A Fully Abstract May Testing Semantics for Concurrent Objects

Alan Jeffrey*
School of CTI
DePaul University
Chicago, IL, USA
ajeffrey@cs.depaul.edu

Bell Labs
Lucent Technologies
Lisle, IL, USA
ajeffrey@bell-labs.com

Julian Rathke†
School of Informatics
University of Sussex
Brighton, UK
julian.rathke@sussex.ac.uk

October 2004

Abstract

This paper provides a fully abstract semantics for a variant of the concurrent object calculus. We define may testing for concurrent object components and then characterise it using a trace semantics inspired by UML interaction diagrams. The main result of this paper is to show that the trace semantics is fully abstract for may testing. This is the first such result for a concurrent object language.

1 Introduction

Abadi and Cardelli's [6] object calculus is a minimal language for investigating features of object languages such as encapsulated state, subtyping, and self variables. Gordon and Hankin [10] added concurrent features to the object calculus, to produce the concurrent object calculus.

Prior work on the object calculus has concentrated on the operational behaviour of object systems, and type systems which provide type safety guarantees. The closest paper to ours is Gordon and Rees's [11] fully abstract semantics for the immutable single-threaded object calculus. There has been no work on providing fully abstract semantics for concurrent mutable objects.

In this paper, we present the first fully abstract testing semantics for a variant of Gordon and Hankin's concurrent object calculus without subtyping. The lack of subtyping here affords a simpler presentation of the labelled transitions and traces but we anticipate that the proof techniques used here are robust enough to cater for subtyping also. This semantics was inspired by UML interaction diagrams [27], which are a common tool for visualising interactions with object systems.

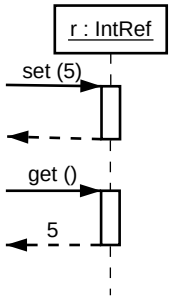
1.1 Interaction diagrams

Interaction diagrams (in particular sequence diagrams) were developed by Jacobson, and are now part of the Unified Modeling Language standard [27]. Interaction diagrams record the messages

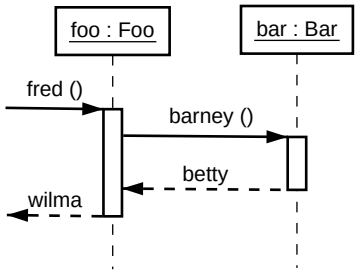
*This material is based upon work supported by the National Science Foundation under Grant No. 0430175
†Support provided by Nuffield Foundation Grant NAL/00226/G

sent between objects of a component in an object system. These messages include method calls and returns (interaction diagrams include other forms of message, but we will not use these in this paper).

A simple interaction with an integer reference object `r` of type `IntRef` has it receive two incoming method calls `set(5)` and `get()`, for which it produces appropriate return values:



A more complex interaction allows a method call on one object to call methods on other objects:



Here, the object `foo` has one incoming call to `fred()`, makes one outgoing call to `barney()`, receives the result `betty` back, then returns `wilma` itself. This illustrates the four messages which may be sent during an interaction: incoming and outgoing method calls, and matching outgoing and incoming returns.

In this paper, we use a textual representation of an interaction, as a trace, which is just a sequence of messages. In the above example, `foo` has the trace:

$$\langle \mathsf{call}\,\mathsf{foo.fred}() \rangle ?$$
$$\langle \mathsf{call}\,\mathsf{bar.barney}() \rangle !$$
$$\langle \mathsf{return}\,\mathsf{betty} \rangle ?$$
$$\langle \mathsf{return}\,\mathsf{wilma} \rangle !$$

where we mark incoming messages with ? and outgoing messages with !. The object `bar` has the matching trace:

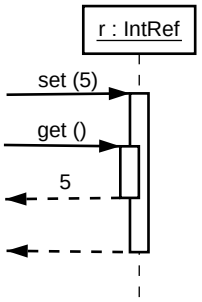
$$\langle \mathsf{call}\,\mathsf{bar.barney}() \rangle ?$$
$$\langle \mathsf{return}\,\mathsf{betty} \rangle !$$

and so composing these two traces together, we get that the whole system has the trace:

$$\langle \mathsf{call}\,\mathsf{foo.fred}() \rangle ?$$
$$\langle \mathsf{return}\,\mathsf{wilma} \rangle !$$

There are two additions we will make to the UML message notation: adding thread identifiers, and making name scope more explicit.

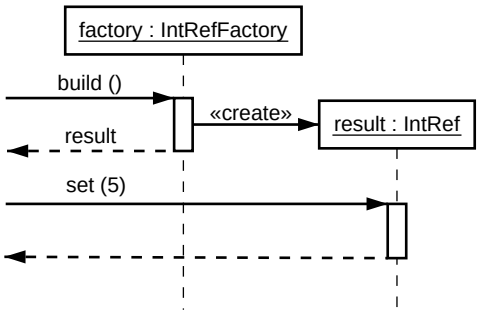
Sequence diagrams can be used for multi-threaded applications, for example:



Here, two threads independently call methods of the object `r`, creating a race condition. In our textual representation, we give the threads names, and we decorate each message with the thread responsible for the message:

$$\begin{aligned}&\mathsf{thread1}\langle \mathsf{call}\, \mathsf{r.set}(5)\rangle ?\\ &\mathsf{thread2}\langle \mathsf{call}\, \mathsf{r.get}()\rangle ?\\ &\mathsf{thread2}\langle \mathsf{return}\, 5\rangle !\\ &\mathsf{thread1}\langle \mathsf{return}\rangle !\end{aligned}$$

The other addition we make to the notation is to make the scope of names more explicit. For example, consider the following interaction with a factory object, which builds new integer reference objects:



In the textual representation of this trace, we need to make clear that the `result` object has not been seen before by the environment (it is a genuinely new object, not a recycled object). We do this by decorating the label with ν to indicate that the `result` object is new:

$$\begin{aligned}&\mathsf{thread1}\langle \mathsf{call}\, \mathsf{factory.build}()\rangle ?\\ &\nu(\mathsf{result} : \mathsf{IntRef})\,.\,\mathsf{thread1}\langle \mathsf{return}\, \mathsf{result}\rangle !\\ &\mathsf{thread1}\langle \mathsf{call}\, \mathsf{result.set}(5)\rangle ?\\ &\mathsf{thread1}\langle \mathsf{return}\rangle !\end{aligned}$$

As well as allowing the system to generate new names on outgoing messages, we allow the environment to generate new names on incoming messages. This style of dealing with fresh names comes originally from the π-calculus [22, 21], and has since been used in other languages, notably the ν-calculus [25].

We have now presented informally all of the machinery required by our semantics for objects:

- The semantics of a system is given by a set of traces, where a trace is a sequence of messages corresponding to one interaction.

- Messages are incoming or outgoing message calls, or matching outgoing or incoming returns.
- Messages are decorated with thread identifiers.
- Messages may include fresh names.

We have only used a very small subset of sequence diagrams, which in turn is a very small subset of UML, but in this paper we will show that this small subset is very expressive, and in particular provides a fully abstract semantics.

1.2 The object calculus

The object calculus is a minimal language for modelling object-based programming. Abadi and Cardelli [6] provided a type system and operational semantics for a variety of object calculi, and proved type safety for them. Gordon and Hankin [10] have since extended this language to include concurrent features.

In this paper, we shall investigate a variant of Gordon and Hankin's concurrent object calculus, which includes:

- A heap of named objects and threads.
- Threads can call or update object methods, can compare object or thread names for equality, can create new objects and threads and can discover their own thread name.
- An operational semantics based on the π-calculus [22, 21], and a simple type system.
- A trace semantics as discussed in Section 1.1.

We are not considering many of the more advanced features of the object calculus or the concurrent object calculus, such as recursive types, object cloning and object locking. This is just for simplicity, and we do not see any technical problems with incorporating these features into our language.

In another strand of research Di Blasio and Fisher [7] also designed a calculus for modelling imperative, concurrent object-based systems. As with Abadi and Cardelli's object calculus and its various extensions, the emphasis in Di Blasio and Fisher's work is again on type systems and safety properties for them.

1.3 Full abstraction

The problem of full abstraction was first introduced by Milner [20], and investigated in depth by Plotkin [26]. Full abstraction was first proposed for variants of the λ-calculus, but has since been investigated for process algebras [12], the π-calculus [9, 13], the ν-calculus [25, 17], Concurrent ML [8, 18], and the immutable object calculus [11].

One way to define a semantics for a programming language is to define:

- A language of *typed components* C which can be *composed* $C_1 \parallel C_2$. (In this paper, components are programs in the concurrent object calculus.)
- A notion of when a component *is successful*. (In this paper, we use a special `succ` method call to indicate a successful component although the theory is robust enough that any other suitable observable would suffice).

We can then define the *may testing preorder* [24, 12] as $C_1 \sqsubseteq_{\text{may}} C_2$ whenever:

> for any appropriately typed C
> if $C_1 \parallel C$ is successful then $C_2 \parallel C$ is successful

Unfortunately, although it is very simple to define, and is quite intuitive, may testing is often very difficult to reason about directly, because of the quantification over 'any appropriately typed C'. In practice, we require a proof technique which we can use to show results about may testing.

One approach is to use a *trace* semantics, given by defining possible executions of components $C \stackrel{s}{\Longrightarrow} C'$ where s is a sequence of messages. We then write $\mathsf{Traces}(C)$ for the set of all traces of C. We say that:

- Traces are *sound* for may testing when $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ implies $C_1 \sqsubseteq_{\text{may}} C_2$.
- Traces are *complete* for may testing when $C_1 \sqsubseteq_{\text{may}} C_2$ implies $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$.
- Traces are *fully abstract* for may testing when they are both sound and complete.

A fully abstract trace model can be a useful tool in understanding a behavioural equivalence in the sense that, in order to be sound, the traces used to build the model must, at minimum, account for all of the possible interactions a system of objects may have with its environment and, in order to be complete, the interactions described by the traces must be genuine. This is taken to mean that for each interaction described by a trace there is an actual system of objects which can play the role of the environment in that interaction. Therefore, to obtain a fully abstract trace model it is necessary to describe all possible interactions accurately.

Establishing full abstraction for a language which includes features such as higher-order programming, new name generation, and heap-based objects is often non-trivial. For example, Pitts and Stark introduced the ν-calculus [25], as a minimal higher-order language with name generation, by extending the simply typed λ-calculus with an abstract type of names, together with a name generator and an equality test. Even this minimal language is remarkably difficult to reason about, and there is no known fully abstract semantics for it [18].

1.4 Contribution of this paper

In this paper, we present a variant of Gordon and Hankin's concurrent object calculus, which is in turn an extension of Abadi and Cardelli's object calculus. The only significant departures from Gordon and Hankin's concurrent object calculus is that we use named threads, where they use anonymous threads and we restrict the calculus to disallow subtyping and recursive types. Whilst this latter restriction does move us away from the essence of object-oriented programming it is imposed so as to keep the technical presentation as simple as possible at this stage. The re-introduction of these features into the type system would affect the behavioural theory in what we expect to be a predictable way and anticipate that techniques employed in [14] and those presented here can be combined to give a similar treatment for a concurrent object language with subtyping.

$$
\begin{array}{lrcl}
\text{Components:} & C & ::= & \mathbf{0} \mid C \parallel C \mid \nu(n:T).C \mid n[O] \mid n\langle t\rangle \\
\text{Objects:} & O & ::= & l = M, \ldots, l = M \\
\text{Methods:} & M & ::= & \varsigma(n:T).\lambda(x:T,\ldots,x:T).\langle t\rangle \\
\text{Threads:} & t & ::= & v \mid \mathsf{stop} \mid \mathsf{let}\ x:T = e\ \mathsf{in}\ t \\
\text{Expressions:} & e & ::= & t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid v.l(v,\ldots,v) \mid n.l \Leftarrow M \mid \\
 & & & \quad \mathsf{new}[O] \mid \mathsf{new}\langle t\rangle \mid \mathsf{currentthread} \\
\text{Values:} & v & ::= & x \mid n \\
\text{Types:} & T & ::= & \mathsf{thread} \mid \mathsf{none} \mid [l:L,\ldots,l:L] \\
\text{Method types:} & L & ::= & (T,\ldots,T) \to T
\end{array}
$$

We assume grammars for variables x, y, names n, p and method identifiers l.
In objects and object types, we require method identifiers l to be unique, and viewed up to reordering.

Figure 1: Syntax of the concurrent object calculus

We provide the calculus with an operational semantics, and a trace semantics, and then show that the trace semantics is fully abstract for may testing. This is the first full abstraction result for a concurrent object-based language.

2 Concurrent objects

In this section, we will present the syntax, static semantics and dynamic semantics of our concurrent object calculus. This is a variant of Gordon and Hankin's concurrent object calculus with named rather than anonymous threads.

2.1 Syntax

The syntax for the concurrent object calculus we will use in this paper is given in Figure 1. We make use of a number of distinct syntactic categories of identifiers, namely, object and thread *Names*, ranged over by n and p (the latter is typically used to indicate an object), *Variables*, ranged over by x, y, z, and *Method Identifiers*, ranged over by l. The operators let and λ act as binders for *Variables* and ς and ν act as binders for *Names*. *Method Identifiers* can not be bound. Note that, at the level of components, there is no facility for binding variables. We will work with terms up to α-conversion of both *Names* and *Variables* in the conventional way. We also make use of capture-free substitution of values for variables or names for names, again defined in the conventional way, and written $t[v/x]$ or $t[p/n]$ as appropriate.

In examples, we will often make use of base types such as integers and booleans: these are not part of our formal system, but will make examples easier to present. They could be comfortably included in the language without changing the theory significantly. We will also make use of some syntax sugar:

We will elide types from variable and name binders, where they can be reconstructed. We write $e;t$ as syntax sugar for $\mathsf{let}\ x = e\ \mathsf{in}\ t$ when x is a fresh variable. We use Abadi and Cardelli's

definition of fields f as zero-argument methods:

- A field declaration $f = v$ in an object is syntax sugar for a method declaration $f = \varsigma(n : T) . \lambda() . \langle v \rangle$.
- A field type $f : T$ in an object type is syntax sugar for a method type $f : () \to T$.
- A field access expression $v.f$ is syntax sugar for a method call $v.f()$.
- A field update expression $n.f := v$ is syntax sugar for a method update $n.f \Leftarrow (\varsigma(p : T) . \lambda() . \langle v \rangle)$.

