A Fully Abstract Trace Semantics for General References

J. Laird[*]
e-mail:jiml@sussex.ac.uk

Dept. of Informatics, University of Sussex, UK.

Abstract. We describe a fully abstract trace semantics for a functional language with locally declared general references (a fragment of Standard ML). It is based on a bipartite LTS in which states alternate between program and environment configurations and labels carry only (sets of) basic values, location and pointer names. Interaction between programs and environments is either direct (initiating or terminating subprocedures) or indirect (by the overwriting of shared locations): actions reflect this by carrying updates to the shared part of the store.
The trace-sets of programs and contexts may be viewed as deterministic strategies and counter-strategies in the sense of game semantics: we prove soundness of the semantics by showing that the evaluation of a program in an environment tracks the interaction between the corresponding strategies. We establish full abstraction by proving a definability result: every bounded deterministic strategy of a given type is the trace-set of a configuration of that type.

1 Introduction

The conjunction of functional programming and general references is a powerful one — for example, it can describe both object-oriented and aspect-oriented [11] computation by translation. So it is not, perhaps, surprising that the behaviour of functional programs with locally bound references is difficult to reason about; they may exhibit a variety of subtle phenomena such as aliassing, and self-referencing and self-updating variables. In some respects, the higher-order "pointer-passing" exhibited by such programs is analogous to process-passing in a concurrent setting. The most significant differences between pointer-passing and process-passing are that the former typically takes place sequentially, between a program and its environment rather than between processes in parallel, and that pointers and locations may be passed through the medium of the store, where they persist until overwritten. In this paper, we describe a labelled transition system for a sequential functional language with general references which captures these aspects of reference passing, and use it to give a sound, complete and direct characterization of contextual equivalence.

Another objective of this research is develop the correspondence between labelled transition systems and another, more abstract form of trace-based representation of references. *Game semantics* has been used to give a denotational model [1] of essentially the same functional language with general references as the the one studied here. The principal difference is that the language in [1] includes "bad references" — objects of reference type which do not behave as location names — due to the encoding of imperative variables as read-write methods rather than pointers. Because of this it does not precisely characterize the way

[*] Supported by UK EPSRC grant GR/S72181.

names may be passed between functions and used to interact via the store in languages like Standard ML with only good variables. By passing location names directly, we are able to capture interaction via the shared part of the store, yielding a full abstraction result for such a language.

The languages modelled here and in [1] are equivalent when restricted to terms of purely functional type (which in any case suffice to represent bad variables). We first present a basic trace semantics which is sound and complete for this fragment, and in which there is a simple correspondence between actions over a given type, and moves in the corresponding game, which extends to relate (α-equivalence classes of) traces to justified, well-bracketed sequences (using pointer-names to determine justification pointers) and deterministic trace-sets to strategies. We prove a game-semantics-style "definability result", showing that every bounded "deterministic strategy" is the trace-set of a term.

The full LTS is based on the same set of actions and so is a conservative extension of the basic one as far as the functional fragment is concerned. It accounts for interaction through the store by adding an update of every shared location (as a pointer or atomic value) to each action. The soundness, definability and full abstraction results extend straightforwardly to this setting.

1.1 Related Work

Labelled transition systems for similar functional languages have been presented, including one for a language with (global) general references [6]. In this system, labels carry the contexts which are used to test the term; in the one describe here, labels are more basic, containing only (sets of) atomic values, and so the resulting trace semantics could be said to be more abstract. In this respect it resembles the triggering semantics of the higher-order π-calculus [10], which motivated its development. There are close parallels with the trace semantics of a core fragment of Java presented in [7], including the use of a merge operation on context stacks.

We may compare labelled transition systems with various approaches to reasoning about contextual equivalence in functional-imperative languages, such as the bisimulation based methods of Koutavas and Wand [8], which are sound and complete for an (untyped) language similar to the one described here, or the Hoare-logic style approach of Honda, Berger and Yoshida [3]. These could be seen as characterizes equivalence using a more restricted set of contexts, but trace semantics actually gives an explicit description of the (observationally relevant) behaviour of a term.

The correspondence between game semantics and pointer-based evaluation of functional languages was first investigated in [4], where the interaction between the strategies denoting two terms of the λ-calculus was shown to corespond to the "run" of the Krivine abstract machine evaluating the application of one to the other. This is very similar to the proof of a result used here to establish that trace inclusion is a precongruence: that the result of evaluating a program in a given environment is determined by the "parallel composition" of their traces. Another route to a trace-based representation of functional programs is by representing them as terms of the π-calculus, for which a sequential fragment has been identified by typing [2] and the connections with game semantics examined in [5].

2 A Functional Language with General References

The language we study, $\mathcal{L}$, is a small fragment of Standard ML. It is essentially the language modelled in [1], without bad variables (and restricted to the ground types `com` and `bool`, for simplicity) — a simply-typed functional language with assignment, dereferencing and declaration of general references. Thus types are given by the grammar:
$T ::= \mathtt{com} \mid \mathtt{bool} \mid T \Rightarrow T \mid \mathtt{var}[T]$
Terms are those of the simply-typed λ-calculus over these types, with the following constants:

Assignment $\mathtt{assign}_T : \mathtt{var}[T] \Rightarrow T \Rightarrow \mathtt{com}$
Dereferencing $!_T : \mathtt{var}[T] \Rightarrow T$
Variable declaration $\mathtt{ref}_T : T \Rightarrow \mathtt{var}[T]$
Conditional $\mathtt{If}_T : \mathtt{bool} \Rightarrow T \Rightarrow T \Rightarrow T$
Equality Testing $\mathtt{eq}_T : \mathtt{var}[T] \Rightarrow \mathtt{var}[T] \Rightarrow \mathtt{bool}$
Basic Values $() : \mathtt{com}$ and $\mathtt{tt}, \mathtt{ff} : \mathtt{bool}$

We assume that each variable has a unique type, so that any well-formed term M has a unique type $\mathtt{ty}(M)$ (although we shall generally omit typing annotations where they are not necessary). We sugar the language by writing $M := N$ for $(\mathtt{assign}\, M)\, N$, $M; N$ for $(\lambda x.N)\, M$ with x not free in N, $\mathrm{new}\, x := M.N$ for $(\lambda x.N)\,(\mathtt{ref}\, M)$ (or just $\mathrm{new}\, x.N$ if M is an arbitrary value, which we may select at any type), Ω for $(\mathrm{new}\, x.x := \lambda y.(!x\,()).!x)\,()$, and $M = N$ for $(\mathtt{eq}\, M)\, N$.

