 Streicher 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 so far 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 [18, 19] has extended the reach of polymorphic games to languages with higher-order state.

An important aspect of previous models [9, 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.

\[ \forall \alpha. \, (\text{ref } \alpha \times \text{ref } \text{Int}) \rightarrow \alpha \]

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, \(\alpha\) is instantiated with Int and \(x\) and \(y\) happen to represent the same location. The following term would take advantage of such a coincidence,

\[ \Lambda \alpha. \, \lambda (x,y)^{\text{ref } \alpha \times \text{ref } \text{Int}}. \, y := 42; !x \]

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 implement our example in a way that it can discover such coincidences and effectively deduce that \(\alpha = \text{Int}\). Let us write \(y++\) below for \(y := !y + 1\).

\[ \Lambda \alpha. \, \lambda (x,y)^{\text{ref } \alpha \times \text{ref } \text{Int}}. \, \text{let } x' = \text{ref } x, y' = \text{ref } !y \text{ in} \]

\[ y++; x := x'; \]

\[ \text{if } y' = !y \text{ then } y := 42; !x \text{ else } !x \]

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 \(\alpha\) and \text{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 vice versa, 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 23 [24] 25 , later refined into Kripke logical relations 2 [26] 28 , 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 22 . 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 26 . 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. 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:

\[ \theta, \theta' ::= \alpha \mid \text{Unit} \mid \text{Int} \mid \text{ref } \theta \mid \theta \times \theta' \mid \theta \rightarrow \theta' \mid \forall \alpha. \theta \mid \exists \alpha. \theta \]

where \( \alpha \in \text{TVar} \), and \( \text{TVar} \) a countably infinite set of type variables. As usual, a type is closed if its all type variables 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 \( \text{Loc of locations} \) and some standard collection of binary integer operators, which we generally denote by \( \oplus \). We use the following macros: let \( x = N \) 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 \( \Delta \) is a location context, i.e. a finite partial function from locations to closed types; \( \Gamma \) a variable context, and \( \Sigma \) a set of type variables containing all free type variables of \( \Gamma \). Given a closed evaluation context \( E \), we write \( \Delta; \Sigma; \Gamma \vdash E : \theta' \) 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 \( \Sigma \) 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 \( S : \text{Loc} \rightarrow \text{Val} \) from locations to \( v, u ::= () \mid n \mid x \mid \lambda x.\theta.M \mid \Lambda \alpha.M \mid \text{pack}(\theta, v) \mid \{v, u\} \)

values. We define the following notion for stores, which we shall also be using for general partial maps:

- The empty store is written \( \varepsilon \). Adding a new element \((I, v)\) to a store \( S \) is written \( S \cdot [I \mapsto v] \), and is defined only if \( I \notin \text{dom}(S) \).
- We also define \( S[l \mapsto v] \), for \( l \in \text{dom}(S) \), 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 \( S; \Sigma ; \vdash \lambda l.\theta : \forall \alpha.\theta \) for all \( l \in \text{dom}(S) \). Given a set \( L \) of locations and a store \( S \), we define the image of \( L \) by \( S \), written \( S^* (L) \), as \( S^* (L) = \cup_{l \in L} S^*(L) \) with \( S^{l+1}(L) = S^l(L) \cup \{ l \in L \mid l \text{ contained in } S^l(L) \} \) and \( S^0(L) = L \). \( S \) is closed just if \( \text{dom}(S) = S^*(\text{dom}(S)) \).

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

Remark 2. We have equipped our language with a constructor 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 \( \Sigma \) be closed. Two terms \( \Delta; \Sigma; \Gamma \vdash \lambda M_1.M_2 : \theta \) are contextually equivalent, written \( \Delta; \Sigma; \Gamma \vdash \lambda M_1.M_2 : \theta \), if for all contexts \( C \), all \( \Sigma' \supseteq \Sigma \) and all closed \( S : \Sigma' \) such that \( \Delta; \Sigma ; C[M_1 : \text{Unit}] ; \Gamma \vdash \lambda M_1.M_2 : \theta \) and \( \text{ref } \theta \).

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:

\[ A = \text{Loc} \uplus \text{TVar} \uplus \bigcup_{\theta \in \text{FTVar}} \text{Fun}_\theta \uplus \bigcup_{\theta \in \text{TVar}} \text{Pol}_\theta \]

where \( \theta \) ranges over function types and each of the components in this countable union is itself a countable set. We let \( \text{Fun} = \bigcup_{\theta \in \text{FTVar}} \text{Fun}_\theta \) and \( \text{Pol} = \bigcup_{\theta \in \text{TVar}} \text{Pol}_\theta \). We range over elements of \( \text{Loc} \) by \( l \) and variants; over \( \text{TVar} \) by \( \alpha, \text{etc} \); over \( \text{Fun} \) by \( f, g, \text{etc} \); and over \( \text{Pol} \) by \( p, \text{etc} \).

Semantic objects feature elements of \( A \) as atomic entities which, moreover, can be acted upon by finite permutations of \( A \). A nominal set \( A \) is a pair \((X, \ast)\) of a set \( X \) along with an action \( \ast \) from the set of finite component-preserving computations of \( A \) on the set \( X^{\#} \).