In addition, we have restricted many subexpressions of an expression to be values rather than full expressions, for example in a method call $v.l(\vec{v})$ we require the object and the arguments to be values rather than expressions $e.l(\vec{e})$. This makes the operational semantics much easier to define, and does not restrict the expressivity of the language, for example we can define $(e.l(\vec{e})) \equiv (\mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ \vec{x} = \vec{e}\ \mathsf{in}\ x.l(\vec{x}))$. Similarly, the distinction between threads and expressions makes the operational semantics much simpler, but we can treat any expression as a thread by η-converting it: $\langle e \rangle \equiv \langle \mathsf{let}\ x = e\ \mathsf{in}\ x \rangle$.

For the remainder of this section, we will provide an informal description of the syntax:

A component C is a collection of named objects $n[O]$ and threads $n\langle t \rangle$. For example, one possible component consisting of an integer reference p and a thread n which increments the reference is:

$$\begin{array}{l} p[\mathsf{contents} = 5] \parallel \\ n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{contents} := x + 1 \rangle \end{array}$$

We also use the ν-notation of the π-calculus [21] to indicate which names are private, and not known to the outside world. By default, names are public, and have to be marked by ν in order to be considered private. For example, n is private, and p is public in:

$$\begin{array}{l} \nu(n : \mathsf{thread}) . (\\ \quad p[\mathsf{contents} = 5] \parallel \\ \quad n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{contents} := x + 1 \rangle \\ ) \end{array}$$

An object $[O]$ consists of a set of named methods, for example an integer reference with set and get methods might be written:

$$\begin{array}{l} [\\ \quad \mathsf{contents} = 5, \\ \quad \mathsf{set} = \varsigma(\mathsf{this} : \mathsf{IntRef}) . \lambda(x : \mathsf{Int}) . \langle \mathsf{this.contents} := x; x \rangle, \\ \quad \mathsf{get} = \varsigma(\mathsf{this} : \mathsf{IntRef}) . \lambda() . \langle \mathsf{this.contents} \rangle \\ ] \end{array}$$

Each method M consists of a self name as well as a list of parameters and a body. For example, the set method above has self name $(\mathsf{this} : \mathsf{IntRef})$, parameters $(x : \mathsf{Int})$, and body $(\mathsf{this.contents} := x)$. Readers familiar with Abadi and Cardelli's work will note that we are taking parameterized methods as primitive, rather than defining them as syntax sugar. This is necessary for our semantics, which is based on method calls with arguments and return values.

A thread $\langle t \rangle$ consists of a stack of let-expressions, terminated either by a return value:

$$\langle \mathsf{let}\ x_1 : T_1 = e_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n : T_n = e_n\ \mathsf{in}\ v \rangle$$

or by a deadlocked stop thread:

$$\langle \mathsf{let}\ x_1 : T_1 = e_1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n : T_n = e_n\ \mathsf{in}\ \mathsf{stop} \rangle$$

Each expression is either itself a thread, or:

- an if expression if $v_1 = v_2$ then e_1 else e_2,
- a method call $v.l(\vec{v})$,
- a method update $n.l \Leftarrow M$, on a named object
- a new object $\mathsf{new}[O]$,
- a new thread $\mathsf{new}\langle t \rangle$, or
- the current thread name currentthread.

Each value is simply a name or a variable and we defer the discussion of types until Section 2.2.

2.2 Static semantics

The static semantics for our concurrent object calculus is given in Figures 2–6. Most of the rules are straightforward adaptations of those given by Abadi and Cardelli [6]. The main judgement is $\Delta \vdash C : \Theta$ which is read as 'the component C uses names Δ and defi nes names Θ'. For example, if we defi ne $C_1(v)$, C_2 and IntRef as:

$$\begin{array}{l}
C_1(v) \equiv p[\\
\quad \mathsf{contents} = v, \\
\quad \mathsf{set} = \varsigma(\mathsf{this} : \mathsf{IntRef})\,.\,\lambda(x : \mathsf{Int})\,.\,\langle \mathsf{this.contents} := x; x \rangle, \\
\quad \mathsf{get} = \varsigma(\mathsf{this} : \mathsf{IntRef})\,.\,\lambda()\,.\,\langle \mathsf{this.contents} \rangle \\
] \\
C_2 \equiv n\langle \\
\quad \mathsf{let}\ x = p.\mathsf{get}()\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop} \\
\rangle \\
\mathsf{IntRef} \equiv [\\
\quad \mathsf{contents} : \mathsf{Int}, \mathsf{set} : (\mathsf{Int}) \to \mathsf{Int}, \mathsf{get} : () \to \mathsf{Int} \\
]
\end{array}$$

then we can deduce (if $v : \mathsf{Int}$):

$$\begin{array}{rcl}
n : \mathsf{thread} & \vdash & C_1(v) : (p : \mathsf{IntRef}) \\
p : \mathsf{IntRef} & \vdash & C_2 : (n : \mathsf{thread}) \\
& \vdash & (C_1(v) \parallel C_2) : (p : \mathsf{IntRef}, n : \mathsf{thread}) \\
& \vdash & \nu(n : \mathsf{thread})\,.\,(C_1(v) \parallel C_2) : (p : \mathsf{IntRef})
\end{array}$$

We will now introduce an important requirement of our components, that they be *write closed*:

$$\frac{}{\Delta \vdash \mathbf{0} : ()} \qquad \frac{;\Delta, n : T \vdash [O] : T}{\Delta \vdash n[O] : (n : T)} \qquad \frac{;\Delta, n : \mathsf{thread} \vdash t : \mathsf{none}}{\Delta \vdash n\langle t \rangle : (n : \mathsf{thread})}$$

$$\frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash (C_1 \parallel C_2) : (\Theta_1, \Theta_2)} \qquad \frac{\Delta \vdash C : \Theta, n : T}{\Delta \vdash \nu(n : T) . C : \Theta}$$

Figure 2: Rules for judgement $\Delta \vdash C : \Theta$

$$\frac{\Gamma; \Delta \vdash M_1 : T.l_1 \quad \cdots \quad \Gamma; \Delta \vdash M_k : T.l_k}{\Gamma; \Delta \vdash [l_1 = M_1, \ldots, l_k = M_k] : T}$$

Figure 3: Rule for judgement $\Gamma; \Delta \vdash [O] : T$ (when $T = [l_1 : L_1, \ldots, l_k : L_k]$)

$$\frac{\Gamma, x_1 : T_1, \ldots, x_k : T_k; \Delta, n : T \vdash t : U}{\Gamma; \Delta \vdash \varsigma(n : T) . \lambda(x_1 : T_1, \ldots, x_k : T_k) . \langle t \rangle : T.l}$$

Figure 4: Rule for judgement $\Gamma; \Delta \vdash M : T.l$ (when $T = [\ldots, l : (T_1, \ldots, T_k) \to U, \ldots]$ and $T.l$ is the record l selected from T)

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash v_1 : T_1 \quad \Gamma; \Delta \vdash v_2 : T_1 \\ \Gamma; \Delta \vdash e_1 : T_2 \quad \Gamma; \Delta \vdash e_2 : T_2\end{array}}{\Gamma; \Delta \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash v : [\ldots, l : (T_1, \ldots, T_k) \to T, \ldots] \\ \Gamma; \Delta \vdash v_1 : T_1 \quad \cdots \quad \Gamma; \Delta \vdash v_k : T_k\end{array}}{\Gamma; \Delta \vdash v.l(v_1, \ldots, v_k) : T} \qquad \frac{\Gamma; \Delta \vdash n : T \quad \Gamma; \Delta \vdash M : T.l}{\Gamma; \Delta \vdash n.l \Leftarrow M : T}$$

$$\frac{\Gamma; \Delta \vdash [O] : T}{\Gamma; \Delta \vdash \mathsf{new}[O] : T} \qquad \frac{\Gamma; \Delta \vdash t : T}{\Gamma; \Delta \vdash \mathsf{new}\langle t \rangle : \mathsf{thread}} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{currentthread} : \mathsf{thread}}$$

$$\frac{\Gamma; \Delta \vdash e : T_1 \quad \Gamma, x : T_1; \Delta \vdash t : T_2}{\Gamma; \Delta \vdash \mathsf{let}\ x : T_1 = e\ \mathsf{in}\ t : T_2} \qquad \frac{}{\Gamma; \Delta \vdash \mathsf{stop} : T} \qquad \frac{}{\Gamma, x : T, \Gamma'; \Delta \vdash x : T} \qquad \frac{}{\Gamma; \Delta, n : T, \Delta' \vdash n : T}$$

Figure 5: Rules for judgement $\Gamma; \Delta \vdash e : T$

Variable contexts: $\Gamma \quad ::= \quad x : T, \ldots, x : T$ Name contexts: $\Delta, \Theta, \Sigma, \Phi \quad ::= \quad n : T, \ldots, n : T$

In variable contexts, variables must be unique, and are viewed up to reordering.
In name contexts, names must be unique, types must not be none, and are viewed up to reordering.

Figure 6: Syntax of name and variable contexts

Whenever $\Delta \vdash C : \Theta$ contains a subexpression of the form $n.l \Leftarrow M$ with n free, then n appears in Θ.

This is intended to capture the common software engineering requirement that components should not export mutable fields, instead they should export suitable get and set methods. For example, the components C_1 and C_2 above are write closed, since the only updates are to this, but the following component which writes directly to $p.\mathsf{contents}$ is not write closed:

$$C_2' \equiv n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{contents} := x+1; \mathsf{stop}\rangle$$

For the remainder of the paper we will require components to be write closed. This makes developing a fully abstract semantics much simpler, since we do not need to model method update directly.

2.3 Dynamic semantics

The dynamic semantics for our concurrent object calculus is given in Figures 7–10.

We define three relations between components:

- $\equiv$, structural congruence, represents the least congruence on components which includes the axioms in Figure 7.
- $C \xrightarrow{\tau} C'$ when C can reduce to C' by the interaction of a thread and an object (either a method call or a method update).
- $C \xrightarrow{\beta} C'$ when C can reduce to C' by a thread acting independently of any other threads or objects.

We write $C \rightarrow C'$ when either $C \xrightarrow{\tau} C'$ or $C \xrightarrow{\beta} C'$; we write $C \Rightarrow C'$ when $C \rightarrow^* C'$.

The important property of β-reductions is that they do not introduce race conditions (and hence nondeterminism), where τ-reductions may introduce race conditions. This is discussed further in Appendix B.1.

For example, recalling the definition of $C_1(v)$ from Section 2.2 we have:

$$\begin{aligned}
&C_1(5) \parallel n\langle \mathsf{let}\ x = p.\mathsf{get}()\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop}\rangle \\
&\quad \xrightarrow{\tau} C_1(5) \parallel n\langle \mathsf{let}\ x = p.\mathsf{contents}\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop}\rangle \\
&\quad \xrightarrow{\tau} C_1(5) \parallel n\langle \mathsf{let}\ x = 5\ \mathsf{in}\ p.\mathsf{set}(x+1); \mathsf{stop}\rangle \\
&\quad \xrightarrow{\beta}{}^* C_1(5) \parallel n\langle p.\mathsf{set}(6); \mathsf{stop}\rangle \\
&\quad \xrightarrow{\tau} C_1(5) \parallel n\langle p.\mathsf{contents} := 6; 6; \mathsf{stop}\rangle \\
&\quad \xrightarrow{\tau} C_1(6) \parallel n\langle p; 6; \mathsf{stop}\rangle \\
&\quad \xrightarrow{\beta}{}^* C_1(6) \parallel n\langle \mathsf{stop}\rangle
\end{aligned}$$

as expected.

Proposition 2.1 (Subject Reduction) *If $\Delta \vdash C : \Theta$ and $C \Rightarrow C'$ then $\Delta \vdash C' : \Theta$*

Proof: Straightforward. □

$$\mathbf{0} \parallel C \equiv C \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3) \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1$$
$$C_1 \parallel \nu(n:T).C_2 \equiv \nu(n:T).(C_1 \parallel C_2) \qquad \nu(n_1:T_1).\nu(n_2:T_2).C \equiv \nu(n_2:T_2).\nu(n_1:T_1).C$$

Figure 7: Axioms for structural congruence (where n is not free in C_1)

$$\begin{array}{rcll}
n\langle \mathsf{let}\ x:T = v\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle t[v/x]\rangle & \\
n\langle \mathsf{let}\ x:T = (\mathsf{let}\ x_1:T_1 = e_1\ \mathsf{in}\ e_2)\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x_1:T_1 = e_1\ \mathsf{in}\ (\mathsf{let}\ x:T = e_2\ \mathsf{in}\ t)\rangle & \\
n\langle \mathsf{let}\ x:T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x:T = e_1\ \mathsf{in}\ t\rangle & \\
n\langle \mathsf{let}\ x:T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x:T = e_2\ \mathsf{in}\ t\rangle & (v_1 \neq v_2) \\
n\langle \mathsf{let}\ x:T = \mathsf{new}[O]\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & \nu(p:T).(p[O] \parallel n\langle \mathsf{let}\ x:T = p\ \mathsf{in}\ t\rangle) & (p \notin O \text{ or } t) \\
n\langle \mathsf{let}\ x:T = \mathsf{new}\langle f\rangle\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & \nu(p:T).(p\langle f\rangle \parallel n\langle \mathsf{let}\ x:T = p\ \mathsf{in}\ t\rangle) & (p \notin t \text{ or } f) \\
n\langle \mathsf{let}\ x:T = \mathsf{currentthread}\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{let}\ x:T = n\ \mathsf{in}\ t\rangle & \\
n\langle \mathsf{let}\ x:T = \mathsf{stop}\ \mathsf{in}\ t\rangle & \xrightarrow{\beta} & n\langle \mathsf{stop}\rangle & \\
p[O] \parallel n\langle \mathsf{let}\ x:T = p.l(\vec{v})\ \mathsf{in}\ t\rangle & \xrightarrow{\tau} & p[O] \parallel n\langle \mathsf{let}\ x:T = O.l(p)(\vec{v})\ \mathsf{in}\ t\rangle & \\
p[O] \parallel n\langle \mathsf{let}\ x:T = p.l \Leftarrow M\ \mathsf{in}\ t\rangle & \xrightarrow{\tau} & p[O.l \Leftarrow M] \parallel n\langle \mathsf{let}\ x:T = p\ \mathsf{in}\ t\rangle &
\end{array}$$

Figure 8: Axioms for reduction precongruence

$$\frac{C \equiv \xrightarrow{\beta} \equiv C'}{C \xrightarrow{\beta} C'} \qquad \frac{C \xrightarrow{\beta} C'}{C \parallel C'' \xrightarrow{\beta} C' \parallel C''} \qquad \frac{C \xrightarrow{\beta} C'}{\nu(n:T).C \xrightarrow{\beta} \nu(n:T).C'}$$
$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n:T).C \xrightarrow{\tau} \nu(n:T).C'}$$

Figure 9: Rules for reduction precongruence

$$(\vec{l} = \vec{M}, l = M).l(p)(\vec{v}) = t[p/n, \vec{v}/\vec{x}] \qquad (\vec{l} = \vec{M}, l = M').l \Leftarrow M = (\vec{l} = \vec{M}, l = M)$$

Figure 10: Definition of $O.l(p)(\vec{v})$ and $O.l \Leftarrow M$ where $M = \varsigma(n:T).\lambda(\vec{x}:\vec{T}).\langle t\rangle$

2.4 Testing preorder

We will now define the testing semantics for our concurrent object calculus. We will do this by defining a notion of *barb* for a component, and let a successful component be one which communicates on that barb. This is similar to the use of barbs in process algebra [23].