A key distinction will be between functional values (i.e. values of function type), which are λ-abstractions, function-constants and pointer names (variables of function-type) — and non-functional values, which are location names (variables of reference type) and the basic values $(), \mathtt{tt}, \mathtt{ff}$. We mark this distinction by using F, G for functional values and v, u, w for non-functional values; the two notions of value are given by the following grammars:
$F, G := x \mid \lambda x.N \mid\, ! \mid \mathtt{If} \mid \mathtt{eq} \mid \mathtt{assign} \mid (\mathtt{eq}\, v) \mid \mathtt{assign}\, v$
$v, u ::= a \mid () \mid \mathtt{tt} \mid \mathtt{ff}$
where x ranges over pointer names and a over location names. We write V, U for values of arbitrary type. *Atomic* values, for which we write ϕ, ψ, are basic values, pointer or location names.

Our LTS for $\mathcal{L}$ is based on a small-step operational semantics (Table 1). Terms are evaluated in an environment consisting of a set of pointers $\mathcal{B}$ — a finite partial function from pointer names to functional values and a store $\mathcal{S}$ — a finite partial function from location names to values. Non-functional values may be substituted directly, but substitution of a functional value is via a freshly generated pointer name bound to it. The reduction rules are based on typed *evaluation contexts* given by the following grammar:
$E[\_]_T := [\_]_T \mid F\, E[\_]_T \mid E[\_]_T\, M$
By typing the "hole", we ensure that any typable context $E_T[\_]$ has a unique type $\mathtt{ty}(E_T[\_])$. Throughout, we adopt the convention that in any rule $C \longrightarrow C'$, any variable mentioned explicitly in C' which is not mentioned explicitly in C is assumed to be fresh — i.e. not occurring free or bound in C. We write $M \Downarrow$ if $M; \_; \_ \longrightarrow^* (); \mathcal{B}; \mathcal{S}$ for some $\mathcal{B}, \mathcal{S}$. We define standard notions of observational approximation and equivalence: $M \lesssim N$ if $C[M] \Downarrow$ implies $C[N] \Downarrow$ for all compatible closing contexts $C[\_] : \mathtt{com}$. Our full abstraction result is given for closed terms, although the extension to terms with free variables is straightforward — note that $M \lesssim N$ if and only if $\lambda x.M \lesssim \lambda x.N$. We may prove the following Context Lemma using standard techniques.

$E[(\lambda x.M)\ v];\mathcal{B};\mathcal{S}$	$\longrightarrow E[M[v/x]];\mathcal{B};\mathcal{S}$	$E[(\lambda x.M)\ F];\mathcal{B};\mathcal{S}$	$\longrightarrow E[M[y/x]];\mathcal{B},(y,F);\mathcal{S}$
$E[x\ V];\mathcal{B},(x,F);\mathcal{S}$	$\longrightarrow E[F\ V];\mathcal{B},(x,F);\mathcal{S}$	$E[\texttt{ref}\ V];\mathcal{B};\mathcal{S}$	$\longrightarrow E[a];\mathcal{B};\mathcal{S},(a,V)$
$E[!l];\mathcal{B};\mathcal{S},(l,V)$	$\longrightarrow E[V];\mathcal{B};\mathcal{S},(l,V)$	$E[l := U];\mathcal{B};\mathcal{S},(l,V)$	$\longrightarrow E[()];\mathcal{B};\mathcal{S},(l,U)$
$E[l = l];\mathcal{B};\mathcal{S}$	$\longrightarrow E[\texttt{tt}];\mathcal{B};\mathcal{S}$	$E[l = m];\mathcal{B};\mathcal{S}$	$\longrightarrow E[\texttt{ff}];\mathcal{B};\mathcal{S}$
$E[\texttt{If tt}];\mathcal{B};\mathcal{S}$	$\longrightarrow E[\lambda x.\lambda y.x];\mathcal{B};\mathcal{S}$	$E[\texttt{If ff}];\mathcal{B};\mathcal{S}$	$\longrightarrow E[\lambda x.\lambda y.y];\mathcal{B};\mathcal{S}$

Table 1. Reduction Rules for Program Evaluation

Lemma 1 (Context Lemma). *For any closed terms $M, N{:}T$, $M \lesssim N$ if for every evaluation context $E[\_]_T : \texttt{com}$, and store $\mathcal{S}$, $E[M];\_;\mathcal{S} \Downarrow$ implies $E[N];\_;\mathcal{S} \Downarrow$.*

3 Trace Semantics

We shall develop our trace semantics for $\mathcal{L}$ by first giving a basic LTS, and proving that it yields a sound and complete model of the functional fragment of $\mathcal{L}$ — i.e. for terms of type T, where T contains no reference type as a subtype. We shall then extend the LTS by adding a "shared store" component to its actions, and show that the resulting semantics is sound and complete for all terms of $\mathcal{L}$. This two-stage presentation of the trace semantics allows us to capture two aspects of behaviour in $\mathcal{L}$ separately — first, the control-flow between programs at function types, and then shared access to the store. It also allows a clearer perspective on the correspondence with the game semantics of general references. Note that we can give a conservative translation of $\mathcal{L}$ (except for equality testing on reference names) into the functional fragment by representing the references as objects of functional type with the correct assignment/dereferencing behaviour (this is how references are represented in the games model [1]): for example we may replace $\texttt{var}[T]$ with the type $\texttt{bool} \Rightarrow T \Rightarrow T$ and give the following macros for assignment, dereferencing and declaration.

- $\texttt{assign} \mapsto \lambda x.\lambda y.((x\ \texttt{tt})\ y);()$
- $! \mapsto \lambda y.((y\ \texttt{ff})\ c)$ (where c is an arbitrary value)
- $\texttt{ref} \mapsto \lambda x.\mathrm{new}\, n := x.\lambda y.\lambda z.\texttt{If}\ y\ \texttt{then}\ (n := z); z\ \texttt{else}\ !n$

The translation becomes fully abstract if we include a bad variable constructor $\texttt{mkvar} : (\texttt{bool} \Rightarrow T \Rightarrow T) \Rightarrow \texttt{var}[T]$ with the appropriate operational behaviour making $\texttt{var}[T]$ a retract of $\texttt{bool} \Rightarrow T \Rightarrow T$.