Given some \( x \in X \), the set of names featuring in \( x \) form

\[ ^1 \text{A finite permutation } \pi : A \rightarrow A \text{ is component-preserving simply if it preserves the partition of } A, \text{e.g. if } d \in \text{Loc then } \pi(d) \in \text{Loc}. \]
its support, written \( \nu(x) \), which we stipulate to be finite. Formally, \( \nu(x) \) is the smallest subset of \( A \) that such all permutations which elementwise fix \( \nu(x) \) also fix \( x \). We shall sometimes write \( \nu_{\alpha}(x) \), for \( \alpha \in \{L, T, F, P\} \) in order to select a specific kind of names from the support of \( x \). For instance, \( \nu_{\alpha}(x) = \nu(x) \cap \text{Loc} \). Using the same notation, we also write \( \nu_{\alpha}(\theta) \) for the free type variables of \( \theta \). We usually write \( (X, \ast) \) 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:

\[
\text{AValues} \ni v, u := (i | i \mid \bar{f} | p | \alpha | (u, v))
\]

where \( i \in \mathbb{Z}, l \in \text{Loc}, f \in \text{Fun}, p \in \text{Pol} \) and \( \alpha \in \text{TVar} \). 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 \( \text{Loc} \rightarrow \text{AValues} \).

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 \( \theta \), written \( [\theta] \), consists of pairs \((v, \phi)\) of an abstract value \( v \) along with a function \( \phi : \nu_1(v) \rightarrow P(\text{Types}) \), and is given as:

\[
\begin{align*}
\text{Unit} & = (()) \\
\text{Int} & = \{ (n, e) | n \in \mathbb{Z} \} \\
\text{ref } \theta & = \{ (l, \{ (l, \emptyset) \} | l \in \text{Loc} \} \\
\alpha & = \{ (p, r) | p \in \text{Pol}_\alpha \} \\
\theta \rightarrow \theta' & = \{ (f, e) | f \in \text{Fun}_{\alpha \rightarrow \nu} \} \\
[\alpha, \theta] & = \{ (f, e) | f \in \text{Fun}^{\alpha \rightarrow \nu} \} \\
[3 \alpha, \theta] & = \{ ((\alpha', v), \phi) | (v, \phi) \in [\theta_{\alpha'}] \} \\
[1 \alpha \times \theta_2] & = \{ ((v_1, v_2), (\nu_1(v_1), \nu_2(v_2)) \} \\
\end{align*}
\]

The role of \( \phi \) 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, \( \phi \) assigns sets of types to each location instead of a unique type. More generally, a **typing function** is a finite map \( \phi : \text{Loc} \rightarrow P(\text{Types}) \). The type translation is extended to typing environments by mapping each \( \Delta = \{ \alpha_1, \ldots, \alpha_k \} \), \( \Sigma = \{ l_1 : \theta_1, \ldots, l_n : \theta_n \} \) and \( \Gamma = \{ x_i : \theta_1, \ldots, x_k : \theta_k \} \) to:

\[
[\Delta, \Sigma, \Gamma] = \{ ((\alpha, \bar{\nu}, \bar{v}), \{ l_{j, n} : [\theta_1, \ldots, x_k : [\theta_k] \}) | (v_j, \phi_j) \in [\theta_j] \}.
\]

**Extending the syntax for Fun \cup 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 Fun \cup 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 \( (\bar{u}) \), representing a call to a functional name \( f \in \text{Fun} \) with argument \( u \in \text{AValues} \).

- **OQ. Opponent Questions** are of the form \( (u) \), with \( f \in \text{Fun} \) and \( u \in \text{AValues} \); moreover, there are initial opponent questions of the form \( (\bar{u}) \) \( \in \text{AValues} \).

- **PA. Player Answers** are moves of the form \( (\bar{u}) \), with \( u \in \text{AValues} \).

- **OA. Opponent Answers**, which are of the form \( (u) \in AValues \).

On the other hand, value disclosures are partial functions \( \rho \) 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 \((m, \rho, \sigma)\) of a move \( m \), a closed abstract store \( S \) and a finite map \( \rho : \text{Pol} \rightarrow \text{AValues} \). 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 \( \mathcal{E}^E, \theta_n \rightarrow \theta'_n : \cdots : (\mathcal{E}^E, \theta_1 \rightarrow \theta'_1) \), where each \( \mathcal{E}^E \) is an evaluation context of type \( \theta_1 \rightarrow \theta'_1 \);

- or **active**, which are related to Player configurations and are of the form \( (M, \emptyset) \rightarrow \emptyset' \), i.e. they consist of a term \( M \) of type \( \emptyset \) and a passive stack \( \emptyset' \). The empty stack is written \( \emptyset \).

**Definition 6.** A configuration is a tuple \((E, \gamma, \phi, \sigma, \lambda)\) with:

- an evaluation stack \( E \), a typing function \( \phi \) for locations, and a closed store \( S \);

- an environment \( \gamma \) mapping names to values;

- an ownership function \( \lambda \in (\text{Loc} \times (O, \text{Fun}'))' \) ordering played names and mapping them to the party who has introduced them; and

which satisfies the following conditions:

- the relation \( \{ (a, X) | \lambda = \lambda_1 \cdot (a, X) \cdot \lambda_2 \} \) is a partial function and \( \lambda \) has no repetition of names.

- \( \text{dom}(\gamma) = \{ a \in \text{Pol} \cup \text{Fun} \cup \text{TVar} | \lambda(a) = \text{P} \} \)

- \( \text{dom}(\phi) = \{ l \in \text{Loc} \cap \text{dom}(\lambda) \subseteq \text{dom}(S) \}

- for all \( \in \nu(E, \text{cod}(\gamma), \text{cod}(\gamma)) \)

where, because of the first condition above, we write \( \lambda(a) = X \) if \( \lambda = \lambda_1 \cdot (a, X) \cdot \lambda_2 \) for some \( \lambda_1, \lambda_2 \).
In addition, we include special configurations of the form $\langle Δ; Σ; Γ \vdash M : θ \rangle$, one for each typed term $Δ; Σ; Γ \vdash M : θ$.