Let the type barb be defined:

$$\mathsf{barb} = [\mathsf{succ} : () \to \mathsf{none}]$$

for some fresh method name succ. We say that a component *strongly barbs* on $b : \mathsf{barb}$ written $C{\downarrow_b}$ if and only if:

$$C \equiv \nu(\vec{n} : \vec{T})\,.\,(C' \parallel n\langle \mathsf{let}\ x : \mathsf{none} = b.\mathsf{succ}()\ \mathsf{in}\ t\rangle)$$

for $b \notin \vec{n}$ and *barbs* on $b : \mathsf{barb}$ written $C{\Downarrow_b}$ if and only if:

$$C \Rightarrow C'{\downarrow_b}$$

For components C_1 and C_2 such that $\Delta \vdash C_1 : \Theta$ and $\Delta \vdash C_2 : \Theta$, we define the may testing preorder $\Delta \models C_1 \sqsubseteq_{\mathrm{may}} C_2 : \Theta$ if and only if:

$$\text{for any } \Delta', \Theta, b : \mathsf{barb} \vdash C : \Delta \text{ if } (C_1 \parallel C){\Downarrow_b} \text{ then } (C_2 \parallel C){\Downarrow_b}$$

This is a straightforward adaptation of the standard [12] definition of may testing for concurrent systems.

3 Trace semantics

The trace semantics for the concurrent object calculus is given by a labelled transition system (lts) with judgements:

$$(\Delta \vdash C : \Theta) \xrightarrow{\alpha} (\Delta' \vdash C' : \Theta')$$

The lts is given for components extended by introducing two new expressions:

$$e ::= \cdots \mid \mathsf{block} \mid \mathsf{return}\,(v : T)$$

These new threads are included purely to assist in the description of the lts and are intended to represent a command for a thread to wait for some unknown interaction with the environment and a command for a thread to report a value to the environment and then to go back to a blocked state. There are no reductions associated with these commands and they may be typed as:

$$\frac{}{\Gamma;\Delta \vdash \mathsf{block} : T} \qquad \frac{\Gamma;\Delta \vdash v : U}{\Gamma;\Delta \vdash \mathsf{return}\,(v : U) : T}$$

where T and U are any types. The lts for our concurrent object language are given in Figures 11–14. The form of the actions is discussed in Section 1. The actions are generated using the axioms in Figure 11, as follows:

$$(\Delta, n : \mathsf{thread} \vdash C : \Theta) \xrightarrow{n\langle \mathsf{call}\, p.l(\vec{v})\rangle ?} (\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = p.l(\vec{v})\ \mathsf{in}\ \mathsf{return}\,(x : T)\rangle : (n : \mathsf{thread}, \Theta)) \quad (\text{when } ; \Delta, n : \mathsf{thread}, \Theta \vdash p.l(\vec{v}) : T \text{ and } p \in \Theta)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{call}\, p.l(\vec{v})\rangle ?} (\Delta \vdash C \parallel n\langle \mathsf{let}\, y : U = p.l(\vec{v})\ \mathsf{in}\ \mathsf{let}\, x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle : \Theta) \quad (\text{when } ; \Delta, \Theta \vdash p.l(\vec{v}) : U \text{ and } p \in \Theta)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{return}\, v\rangle ?} (\Delta \vdash C \parallel n\langle t[v/x]\rangle : \Theta) \quad (\text{when } ; \Delta, \Theta \vdash v : T)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = p.l(\vec{v})\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{call}\, p.l(\vec{v})\rangle !} (\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta) \quad (\text{when } p \in \Delta)$$

$$(\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{return}\,(v : U)\ \mathsf{in}\ t\rangle : \Theta) \xrightarrow{n\langle \mathsf{return}\, v\rangle !} (\Delta \vdash C \parallel n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle : \Theta)$$

Figure 11: Axioms for labelled transition system $(\Delta \vdash C : \Theta) \xrightarrow{\alpha} (\Delta' \vdash C' : \Theta')$

$$\frac{C \xrightarrow{\tau} C'}{(\Delta \vdash C : \Theta) \xrightarrow{\tau} (\Delta \vdash C' : \Theta)} \qquad \frac{C \xrightarrow{\beta} C'}{(\Delta \vdash C : \Theta) \xrightarrow{\beta} (\Delta \vdash C' : \Theta)}$$

$$\frac{(\Delta \vdash C : (\Theta, n : T)) \xrightarrow{a} (\Delta' \vdash C' : (\Theta', n : T))}{(\Delta \vdash \nu(n : T).C : \Theta) \xrightarrow{a} (\Delta' \vdash \nu(n : T).C' : \Theta')} (n \text{ is not free in } a)$$

$$\frac{(\Delta \vdash C : (\Theta, n : T)) \xrightarrow{\gamma !} (\Delta' \vdash C' : \Theta')}{(\Delta \vdash \nu(n : T).C : \Theta) \xrightarrow{\nu(n:T).\gamma !} (\Delta' \vdash C' : \Theta')} (n \text{ is free in } \gamma)$$

$$\frac{(\Delta, n : T \vdash C : \Theta) \xrightarrow{\gamma ?} (\Delta' \vdash C' : \Theta')}{(\Delta \vdash C : \Theta) \xrightarrow{\nu(n:T).\gamma ?} (\Delta' \vdash C' : \Theta')} (n \text{ is free in } \gamma, T \text{ is not none})$$

Figure 12: Rules for labelled transition system $(\Delta \vdash C : \Theta) \xrightarrow{\alpha} (\Delta' \vdash C' : \Theta')$

$$\frac{C \Rightarrow C'}{(\Delta \vdash C : \Theta) \stackrel{\varepsilon}{\Longrightarrow} (\Delta \vdash C' : \Theta)} \qquad \frac{(\Delta \vdash C : \Theta) \stackrel{a}{\longrightarrow} (\Delta' \vdash C' : \Theta')}{(\Delta \vdash C : \Theta) \stackrel{a}{\Longrightarrow} (\Delta' \vdash C' : \Theta')}$$

$$\frac{(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C' : \Theta') \stackrel{s'}{\Longrightarrow} (\Delta'' \vdash C'' : \Theta'')}{(\Delta \vdash C : \Theta) \stackrel{ss'}{\Longrightarrow} (\Delta'' \vdash C'' : \Theta'')}$$

Figure 13: Rules for trace semantics $(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C' : \Theta')$

Basic labels:	γ	::=	$n\langle \mathsf{call}\, p.l(\vec{v})\rangle \mid n\langle \mathsf{return}\, v\rangle \mid \nu(n:T)\,.\,\gamma$
Visible labels:	a	::=	$\gamma? \mid \gamma!$
Labels:	α	::=	$a \mid \tau \mid \beta$
Traces:	q,r,s	::=	$a \cdots a$

Figure 14: Syntax of labels and traces

- A call input action, $n\langle \mathsf{call}\, p.l(\vec{v})\rangle?$, represents the environment calling a method on an object defined in the component. It can be generated by a component C in two ways: either the thread n is not defined in C or it is a blocked thread in C. In both cases p is an object defined in C for which the method call $p.l(\vec{v})$ is well-typed. Computation at thread n proceeds by executing this call and returning any result to the environment.

- A return input action, $n\langle \mathsf{return}\, v\rangle?$, represents the environment returning a result to the component. It can be generated by a component C containing a blocked thread n waiting for a result of appropriate type; the thread then unblocks. A thread can only reach such a blocked state by previously having performed a call output action for which this is the corresponding return.

- A call output action, $n\langle \mathsf{call}\, p.l(\vec{v})\rangle!$, represents the component calling a method on an object for which it does not have a definition and so is expecting the environment to provide an appropriate definition. It can be generated by a component C containing a running thread n whose next command is a method call $p.l(\vec{v})$. The calling thread then enters a blocked state waiting for a response from the environment.

- A return output action, $n\langle \mathsf{return}\, v\rangle!$, represents the component returning a result to the environment. It can be generated by a component C containing a running thread n whose next command is a return statement $\mathsf{return}\, v$. A thread can only reach such a return state by previously having received a call input action for which this is the corresponding return.

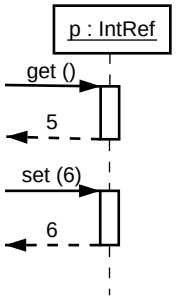
For example if we define:

$$\Theta \equiv (p : \mathsf{IntRef})$$
$$\Theta' \equiv (p : \mathsf{IntRef}, n : \mathsf{thread})$$

then (where $C_1(v)$ is defined in Section 2.2) we have:

$$
\begin{array}{l}
(\vdash C_1(5) : \Theta) \\
\quad \xrightarrow{\nu(n{:}\mathsf{thread}).n\langle \mathsf{call}\, p.\mathsf{get}()\rangle?} \\
(\vdash (C_1(5) \parallel n\langle \mathsf{let}\, x = p.\mathsf{get}()\ \mathsf{in}\ \mathsf{return}\, x\rangle) : \Theta') \\
\quad \Rightarrow \\
(\vdash (C_1(5) \parallel n\langle \mathsf{return}\, 5\rangle) : \Theta') \\
\quad \xrightarrow{n\langle \mathsf{return}\, 5\rangle!} \\
(\vdash (C_1(5) \parallel n\langle \mathsf{block}\rangle) : \Theta') \\
\quad \xrightarrow{n\langle \mathsf{call}\, p.\mathsf{set}(6)\rangle?} \\
(\vdash (C_1(5) \parallel n\langle \mathsf{let}\, x = p.\mathsf{set}(6)\ \mathsf{in}\ \mathsf{return}\, x\rangle) : \Theta') \\
\quad \Rightarrow \\
(\vdash (C_1(6) \parallel n\langle \mathsf{return}\, 6\rangle) : \Theta') \\
\quad \xrightarrow{n\langle \mathsf{return}\, 6\rangle!} \\
(\vdash (C_1(6) \parallel n\langle \mathsf{block}\rangle) : \Theta')
\end{array}
$$

which corresponds to the interaction diagram:



Note that these traces are typed, in order to avoid undesirable traces, which correspond to interaction with an ill-typed environment, such as:

$$
\begin{array}{l}
(\vdash (C_1(5) \parallel n\langle \mathsf{block}\rangle) : \Theta') \\
\quad \xrightarrow{n\langle \mathsf{call}\, p.\mathsf{set}(\text{“}\mathit{hello}\text{”})\rangle?} \\
(\vdash (C_1(5) \parallel n\langle \mathsf{let}\, x = p.\mathsf{set}(\text{“}\mathit{hello}\text{”})\ \mathsf{in}\ \mathsf{return}\, x\rangle) : \Theta') \\
\quad \Rightarrow \\
(\vdash (C_1(\text{“}\mathit{hello}\text{”}) \parallel n\langle \mathsf{return}\, \text{“}\mathit{hello}\text{”}\rangle) : \Theta') \\
\quad \xrightarrow{n\langle \mathsf{return}\, \text{“}\mathit{hello}\text{”}\rangle!} \\
(\vdash (C_1(\text{“}\mathit{hello}\text{”}) \parallel n\langle \mathsf{block}\rangle) : \Theta')
\end{array}
$$

For any component $(\Delta \vdash C : \Theta)$ we define its traces to be:

$$\mathsf{Traces}(\Delta \vdash C : \Theta) = \{s \mid (\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C' : \Theta')\}$$

We will now show that this trace semantics is fully abstract for may testing.

4 Soundness of traces for may testing

Having defined our trace semantics we must demonstrate that it provides a sound characterisation of our notion of equivalence, that is, may testing. Specifically we must show that whenever the traces of a well-typed component are contained in another's then the components must be related in the may testing preorder. We immediately see some difficulty in proving this directly as the traces are defined using terms over an extended syntax whereas testing is defined purely in the base language. However, the extensions made to the syntax represent *interaction points*, between a component and a putative testing component. Therefore, given an actual testing component we may *merge* the original component and the test together at these interaction points, thereby recovering the term in the base language which would have been reached had the component and test actually interacted. This operation of merging is defined below:

4.1 The merge operator

Define the partial *merge* operator $C_1 \between C_2$ on components as the symmetric operator defined up to $\equiv$ where:

$$\begin{array}{rcl} \mathbf{0} \between C & = & C \\ (\nu(p:T).C_1) \between C_2 & = & \nu(p:T).(C_1 \between C_2) \\ (p[O] \parallel C_1) \between C_2 & = & p[O] \parallel (C_1 \between C_2) \\ (p\langle t\rangle \parallel C_1) \between C_2 & = & p\langle t\rangle \parallel (C_1 \between C_2) \\ (n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2) & = & n\langle t_1 \between t_2\rangle \parallel (C_1 \between C_2) \end{array}$$

when $n \notin \mathsf{dom}\,(C_1, C_2)$ and $p \notin \mathsf{fn}\,(C_2)$.