The trace semantics is given by a bipartite LTS in which nodes are partitioned between *program-configurations* and *environment-configurations.* The former are tuples $(M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E})$ consisting of a program M, internal pointers B and store $\mathcal{S}$, a set of "output pointers" $\mathcal{O}$, a set of typed "input pointer" names $\mathcal{I}$ and an "execution stack" $\mathcal{E}$ of evaluation contexts, the sets $\mathsf{dom}(\mathcal{B}), \mathsf{dom}(\mathcal{O}), \mathcal{I}$ being pairwise disjoint. An environment-configuration E is a tuple $(\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E})$ — there is no program.

The type of the program configuration $(M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E})$ is the tuple $(\mathsf{ty}(M);\mathsf{dom}(\mathcal{O});\mathcal{I};\mathsf{ty}(\mathcal{E}))$, where $\mathsf{ty}(\mathcal{E})$ is the sequence of hole and result types of the evaluation contexts on $\mathcal{E}$ — i.e. $\mathsf{ty}(E_1[\_]_{S_1}){:}\ldots{:}\mathsf{ty}(E_n[\_]_{S_n})| = S_1 {\cdot} \mathsf{ty}(E_1[\_]_{S_1}) {\cdot} \ldots S_n {\cdot} \mathsf{ty}(E_n[\_]_{S_n})$. The type of an environment configuration $(\mathcal{B},\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E})$ is the tuple $(\mathsf{dom}(\mathcal{O}),\mathcal{I},\mathsf{ty}(\mathcal{E}))$. A configuration is functional if none of its subtypes is a reference type.

The actions of the LTS are either unlabelled (silent) or labelled. The unlabelled actions are the reductions of the small-step semantics — i.e.:

$$\frac{M;\mathcal{B};\mathcal{S} \longrightarrow M';\mathcal{B}';\mathcal{S}'}{M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E} \longrightarrow M';\mathcal{B}';\mathcal{S}';\mathcal{O};\mathcal{I};\mathcal{E}}$$

Labelled actions come in complementary pairs $\alpha, \overline{\alpha}$ (an input action and an output action). We write $s^{\perp}$ for the trace in which each output action is swapped for its corresponding input and vice-versa. Input actions send E-configurations to P-configurations and output actions send P-configurations to E-configurations and so traces alternate between them. Actions take one of the following two forms.

– Query: $x\langle\phi\rangle, \overline{x\langle\phi\rangle}$, where $a : S \Rightarrow T$ is a pointer name and $\phi : S$ is an atomic value.
– Response: $\langle\phi\rangle, \overline{\langle\phi\rangle}$, where ϕ is an atomic value.

In each case the atomic value inside the angled brackets is either non-functional (a value passed directly) or functional (a freshly generated pointer name which can be used to pass information back to the generating term). Thus there are eight transitions (input/output, functional/non-functional query/response) generated as follows:

Query Output A program which encounters an output pointer name uses it to send a query containing its argument to the environment, and pushes the evaluation context onto the execution stack. If the argument to the pointer is non-functional it is passed directly. If it is functional, it is passed as a fresh output pointer.

$$E[x\,v]; \mathcal{B}, \mathcal{S}; \mathcal{O}; \mathcal{I}, x; \mathcal{E} \xrightarrow{\overline{x\langle v\rangle}} \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}, x; E[\_] \colon \mathcal{E},$$
$$E[x\,F]; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}, x; \mathcal{E} \xrightarrow{\overline{x\langle y\rangle}} \mathcal{B}; \mathcal{S}; \mathcal{O}, (y, F); \mathcal{I}, x; E[\_] \colon \mathcal{E}$$

Query Input An environment which receives a query $x\langle\phi\rangle$ evolves into a program by following the input pointer x and applying the result to ϕ.

$$\mathcal{B}; \mathcal{S}; \mathcal{O}, (x, F); \mathcal{I}; \mathcal{E} \xrightarrow{x\langle u\rangle} F\,u; \mathcal{B}; \mathcal{S}; \mathcal{O}, (x, F); \mathcal{I}; \mathcal{E},$$
$$\mathcal{B}; \mathcal{S}; \mathcal{O}, (x, F); \mathcal{I}; \mathcal{E} \xrightarrow{x\langle y\rangle} F\,y; \mathcal{B}; \mathcal{S}; \mathcal{O}, (x, F); \mathcal{I}, y; \mathcal{E}$$

Response Output A program which produces a value passes this as a response action, either directly (non-functional values) or as a pointer to which it is bound.

$$v; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E} \xrightarrow{\overline{\langle v\rangle}} \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E}$$
$$F; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E} \xrightarrow{\overline{\langle y\rangle}} \mathcal{B}; \mathcal{S}; \mathcal{O}, (a, F); \mathcal{I}; \mathcal{E}$$

Response Input An environment which receives a response evolves into a program by placing it inside an evaluation context popped from the top of the stack.

$$\mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; E[\_] \colon \mathcal{E} \xrightarrow{\langle v\rangle} E[v]; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E}$$
$$\mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; E[\_] \colon \mathcal{E} \xrightarrow{\langle y\rangle} E[y]; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}, y; \mathcal{E}$$

We obtain a bipartite LTS with the same set of labels, but having configuration types as nodes, by simply restricting each configuration in the above actions to its type — i.e.

$$\frac{M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E} \xrightarrow{\alpha} \mathcal{B};\mathcal{S};\mathcal{O}';\mathcal{I}';\mathcal{E}'}{(\mathsf{ty}(M);\mathsf{dom}(\mathcal{O});\mathcal{I};\mathsf{ty}(\mathcal{E})) \xrightarrow{\alpha} (\mathsf{dom}(\mathcal{O}');\mathcal{I}';\mathsf{ty}(\mathcal{E}'))} \qquad \frac{\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E} \xrightarrow{\alpha} M;\mathcal{B};\mathcal{S};\mathcal{O}';\mathcal{I}';\mathcal{E}'}{(\mathsf{dom}(\mathcal{O});\mathcal{I};\mathsf{ty}(\mathcal{E})) \xrightarrow{\alpha} (\mathsf{ty}(M);\mathsf{dom}(\mathcal{O}');\mathcal{I}';\mathsf{ty}(\mathcal{E}'))}$$