Thus, a configuration registers syntactic and semantic information on the execution of a term necessary to produce its traces. $E$ 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 $E[f v]$ 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 $AVal(u,θ)$ 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:

$$AVal(u, θ) = \{(v, γ, φ) : v ∈ U \text{ or } φ ∈ Γ \}$$

For uniformity, it makes sense to view types as values of special “universe” type $U$ and set $AVal(θ, U) = \{(α, [α \mapsto θ], φ) : α ∈ TVar\}$. 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:

$$arg(θ' \rightarrow θ) = θ' \quad arg(ψ α.θ) = U$$
$$ret_u(θ' \rightarrow θ) = θ \quad ret_u(φ α.θ) = θ[α/φ]$$

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 $AStore(S, φ)$ 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 6:

$$AStore(S, φ) = \bigcup_{I ∈ dom(φ)} \{(I [v \mapsto γ'] , φ') : (v, γ', φ') ∈ AVal(S(I), φ(I))\}$$

1 2 we also substitute via ρ, but this we discuss in the next section.

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 $Z[δ]$ 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 $Tr(C)$ be the set of all traces produced from $C$. Terms are translated by setting

$$[Δ; Σ; Γ \vdash M : θ] = comp(Tr(Δ; Σ; Γ \vdash M : θ))$$

for each typed term $Δ; Σ; Γ \vdash M : θ$, where $comp$ 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 $X'$, along with any new location names revealed in the store $S'$ to be played. Since this is a P-move then, all names in $X'$ 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 $γ' = γ \cdot γ_u \cdot γ_S$ and $φ' = φ \cup φ_u \cup φ_S$ (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 $P$ 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. $v(γ')$ and the same goes for the abstract store $S'$). This is enforced by the use of $v$ 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...
\[(\text{INT})\quad \langle (M, \theta) : E, \gamma, \phi, S, \lambda \quad \frac{}{} \quad (M', \theta) : E, \gamma, \phi', S', \lambda \rangle, \quad \text{given} \ (M, S) \to (M', S').\]

\[(\text{PA})\quad \langle (u, \theta) : E, \gamma, \phi, S, \lambda \quad \frac{g, \phi \in \text{AVal}(u, \theta)}{} \quad \langle E, \gamma, \phi' \quad \text{,} \quad \phi, S, \lambda \rangle \rangle \quad \text{, given} \ (v, \gamma_v, \phi_v) \in \text{AVal}(u, \theta)\). \]

\[(\text{PQ})\quad \langle \begin{array}{c} \left\langle \left( f [u: \theta] : E, \gamma, \phi, S, \lambda \quad \frac{g' \in \text{Fun}_{A_\theta}(u, \theta)}{f} \quad \left( E, \gamma', \phi' \right) : E, \gamma', \phi' \quad \text{,} \quad S', \lambda' \rangle \right. \\
\left. \left. \quad \text{given} \ f \in \text{Fun}_{A_\theta} \right\rangle \quad \frac{g \in \text{Fun}_{A_\theta} \land (f) \quad \theta' = \text{ret}(\theta)}{f \quad \text{,} \quad g} \quad \text{,} \quad \lambda' = \text{ret}(\lambda) \rangle \end{array} \]

\[(\text{OA})\quad \langle \begin{array}{c} \left\langle \left( E, \gamma, \phi, S, \lambda \quad \frac{g \in \text{Fun}_{\theta}(f, \theta)}{E, \gamma, \phi' \quad \text{,} \quad \phi', S', \lambda'} \right. \\
\left. \left. \quad \text{given} \ v, (\phi, \theta) \in \text{AVal}(u, \theta) \right\rangle \quad \frac{g' \in \text{Fun}_{\theta}(f, \theta)}{g} \quad \theta' = \text{ret}(\theta) \rangle \end{array} \]

\[(\text{OQ})\quad \langle \begin{array}{c} \left\langle \left( E, \gamma, \phi, S, \lambda \quad \frac{g \in \text{Fun}_{\theta}(f, \theta)}{E, \gamma, \phi' \quad \text{,} \quad \phi', S', \lambda'} \right. \\
\left. \left. \quad \text{given} \ v, (\phi, \theta) \in \text{AVal}(u, \theta) \right\rangle \quad \frac{g' \in \text{Fun}_{\theta}(f, \theta)}{g} \quad \theta' = \text{ret}(\theta) \rangle \end{array} \]

\[(\text{INT})\quad \langle \begin{array}{c} \left\langle \left( \Delta; \Sigma, \Gamma \to M : \theta \quad \frac{g \in \text{Fun}_{\theta}(f, \theta)}{\langle M, \theta \quad \frac{}{} \quad \langle \Delta, \Sigma, \Gamma \quad \frac{}{} \quad \text{,} \quad \theta \rangle \rangle \right. \\
\left. \left. \quad \text{given} \ \text{dom}(\Gamma) = \{ \cdot x_1, \cdot x_2 \}, \ (v, \phi) \in \langle \Delta, \Sigma, \Gamma \rangle \text{,} \quad \text{and} \ v = (\alpha, \tilde{t}, \tilde{u}) \rangle \quad \frac{\tilde{Z} \quad \text{above}}{\tilde{Z} \quad \text{is a term, context or stack, and} \quad \tilde{Z} = \{(z, \tilde{Z}(z)) \mid z \in \text{dom}(Z)\} \quad \text{if} \ \tilde{Z} \quad \text{is a map into terms.}} \end{array} \]