We overload notation and define the partial merge operator $t_1 \between t_2$ on threads as the symmetric operator where:

$$\begin{array}{rcl} (\mathsf{let}\, x:T = \mathsf{block}\,\mathsf{in}\, t) \between \mathsf{stop} & = & \mathsf{stop} \\ (\mathsf{let}\, x:T = \mathsf{block}\,\mathsf{in}\, t_1) \between (\mathsf{let}\, y:U = \mathsf{return}\,(v:T)\,\mathsf{in}\, t_2) & = & (\mathsf{let}\, y:U = \mathsf{block}\,\mathsf{in}\, t_2) \between (t_1[v/x]) \\ (\mathsf{let}\, x:T = \mathsf{block}\,\mathsf{in}\, t_1) \between (\mathsf{let}\, y:U = e\,\mathsf{in}\, t_2) & = & \mathsf{let}\, y:U = e\,\mathsf{in}\, ((\mathsf{let}\, x:T = \mathsf{block}\,\mathsf{in}\, t_1) \between t_2) \end{array}$$

when e is block/return free and $y \notin \mathsf{fv}\,(t_1)$.

Lemma 4.1 *If* $\Delta \vdash (C_1 \parallel C_2) : \Theta$ *then* $(C_1 \between C_2) \equiv (C_1 \parallel C_2)$.

Proof: An induction on the definition of $C_1 \between C_2$. □

Lemma 4.2 *If* $C_1 \between C_2 \equiv C$ *and* $C_1{\downarrow_b}$ *then* $C{\downarrow_b}$.

Proof: An induction on the definition of $C_1 \between C_2$. □

4.2 Trace composition and decomposition

Given a trace s we write $\overline{s}$ for the complementary trace:

$$\overline{\varepsilon} = \varepsilon \qquad \overline{s_1 s_2} = \overline{s_1}\,\overline{s_2} \qquad \overline{\gamma?} = \gamma! \qquad \overline{\gamma!} = \gamma?$$

Proposition 4.3 (Trace composition/decomposition) *For any components* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma)$ *and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma)$ *such that* $C_1 \between C_2 \equiv C$*, we have:*

1. *If* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xrightarrow{s} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$
   and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \xrightarrow{\overline{s}} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$
   then $C \Rightarrow C'$ *where* $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C'$.

2. *If* $C \Rightarrow C'$ *then there exists some trace s such that* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$
   and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \stackrel{\overline{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ *where* $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C'$.

Proof: Given in Appendix A. □

Corollary 4.4 *For any components* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma)$ *and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma)$ *such that* $C_1 \between C_2 \equiv C$ *and* $C \Downarrow_b$ *there exists some trace s such that* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$ *and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \xrightarrow{\overline{s}} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ *where either* $C_1' \downarrow_b$ *or* $C_2' \downarrow_b$.

Proof: We know that $C \Downarrow_b$ which tells us that $C \Rightarrow C''$ for some C'' such that $C'' \downarrow_b$. We use Proposition 4.3 Part 2, to obtain a trace s_1 such that

$$(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xrightarrow{s_1} (\Delta'', \Phi \vdash C_1'' : \Theta'', \Sigma'')$$
$$(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \xrightarrow{\overline{s_1}} (\Theta'', \Phi \vdash C_2'' : \Delta'', \Sigma'')$$

where $\nu(\Delta'', \Theta'', \Sigma'' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1'' \between C_2'') \equiv C''$. Given that $C'' \downarrow_b$ we know that $(C_1'' \between C_2'') \downarrow_b$ also. By the definition of $\between$ we see that one of the following (or their symmetric counterparts) must hold:

- $C_1'' \downarrow_b$ and we are done, or
- $C_1'' \equiv \nu(\Delta_1) \,.\, (n\langle t_1 \rangle \parallel C_1''')$ and $C_2'' \equiv \nu(\Delta_2) \,.\, (n\langle t_2 \rangle \parallel C_2''')$ where $n\langle t_1 \between t_2 \rangle \downarrow_b$. We now proceed by induction on the definition of $t_1 \between t_2$ to show that for all such C_1'' and C_2'', we can find s_2 where:

  $$(\Delta'', \Phi \vdash C_1'' : \Theta'', \Sigma'') \xrightarrow{s_2} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$$
  $$(\Theta'', \Phi \vdash C_2'' : \Delta'', \Sigma'') \xrightarrow{\overline{s_2}} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$$

  and either $C_1' \downarrow_b$ or $C_2' \downarrow_b$. There are two cases (up to symmetry of $\between$):

  - If $t_1 = \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t_1'$ and $t_2 = \mathsf{let}\ y : U = b.\mathsf{succ}()\ \mathsf{in}\ t_2'$ then $C_2'' \downarrow_b$.
  - If $t_1 = \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t_1'$ and $t_2 = \mathsf{let}\ y : U = \mathsf{return}\,(v : T)\ \mathsf{in}\ t_2'$ then we have:

    $$(\Delta'', \Phi \vdash C_1'' : \Theta'', \Sigma'') \xrightarrow{\nu(\Delta_2').n\langle \mathsf{return}\, v \rangle ?} (\Delta'', \Delta_2', \Phi \vdash \nu(\Delta_1) \,.\, (n\langle t_1'[v/x] \rangle \parallel C_1''') : \Theta', \Sigma')$$
    $$(\Theta'', \Phi \vdash C_2'' : \Delta'', \Sigma'') \xrightarrow{\nu(\Delta_2').n\langle \mathsf{return}\, v \rangle !} (\Theta'', \Phi \vdash \nu(\Delta_2'') \,.\, (n\langle \mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t_2' \rangle \parallel C_2''') : \Delta'', \Delta_2', \Sigma'')$$

    where $\Delta_2 = (\Delta_2', \Delta_2'')$ and moreover:

    $$n\langle t_1 \between t_2 \rangle \equiv n\langle (\mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t_2') \between t_1[v/x] \rangle \downarrow_b$$

    so by inductive hypothesis:

    $$(\Delta'', \Phi \vdash C_1'' : \Theta'', \Sigma'') \xrightarrow{\nu(\Delta_2').n\langle \mathsf{return}\, v \rangle ?} \xrightarrow{s_2} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$$
    $$(\Theta'', \Phi \vdash C_2'' : \Delta'', \Sigma'') \xrightarrow{\nu(\Delta_2').n\langle \mathsf{return}\, v \rangle !} \xrightarrow{\overline{s_2}} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$$

    and either $C_1' \downarrow_b$ or $C_2' \downarrow_b$, as required. □

4.3 Proof of soundness

Theorem 4.5 (Soundness of traces for may testing) *If* $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$ *then* $\Delta \models C_1 \sqsubseteq_{may} C_2 : \Theta$

Proof: Suppose that $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$ and that we have $(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta)$ such that $(C_1 \parallel C_0)\Downarrow_b$; we must show that $(C_2 \parallel C_0)\Downarrow_b$ also.

Now, since $(C_1 \parallel C_0)\Downarrow_b$, we can use Corollary 4.4 to get:

$$(\Delta, b : \mathsf{barb} \vdash C_1 : \Theta) \stackrel{s}{\Longrightarrow} (\Delta', b : \mathsf{barb} \vdash C_1' : \Theta', \Sigma')$$
$$(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta) \stackrel{\bar{s}}{\Longrightarrow} (\Theta', b : \mathsf{barb} \vdash C_0' : \Delta', \Sigma')$$

and one of the following cases holds:

- **Case** $(C_1'\downarrow_b)$. Since $C_1'\downarrow_b$ we can find a label $\omega!$ of the form:

  $$\omega! = \nu(\vec{n} : \vec{T})\,.\,n\langle \mathsf{call}\, b.\mathsf{succ}()\rangle!$$

  such that:

  $$(\Delta', b : \mathsf{barb} \vdash C_1' : \Theta', \Sigma') \stackrel{\omega!}{\longrightarrow}$$

  Since $\mathsf{Traces}(\Delta \vdash C_1 : \Theta) \subseteq \mathsf{Traces}(\Delta \vdash C_2 : \Theta)$ we have:

  $$(\Delta, b : \mathsf{barb} \vdash C_2 : \Theta) \stackrel{s}{\Longrightarrow} (\Delta', b : \mathsf{barb} \vdash C_2' : \Theta', \Sigma') \stackrel{\omega!}{\longrightarrow}$$

  and hence $C_2'\downarrow_b$. By Lemma 4.1 we know that $C_2 \parallel C_0 \equiv C_2 \between C_0$ and so by Proposition 4.3 we have: $(C_2 \parallel C_0) \Rightarrow C''$ where:

  $$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta)\,.\,(C_2' \between C_0') \equiv C''$$

  By Lemma 4.2, since $C_2'\downarrow_b$ we have that $C''\downarrow_b$, and so $(C_2 \parallel C_0)\Downarrow_b$ as required.

- **Case** $(C_0'\downarrow_b)$. Similar to the above. □

5 Completeness of traces for may testing

We now turn to the question of whether trace inclusion captures the may testing preorder exactly. We have already shown that trace inclusion implies may testing inclusion, and so we must consider the converse—completeness.

A key step in demonstrating completeness of traces for may testing is to find, for each trace, a component which exhibits that trace; we call this problem *definability*. However, we only actually require definability for traces which originated from well-typed components. To identify these we present a type system for traces $\Delta \vdash s : \mathsf{trace}\ \Theta$ which captures exactly those we require.

Due to asynchrony in the labelled transition system, to demonstrate definability, we found it necessary to define an information order $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$ for typed traces which incorporates prefixing, input-enabling, and commutativity of certain actions.

In the next section we introduce the type system for traces and demonstrate that every trace from a well-typed component is in fact well-typed. In the section which follows this we introduce the information order on traces and prove the properties required of it.

$$\frac{}{\Delta \vdash \varepsilon : \mathsf{trace}\ \Theta}$$

$$\frac{\begin{array}{c} n \text{ is input-enabled in } \Delta \vdash s : \mathsf{trace}\ \Theta \\ \mathsf{dom}\ (\Delta') \subseteq \mathsf{fn}\ (n\langle \mathsf{call}\, p.l(\vec{v})\rangle) \\ ;\Theta, \Theta(s) \vdash p : [\ldots, l : (\vec{T}) \to T, \ldots] \\ ;\Delta, \Theta, \Delta(s), \Theta(s), \Delta' \vdash \vec{v} : \vec{T} \\ ;\Delta, \Theta, \Delta(s), \Theta(s), \Delta' \vdash n : \mathsf{thread} \end{array}}{\Delta \vdash s\nu(\Delta')\,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle? : \mathsf{trace}\ \Theta} \qquad \frac{\begin{array}{c} n \text{ is output-enabled in } \Delta \vdash s : \mathsf{trace}\ \Theta \\ \mathsf{dom}\ (\Theta') \subseteq \mathsf{fn}\ (n\langle \mathsf{call}\, p.l(\vec{v})\rangle) \\ ;\Delta, \Delta(s) \vdash p : [\ldots, l : (\vec{T}) \to T, \ldots] \\ ;\Delta, \Theta, \Delta(s), \Theta(s), \Theta' \vdash \vec{v} : \vec{T} \\ ;\Delta, \Theta, \Delta(s), \Theta(s), \Theta' \vdash n : \mathsf{thread} \end{array}}{\Delta \vdash s\nu(\Theta')\,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle! : \mathsf{trace}\ \Theta}$$

$$\frac{\begin{array}{c} \Delta \vdash s : \mathsf{trace}\ \Theta \\ \mathsf{pop}\, n\,(s) = \nu(\Delta')\,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle? \\ \mathsf{dom}\ (\Theta') \subseteq \mathsf{fn}\ (v) \\ ;\Theta, \Theta(s) \vdash p : [\ldots, l : (\vec{T}) \to T, \ldots] \\ ;\Delta, \Theta, \Delta(s), \Theta(s), \Theta' \vdash v : T \end{array}}{\Delta \vdash s\nu(\Theta')\,.\, n\langle \mathsf{return}\, v\rangle! : \mathsf{trace}\ \Theta} \qquad \frac{\begin{array}{c} \Delta \vdash s : \mathsf{trace}\ \Theta \\ \mathsf{pop}\, n\,(s) = \nu(\Theta')\,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle! \\ \mathsf{dom}\ (\Delta') \subseteq \mathsf{fn}\ (v) \\ ;\Delta, \Delta(s) \vdash p : [\ldots, l : (\vec{T}) \to T, \ldots] \\ ;\Delta, \Theta, \Delta(s), \Theta(s), \Delta' \vdash v : T \end{array}}{\Delta \vdash s\nu(\Delta')\,.\, n\langle \mathsf{return}\, v\rangle? : \mathsf{trace}\ \Theta}$$

Figure 15: Rules for judgement $\Delta \vdash s : \mathsf{trace}\ \Theta$

$$\frac{\Delta \vdash s : \mathsf{trace}\ \Theta \quad \mathsf{pop}\, n\,(s) = \gamma!}{n \text{ is input-enabled in } \Delta \vdash s : \mathsf{trace}\ \Theta} \qquad \frac{\Delta \vdash s : \mathsf{trace}\ \Theta \quad \mathsf{pop}\, n\,(s) = * \quad n \notin \Delta, \Delta(s)}{n \text{ is input-enabled in } \Delta \vdash s : \mathsf{trace}\ \Theta}$$

$$\frac{\Delta \vdash s : \mathsf{trace}\ \Theta \quad \mathsf{pop}\, n\,(s) = \gamma?}{n \text{ is output-enabled in } \Delta \vdash s : \mathsf{trace}\ \Theta} \qquad \frac{\Delta \vdash s : \mathsf{trace}\ \Theta \quad \mathsf{pop}\, n\,(s) = * \quad n \notin \Theta, \Theta(s)}{n \text{ is output-enabled in } \Delta \vdash s : \mathsf{trace}\ \Theta}$$

Figure 16: Rules for judgement n is input/output-enabled in $\Delta \vdash s : \mathsf{trace}\ \Theta$

5.1 Types for traces

In this section we will define a judgement $\Delta \vdash s : \mathsf{trace}\ \Theta$ which describes those traces s which may be exhibited by well-typed components $\Delta \vdash C : \Theta$.

The type rules for traces make use of some auxiliary notions which we define below.

We write $C \equiv C[D]$ to mean

$$C \equiv \nu(\Delta)\,.\,(D \parallel C')$$

for some Δ, C'.