Our full abstraction result is proved for sets of *complete* traces: a trace s is complete for a program-configuration with stack $\mathcal{E}$ if the number of response actions in s is (one) greater than the number of queries plus the depth of the stack $\mathcal{E}$. An environment-trace is complete if the number of responses is equal to the number of queries plus the depth of the stack. It is straightforward to show that if $C \overset{s}{\Rightarrow} C'$ then s is complete if and only if C' is an environment-configuration with empty stack. We write $[\![C]\!]$ (resp. $[\![\theta]\!]$) for the set of complete traces of the configuration C (resp configuration type θ). By definition, if $C : \theta$ then $[\![C]\!] \subseteq [\![\theta]\!]$. Moreover, we will show that if $s \in [\![\theta]\!]$ then there exists $C : \theta$ with $s \in [\![C]\!]$. For a closed term $M : T$ we define $[\![M]\!] = [\![M; \_; \_; \_; \_; \varepsilon]\!]$ — in this case a trace is complete if the number of responses is one greater than the number of queries. As an example, we give the derivation of a complete trace — $\overline{\langle y\rangle} y\langle g\rangle \overline{g\langle \mathtt{tt}\rangle} \langle ()\rangle \overline{\langle ()\rangle}$ — of $\lambda f.(f\ \mathtt{tt})\colon (\mathtt{bool} \Rightarrow \mathtt{com}) \Rightarrow \mathtt{com}$:

$\lambda f.(f\ \mathtt{tt}); \_; \_; \_; \_; \varepsilon \xrightarrow{\overline{\langle y\rangle}} \_; \_; (y, \lambda f.(f\ \mathtt{tt})); \_; \varepsilon \xrightarrow{y\langle g\rangle} (\lambda f.(f\ \mathtt{tt}))\ g; \_; \_; (y, \lambda f.(f\ \mathtt{tt})); g; \varepsilon$
$\longrightarrow h\ \mathtt{tt}; (h, g); \_; (y, \lambda f.(f\ \mathtt{tt})); g; \varepsilon \longrightarrow (\lambda z.z)\ (g\ \mathtt{tt}); (h, g); \_; (y, \lambda f.(f\ \mathtt{tt})); g; \varepsilon$
$\xrightarrow{\overline{g\langle \mathtt{tt}\rangle}} (h, g); \_; (y, \lambda f.(f\ \mathtt{tt})); g; \lambda z.z\, [\_] \xrightarrow{\langle ()\rangle} (\lambda z.z)\ (); (h, g); \_; (y, \lambda f.(f\ \mathtt{tt})); g; \varepsilon$
$\longrightarrow (); (h, g); \_; (y, \lambda f.(f\ \mathtt{tt})); g; \varepsilon \xrightarrow{\overline{\langle ()\rangle}} (h, g); \_; (y, \lambda f.(f\ \mathtt{tt})); g; \varepsilon.$

We will not examine the correspondence between the set of traces of a term and its denotation as a strategy in the games model of [1] in detail. However, readers familiar with game semantics may note that there is a simple correspondence between (α-equivalence classes of) traces and well-bracketed *alternating justified sequences* in dialogue games: we replace each output/input query/response with a Player/Opponent Question/Answer, and add a "justification pointer" from each query $x\langle\phi\rangle$ or $\overline{x\langle\phi\rangle}$ to the query/response in which x was introduced (if any), and from each response to the pending question. This correspondence sends the trace-set of each term to (the complete sequences of) a deterministic strategy on the associated "arena". We may show that it preserves the meaning of functional $\mathcal{L}$-terms.

3.1 Soundness and Completeness

We will now prove inequational soundness for terms in the functional fragment — i.e. $[\![M]\!] \subseteq [\![N]\!]$ implies $M \lesssim N$. To show that complete trace inclusion is a precongruence, we prove that the "parallel composition" of two *compatible* functional configurations produces the response $\overline{\langle ()\rangle}$ if and only if evaluating them in combination converges to $()$.

Definition 1. *Let $P = (M; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E})$ be a functional program-configuration, and $E = (\mathcal{B}'; \mathcal{S}'; \mathcal{O}'; \mathcal{I}'; \mathcal{E}')$ a functional environment-configuration. We say that P and E are* compatible *if:*

1. $\mathsf{dom}(\mathcal{O}) = \mathcal{I}'$ *and* $\mathsf{dom}(\mathcal{O}') = \mathcal{I}$,
2. $\mathsf{dom}(\mathcal{B}) \cap \mathsf{dom}(\mathcal{B}') = \mathsf{dom}(\mathcal{S}) \cap \mathsf{dom}(\mathcal{S}') = \varnothing$
3. $\mathsf{ty}(M) \cdot \mathsf{ty}(\mathcal{E}) = \mathsf{ty}(\mathcal{E}') \cdot \mathtt{com}$ *or* $\mathsf{ty}(M) \cdot \mathsf{ty}(\mathcal{E}) \cdot \mathtt{com} = \mathsf{ty}(\mathcal{E}')$.

if P and E are compatible, we write $P|E\downarrow$ if there exists a trace (a *witness*) s such that $s\overline{\langle ()\rangle} \in [\![P]\!]$ and $s^{\perp} \in E$ or $s \in [\![P]\!]$ and $s^{\perp}\overline{\langle ()\rangle} \in E$.

Definition 2. *The* merge *of P and E is a program-in-environment defined: $P \mathbin{/\!\!\!\wedge\!\!\!\backslash} E = (\mathcal{E} \mathbin{|||} \mathcal{E}')[M], \mathcal{B} \cup \mathcal{B}' \cup \mathcal{O} \cup \mathcal{O}', \mathcal{S} \cup \mathcal{S}'$, where $(\mathcal{E} \mathbin{|||} \mathcal{E}')[\_]$ — the unfolding of $\mathcal{E}, \mathcal{E}'$ — is an evaluation context, defined (up to α-equivalence) as follows:*

– $(\mathcal{E} \mathbin{|||} \varepsilon)[\_] = [\_]$

– $(\mathcal{E} \mathbin{|||} E[\_]\colon \mathcal{E}') = (\mathcal{E}' \mathbin{|||} \mathcal{E})[\lambda x.E[x]\ \_]$ *(where x is not free in $E[\_]$).*

Then $(\mathcal{E} \mathbin{|||} \mathcal{E}')[\_] :$ `com` *by the compatibility condition on stack types.*

We need to show that if P, E are compatible configurations then $P|E \downarrow$ if and only if $P \mathbin{/\!\!\wedge\!\!\backslash} E \Downarrow$. We prove this by altering the operational semantics so that it is equivalent with respect to termination, but more closely tracks the interaction between term and environment. Specifically, we replace the rule $E[x\,V];\mathcal{B},(x,F);\mathcal{S} \longrightarrow E[F\,V];\mathcal{B},(x,F);\mathcal{S}$ with two rules, depending on whether the value V is functional or non-functional.

$$\begin{array}{l} E[x\,v];\mathcal{B},(x,F);\mathcal{S} \longrightarrow (\lambda z.E[z])\ (F\,v);\mathcal{B},(x,F);\mathcal{S} \\ E[x\,G];\mathcal{B},(x,F);\mathcal{S} \longrightarrow (\lambda z.E[z])\ (F\,y);\mathcal{B},(x,F),(y,G);\mathcal{S} \end{array}$$

Writing $M;\mathcal{B};\mathcal{S} \Downarrow'$ if $M;\mathcal{B};\mathcal{S}$ reduces to $();\mathcal{B}';\mathcal{S}'$ under the modified rules, we may prove equivalence to the original notion of termination using a simulation relation between the two reduction systems.