\[\tilde{Z} \quad \text{above}, \quad \tilde{Z} = \{ [\tilde{z}, \tilde{Z}((\tilde{z}))] \mid \tilde{z} \in \text{dom}(\tilde{Z}) \} \quad \text{if} \ \tilde{Z} \quad \text{is a map into terms.}\]

\[\begin{array}{c} \text{P1} \quad \lambda' = \{ \alpha, [P] \mid a \in \nu (v, S', \rho) \} \quad \text{and} \quad P = \nu (v, S', \rho) \quad \text{and} \quad each \nu (\lambda')_{\text{loc}} \text{is single-played in} \ (v, S', \rho) \\
\text{P2} \quad \text{for all} \ f \in \text{Fun}_{\nu}(S', v, \rho), \ f \notin \nu (v, S', \rho), \ \alpha \notin \nu (v, S', \rho) \\
\text{P3} \quad \nu_{\text{loc}}(\lambda') = S' \quad \text{and} \quad (S', \gamma, S', \lambda) \in \text{IntStore}(S', \gamma, S', \lambda) \\
\text{P4} \quad \text{for all} \ p \in \text{Fun}_{\nu}(S', v, \rho), \ \text{with} \ p \in \text{Pol}_{\nu}, \ \lambda (a) = P \quad \text{iff} \ p \notin \nu (v, S', \rho) \quad \text{and} \ \text{fresh}\]
4. Type Disclosure, Casts and $\kappa$-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 ref $\theta_1$, ref $\theta_2$, then a casting function between $\theta_1$ and $\theta_2$ is readily available. For instance, if the two variables are $x_1 : \text{ref } \theta_1$, here is a casting function from $\theta_1$ to $\theta_2$:

$$\text{cast}_1 \equiv \lambda z_1 : \theta_1. \ x_1 \equiv z_1 | x_2 : \theta_1 \to \theta_2$$

Clearly, if the same location $l$ flows in $x_1$ and $x_2$ then we obtain $\text{cast}_1([l/x_1, l/x_2])$ 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:

$$C \equiv \text{let } x = \text{ref } 0 \in (\lambda \alpha. \ \lambda y_1 : \text{ref } \alpha. \ \lambda y_2 : \text{ref } \text{Int}. \ \bullet) \text{Int } x \ x \text{ whereby } \theta_1 = \alpha \text{ and } \theta_2 = \beta.$$  

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 $f : (\text{ref } \text{Int} \times \text{ref } \text{Int}) \to \text{Unit}, g : \forall \alpha. \ \text{ref } \alpha \to \text{ref } \alpha$ and $\phi : \forall \alpha, \alpha'. (\alpha' \to \alpha) \times (\text{ref } \alpha \to \text{ref } \alpha') \to \alpha$.

Let $x, y = \text{ref } 0 \in f(x, y)$; let $u = g \text{Int } x$ in if $(u = y) 1 2 \equiv? let x, y = \text{ref } 0 \in f(x, y)$; let $u = g \text{Int } x$ in if $(u = y) 3 2 \equiv? let x = \text{ref } (\lambda y_1.1)$ in let $u = h \text{Int } x, (x, 0)$ in if $u 1 2 \equiv? let x = \text{ref } (\lambda y_1.1)$ in let $u = h \text{Int } x, (x, 0)$ 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-unicty of typing functions $\phi$. In order to restrict the behaviour of $\text{O}$ in the interaction to plausible computations, we shall impose some validity conditions to $\phi$; after all, not all types can be instantiations of the same type variable (for instance, $\phi \ell = \{\text{ref } \text{Int}, \text{ref } \text{Unit}\}$ 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 $\subseteq$ on types, indexed by an ordered set $\Phi$ of type variables, as:

$$\begin{array}{cccccc}
\theta \subseteq \theta & \theta_1 \subseteq \theta_2 & \theta_1 \leq \theta_2 & \nu_\theta(\theta) <_{\kappa} \alpha & \theta \leq \theta' & \\
& \theta \subseteq \theta & \theta_1 \subseteq \theta_2 & \theta_1 \leq \theta_2 & \theta_1 \leq \theta_2 & \\
\end{array}$$

Casts yield other casts. For example, a cast from $\theta_1 \times \theta_2$ to $\theta'_1 \times \theta'_2$ yields subcasts from $\theta_1$ to $\theta'_1$, and from $\theta_2$ to $\theta'_2$. We formalise this as follows. Given a cast relation $\kappa$, we define its closure $\kappa^*$ by:

$$(\theta, \theta') \in \kappa^* \Rightarrow (\theta, \theta') \in \kappa \quad (\theta, \theta') \in \kappa \quad (\theta, \theta') \in \kappa^*$$

$$(\theta_1 \times \theta_2, \theta'_1 \times \theta'_2) \in \kappa^* \Rightarrow (\theta_1 \times \theta_2, \theta'_1 \times \theta'_2) \in \kappa$$

$$(\theta, \theta') \in \kappa^* \Rightarrow (\theta', \theta) \in \kappa^*$$

$$(\theta_1 \times \theta_2, \theta'_1 \times \theta'_2) \in \kappa^* \Rightarrow (\theta_1 \times \theta_2, \theta'_1 \times \theta'_2) \in \kappa^*$$

4.2 The starred conditions

We next look at the use of environments $\rho$ and the conditions $\text{O}^*$ and $\text{P}^*$ which govern type disclosure in the interaction reduction.