Define the *thread* of an action as:

$$\mathsf{thread}\ (\nu(\Delta)\,.\, n\langle\cdots\rangle?) = \mathsf{thread}\ (\nu(\Theta)\,.\, n\langle\cdots\rangle!) = n$$

Define the *threads* of a trace as:

$$\mathsf{threads}\ (a_1 \cdots a_n) = \{\mathsf{thread}\ (a_1), \ldots, \mathsf{thread}\ (a_n)\}$$

Each thread in a well-typed trace maintains a stack discipline: each return action in the trace pops a corresponding call action that appears earlier in the trace. In the single-threaded case this corresponds to the well-bracketing condition of game semantics [3, 16]. To formalise this we first define what it means for a thread to be *balanced* within a trace: each call action has been completed with a corresponding return action. For a given thread n and trace s, define n is *balanced in* s as:

- If $n \notin \mathsf{threads}\,(s)$ then n is balanced in s.
- If n is balanced in s_1 and s_2 then n is balanced in $s_1 s_2$.
- If n is balanced in s then n is balanced in $\nu(\Delta)\,.\,n\langle \mathsf{call}\, p.l(\vec{n})\rangle?\, s\, \nu(\Theta)\,.\,n\langle \mathsf{return}\, v\rangle!$.
- If n is balanced in s then n is balanced in $\nu(\Theta)\,.\,n\langle \mathsf{call}\, p.l(\vec{n})\rangle!\, s\, \nu(\Delta)\,.\,n\langle \mathsf{return}\, v\rangle?$.

In any well-typed trace ending in a return action we need to find the corresponding call action earlier in the trace, for which we define a *pop* function. To make this function total on well-typed traces we define it to return a dummy value $*$ on balanced traces. Define $\mathsf{pop}\, n\,(s)$ as:

- If n is balanced in s then $\mathsf{pop}\, n\,(s) = *$.
- If n is balanced in s and $a = \nu(\Delta)\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle?$ then $\mathsf{pop}\, n\,(r\, a\, s) = a$.
- If n is balanced in s and $a = \nu(\Theta)\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle!$ then $\mathsf{pop}\, n\,(r\, a\, s) = a$.

A feature of our traces is that they model the creation of fresh object references. The following two definitions allow us to identify those fresh references which were generated by the component ($\Theta(s)$) and those which were generated by the testing environment ($\Delta(s)$). Define $\Delta(s)$ to be the bound input names of s:

$$\begin{array}{rcl} \Delta(\varepsilon) & = & \varepsilon \\ \Delta(\nu(\vec{n} : \vec{T})\,.\,a!\, s) & = & \Delta(s) \\ \Delta(\nu(\vec{n} : \vec{T})\,.\,a?\, s) & = & \vec{n} : \vec{T}, \Delta(s) \end{array}$$

and $\Theta(s)$ to be the bound output names of s:

$$\begin{array}{rcl} \Theta(\varepsilon) & = & \varepsilon \\ \Theta(\nu(\vec{n} : \vec{T})\,.\,a?\, s) & = & \Theta(s) \\ \Theta(\nu(\vec{n} : \vec{T})\,.\,a!\, s) & = & \vec{n} : \vec{T}, \Theta(s) \end{array}$$

The type system for traces is given in Figures 15 and 16. The type rules correspond to the typing side-conditions of the rules in the labelled transition system itself, along with extra conditions to enforce the stack discipline.

It will be useful to prove two technical lemmas before we can prove that Trace Subject Reduction (Proposition 5.3) holds.

Lemma 5.1

1. *If n is balanced in s and:*

$$(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C'[n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle] : \Theta')$$

then $C \equiv C[n\langle \mathsf{let}\, x : T = \mathsf{block}\ \mathsf{in}\ t\rangle]$.

2. *If n is balanced in s and $\vec{e}'$ are block/return-free, and:*

$$(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C'[n\langle \mathsf{let}\ \vec{x}' : \vec{T}' = \vec{e}'\ \mathsf{in}\ \mathsf{let}\ y : U = \mathsf{return}\,(v : T)\ \mathsf{in}\ t\rangle] : \Theta')$$

then $C \equiv C[n\langle \mathsf{let}\ \vec{x} : \vec{T} = \vec{e}\ \mathsf{in}\ \mathsf{let}\ y : U = \mathsf{return}\,(v : T)\ \mathsf{in}\ t\rangle]$ where $\vec{e}$ is block/return free.

Proof: Easy induction on s. □

Lemma 5.2

1. *If C is block/return free and $(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} \xrightarrow{\nu(\Theta').n\langle \mathsf{return}\, v\rangle!}$ then $s = s_1\, \nu(\Delta').n\langle \mathsf{call}\, p.l(\vec{v})\rangle?\, s_2$ where n is balanced in s_2 .*
2. *If C is block/return free and $(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} \xrightarrow{\nu(\Delta').n\langle \mathsf{return}\, v\rangle?}$ then $s = s_1\, \nu(\Theta').n\langle \mathsf{call}\, p.l(\vec{v})\rangle!\, s_2$ where n is balanced in s_2 .*

Proof: We prove these properties simultaneously by an induction on the length of s. We only show the argument for Part 1 as Part 2 can be shown in a similar manner. By analysis of the rules of the lts, we have:

$$(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta'' \vdash C''[n\langle \mathsf{let}\ x : T = \mathsf{return}\,(v : U)\ \mathsf{in}\ t\rangle] : \Theta'') \xrightarrow{\nu(\Theta').n\langle \mathsf{return}\, v\rangle!}$$

Now, partition s into $s_3\ s_2$ picking s_2 to be the longest suffix of s in which n is balanced. We then use Lemma 5.1 to get that:

$$(\Delta \vdash C : \Theta) \stackrel{s_3}{\Longrightarrow} (\Delta'' \vdash C''[n\langle \mathsf{let}\ \vec{x} : \vec{T} = \vec{e}\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(v' : U)\ \mathsf{in}\ t\rangle] : \Theta'') \stackrel{s_2}{\Longrightarrow} \xrightarrow{\nu(\Theta').n\langle \mathsf{return}\, v\rangle!}$$

We now proceed by analysis of s_3:

- s_3 is not of the form ε since C is block/return free.
- s_3 is not of the form $s_1\ a$ with $\mathsf{thread}\,(a) \neq n$, since s_2 is required to be the longest suffix of s in which n is balanced.
- s_3 is not of the form $s_1\ \gamma!$ since $n\langle \mathsf{let}\ \vec{x} : \vec{T} = \vec{e}\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(v' : U)\ \mathsf{in}\ t\rangle$ is not of the form $n\langle \mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t'\rangle$.
- s_3 is not of the form $s_1\, \nu(\Delta'').n\langle \mathsf{return}\, v'\rangle?$ since otherwise, by applying Part 2 of the inductive hypothesis we can partition s_1 into $s_1'\ \nu(\Theta'').n\langle \mathsf{call}\, p'.l'(\vec{v}')\rangle!\ s_2'$ where n is balanced in s_2', hence n is balanced in $\nu(\Theta'').n\langle \mathsf{call}\, p'.l'(\vec{v}')\rangle!\ s_2'\ \nu(\Delta'').n\langle \mathsf{return}\, v'\rangle?\ s_2$, contradicting the requirement that s_2 is the longest such suffix of s.
- So, by a process of elimination, s_3 is of the form $s_1\, \nu(\Delta').n\langle \mathsf{call}\, p.l(\vec{v})\rangle?$ as required. □

Proposition 5.3 (Trace Subject Reduction) *If $\Delta \vdash C : \Theta$ is block/return free and $(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C' : \Theta')$ then $\Delta \vdash s : \mathsf{trace}\ \Theta$ and $\Delta' \vdash C' : \Theta'$.*

Proof: We proceed by induction on the derivation of $(\Delta \vdash C : \Theta) \stackrel{s}{\Longrightarrow} (\Delta' \vdash C' : \Theta')$.

It is relatively easy to check that $\Delta' \vdash C' : \Theta'$ where $\stackrel{s}{\Longrightarrow}$ is given by a single axiom instance. We use the inductive hypothesis and Proposition 2.1 to deal with the more general case. We now show $\Delta \vdash s : \mathsf{trace}\ \Theta$. The base case in which s is empty is trivial. Suppose instead that s is non-empty: we perform a case-analysis on the last action of s.

Case $s = s'\ \nu(\Delta')\,.\,n\langle \mathsf{call}\ p.l(\vec{v})\rangle?$. We know that

$$(\Delta \vdash C : \Theta) \stackrel{s'}{\Longrightarrow} (\Delta, \Delta(s') \vdash C' : \Theta, \Theta(s')) \xrightarrow{\nu(\Delta').n\langle \mathsf{call}\ p.l(\vec{v})\rangle?}$$

so we have that either

$$C' \equiv \nu(\Delta')\,.\,\nu(\Delta'')\,.\,n\langle \mathsf{let}\ x : T = \mathsf{block\ in}\ t\rangle \parallel C''$$

or $n \in \Delta, \Delta(s')$ and n is a fresh thread to s'. We can apply the inductive hypothesis to s' to see that $\Delta \vdash s' : \mathsf{trace}\ \Theta$ and we consider $\mathsf{pop}n\,(s')$: if $n \in \Delta, \Delta(s')$ and n is a fresh thread to s' then $\mathsf{pop}n\,(s')$ is necessarily $*$. Otherwise we know that $C' \equiv \nu(\Delta')\,.\,\nu(\Delta'')\,.\,n\langle \mathsf{let}\ x : T = \mathsf{block\ in}\ t\rangle \parallel C''$ and therefore the last action which could have occurred at n must have been an output, that is, $\mathsf{pop}n\,(s') = \gamma!$. In both cases we see that

$$n \text{ is input enabled in } \Delta \vdash s' : \mathsf{trace}\ \Theta \tag{1}$$

We know that $(\Delta, \Delta(s') \vdash C' : \Theta, \Theta(s')) \xrightarrow{\nu(\Delta').n\langle \mathsf{call}\ p.l(\vec{v})\rangle?}$ and we know that the side-conditions on the transition rule for $\nu(\Delta')\,.\,\gamma?$ actions guarantees that

$$\mathsf{dom}\ (\Delta') \subseteq \mathsf{fn}\ (\vec{v}) \tag{2}$$

We also know that the side-conditions on the rule for call-input actions guarantees that

$$;\Delta, \Delta(s'), \Theta, \Theta(s'), \Delta' \vdash p.l(\vec{v}) : T \text{ and } p \in \Theta, \Theta(s')$$

We use this to see that

$$;\Theta, \Theta(s'), \Delta' \vdash p : [\ldots l : (\vec{T}) \to T] \tag{3}$$

and

$$;\Delta, \Delta(s'), \Theta, \Theta(s'), \Delta' \vdash \vec{v} : \vec{T} \tag{4}$$

Lastly, it is easy to see that

$$;\Delta, \Delta(s'), \Theta, \Theta(s'), \Delta' \vdash n : \mathsf{thread} \tag{5}$$

We collect the statements (1)–(5) together to see that they form the hypotheses of the type rule which allows us to conclude

$$\Delta \vdash s'\ \nu(\Delta')\,.\,n\langle \mathsf{call}\ p.l(\vec{v})\rangle? : \mathsf{trace}\ \Theta$$

as required.

Case $s = s'\ \nu(\Theta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle!$. Similar to previous case.

Case $s = s'\ \nu(\Theta')\,.\,n\langle \mathsf{return}\, v\rangle!$. We know that

$$(\Delta \vdash C : \Theta) \stackrel{s'}{\Longrightarrow} (\Delta, \Delta(s') \vdash C' : \Theta, \Theta(s')) \xrightarrow{\nu(\Theta').n\langle \mathsf{return}\, v\rangle!}$$

so we have that

$$C' \equiv C'[n\langle \mathsf{let}\ x : T = \mathsf{return}\,(v : U)\ \mathsf{in}\ t\rangle]$$

We can apply the inductive hypothesis to obtain

$$\Delta \vdash s' : \mathsf{trace}\,\Theta \tag{1}$$

and we notice that because C is block/return free we can apply Lemma 5.2 to get:

$$s' = s_1\ \nu(\Delta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle?\ s_2$$

where n is balanced in s_2. Given this, we see that

$$\mathsf{pop}\, n\,(s_1\ \nu(\Delta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle?\ s_2) = \nu(\Delta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle?$$

hence

$$\mathsf{pop}\, n\,(s') = \nu(\Delta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle? \tag{2}$$

Again, the side-conditions on the transition rule for $\nu(\Theta')\,.\,\gamma!$ guarantee that

$$\mathsf{dom}\,(\Theta') \subseteq \mathsf{fn}\,(v) \tag{3}$$

We also know, by (1) and the fact that prefi xes of well-typed traces are also well-typed, that

$$\Delta \vdash s_1\ \nu(\Delta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v})\rangle? : \mathsf{trace}\,\Theta$$

and we see that this must have been inferred using a hypothesis

$$;\Theta, \Theta(s_1) \vdash p : [\ldots l : (\vec{U}) \to U' \ldots]$$

which, by weakening, gives us

$$;\Theta, \Theta(s') \vdash p : [\ldots l : (\vec{U}) \to U' \ldots] \tag{4}$$

Lastly, because

$$(\Delta, \Delta(s') \vdash C' : \Theta, \Theta(s'))$$

and

$$C' \equiv C'[n\langle \mathsf{let}\ x : T = \mathsf{return}\,(v : U)\ \mathsf{in}\ t\rangle]$$

we see that

$$;\Delta, \Delta(s'), \Theta, \Theta(s'), \Theta' \vdash v : U$$

So, by Lemma 5.1 together with the typing side-conditions for call-input transitions, we have that $U = U'$, and so

$$;\Delta, \Delta(s), \Theta, \Theta(s), \Theta' \vdash v : U \tag{5}$$

We collect the statements (1)–(5) together to see that they form the hypotheses of the type rule which allows us to conclude

$$\Delta \vdash s'\ \nu(\Theta')\,.\,n\langle \mathsf{return}\, v\rangle ! : \mathsf{trace}\ \Theta$$

as required.

Case $s = s'\ \nu(\Delta')\,.\,n\langle \mathsf{return}\, v\rangle ?$. Similar to previous case. □

5.2 Information order on traces

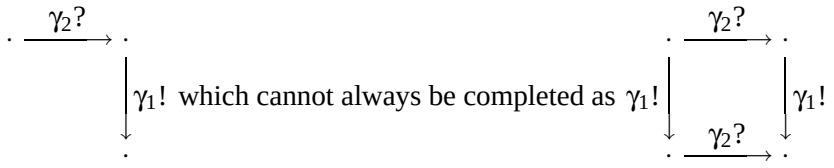
Information orders are an established technique for characterising observable behaviours of programs: behaviour B has less information than behaviour B' if any program which exhibits behaviour B' also exhibits behaviour B. They have been used extensively in denotational models for higher-order languages, notably Scott's treatment of information systems for the λ-calculus [28] and Abramsky's domain theory in logical form and its application to the lambda-calculus [1, 2]. They are also evident in much work on synchronous process languages, for example the information order characterising may testing is simply prefix order on traces [12], and on must testing is the order on failures–divergences pairs [5]. The first use of an information order to characterise observable behaviour in an asynchronous language appears in [4].