Proposition 1. *For any program-in-environment $M;\mathcal{B};\mathcal{S} \Downarrow$ if and only if $M;\mathcal{B};\mathcal{S} \Downarrow'$.*

Proposition 2. *If P, E are compatible configurations then $P|E \downarrow$ if and only if $P \mathbin{/\!\!\wedge\!\!\backslash} E \Downarrow'$.*

Proof. We prove the implication from left to right by induction on the length of the witness s to $P|E \downarrow$. If this is the empty trace then $\overline{\langle()\rangle}$ is a complete trace in $[\![P]\!]$ and the empty trace is a complete trace in $[\![E]\!]$ — i.e. $\mathcal{E} = \mathcal{E}' = \varepsilon$ and so $(\mathcal{E} \mathbin{|||} \mathcal{E}')[M] = M$ and $M, \mathcal{B}, \mathcal{S} \twoheadrightarrow (), \mathcal{B}'','\,xs''$ and so $(\mathcal{E} \mathbin{|||} \mathcal{E}')[M], \mathcal{B} \cup \mathcal{B}', \mathcal{O} \cup \mathcal{O}', \mathcal{S} \cup \mathcal{S}' \Downarrow$ as required. Otherwise $s = \overline{\alpha}t$ and $P \twoheadrightarrow P' \xrightarrow{\overline{\alpha}} E'$ and $E \xrightarrow{\alpha} P''$ for some P'', E' such that $P''|E' \downarrow$, and so $P'' \mathbin{/\!\!\wedge\!\!\backslash} E' \Downarrow$ by hypothesis. By considering each pair of actions $\overline{\alpha}, \alpha$, we show that $P \mathbin{/\!\!\wedge\!\!\backslash} E \twoheadrightarrow P''| \mathbin{/\!\!\wedge\!\!\backslash} E'$ and so $P \mathbin{/\!\!\wedge\!\!\backslash} E \Downarrow$ as required.

We prove the implication from right to left by induction on the length of the reduction of $P \mathbin{/\!\!\wedge\!\!\backslash} E$ to $()$. If $(\mathcal{E} \mathbin{|||} \mathcal{E}')[M] = ()$ then the witness to $P|E \downarrow$ is ε. If $P \longrightarrow P'$ then we may apply the induction hypothesis to P', E. Otherwise, $P \xrightarrow{\overline{\alpha}} E'$ and $E \xrightarrow{\alpha} P'$. By considering the possible α, we show that $P \mathbin{/\!\!\wedge\!\!\backslash} E \longrightarrow P' \mathbin{/\!\!\wedge\!\!\backslash} E'$ and hence $P'|E' \downarrow$ by hypothesis, and so $P|E \downarrow$ as required.

Proposition 3 (Soundness). *If $[\![M]\!] \subseteq [\![N]\!]$ then $M \lesssim N$.*

Proof. Suppose $M \not\lesssim N$. Then by the Context Lemma there exists $E[\_];\mathcal{B};\mathcal{S}$ such that $E[M];\mathcal{B};\mathcal{S} \Downarrow$ and $E[N];B;\mathcal{S} \not\Downarrow$. Hence there exists $s\overline{\langle()\rangle} \in [\![\mathcal{B};\mathcal{S};\_;\_;E[\_]]\!]$ such that $s^\perp \in [\![M]\!]$ and $s^\perp \notin N$ — i.e. $[\![M]\!] \not\subseteq [\![N]\!]$ as required.

We now show that our trace semantics is complete for functional terms by proving a *definability* result: every bounded α-equivalence class of complete traces over a given configuration type which branches only on input actions is the set of traces generated by a configuration of the corresponding type.

Definition 3. *A* strategy *on a configuration type θ is a subset $\sigma \subseteq [\![\theta]\!]$ which is closed under α-equivalence* [1] *and complete prefixes, and which branches only on input actions — i.e. if $s\overline{a}t, s'\overline{a}'t \in \sigma$ and $s \sim_\alpha s'$ is even-length then $sa \sim_\alpha s'a'$. A strategy on a program-configuration type is a p-strategy, a strategy on an environment-configuration type is an e-strategy.*

[1] Traces in $[\![\theta]\!]$ are α-equivalent if they are obtainable by permutation of variables not in θ.

It is straightforward to show that if $C\colon\theta$ then $[\![C]\!]$ is a strategy on θ.

Proposition 4. *For any configuration type θ, every bounded strategy σ on θ is the trace of a configuration C_σ of type θ.*

Proof. By induction on the maximum length of trace, with the additional hypothesis that if $C_\sigma = (M_\sigma; \mathcal{B}_\sigma; \mathcal{O}_\sigma; \mathcal{I}_\sigma; \mathcal{E}_\sigma)$ or $(\mathcal{B}_\sigma; \mathcal{O}_\sigma; \mathcal{I}_\sigma; \mathcal{E}_\sigma)$ then $\mathcal{B} = \varnothing$ and:

– $\mathcal{O}_\sigma = \{(x_i, \lambda z.!q_i\, z) \mid i \leq m\}$, where each q_i is a distinct location name.
– $\mathcal{E}_\sigma = (\lambda z.!r_n\, z)\,[\_]\colon \ldots \colon (\lambda z.!r_1\, z)\,[\_]$, where each r_i is a distinct location name.
– $\mathsf{dom}(\mathcal{S}_\sigma) = \{q_i \mid i \leq m\} \cup \{r_j \mid j \leq n\}$