Each move $(m, S, \rho)$ 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 $\phi$ to a (valid) superset $\phi \cup \phi''$: this is due to the fact that locations $l$ which up until then had types $\phi(l)$ are put in positions which expect types $\theta \in \phi(l)$ (e.g. in return position of some $f \in \text{Func}_{\kappa'}$).

This means that a type $\theta$ can have several minimal types in its cast class, and each of them needs to be taken in account when computing abstract values to be played in a move. Hence, minimal types are central to the (full) definitions of $\text{AVaL, AStore}$, etc.

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

In the sequel we will be using a very specific set $\Phi$, which we shall be leaving implicit. For any configuration $C$ with components $\lambda$ and $\phi$, we say that $\phi$ is valid if it is so with respect to the ordered set $\Phi$ of type variables obtained from $\lambda: \Phi = \pi_l(\lambda) \uplus \text{TVar}$.

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 $\kappa$, which are simply relations over types. The fact that $(\theta, \theta') \in \kappa$ means that we can cast values of type $\theta$ to $\theta'$.

$\kappa$ is the one we obtain via the updated typing function $\phi \cup \phi''$. Morever, as explained above, each concrete value $\gamma(p)$ of a polymorphic name $p$ needs to be partially revealed.

The degree to which the code of $\gamma(p)$ will be revealed is determined by the function $\text{AEEnv}$. That is, $\text{AEEnv}(\gamma(p))$, $\phi'$ comprises a new abstract environment $(\rho, \gamma(p), \phi')$ for these newly revealed values, that is moreover unique up to permutation of fresh names. The
first component ($\rho$) is the map from polymorphic names to their revealed values. The other two components record the location types ($\phi_j$) and value abstraction ($\gamma_j$) occurring via this disclosure. In the case of O, a similar disclosure occurs, only that this time there is no $\gamma$ to guide the revealed values; rather, $\delta$ supplies the disclosure in a non-deterministic fashion.

The definition of $\text{AEnv}$ and its O-counterpart are given below.

\[ \text{AEnv}(\gamma)_{\kappa,\tau} = \bigcup_{p \in \text{dom} \left( \gamma \right)} \left\{ \left( \left[ \text{E} \right]_{\kappa,\tau} \right) \cdot \phi \mid \phi = \bigcup_{\gamma \in \text{X}_{\phi,\gamma}} \phi \right\} \]  

\[ \text{E}[\phi]_{\kappa,\tau} = \bigcup_{p \in \text{E} \cdot \phi} \left\{ \left( \left[ \text{E} \right]_{\kappa,\tau} \right) \cdot \phi \mid \phi = \bigcup_{\gamma \in \text{X}_{\phi,\gamma}} \phi \right\} \]  

\[ \text{AVal}(u, \gamma)_{\kappa} = \left\{ \left( v, \gamma, \phi \right) \mid \phi = \bigcup_{\gamma \in \text{X}_{\phi,\gamma}} \phi \in \text{AVal}(u, \theta) \right\} \]  

\[ \text{AStore}(S, \phi)_{\kappa} = \bigcup_{\text{dom} \left( \phi \right) \cdot \text{X}_{\phi,\gamma}} \left\{ \left[ \left( \left[ \text{E} \right] \cdot \phi \right) \cdot \phi \mid \phi = \bigcup_{\gamma \in \text{X}_{\phi,\gamma}} \phi \right\} \right\} \]  

with $\text{dom}(\gamma) \in \text{Pol}$ and $\text{P}_{\phi} = \min \kappa' \left( \alpha \right) \cdot \min \kappa(\alpha)$ for $p \in \text{Pol}_{\alpha}$.

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 $\text{S} = \text{S} \cdot \phi$. We can now specify which configurations are valid. Let $\text{S} = \text{S} \cdot \phi$; then $\Delta' : \phi$ is valid if it is also valid in $\phi$.

We show that our model is sound, i.e., equality of term denotations implies contextual equivalence. In fact, we prove a stronger result (Theorem 5), whereby equality is replaced by a larger equivalence relation which rules out some over-distinguishing O behaviours.

### 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 5), 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 need to generalise it by allowing context-to-context to be multi-valued, i.e., use valid typing functions $\phi$ (instead of $\Sigma$), together with the new typing rule:

\[ \Delta; \phi; 1 \vdash M : \theta \vdash \phi \vdash \phi \vdash M : \theta' \]

We write $\Gamma ; \phi$ if $\forall \in \text{dom}(\Sigma). \exists \phi \in \phi(1). \forall \theta(\phi); \phi; 1 \vdash \phi(\theta) ; : \theta$.

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

**Lemma 13.** Given $\Delta; \phi; \Gamma ; \vdash M : \theta$ and $\Sigma ; \phi$ such that for all $p \in \nu(M, S) \cap \text{Pol}_{\alpha}$, $\min \left( \text{Cast}(\phi)(\alpha) \right) \supseteq \left( \alpha \right)$ either $(M, S)$ diverges or there exists $(M', S')$ irreducible such that:

- $(M, S) \Rightarrow^* (M', S')$.

- $M'$ is either equal to a value $v$ or to a callback $\text{E}[f, v]$.