In the current setting we use the information order to characterise three important properties of the concurrent object calculus:

1. Prefix ordering: any component with trace sr also has trace s.

2. Input-enabling: any component with trace s also has trace $s\gamma?$ whenever this latter trace is well-typed. This property arises in our language because a component must always be prepared to accept an incoming method call on any of its objects, and similarly a component which has made an outgoing method call must be prepared to accept any incoming result. This feature is similar to input-enabling for I/O automata [19] and input receptivity in the asynchronous π-calculus of [15].

3. Commutativity of actions originating in different threads: under certain conditions, any component with the trace $saa'r$ also has the trace $sa'ar$. A good survey of such properties is given, in an abstract setting, by Selinger [29]. In our setting these commutativity properties are generated by the diamond properties shown in Figure 17. Note that these allow almost all actions to be commuted with the exception of:



The information preorder on traces $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$ is generated by axioms (where in each case we require both sides of the inequation to be well-typed traces):

$$\Delta \vdash s \sqsubseteq sr : \mathsf{trace}\ \Theta$$
$$\Delta \vdash s\gamma? \sqsubseteq s : \mathsf{trace}\ \Theta$$
$$\Delta \vdash s\gamma_1?\gamma_2!r \sqsubseteq s\gamma_2!\gamma_1?r : \mathsf{trace}\ \Theta$$
$$\Delta \vdash s\nu(\Delta).\gamma_1?\gamma_2?r \sqsubseteq s\nu(\Delta).\gamma_2?\gamma_1?r : \mathsf{trace}\ \Theta$$
$$\Delta \vdash s\nu(\Theta).\gamma_1!\gamma_2!r \sqsubseteq s\nu(\Theta).\gamma_2!\gamma_1!r : \mathsf{trace}\ \Theta$$

One consequence of the requirement that related traces are well-typed is that in the latter three axioms γ_1 and γ_2 are actions generated by different threads, and so correspond to the diamond properties in Figure 17.

Proposition 5.4 (Information Order Closure) *If* $(\Delta \vdash C : \Theta) \xrightarrow{s}$ *and* $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$ *then* $(\Delta \vdash C : \Theta) \xrightarrow{r}$.

Proof: Show that the diagrams in Figure 17 can be completed. The result follows by an induction on the derivation of $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$. □

We finish this section with a technical lemma used in the proof of completeness.

Lemma 5.5 (Information Order Duality) *If* $\Delta \vdash r\gamma! \sqsubseteq s\gamma! : \mathsf{trace}\ \Theta$ *and* $\mathsf{fn}(\gamma) \cap \Theta(r) = \emptyset$ *and* $\gamma! \notin s, r$ *then* $\Theta \vdash \overline{s} \sqsubseteq \overline{r} : \mathsf{trace}\ \Delta$.

Proof: We write $\Delta \vdash r \sqsubseteq^n s : \mathsf{trace}\ \Theta$ if $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$ can be derived using n instances of transitivity and no reflexivity. It is sufficient to show, by induction on n, that

$$\Delta \vdash r_1\gamma!r_2 \sqsubseteq^n s\gamma! : \mathsf{trace}\ \Theta \text{ implies } \Theta \vdash \overline{s} \sqsubseteq \overline{r_1} : \mathsf{trace}\ \Delta$$

whenever $\mathsf{fn}(\gamma) \cap \Theta(r_1) = \emptyset$ and $\gamma! \notin s, r_1$. The base case, $n = 0$, asks that $\Delta \vdash r_1\gamma!r_2 \sqsubseteq s_1\gamma! : \mathsf{trace}\ \Theta$ be derived from axioms alone. The argument is similar to that used in the inductive case so we omit it here. Suppose then that $\Delta \vdash r_1\gamma!r_2 \sqsubseteq^{n+1} s\gamma! : \mathsf{trace}\ \Theta$, that is

$$\Delta \vdash r_1\gamma!r_2 \sqsubseteq^0 q \sqsubseteq^n s\gamma! : \mathsf{trace}\ \Theta$$

for some q. We examine each of the five axioms in turn (for brevity we will elide the type environments in the judgements $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$):

(i) Suppose q is $r_1\gamma!r_2r$ so that

$$r_1\gamma!r_2 \sqsubseteq^0 r_1\gamma!r_2r \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to $q = r_1\gamma!r_2r$ to obtain $\overline{s} \sqsubseteq \overline{r_1}$ as required.

(ii) Suppose r_2 is $r_2'\gamma'?$ and q is $r_1\gamma!r_2'$ so that

$$r_1\gamma!r_2'\gamma'? \sqsubseteq^0 r_1\gamma!r_2' \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to finish.

The following diagrams can be completed (when $\mathsf{thread}\ (\gamma_1) \neq \mathsf{thread}\ (\gamma_2)$):

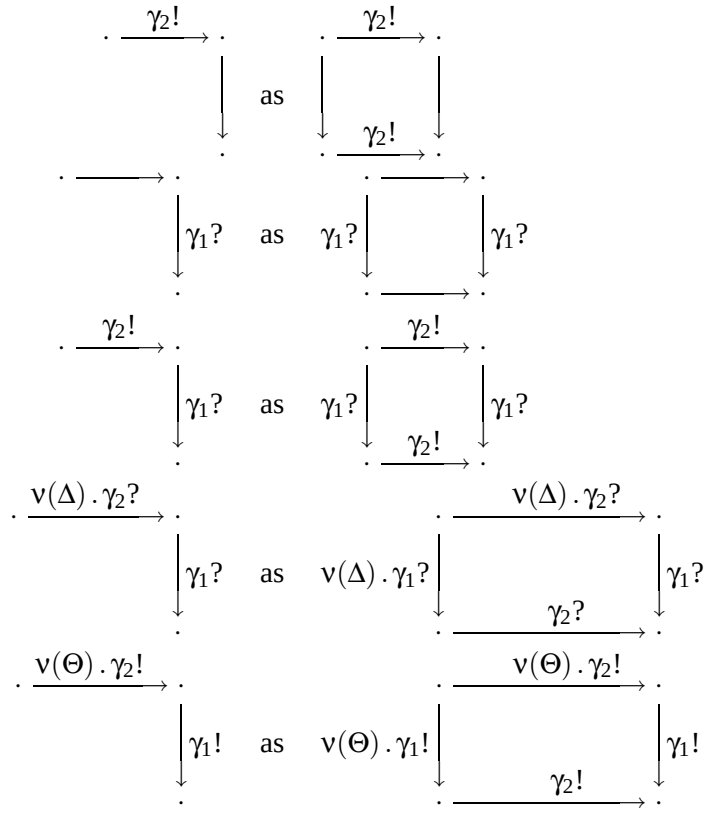


Figure 17: Diamond properties of the labelled transition system

(iii) (a) Suppose r_1 is $r_1'\gamma_1?\gamma_2!r_1''$ and q is $r_1'\gamma_2!\gamma_1?r_1''\gamma!r_2$ so that

$$r_1'\gamma_1?\gamma_2!r_1''\gamma!r_2 \sqsubseteq^0 r_1'\gamma_2!\gamma_1?r_1''\gamma!r_2 \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to see that

$$\bar{s} \sqsubseteq \vec{r_1'}\gamma_2?\gamma_1!\vec{r_1''} \sqsubseteq \vec{r_1'}\gamma_1!\gamma_2?\vec{r_1''} = \bar{r_1}$$

as required.

(b) Suppose r_2 is $r_2'\gamma_1?\gamma_2!r_2''$ and q is $r_1\gamma!r_2'\gamma_2!\gamma_1?r_2''$ so that

$$r_1\gamma!r_2'\gamma_1?\gamma_2!r_2'' \sqsubseteq^0 r_1\gamma!r_2'\gamma_2!\gamma_1?r_2'' \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to see $\bar{s} \sqsubseteq \bar{r_1}$ as required.

(c) Suppose r_1 is $r_1'\gamma'?$ and q is $r_1'\gamma!\gamma'?r_2$ so that

$$r_1'\gamma'?\gamma!r_2 \sqsubseteq^0 r_1'\gamma!\gamma'?r_2 \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to obtain $\bar{s} \sqsubseteq \vec{r_1'}$ and use the first axiom and transitivity to see $\bar{s} \sqsubseteq \vec{r_1'} \sqsubseteq \vec{r_1'}\gamma'! = \bar{r_1}$.

(iv) (a) Suppose r_1 is $r_1'\nu(\Delta).\gamma_1?\gamma_2?r_1''$ and q is $r_1'\nu(\Delta).\gamma_2?\gamma_1?r_1''\gamma!r_2$ so that

$$r_1'\nu(\Delta).\gamma_1?\gamma_2?r_1''\gamma!r_2 \sqsubseteq^0 r_1'\nu(\Delta).\gamma_2?\gamma_1?r_1''\gamma!r_2 \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to obtain $\bar{s} \sqsubseteq \vec{r_1'}\nu(\Delta).\gamma_2!\gamma_1!\vec{r_1''}$ and we note that

$$\vec{r_1'}\nu(\Delta).\gamma_2!\gamma_1!\vec{r_1''} \sqsubseteq \vec{r_1'}\nu(\Delta).\gamma_1!\gamma_2!\vec{r_1''} = \bar{r_1}$$

as required.

(b) Suppose r_2 is $r_2'\nu(\Delta).\gamma_1?\gamma_2?r_2''$ and q is $r_1\gamma!r_2'\nu(\Delta).\gamma_2?\gamma_1?r_2''$ so that

$$r_1\gamma!r_2'\nu(\Delta).\gamma_1?\gamma_2?r_2'' \sqsubseteq^0 r_1\gamma!r_2'\nu(\Delta).\gamma_2?\gamma_1?r_2'' \sqsubseteq^n s\gamma!.$$

We apply the inductive hypothesis to obtain $\bar{s} \sqsubseteq \bar{r_1}$ as required.

(v) (a) Suppose r_1 is $r_1'\nu(\Theta).\gamma_1!\gamma_2!r_1''$ and q is $r_1'\nu(\Theta).\gamma_2!\gamma_1!r_1''\gamma!r_2$, for which the proof follows as for Case (iv)(a).

(b) Suppose r_2 is $r_2'\nu(\Theta).\gamma_1!\gamma_2!r_2''$ and q is $r_1\gamma!r_2'\nu(\Theta).\gamma_2!\gamma_1!r_2''$, for which the proof follows as for Case (iv)(b).

(c) Suppose r_1 is $r_1'\nu(\Theta).\gamma'!$ and q is $r_1'\nu(\Theta).\gamma!\gamma'!r_2$ so that

$$r_1'\nu(\Theta).\gamma'!\gamma!r_2 \sqsubseteq^0 r_1'\nu(\Theta).\gamma!\gamma'!r_2 \sqsubseteq^n s\gamma!.$$

We know that $\mathrm{fn}(\gamma) \cap \Theta(r_1) = \emptyset$. This implies that Θ must be empty. Therefore we can apply the inductive hypothesis to obtain $\bar{s} \sqsubseteq \vec{r_1'}$ and then note $\vec{r_1'} \sqsubseteq \vec{r_1'}\nu(\Theta).\gamma'? = \bar{r_1}$ by the first axiom.

(d) Suppose r_2 is $\gamma''!r_2'$, γ is $\nu(\Theta).\gamma'$ and q is $r_1\nu(\Theta).\gamma''!\gamma'!r_2'$ so that

$$r_1\nu(\Theta).\gamma'\gamma''!r_2' \sqsubseteq^0 r_1\nu(\Theta).\gamma''!\gamma'!r_2' \sqsubseteq^n s\nu(\Theta).\gamma'!.$$

We first show a subsidiary result (as an induction on the derivation of $\sqsubseteq$), that:

$$\text{if } r_3\ \nu(n:T).\gamma_3!\ r_4\ \gamma_4!\ r_5 \sqsubseteq s_3\ \nu(n:T).\gamma_5!\ s_4 \text{ then } s_4 \neq \varepsilon \qquad (1)$$

from which it follows that Θ is empty. The inductive hypothesis tells us that $\bar{s} \sqsubseteq \bar{r}\nu(\Theta).\gamma''?$ and we note that $\bar{s} \sqsubseteq \bar{r_1}\nu(\Theta).\gamma''? \sqsubseteq \bar{r_1}$ follows from the second axiom. □

5.3 Definability of traces

For a well-typed trace $\Delta \vdash s : \mathsf{trace}\,\Theta$ we give the definition of a component $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ in Figure 18. It is this component that we will show to exhibit the trace s and only traces r such that $r \sqsubseteq s$.

The definition of $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ is rather lengthy so we offer an indication of how it is constructed. Firstly, we construct two objects called Ref and State. The former contains a field holding a pointer to the latter. The State object provides type-indexed families of methods called out, inReturn, and inCall. These are all defined inductively over prefixes r of the trace s and we write, for instance, $\mathsf{State}(\Delta \vdash r \leq s : \mathsf{trace}\,\Theta)$ to represent the state of the State object after the component has performed the actions in r. The initial state of the State object is given by $\mathsf{State}(\Delta \vdash \varepsilon \leq s : \mathsf{trace}\,\Theta)$.

We also provide object and thread definitions for all those references for which the type demands it, i.e. those in Θ. The object definitions provide methods according to the object types, where the method bodies simply indirectly re-route all calls to the appropriate State.inCall. The thread definitions make indirect calls to State.out. It it through these that traces are begun.

The bodies for the out, inReturn, and inCall methods depend on the next action in the trace we are providing a definition for. For instance, if the next action to be performed is an output $n\langle \mathsf{call}\, p.l(\vec{v})\rangle!$ then all of the bodies will be a stopped thread save for out which will have a method body which will check that the calling thread is n and, if so, update Ref to point to a new State object which will perform the next action in the trace. It will then indirectly call State.inReturn with the result of calling $p.l(\vec{v})$ (on dangling p) to listen for an input interaction (cf. the labelled transition rule for output, any subsequent action at this thread must be an input). Having successfully observed an input interaction, the line of interrogation in this thread is complete so it must reset itself by returning to a state in which it makes an indirect call to State.out. Similar definitions are given for each type of action.

We provide no synchronisation in the $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ component so that there is no guarantee that the reductions will follow the precise sequence of calls needed to exhibit the trace. However, with respect to may testing, this is irrelevant as we are only looking for one possible successful sequence of execution. We do guarantee the existence of this in Proposition 5.8.