So suppose σ is a p-strategy, then either $\sigma = \{\varepsilon\}$ (in which case set $M = \Omega$) or $\sigma = \{\varepsilon\} \cup \alpha \cdot \tau$, where τ is an e-strategy, which is definable as above by hypothesis. We define $\mathcal{O}_\sigma, \mathcal{B}_\sigma, \mathcal{E}_\sigma$ as above, set $\mathcal{S}_\sigma = \mathcal{S}_\tau \lceil q_1, \ldots q_m, r_1, \ldots, r_n$ and define M_σ by analysis of α:

– If $\alpha = \overline{y\langle x_{m+1}\rangle}$ then $M_\sigma = (\lambda z.!r_{n+1}\, z)\,(y\, \lambda z.!q_{m+1}\, z)$.
– If $\alpha = \overline{y\langle v\rangle}$ then $M_\sigma = (\lambda z.!r_{n+1}\, z)\,(y\, v)$.
– If $\alpha = \overline{\langle x_{m+1}\rangle}$ then $M_\sigma = \lambda z.!q_{m+1}\, z$
– If $\alpha = \overline{\langle v\rangle}$ then $M_\sigma = v$.

Then $C_\sigma \xrightarrow{\overline{\alpha}} C'$, for some C' which is trace-equivalent to τ.

An action α is enabled at θ if there exists s such that $\alpha s \in [\![\theta]\!]$. If σ is an e-strategy, let $\widehat{\sigma}$ be the function from enabled actions to p-strategies defined $\widehat{\sigma}(\alpha) = \{s \mid \alpha s \in \sigma\}$. By inductive hypothesis, for each enabled action α, $\widehat{\sigma}(\alpha)$ is definable as a configuration of the specified kind. We define $\mathcal{O}_\sigma, \mathcal{B}_\sigma, \mathcal{E}_\sigma$ as above, and define $\mathcal{S}_\sigma$ (with $\mathsf{dom}(\mathcal{S}_\sigma) = \{q_1, \ldots, q_m, r_n\}$) as follows: Let $\text{assign}(\phi, m, n) = q_1 := \mathcal{S}_{\widehat{\sigma}(x_i\langle\phi\rangle)}(q_1); \ldots; q_m := \mathcal{S}_{\widehat{\sigma}(x_i\langle\phi\rangle)}(q_m); r_n := \mathcal{S}_{\widehat{\sigma}(x_i\langle\phi\rangle)}(r_n)$. For each $x_i : S_i \Rightarrow T_i \in \mathsf{dom}(\mathcal{O})$,

– If $S_i = \mathtt{com}$, then $\mathcal{S}_\sigma(q_i) = \lambda z.\text{assign}((), m, n); M_{\widehat{\sigma}(x_i\langle()\rangle)}$
– If $S_i : \mathtt{bool}$ then $\mathcal{S}_\sigma(q_i) =$
$\mathtt{If}\ z\ \mathtt{then}\ \text{assign}(\mathtt{tt}, m, n); M_{\widehat{\sigma}(x_i\langle\mathtt{tt}\rangle)}\ \mathtt{else}\ \text{assign}(\mathtt{ff}, m, n); M_{\widehat{\sigma}(x_i\langle\mathtt{ff}\rangle)}$
– $\mathcal{S}_\sigma(q_i) = \lambda y.\text{assign}(y, m, n); M_{\widehat{\sigma}(x_i\langle y\rangle)}$ otherwise.

Suppose $L = S \cdot T \cdot L'$. Let $\mathcal{S}_\sigma(r_i) = \lambda x.\Omega$ for $i < n$ and:

– If $S : \mathtt{com}$ then $\mathcal{S}_\sigma(r_n) = \lambda z.\text{assign}((), m, n); M_{\widehat{\sigma}(\langle()\rangle)}$,
– If $S : \mathtt{bool}$ then let $\mathcal{S}_\sigma(r_n) =$
$\lambda z.\mathtt{If}\ z\ \mathtt{then}\ \text{assign}(\mathtt{tt}, m, n-1); M_{\widehat{\sigma}(\langle\mathtt{tt}\rangle)}\ \mathtt{else}\ \text{assign}(\mathtt{ff}, m, n-1); M_{\widehat{\sigma}(\langle\mathtt{ff}\rangle)}$,
– $\mathcal{S}_\sigma(r_n) = \lambda y.\text{assign}(y, m, n-1); M_{\widehat{\sigma}(\langle y\rangle)}$ otherwise.

Then for each α there exists C' such that $C_\sigma \xrightarrow{\alpha} C'$, where C' is trace-equivalent to $C_{\widehat{\sigma}(\alpha)}$.

We may now complete the proof of full abstraction. We may prove the following lemma by induction on trace length.

Proposition 5. *For any closed terms $M, N : T$, if $M \lesssim N$ then $[\![M]\!] \subseteq [\![N]\!]$.*

Proof. Suppose $[\![M]\!] \not\subseteq [\![N]\!]$. Let s be a minimal length complete sequence in $[\![M]\!]$ such that $s \notin [\![N]\!]$. It is straightforward to show that if $t \in [\![\Gamma; \Delta; T; L]\!]$ then $t^{\perp}\overline{\langle()\rangle} \in [\![\Delta; \Gamma; T : L \cdot \mathtt{com}]\!]$, and so in particular $s^{\perp}\overline{\langle()\rangle} \in [\![\_; \_; T \cdot \mathtt{com}]\!]$. So by Proposition 4, there exists a configuration $E = (\_; \mathcal{S}; \_; \_; E[\_]_T)$ such that $[\![E]\!]$ consists of the complete prefixes of $\{s^{\perp}\overline{\langle()\rangle}\}$. Then $(M, \_; \_; \_; \_)|E \downarrow$ and $(N; \_; \_; \_; \_)|E \not\downarrow$, and so $E[M]; \_; \mathcal{S} \Downarrow$ and $E[M]; \_; \mathcal{S} \not\Downarrow$. Then if the location names occurring in $E[\_], \mathcal{S}$ are $l_1, \ldots, l_k$, and $\mathsf{dom}(\mathcal{S}) = (l_{j_1}, \ldots, l_{j_n})$, the required separating context is new $l_1 \ldots$ new $l_k.l_{j_1} := \mathcal{S}(l_{j_1}); \ldots; l_{j_n} := \mathcal{S}(l_{j_n}); E[\_]$.

4 Trace Semantics: Reference Types

The basic LTS fails to capture the behaviour of terms at reference types because it takes no account of implicit interaction between program and environment through shared locations in the store. We extend our LTS so that it is sound and complete for terms of all types by retaining the same basic set of actions but extending them with a component characterizing the action of the term/environment on shared locations. Since programs of purely functional type never can share the name of any location with the environment, the store component for such terms is always empty, so this is (essentially) a conservative extension of the basic LTS as far as such terms are concerned.