- there exists $\phi'$ disjoint from $\phi$ such that $\Delta; \phi; \phi' ; : \Gamma ; \vdash M : \theta$ and $\Sigma; \phi'$.

Using this extended system, we can type evaluation stack configurations. A passive evaluation stack $\text{E} = (\text{E}_n, \text{E}_n \text{E}_n \Rightarrow \text{E}_n') : \vdash : (\text{E}_1, \text{E}_1 \Rightarrow \text{E}_1')$ is said to be well-typed w.r.t. a typing function $\phi$ and a type environment $\delta : \text{TVar} \Rightarrow \text{TVar}$ if, for all $1 \leq i \leq n$:

\[ \Delta; \phi ; \text{TVar} \Rightarrow \text{TVar} \]  

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

**Definition 14.** We call $\{(\gamma, \phi, \delta, \lambda, \alpha)\}$ a valid configuration if:

- $\text{dom}(\gamma) = \{ \alpha \in \text{Pol} \cup \text{Fun} \cup \text{TVar} \mid \text{Loc}(\alpha) \neq P \}$.

- $\text{dom}(\phi) = \text{dom}(\lambda) \cap \text{Loc} \subseteq \text{dom}(S)$.

- for all $p \in \nu(\text{E}, \text{cond}(\gamma)) \cap \text{Pol}_{\alpha}, \min \left( \text{Cast}(\phi)(\alpha) \right) = \left( \alpha \right)$.

We write $C \Rightarrow^* C'$ when $C \Rightarrow C'' \Rightarrow C'$ for some configuration $C''$. Validity of configurations is preserved as follows.

**Lemma 15.** If $C \Rightarrow^* C'$ 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 \((E, E_0, \gamma, \alpha, \phi, S, P, S_0)\), where:

- \(E, E_0\) are evaluation stacks (one passive and one active);
- \(\gamma, \gamma_0\) are environments; and \(S, S_0\) are stores;
- \(\phi\) 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 \(C \Rightarrow C_n\). We say that a composite configuration \(C\) terminates with the trace \(t\), written \(\bot\), if there exists a store \(S\) such that \(C \xrightarrow{((t, S), \phi)} \bot\).

We now define how to merge configurations \(C_1, C_2\) into a composite one. For each \(X \in \{O, P\}\) we write \(X^2\) for its dual \(\{\{X, X^2\} = \{O, P\}\}\), and extend this to \(\lambda^2 = (\lambda^1)^2\).

**Definition 16.** Given a pair of environments \((\gamma, \gamma_0)\) from \(\Lambda\) to \(\text{Loc}\) to values, we say these are compatible when:

- \(\operatorname{dom}(\gamma) \cap \operatorname{dom}(\gamma_0) = \emptyset\);
- for all \(X \in \{O, P\}\), \(\nu(\gamma_X(a)) \subseteq \operatorname{dom}(\gamma_X)\),
- setting \(\gamma^0 = \gamma \cdot \gamma_0\) and \(\gamma^1 = \{(a, v) \mid (a, v) \in \gamma^2\}(t > 0)\), there is an integer \(n\) such that \(\nu(\operatorname{code}(\gamma^0))\) is valid.

A pair of valid configurations \((C_1, C_2)\) are called compatible if, given \(X \in \{O, P\}\) such that \(\gamma_X = \gamma_X^0 = \gamma_X^1\):

- \(\phi_F = \phi_O\) and \(\lambda_F = \lambda_O\);
- \((\gamma, \gamma_0)\) are compatible and \(\operatorname{dom}(\gamma_0) \cdot \gamma_0 = \operatorname{dom}(\lambda_{P})\) or \(\operatorname{dom}(\lambda_{O})\),
- \(\gamma_0(S) \cap \operatorname{dom}(\gamma_0) = \operatorname{dom}(\lambda_{P}) \cap \operatorname{Loc}\),
- the merge \((E, E_0, \gamma, \gamma_0, \phi, S, P, S_0)\) of \(C_1\) and \(C_2\) is valid.

We write \(C_1 \bowtie C_2\) for the model abstracts away any actual values produced by \(C_1\) or \(C_2\), and there are polymorphic names whose value have been disclosed by their disclosed value; we extend equivalence to sets of traces in an elementwise fashion.

**Theorem 19.** For all pairs of compatible configurations \(C_1\) and \(C_2\), \(C_1 \bowtie C_2 \iff C_1 \bowtie C_2\).

5.3 Soundness result

We need two final pieces of machinery for soundness. The first result is so-called \(\text{cui-equiv}\), which allows one to characterise contextual equivalence by restricting focus to evaluation contexts.

**Definition 20.** Let \(\Sigma\) be a location context. Two terms \(\Delta, \Sigma; \Gamma \vdash M_1, M_2 : \theta\) are \(\text{cui-equiv}\), written \(\Delta, \Sigma; \Gamma \vdash M_1 \equiv_{\text{cui}} M_2 : \theta\), when for all typing substitutions \(\gamma \vdash \delta : \Delta, \Sigma\), closed contexts \(\Sigma' \subseteq \Sigma\), value substitutions \(\gamma \vdash \delta : \Gamma(\delta)\) and evaluation contexts \(\Delta \vdash E : \theta \vdash \theta'\), we have \(E[M_1(\gamma(\delta))] \vdash E[M_2(\gamma(\delta))] \vdash S\).