In Figure 21, we give an example of the definability component for a two-action trace, and show how the trace is generated. Note that this component has many other traces, due to input-enabling, but that all of these traces are below the given trace in the information order.

To be of use in the completeness proof we need to know that $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ is well-typed. This is the subject of the next two lemmas.

Lemma 5.6 *If* $\Gamma;\Delta \vdash \vec{v} : \vec{U}$ *and* $\Gamma;\Delta,\Delta' \vdash \vec{p} : \vec{U}$ *and* $\Gamma;\Delta,\Delta' \vdash t : T$ *then* $\Gamma;\Delta \vdash (\mathsf{if}\ \Delta \vdash (\vec{v}) = \nu(\Delta').(\vec{p})\ \mathsf{then}\ t) : T$.

Proof: Straightforward induction on the definition of $\mathsf{if}\ \Delta \vdash (\vec{v}) = \nu(\Delta').(\vec{p})\ \mathsf{then}\ t$. □

Lemma 5.7 *If* $\Delta \vdash s : \mathsf{trace}\,\Theta$ *then* $\Delta \vdash \mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta) : \Theta$.

Proof: By examining the definition of $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ we see that we are required to show that

(i) $\Delta, \Theta, \Theta(s), \mathsf{state}_\varepsilon : \mathsf{State} \vdash \mathsf{ref}[\mathsf{val} = \mathsf{state}_\varepsilon] : (\mathsf{ref} : \mathsf{Ref})$

$$
\begin{array}{l}
\mathsf{Comp}\ (\Delta \vdash s : \mathsf{trace}\ \Theta) = \nu(\Theta(s), \mathsf{ref} : \mathsf{Ref}, \mathsf{state}_\varepsilon : \mathsf{State})\,.\,(\\
\quad \mathsf{ref}[\mathsf{val} = \mathsf{state}_\varepsilon] \parallel \\
\quad \mathsf{state}_\varepsilon[\mathsf{State}(\Delta \vdash \varepsilon \leq s : \mathsf{trace}\ \Theta)] \parallel \\
\quad \prod\{p[l_i = \mathsf{ref.val.inCall}_{p.l_i:L_i} \mid i = 1 \ldots n] \mid p : [l_i : L_i \mid i = 1 \ldots n] \in \Theta, \Theta(s)\} \parallel \\
\quad \prod\{n\langle \mathsf{ref.val.out}_{\mathsf{none}}() \rangle \mid n : \mathsf{thread} \in \Theta, \Theta(s)\} \\
)
\end{array}
$$

$$
\mathsf{Ref} = [\mathsf{val} : \mathsf{State}]
$$

$$
\mathsf{State} = [\mathsf{out}_T : () \to T, \mathsf{inReturn}_T : (T) \to T, \mathsf{inCall}_{p.l:L} : L]
$$

$$
\begin{array}{l}
\mathsf{State}(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) = (\\
\quad \mathsf{out}_T = \mathsf{Out}_T(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta), \\
\quad \mathsf{inReturn}_T = \mathsf{InReturn}_T(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta), \\
\quad \mathsf{inCall}_{p.l:L} = \mathsf{InCall}_{p.l:L}(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) \\
)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Out}_T(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) = \lambda()\,.\,(\\
\quad \text{when } r\,a \leq s \text{ and } a = \nu(\Theta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v}) \rangle ! \text{ and } ; \Delta, \Theta, \Delta(r), \Theta(r), \Theta' \vdash p.l(\vec{v}) : U : \\
\qquad \text{if currentthread} = n \text{ then} \\
\qquad\quad \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\ \Theta)]; \\
\qquad\quad \mathsf{ref.val.inReturn}_U(p.l(\vec{v})); \\
\qquad\quad \mathsf{ref.val.out}_T() \\
\quad \text{when } r\,a \leq s \text{ and } a = \nu(\Theta')\,.\,n\langle \mathsf{return}\, v \rangle ! \text{ and } ; \Delta, \Theta, \Delta(r), \Theta(r), \Theta' \vdash v : T : \\
\qquad \text{if currentthread} = n \text{ then} \\
\qquad\quad \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\ \Theta)]; \\
\qquad\quad v \\
\quad \text{otherwise :} \\
\qquad \mathsf{stop} \\
)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{InReturn}_T(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) = \lambda(x : T)\,.\,(\\
\quad \text{when } r\,a \leq s \text{ and } a = \nu(\Delta')\,.\,n\langle \mathsf{return}\, v \rangle ? \text{ and } ; \Delta, \Theta, \Delta(r), \Theta(r), \Delta' \vdash v : T : \\
\qquad \text{if } \Delta, \Theta, \Delta(r), \Theta(r) \vdash (\text{currentthread}, x) = \nu(\Delta')\,.\,(n, v) \text{ then} \\
\qquad\quad \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\ \Theta)]; \\
\qquad\quad v \\
\quad \text{otherwise :} \\
\qquad \mathsf{stop} \\
)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{InCall}_{p.l:(\vec{T}) \to T}(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) = \lambda(\vec{x} : \vec{T})\,.\,(\\
\quad \text{when } r\,a \leq s \text{ and } a = \nu(\Delta')\,.\,n\langle \mathsf{call}\, p.l(\vec{v}) \rangle ? \text{ and } ; \Delta, \Theta, \Delta(r), \Theta(r), \Delta' \vdash \vec{v} : \vec{T} : \\
\qquad \text{if } \Delta, \Theta, \Delta(r), \Theta(r) \vdash (\text{currentthread}, \vec{x}) = \nu(\Delta')\,.\,(n, \vec{v}) \text{ then} \\
\qquad\quad \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\ \Theta)]; \\
\qquad\quad \mathsf{ref.val.out}_T() \\
\quad \text{otherwise :} \\
\qquad \mathsf{stop} \\
)
\end{array}
$$

Figure 18: Definition of $\mathsf{Comp}\ (\Delta \vdash s : \mathsf{trace}\ \Theta)$

$$
\begin{array}{rcl}
\text{if } \Delta \vdash () = \nu() \,.\, () \text{ then } t & = & t \\
\text{if } \Delta \vdash (v,\vec{v}) = \nu(p:U,\vec{n}:\vec{T}) \,.\, (p,\vec{p}) \text{ then } t & = & \text{if } v \notin \Delta^{-1}(U) \text{ then} \\
 & & \quad (\text{if } \Delta, p:U \vdash (\vec{v}) = \nu(\vec{n}:\vec{T}) \,.\, (\vec{p}) \text{ then } t)[v/p] \text{ else stop} \\
\text{if } \Delta \vdash (v,\vec{v}) = \nu(\vec{n}:\vec{T}) \,.\, (p,\vec{p}) \text{ then } t & = & \text{if } v = p \text{ then } (\text{if } \Delta \vdash (\vec{v}) = \nu(\vec{n}:\vec{T}) \,.\, (\vec{p}) \text{ then } t) \text{ else stop}
\end{array}
$$

Figure 19: Definition of if $\Delta \vdash (\vec{v}) = \nu(\vec{n}:\vec{T}) \,.\, (\vec{p})$ then t (when $p \notin \vec{n}$).

$$
\begin{array}{rcl}
\text{if } v \notin ()^{-1}(U) \text{ then } t \text{ else stop} & = & t \\
\text{if } v \notin (n:U,\Delta)^{-1}(U) \text{ then } t \text{ else stop} & = & \text{if } v = n \text{ then stop else } (\text{if } v \notin \Delta^{-1}(U) \text{ then } t \text{ else stop}) \\
\text{if } v \notin (n:T,\Delta)^{-1}(U) \text{ then } t \text{ else stop} & = & \text{if } v \notin \Delta^{-1}(U) \text{ then } t \text{ else stop}
\end{array}
$$

Figure 20: Definition of if $v \notin \Delta^{-1}(U)$ then t else stop (when $T \neq U$).

(ii) $\Delta, \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash \mathsf{state}_\varepsilon[\mathsf{State}(\Delta \vdash \varepsilon \leq s : \mathsf{trace}\ \Theta)] : (\mathsf{state}_\varepsilon : \mathsf{State})$

(iii) $\Delta, \Theta, \Theta(s) \setminus p, \mathsf{ref} : \mathsf{Ref}, \mathsf{state}_\varepsilon : \mathsf{State} \vdash p[l_i = \mathsf{ref.val.inCall}_{p.l_i:L_i} \mid i = 1 \ldots n] : (p : [l_i : L_i \mid i = 1 \ldots n])$ for each $p \in \Theta, \Theta(s)$

(iv) $\Delta, \Theta, \Theta(s) \setminus n, \mathsf{ref} : \mathsf{Ref}, \mathsf{state}_\varepsilon : \mathsf{State} \vdash n\langle \mathsf{ref.val.out}_{\mathsf{none}}() \rangle : (n : \mathsf{thread})$ for each $n \in \Theta, \Theta(s)$.

It is easy to check that all but (ii) follow from the definitions of the types State and Ref. We show (ii) by establishing

$$;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash [\mathsf{State}(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta)] : \mathsf{State}$$

by induction on the length of s less the length of r. The base case (when $s = r$) follows as each method body of $\mathsf{State}(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta)$ is stop and hence can be given any type. The inductive case relies on the following properties:

(a) $;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash \mathsf{Out}_T(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) : () \to T$

(b) $;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash \mathsf{InReturn}_T(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) : (T) \to T$

(c) $;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash \mathsf{InCall}_{p.l:L}(\Delta \vdash r \leq s : \mathsf{trace}\ \Theta) : L$

We only show how to establish (a) here as the remaining two cases can be dealt with similarly. Suppose then that $r\,a \leq s$ with $a = \nu(\Theta') \,.\, n\langle \mathsf{return}\, v \rangle!$ and $;\Delta, \Delta(r), \Theta, \Theta(r), \Theta' \vdash v : T$ It is easy to see by the inductive hypothesis that

$$;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\ \Theta)]; v : T$$

holds, and also that $;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash \mathsf{currentthread} : \mathsf{thread}$ and

$$;\Delta, \Delta(r), \Theta, \Theta(s), \mathsf{ref} : \mathsf{Ref} \vdash n : \mathsf{thread}.$$

This latter fact follows from $\Delta \vdash r\,a : \mathsf{trace}\ \Theta$ guaranteeing

$$;\Delta, \Delta(r), \Theta, \Theta(r), \mathsf{ref} : \mathsf{Ref} \vdash n : \mathsf{thread}.$$

We show the component given by $\mathsf{Comp}\ (r : \mathsf{IntRef} \vdash s : \mathsf{trace}\ )$ where s is

$$\begin{array}{l} \nu(n)\,.\,n\langle \mathsf{call}\, r.\mathsf{get}()\rangle? \\ n\langle \mathsf{return}\, 5\rangle! \end{array}$$

Let C_0 be defined as

$$\begin{array}{l} r[\\ \quad \mathsf{contents} = \mathsf{ref.val.inCall}_{r.\mathsf{contents}:()\to\mathsf{Int}}, \\ \quad \mathsf{get} = \mathsf{ref.val.inCall}_{r.\mathsf{get}:()\to\mathsf{Int}}, \\ \quad \mathsf{set} = \mathsf{ref.val.inCall}_{r.\mathsf{set}:(\mathsf{Int})\to\mathsf{Int}} \\ ] \parallel s_0[\\ \quad \mathsf{inCall}_{r.\mathsf{get}:()\to\mathsf{Int}} = \lambda().\langle \mathsf{ref.val} := [\\ \qquad \mathsf{out}_{\mathsf{Int}} = \lambda()\,.\,\langle \mathsf{ref.val} := [\ldots]; 5\rangle, \\ \qquad \ldots \\ \quad\ \ ]; \mathsf{ref.val.out}_{\mathsf{Int}}()\rangle, \\ \quad \ldots \\ ] \end{array}$$

where $\ldots$ is used to elide method definitions whose body is stop. Then we have

$$\mathsf{Comp}\ (r : \mathsf{IntRef} \vdash s : \mathsf{trace}\ ) = \nu(s_0)\,.\,(C_0 \parallel \mathsf{ref}[\mathsf{val} = s_0])$$

We now show how this component generates the trace s above. Let

$$\begin{array}{l} C_1 = C_0 \parallel s_1[\mathsf{out}_{\mathsf{Int}} = \lambda()\,.\,\langle \mathsf{ref.val} := [\ldots]; 5\rangle, \ldots] \\ C_2 = C_1 \parallel s_2[\ldots] \end{array}$$

We have

$$\begin{array}{lll} \nu(s_0)\,.\,(C_0 \parallel \mathsf{ref}[\mathsf{val} = s_0]) & \xrightarrow{\nu(n).n\langle \mathsf{call}\, r.\mathsf{get}()\rangle?} & \nu(s_0)\,.\,(C_0 \parallel \mathsf{ref}[\mathsf{val} = s_0] \parallel n\langle \mathsf{return}\,(r.\mathsf{get}() : \mathsf{Int})\rangle) \\ & \Longrightarrow & \nu(s_0, s_1)\,.\,(C_1 \parallel \mathsf{ref}[\mathsf{val} = s_1] \parallel n\langle \mathsf{return}\,(\mathsf{ref.val.out}_{\mathsf{Int}}() : \mathsf{Int})\rangle) \\ & \Longrightarrow & \nu(s_0, s_1, s_2)\,.\,(C_2 \parallel \mathsf{ref}[\mathsf{val} = s_2] \parallel n\langle \mathsf{return}\,(5 : \mathsf{Int})\rangle) \\ & \xrightarrow{n\langle \mathsf{return}\, 5\rangle!} & \nu(s_0, s_1, s_2)\,.\,(C_2 \parallel \mathsf{ref}[\mathsf{val} = s_2] \parallel n\langle \mathsf{block}\rangle) \end{array}$$

Figure 21: Example of definability component

We can now apply the previous Lemma to see that

$$;\Delta,\Delta(r),\Theta,\Theta(s),\mathsf{ref}:\mathsf{Ref}\vdash \quad \begin{array}{l}\mathsf{if}\ \Delta\vdash(\mathsf{currentthread})=\nu().(n)\ \mathsf{then}\\ \qquad \mathsf{ref.val}:=\mathsf{new}[\mathsf{State}(\Delta\vdash r\,a\leq s:\mathsf{trace}\,\Theta)];v\end{array}\quad :T$$

which gives us that $;\Delta,\Delta(r),\Theta,\Theta(s),\mathsf{ref}:\mathsf{Ref}\vdash \mathsf{Out}_T(\Delta\vdash r\leq s:\mathsf{trace}\,\Theta):()\to T$ as required.