We extend the notion of configuration with a set of shared location names $\mathcal{N}$, i.e. a program-configuration is a tuple $(M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E};\mathcal{N})$ and an environment-configuration a tuple $(\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E};\mathcal{N})$, with $\mathcal{N} \subseteq \mathsf{dom}(\mathcal{S})$[2] in each case. We add the set of shared names to the configuration type — i.e. $(M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E};\mathcal{N})$ has type $(\mathsf{ty}(M);\mathsf{dom}(\mathcal{O});\mathcal{I};\mathsf{ty}(\mathcal{E});\mathcal{N})$.

Labels of the extended LTS are pairs $(\alpha,\mathcal{X})$, where α is an output/input query/response and $\mathcal{X}$ is a closed, atom-valued store — a finite partial function from location names to atomic values (basic values, location names, or (fresh) pointer names) such that

- if $(a,b) \in \mathcal{X}$, where b is a location name, then $b \in \mathsf{dom}(\mathcal{X})$.
- if $(a,x),(b,y) \in \mathcal{X}$, where x,y are pointer names then $a \neq b$ implies $x \neq y$.

The complement of $(\alpha,\mathcal{X})$ is $(\overline{\alpha},\mathcal{X})$. For output actions $(\overline{\alpha},\mathcal{X})$ from a program configuration with store $\mathcal{S}$, the store component $\mathcal{X}$ is generated by:

- updating the set of shared location names (to include the primary content of the action if it is a location name, and any names now accessible through the store),
- for every shared location l containing a non-functional value, setting $\mathcal{X}(l) = \mathcal{S}(m)$.
- for every shared location m containing a functional value, generating a fresh output pointer y to $\mathcal{S}(m)$, and setting $\mathcal{X}(m) = y$.

The store component of an input action is used to update the store and set of shared names.

Definition 4. *Given a (finite) set of location names $\mathcal{N}$, an action α and a store $\mathcal{S}$, we define the set $\mathsf{acc}(\mathcal{N},\alpha,\mathcal{S})$ of names in $\mathcal{S}$ accessible from $(\mathcal{N},\alpha)$ to be the least set containing $\mathcal{N}$ and the content of α (if it is a location name), and such that for any location names $(l,m) \in \mathcal{S}$, $l \in \mathsf{acc}(\mathcal{N},\mathcal{S})$ implies $m \in \mathsf{acc}(\mathcal{N},\mathcal{S})$.*
Given a set S of location names, we write S_{fn} for the subset of S consisting of names of locations storing values of functional type and S_{nf} for the subset of names of non-functional type, $\mathcal{S}_{\mathsf{fn}}$ for $\mathcal{S}{\restriction}\mathsf{dom}(\mathcal{S})_{\mathsf{fn}}$ and $\mathcal{S}_{\mathsf{nf}}$ for $\mathcal{S}{\restriction}\mathsf{dom}(\mathcal{S})_{\mathsf{nf}}$.

We may now define the actions of the extended LTS. Unlabelled actions are still the reductions of the operational semantics. For each basic output transition $M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E} \xrightarrow{\overline{\alpha}} \mathcal{B};\mathcal{S};\mathcal{O}';\mathcal{I};\mathcal{E}'$ we define an action:

$$M;\mathcal{B};\mathcal{S};\mathcal{O};\mathcal{I};\mathcal{E};\mathcal{N} \xrightarrow{(\overline{\alpha},\mathcal{X})} \mathcal{B};\mathcal{S};\mathcal{O}',\{(\mathcal{X}(a),\mathcal{S}(a)) \mid a \in \mathcal{N}'_{\mathsf{fn}}\};\mathcal{I};\mathcal{E}';\mathcal{N}'$$

[2] Note that this condition will not always be fulfilled, e.g. if a configuration shares a name for an unassigned location. But it must hold in any configuration reachable from a closed term.

where $\mathcal{N}' = \mathsf{acc}(\mathcal{N}, \overline{\alpha}, \mathcal{S})$ and $\mathcal{X}$ is any atom-valued store such that $\mathsf{dom}(\mathcal{S}') = \mathsf{dom}(\mathcal{S})$, $\mathcal{X}_{\mathsf{nf}} = \mathcal{S}_{\mathsf{nf}}$, and $\mathsf{ran}(\mathcal{X}_{\mathsf{fn}}) \cap (\mathsf{dom}(\mathcal{B}) \cup \mathsf{dom}(\mathcal{O}) \cup \mathcal{I}) = \varnothing$.
For each basic input transition $\mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E} \xrightarrow{\alpha} M; \mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}'; \mathcal{E}'$ we define an action:

$$\mathcal{B}; \mathcal{S}; \mathcal{O}; \mathcal{I}; \mathcal{E}; \mathcal{N} \xrightarrow{(\alpha, \mathcal{X})} M; \mathcal{B}; \mathcal{S}[\mathcal{X}]; \mathcal{O}; \mathcal{I}', \mathsf{ran}(\mathcal{X}_{\mathsf{fn}}); \mathcal{E}'; \mathsf{dom}(\mathcal{X})$$

where $\mathcal{X}$ is any closed atom-valued store such that: $\mathcal{N} \subseteq \mathsf{dom}(\mathcal{X})$, if α contains a location l then $l \in \mathsf{dom}(\mathcal{X})$, and $\mathsf{ran}(\mathcal{X}_{\mathsf{fn}}) \cap (\mathsf{dom}(\mathcal{B}) \cup \mathsf{dom}(\mathcal{O}) \cup \mathcal{I}') = \varnothing$.

As in the previous section, the LTS restricts to configuration types, with the same notion of complete trace.

5 Full Abstraction

We may now establish full abstraction by extending the proofs of soundness and completeness for the functional fragment. We modify the notion of compatibility of configurations $P = (M; \mathcal{B}; \mathcal{S}; \mathcal{S}; \mathcal{I}; \mathcal{E}; \mathcal{N})$ and $E = (\mathcal{B}'; \mathcal{S}'; \mathcal{O}'; \mathcal{I}'; \mathcal{E}; \mathcal{N}')$ (Def. 1) by replacing the requirement (2) that $\mathcal{S}$ and $\mathcal{S}'$ should be disjoint with $\mathsf{dom}(\mathcal{S}) \cap \mathsf{dom}(\mathcal{S}') = \mathcal{N}' = \mathcal{N}$. We redefine the merge operation: $P \curlywedge E = (\mathcal{E} \parallel\!\mid \mathcal{E}')[M]; \mathcal{B}, \mathcal{B}', \mathcal{O}, \mathcal{O}'; \mathcal{S}'[\mathcal{S}]$.