**Theorem 21.** \(\Delta, \Sigma; \Gamma \vdash M_1 \equiv_{\text{cui}} M_2 : \theta\) iff \(\Delta, \Sigma; \Gamma \vdash M_1 \equiv\theta\).

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 where 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 blinds 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 \(t \rightarrow t'\), if \(t = t_1 \cdot (m, S, \rho) \cdot t_2, t' = t_3 \cdot t_4, m, \rho\) is a \(P\)-move, and there are polymorphic names \(p, p'\) such that:

- \(p \notin (t') \cap (m, S, \operatorname{code}(\rho))\) is introduced in a \(P\)-move of \(t_1\),
- \(p' \notin (t_1)\) and \(t_2\) is \((m, S, \rho)\cdot t_4\) where we first replace a single occurrence of \(p\) in \((m, S, \operatorname{code}(\rho))\) by \(p'\), and then replace any \([p' \vdash v]\) in the resulting subtrace by \([p \vdash v' \cdot [p' \vdash v]\)

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

**Definition 22.** Two traces \(t_1, t_2\) are said to be equivalent, written \(t_1 \sim t_2\), if \(t_1 \sim t_2\), where \(t \sim t'\) is defined as:

\[
\begin{align*}
\mathcal{F}(t_1) & \sim \mathcal{F}(t_2), \\
\mathcal{F}(p \vdash \gamma) & \sim \mathcal{F}(p \vdash \gamma), \\
\mathcal{F}(p \vdash \gamma) & \times \mathcal{F}(p \vdash \gamma), \\
(\mathcal{F}(t_1) & \times \mathcal{F}(t_2)), \\
\end{align*}
\]

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

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

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 \(M_1 \equiv M_2\).

Prove. Suppose \([M_1] \sim [M_2]\). Using Theorem 21 we show that \(M_1 \equiv_{\text{cui}} M_2\). Let us take \(\delta\), \(\Sigma' \supseteq \Sigma, \gamma, \Delta, E\) and as in Definition 20 suppose that \(E[M_1(\gamma(\delta))]\) and write \(C_{P_1}\) for the P-configuration \((\gamma_{\text{ciu}}(\alpha, \lambda), \phi, S, \Delta, \Sigma, \Gamma \vdash M_1 : \theta)\).
Let $C_O = \{ (E, \theta \leadsto \theta') \}$, $\gamma' = \{ (u_i, v_i) \}$ and $\gamma = \{ (u_i, v_i) \}$. From Lemma 17, there exists a complete trace $t$ such that $C_{P_1} \equiv C_O \downarrow$. Then, from Theorem 19, $C_{P_1}(C_O \downarrow)$, so that $t \in \text{Tr}(C_{P_1})$ and $t' \in \text{Tr}(C_O)$. Writing $C_{P_2}$ for the Player configuration $(M_2, e, \phi, S, \lambda)$, from the hypothesis of the theorem, there exists a complete trace $t' \sim t$ such that $t' \in \text{Tr}(C_{P_2})$.

From Proposition 20, $t' \in \text{Tr}(C_O)$, so that $C_{P_2}(C_O \downarrow)$, and using Theorem 19 (in the other direction), we get that $C_{P_2} \equiv C_O \downarrow$. 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 over-approximates the casts that can be implemented by the context in practice, as inhabitation constraints are not taken into account. For instance, a cast from $\theta \leadsto \theta_1$ to $\theta \leadsto \theta_2$ does not yield one from $\theta_1$ to $\theta_2$ unless a value of type $\theta$ 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 $\alpha$ inside a ref $\theta$ must be reachable inhabited: in order for a value of type ref $\theta$ to be played in a trace, a value of type $\alpha$ must have been played before.

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

$$\text{good}_\Gamma(\text{ref } \theta) = \begin{cases} \text{true} & \Gamma \land \theta \text{ is quantifier-free} \\ \text{false} & \text{otherwise} \end{cases}$$

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

- $\text{gtv}(\alpha) = \{\alpha\}$
- $\text{gtv}(\theta \circ \phi) = \text{gtv}(\theta) \cup \text{gtv}(\phi')$
- $\text{gtv}(\text{ref } \alpha) = \text{gtv}(\alpha)$
- $\text{gtv}(\text{ref } \theta) = \text{gtv}(\theta)$

and $\text{gtv}(\theta) = \emptyset$ otherwise. We extend goodness to type interfaces by setting, given $\Sigma = \{ \{1_i, \theta_1, \ldots, \theta_n, \gamma\}, \Gamma = \{ \{x_i, \theta_i, \ldots, x_m, \theta_m\} \}$:

$$\text{good}(\Delta; \Sigma; \Gamma \vdash \theta : M) : \text{such that } \text{good}(\Delta; \Sigma; \Gamma) \land \theta \rightarrow \theta.$$

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

Moreover, we call an initial configuration $(\Delta; \Sigma; \Gamma \vdash M : \theta)$ good just if its interface is, while a valid configuration $(\Delta, \gamma, \phi, S, \lambda)$ is good just if, taking $X_\Delta = \{ \alpha \mid \nu(\lambda) \cap P_\Delta \neq \emptyset \}$, $\nu(\phi) \subseteq X_\Delta$ and $\text{good}_\Delta(\theta)$ hold, for all $\theta \in \text{cod}(\lambda) \cup \{ \theta \mid \nu(\lambda) \cap P_\Delta \neq \emptyset \}$.

We can then check that goodness is preserved under reduction.

In this restricted fragment, we can always implement all possible casts anticipated from the cast closure construction of Section 4. Moreover, a cast-term from $\theta$ to $\theta'$ based on aliased pairs $(\theta_1, \theta_1), \ldots, (\theta_n, \theta_n)$ and inhabited variables $\alpha_1, \ldots, \alpha_m$ is a cast-term $\text{cast}_{\theta \rightarrow \theta'}$ such that:

- $\Delta; x_i : \text{ref } \theta_i, y_i : \text{ref } \theta'_i, z_j : \text{ref } \theta_j \vdash \text{cast}_{\theta \rightarrow \theta'}(\Delta; \theta)$

- for any $\Sigma = \{ \{i : \theta_i\}, p_j \in \text{Pol}_\Delta, S : \Sigma \text{ and } \Delta; \Sigma' \vdash \theta : \theta'$, $(\text{cast}_{\theta \rightarrow \theta'}(i, x_i, y_i, z_j) \| S) \rightarrow (v', S', \theta')$ with $v' \equiv v'$, with $S'$ disjoint of $S$. Recall now Cast ($\phi$) from Definition 12 and define its restriction Cast ($\phi$), the closure of $(\Delta, \theta') \vdash \theta : \theta'$ using all cast closure rules from Section 4 apart from (\).

Lemma 27. Let $\phi$ be a valid typing function with $\bar{\alpha}$ all free type variables in $\phi$. Then, for all $(\theta, \theta') \in \text{Cast}(\phi)$ there is a cast-term $\text{cast}_{\theta \rightarrow \theta'}(\Delta; \theta)$.
where $Q_F$ contains a unique location $q_f$ for each function name $f$ in $\text{dom}(\gamma_0)$, $Q_F$ contains the $q_g$, $Q_P$ the $q_p$, and $Q_Q$ the $q_q$.

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 $\text{cnt}$ is an integer counter that counts the remaining $P$-moves in $t$. We set $S_O(\text{cnt}) = ||t||$. $L = \{e_1, \ldots, e_{\text{cnt}(t)}\}$ 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 getvals is to store all names that appear in the trace. For each $i$, getval, is a location of type:

$$3\alpha \times \left((\text{Int} \to \theta_1') \times \cdots \times (\text{Int} \to \theta_{\text{cnt}(t)})\right) \times \left((\text{Unit} \to \theta_1') \times \cdots \times (\text{Unit} \to \theta_{\text{cnt}(t)})\right)$$

where $\alpha$ is the sequence of all free type variables in $\Theta$. Thus, the value of getvals, is an existential package whose first component contains enumerations of all values of type $\theta_j'$, for each $i, j$. This is enough to represent all the available values at each point in the trace. The second component inside the package stored in getvals contains enumerations of all values of type $\theta_j$, for each $i, j$. We encode acceptance of the first two moves in $q_{\gamma_T}$, by setting $S_O(q_{\gamma_T}(||t||))$ to be:

- unpack getval$_{\text{val}_i}$ as $(\alpha', \zeta', h_i)$ in
- let $z$ = castPk$(z', h_i)$ in
- $\lambda x_0.\text{unpack N}_i$ as $(\beta_1, x_i)$ in unpack N$_i$ as $(\beta_1, x_i)$ in
- let val = ref $(\lambda \Omega_1.\ldots, \lambda_\Omega)$ in
- $\text{cnt} \leftarrow \text{Fshvals}; \text{Chkvals}; \text{Newvals}; \text{Setstor}; \text{Play}$ (*)

Since the type of getval$_{\text{val}_i}$ is fully existentially quantified, when we (statically) unpack getval$_{\text{val}_i}$ and get a, $z'$, the $\beta_1$ are distinct from the type variables $\beta_1$ in $\Theta$ and, consequently, each component $z_i : \text{Int} \to \theta_j'$ of $z'$ is not of the expected type $\text{Int} \to \theta_j$. 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 $q_{\gamma_T}$. Hence, we employ the castPk function which casts values of type $\theta_j'$ to $\theta_j$ in $z'$, using the locations in $L$ (each of type $\theta_j$) and their representations in $h$.

Each term $N_i$ is selected in such a way so that, using val and $x_0, x_1, \ldots, x_{\text{cnt}}$, it captures the precise position within $(m_1, S_1, \rho_1)$ which introduces the type variable $\beta_i$. Note that, here and below, in order to access the values of $\rho_i$ 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 $\beta_i (\lambda_\Omega)$. Also:

- $F$ shvals detects the positions inside $(m_1, S_1, \rho_1)$ that introduce fresh names and updates val by adding them as new values in their corresponding types. This yields an updated store $S'_O$.
- $\text{Chkvals}$ checks that $x_0$, the public part of $S_O$ and the values revealed by type disclosure are the ones expected, that is, $x_0, S_1, \rho_1$ 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 $\alpha$ 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.

- Newvals creates all the fresh locations of $(m_2, S_2)$ and stores them in the corresponding index of val. Moreover, for each name $i'$ of arrow type in $(m_2, S_2)$, Newvals includes a code portion creating a reference $q_{\gamma_T}$ 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 $\alpha$, Newvals includes code creating $q_{\gamma_T}$ and adding a function in val according to the type $\gamma_0(\alpha)$ (e.g. if an arrow type then we add $\lambda z. \text{Dup}_i(q_{\gamma_T})$, where $q_{\gamma_T}$ encapsulates an effect which allows its recognition in the future).

- 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 $M_1 \equiv M_2$ then $[M_1] \equiv [M_2]$.

**References**