To adapt the proof of Proposition 2, we once again modify the operational semantics so that it is equivalent to the original, but more closely reflects interaction in the new LTS. Specifically, we replace the rules for evaluating function application with the following reductions which rebind *all* locations in the store to fresh pointers: — i.e. we replace each rule of the form $E[F\,V]; \mathcal{B}; \mathcal{S} \longrightarrow M'; \mathcal{B}'; \mathcal{S}$, where F is a pointer name or λ-abstraction, with:

$$E[F\,V]; \mathcal{B}; \mathcal{S} \longrightarrow M'; \mathcal{B}', \{(\mathcal{S}'(a), \mathcal{S}(a)) \mid a \in \mathsf{dom}(\mathcal{S})_{\mathsf{fn}}\}; \mathcal{S}'$$

where $\mathcal{S}'$ is any atom-valued store with $\mathsf{dom}(\mathcal{S}') = \mathsf{dom}(\mathcal{S})$, $\mathcal{S}'_{\mathsf{nf}} = \mathcal{S}_{\mathsf{nf}}$ and $\mathsf{ran}(\mathcal{S}_{\mathsf{fn}}) \cap \mathsf{dom}(\mathcal{B}') = \varnothing$. Writing $M; \mathcal{B}; \mathcal{S} \Downarrow''$ if $M; \mathcal{B}; \mathcal{S}$ reduces to $(); \mathcal{B}'; \mathcal{S}'$ under the modified rules, we may prove:

Proposition 6. *For any program-in-environment $M; \mathcal{B}; \mathcal{S} \Downarrow$ if and only if $M; \mathcal{B}; \mathcal{S} \Downarrow'$.*

Proposition 7. *If P, E are compatible configurations then $P|E \downarrow$ if and only if $P \curlywedge E \Downarrow$.*

Proof. We prove that reduction of $P \curlywedge E$ with respect to the modified semantics tracks the interaction of $[\![P]\!]$ and $[\![E]\!]$ as in the proof of Proposition 2.

To prove completeness, we extend the proofs of definability for bounded strategies (Proposition 4) to the extended LTS. Again, the proofs are based on those for the functional fragment.

Proposition 8. *For any configuration type θ, every bounded strategy σ on θ is the trace of a configuration C_σ of type θ.*

Proof. (sketch) Following the proof of Proposition 4, p-strategies are defined as programs, and e-strategies as values in the store, whilst output pointers and evaluation contexts on the stack just dereference the associated private locations.

If $\sigma = \{(\overline{\alpha}, \mathcal{X}) \cdot \tau$ is a p-strategy, where τ is a (definable) e-strategy then the term M_σ is the assignment of $\mathcal{X}(a)$ to each location $a \in \mathsf{dom}(\mathcal{X})_{\mathsf{nf}}$, followed by the assignment of

$\lambda z.(!q\,a)$ (where q is a distinct location q) to each location $a \in \mathsf{dom}(\mathcal{X})_{\mathsf{fn}}$, followed by the appropriate term producing the action α.

If σ is an e-strategy, then for each pointer location q associated with a given action α, $\mathcal{S}_\sigma(q)$ is a function which dereferences every accessible shared location, and so determines the shared store $\mathcal{X}$[3], and becomes $M_{\widehat{\sigma}(\alpha,\mathcal{X})}$.

Soundness follows from Proposition 7 as in the proof of Proposition 3. Completeness follows from Proposition 8 as in the proof of Proposition 5. Thus we have proved:

Theorem 1 (Full Abstraction). *$M \lesssim N$ if and only if $[\![M]\!] \subseteq [\![N]\!]$.*

6 Conclusions and Further Directions

We have sketched a correspondence between our trace semantics, and Hyland-Ong style game semantics of functional programming languages. This may be formalized, to give a bijective, meaning-preserving correspondence between strategies, which suggests a series of problems for further investigation. In one direction, we may obtain games models from labelled transition systems. For example, our extension of the basic LTS to reference types by equipping actions with a "store component" suggests a possible form for a games model of general references without bad variables, similar to that already given for ground references and addresses in [9]. Another possibility would be to develop games models of objects based on the LTS described in [7].

In the other direction we may develop LTSs corresponding to existing games models for functional languages with a variety of state and control effects. For example, our LTS may be restricted to less expressive languages such as Idealized Algol, where full abstraction fails, since there are traces which do not correspond to the behaviour of any observing context (they do not satisfy the *visibility* condition). Is there a natural way of structuring the set of output pointers so that only the traces satisfying this condition may arise?

Acknowledgements

Thanks to Julian Rathke and Guy McCusker for discussions and suggestions.

References

1. S. Abramsky, K. Honda, G. McCusker. A fully abstract games semantics for general references. In *Proceedings of the 13th Annual Symposium on Logic In Computer Science, LICS '98*, 1998.
2. M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-calculus. In *Proceedings of TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
3. M. Berger, K. Honda, and N. Yoshida. An observationally complete program logic for imperative higher-order functions. In *Proceedings of LICS '05*. IEEE press, 2005.
4. V. Danos, H. Herbelin and L. Regnier. Games semantics and abstract machines. In *Proceedings of the eleventh International Symposium on Logic In Computer Science, LICS '96*, 1996.

[3] Note that because a name may only be (indirectly) shared through location of more complex type, there is still a bound on the number of distinct shared stores which are enabled.

5. J. M. E. Hyland and C.-H. L. Ong. Pi-calculus, dialogue games and PCF. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 96–107. ACM Press, 1995.
6. Alan Jeffrey and Julian Rathke. Towards a theory of bisimulation for local names. In *Proceedings of LICS '99*. IEEE Press, 1999.
7. Alan Jeffrey and Julian Rathke. Java jr. : Fully abstract trace semantics for a core java language. volume 3444 of *LNCS*, pages 423–438. Springer-Verlag, 2005.
8. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of POPL '06*, pages 141–152, 2006.
9. J. Laird. A game semantics of names and pointers. To appear in Annals of Pure and Applied Logic, 2006.
10. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis, University of Edinburgh, 1993.
11. S. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. To appear in proc. AOSD '07, 2007.