Alternatively, suppose that $r\,a\leq s$ with $a=\nu(\Theta').\,n\langle\mathsf{call}\,p.l(\vec{v})\rangle!$ and $;\Delta,\Delta(r),\Theta,\Theta(r),\Theta'\vdash p.l(\vec{v}):U$. Given that $\mathsf{State.inReturn}_U:(U)\to U$, and that $\mathsf{State.out}_T:()\to T$ we can apply the inductive hypothesis and previous Lemma as above to see that

$$;\Delta,\Delta(r),\Theta,\Theta(s),\mathsf{ref}:\mathsf{Ref}\vdash \quad \begin{array}{l}\mathsf{if}\ \mathsf{currentthread}=n\ \mathsf{then}\\ \qquad \mathsf{ref.val}:=\mathsf{new}[\mathsf{State}(\Delta\vdash r\,a\leq s:\mathsf{trace}\,\Theta)];\\ \qquad \mathsf{ref.val.inReturn}_U(p.l(\vec{v}));\\ \qquad \mathsf{ref.val.out}_T()\end{array}\quad :T$$

as required.

Otherwise the body of $\mathsf{Out}_T(\Delta\vdash r\leq s:\mathsf{trace}\,\Theta)$ is stop and this can be given any type. □

Proposition 5.8 (Definability) *For any* $\Delta\vdash s:\mathsf{trace}\,\Theta$
we have $(\Delta\vdash\mathsf{Comp}\,(\Delta\vdash s:\mathsf{trace}\,\Theta):\Theta)\stackrel{r}{\Longrightarrow}$ *if and only if* $\Delta\vdash r\sqsubseteq s:\mathsf{trace}\,\Theta$.

Proof: Given in Appendix B. □

5.4 Proof of completeness

Theorem 5.9 (Completeness of traces for may testing) *If* $\Delta\models C_1\sqsubseteq_{may}C_2:\Theta$
then $\mathsf{Traces}(\Delta\vdash C_1:\Theta)\subseteq\mathsf{Traces}(\Delta\vdash C_2:\Theta)$.

Proof: Choose any trace s_1 such that:

$$(\Delta\vdash C_1:\Theta)\stackrel{s_1}{\Longrightarrow}(\Delta'\vdash C_1':\Theta')$$

By Proposition 5.3 we have that $\Delta\vdash s_1:\mathsf{trace}\,\Theta$, and it is easy to establish that $\Theta\vdash\overline{s_1}:\mathsf{trace}\,\Delta$.

Pick a fresh $b:\mathsf{barb}$ and let $\omega!$ be:

$$\omega!=\nu(n:\mathsf{thread}).\,n\langle\mathsf{call}\,b.\mathsf{succ}()\rangle!$$

and let C_0 be:

$$C_0=\mathsf{Comp}\,(\Theta,b:\mathsf{barb}\vdash\overline{s_1}\omega!:\mathsf{trace}\,\Delta)$$

Then by Proposition 5.8 we have:

$$(\Theta,b:\mathsf{barb}\vdash C_0:\Delta)\stackrel{\overline{s_1}}{\Longrightarrow}(\Theta',b:\mathsf{barb}\vdash C_0':\Delta')\stackrel{\omega!}{\longrightarrow}$$

and so $C_0'\downarrow_b$. Thus, by Lemma 4.1, Proposition 4.3, and Lemma 4.2 we have $(C_1\parallel C_0)\Downarrow_b$. We know that $\Delta\models C_1\sqsubseteq_{\mathrm{may}}C_2:\Theta$, that $\Theta,b:\mathsf{barb}\vdash C_0:\Delta$, and $(C_1\parallel C_0)\Downarrow_b$ so this implies $(C_2\parallel C_0)\Downarrow_b$. Thus, by Lemma 4.1 and Corollary 4.4 we can find s_2 such that:

$$(\Delta,b:\mathsf{barb}\vdash C_2:\Theta)\stackrel{s_2}{\Longrightarrow}(\Delta'',\Phi''\vdash C_2'':\Theta'',\Sigma'')$$
$$(\Theta,b:\mathsf{barb}\vdash C_0:\Delta)\stackrel{\overline{s_2}}{\Longrightarrow}(\Theta'',\Phi''\vdash C_0'':\Delta'',\Sigma'')$$

and either $C_0'' \downarrow_b$ or $C_2'' \downarrow_b$. Since b was chosen to be fresh, we must have that $C_0'' \downarrow_b$ and hence $(\Theta, b : \mathsf{barb} \vdash C_0 : \Delta) \stackrel{\bar{s}\omega!}{\Longrightarrow}$ so by Proposition 5.8: $\Theta, b : \mathsf{barb} \vdash \bar{s_2}\omega! \sqsubseteq \bar{s_1}\omega! : \mathsf{trace}\ \Delta$ and so by Lemma 5.5 and narrowing: $\Delta \vdash s_1 \sqsubseteq s_2 : \mathsf{trace}\ \Theta$. Thus, by Proposition 5.4 we have: $(\Delta \vdash C_2 : \Theta) \stackrel{s_1}{\Longrightarrow} (\Delta' \vdash C_2' : \Theta')$ as required. □

6 Restricted sub-languages

The proof techniques use to obtain full abstraction here are quite robust and can also be carried out for two restricted sub-languages:

1. The single-threaded sub-language is given by only allowing one name of type thread, and removing new thread creation from the expression language. The definability result for Proposition 5.8 does not use thread creation, so the proof of full abstraction goes through with only minor changes to the proof of Theorem 5.9.

2. The sub-language with only field update (and no method update) can be given the same trace semantics. The definability result for Proposition 5.8 only uses field update, and so the proof of full abstraction goes through unchanged.

Thus, not only do we have a full abstraction result for the concurrent object calculus, we can also specialise the results to become full abstraction result for other related languages.

One change which cannot easily be made is to remove the restriction that components be write closed, since method, and even field, updates are not generally externally observable. It is unlikely that traces which represent write interactions will be definable in the current sense. However, we do believe that the restriction to write closed components is a reasonable one, since it corresponds to existing 'best practice' for component design.

7 Conclusions and future work

In this paper we have presented the first fully abstract semantics for concurrent objects. The semantics is fairly simple, and corresponds loosely to some of the messages used in UML interaction diagrams. We do need to road test the trace semantics with some reasonably sized examples to demonstrate that the calculation of traces is tractable.

There are a number of issues left open:

- Our semantics has much of the flavour of game semantics [3, 16], and this connection should be investigated.

- The trace semantics characterise may testing, rather than the more common must testing or bisimulation equivalence.

- The object calculus presented here does not include subtyping. We believe that the techniques of [14] should be applicable to the provision of a fully abstract semantics even in the presence of subtyping.

A Proof of trace composition and decomposition

We have to prove that for any components $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma)$ and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma)$ such that $C_1 \between C_2 \equiv C$, we have:

1. *Composition*: If $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$
   and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \stackrel{\bar{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ then $C \Rightarrow C'$
   where $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C'$.

2. *Decomposition*: If $C \Rightarrow C'$ then there exists some trace s
   such that $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$
   and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \stackrel{\bar{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ where $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C'$.

A.1 Composition

We show four lemmas, from which Composition follows by a simple induction.

Lemma A.1

1. *If $C_1 \between C_2 \equiv D \parallel E$ then there exist components such that $C_1 \equiv D_1 \parallel E_1$ and $C_2 \equiv D_2 \parallel E_2$ with $D \equiv D_1 \between D_2$ and $E \equiv E_1 \between E_2$.*

2. *If $C_1 \between C_2 \equiv \nu(\vec{n} : \vec{T}) \,.\, C$ then there exist components such that $C_1 \equiv \nu(\vec{n}_1 : \vec{T}_1) \,.\, C_1'$ and $C_2 \equiv \nu(\vec{n}_2 : \vec{T}_2) \,.\, C_2'$ with $(\vec{n} : \vec{T}) = (\vec{n}_1 : \vec{T}_1, \vec{n}_2 : \vec{T}_2)$ and $C \equiv C_1' \between C_2'$.*

Proof: Proved by induction on the derivation of $C_1 \between C_2$. □

Lemma A.2 *If $C_1 \between C_2 \equiv C$ and $C_1 \xrightarrow{\beta} C_1'$ then $C \xrightarrow{\beta} C'$ where $C_1' \between C_2 \equiv C'$.*

Proof: An induction on the proof of $C_1 \xrightarrow{\beta} C_1'$, making use of Lemma A.1. □

Lemma A.3 *If $C_1 \between C_2 \equiv C$ and $C_1 \xrightarrow{\tau} C_1'$ then $C \xrightarrow{\tau} C'$ where $C_1' \between C_2 \equiv C'$.*

Proof: An induction on the proof of $C_1 \xrightarrow{\tau} C_1'$, making use of Lemma A.1. □

Lemma A.4 *If $C_1 \between C_2 \equiv C$ and $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xrightarrow{\gamma?} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$*
and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \xrightarrow{\gamma!} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ then $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C$.

Proof: A case analysis on γ.

- **Case** $(\gamma = \nu(\vec{n} : \vec{T}) \,.\, n\langle \mathsf{call}\, p.l(\vec{v}) \rangle$ and $n \notin \Sigma)$.

  Since $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xrightarrow{\gamma?} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$ and $n \notin \Sigma$, we must have that:

  $$\begin{aligned} C_1' &\equiv C_1 \parallel n\langle \mathsf{let}\, y : T = p.l(\vec{x})\ \mathsf{in}\ \mathsf{return}\,(y : T) \rangle \\ \Delta' &= (\Delta, \vec{n} : \vec{T}) \setminus (n : \mathsf{thread}) \\ \Theta' &= \Theta \\ \Sigma' &= \Sigma, n : \mathsf{thread} \end{aligned}$$

Since $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \xrightarrow{\gamma!} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ we must have that:

$$C_2 \equiv \nu(\vec{n} : \vec{T}) \,.\, \nu(\vec{p} : \vec{U}) \,.\, (C_2'' \parallel n\langle \mathsf{let}\ x : T = p.l(\vec{x})\ \mathsf{in}\ t\rangle)$$
$$C_2' \equiv \nu(\vec{p} : \vec{U}) \,.\, (C_2'' \parallel n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t\rangle)$$

We can then show that:

$$C_1 \between C_2 \equiv \nu(\vec{n} : \vec{T}) \,.\, \nu(\vec{p} : \vec{U}) \,.\, ((C_1 \between C_2'') \parallel n\langle \mathsf{let}\ x : T = p.l(\vec{x})\ \mathsf{in}\ t\rangle)$$

and that:

$$C_1' \between C_2' \equiv \nu(\vec{p} : \vec{U}) \,.\, ((C_1 \between C_2'') \parallel n\langle \mathsf{let}\ x : U = p.l(\vec{x})\ \mathsf{in}\ t\rangle)$$

and so:

$$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C$$

as required.

- **Case** ($\gamma = \nu(\vec{n} : \vec{T}) \,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle$ and $n \in \Sigma$).
  Similar to the previous case.

- **Case** ($\gamma = \nu(\vec{n} : \vec{T}) \,.\, n\langle \mathsf{return}\, v\rangle$).
  Since $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xrightarrow{\gamma?} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$ we must have that:

$$C_1 \equiv \nu(\vec{p}_1 : \vec{U}_1) \,.\, (C_1'' \parallel n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t_1\rangle)$$
$$C_1' \equiv \nu(\vec{p}_1 : \vec{U}_1) \,.\, (C_1'' \parallel n\langle t_1[v/x]\rangle)$$
$$\Delta' = \Delta, \vec{n} : \vec{T}$$
$$\Theta' = \Theta$$
$$\Sigma' = \Sigma$$

  Since $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \xrightarrow{\gamma!} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ we must have that:

$$C_2 \equiv \nu(\vec{n} : \vec{T}) \,.\, \nu(\vec{p}_2 : \vec{U}_2) \,.\, (C_2'' \parallel n\langle \mathsf{let}\ y : U = \mathsf{return}\,(v : T)\ \mathsf{in}\ t_2\rangle)$$
$$C_2' \equiv \nu(\vec{p}_2 : \vec{U}_2) \,.\, (C_2'' \parallel n\langle \mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t_2\rangle)$$

  We then show that:

$$C_1 \between C_2 \equiv \nu(\vec{n} : \vec{T}) \,.\, \nu(\vec{p}_1 : \vec{U}_1) \,.\, \nu(\vec{p}_2 : \vec{U}_2) \,.\, ((C_1'' \between C_2'') \parallel n\langle (\mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t_2) \between (t_1[v/x])\rangle)$$

  and that:

$$C_1' \between C_2' \equiv \nu(\vec{p}_1 : \vec{U}_1) \,.\, \nu(\vec{p}_2 : \vec{U}_2) \,.\, ((C_1'' \between C_2'') \parallel n\langle (\mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t_2) \between (t_1[v/x])\rangle)$$

  and so:

$$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C$$

  as required. □

Composition follows, by induction on the derivation of $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$ and $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \stackrel{\overline{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$, making use of Lemmas A.2, A.3 and A.4.

A.2 Decomposition

We show three lemmas, from which Decomposition follows.

Lemma A.5 *For any* $\Delta, \Phi \vdash C_1 : \Theta, \Sigma$ *and* $\Theta, \Phi \vdash C_2 : \Delta, \Sigma$ *if* $(C_1 \between C_2) \equiv \nu(\vec{n} : \vec{T}) \,.\, (C \parallel n\langle \mathsf{let}\ x : T = e\ \mathsf{in}\ t\rangle)$ *then either we have:*

$$(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash \nu(\vec{n}_1 : \vec{T}_1) \,.\, (C_1' \parallel n\langle \mathsf{let}\ x : T = e\ \mathsf{in}\ t_1\rangle) : \Theta', \Sigma')$$
$$(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \stackrel{\bar{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$$

where:

$$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, \nu(\vec{n}_1 : \vec{T}_1) \,.\, (C_1' \parallel n\langle t_1\rangle) \between C_2' \equiv \nu(\vec{n} : \vec{T}) \,.\, (C \parallel n\langle t\rangle)$$

or symmetrically, swapping the roles of C_1 *and* C_2.

Proof: An induction on the derivation of:

$$(C_1 \between C_2) \equiv \nu(\vec{n} : \vec{T}) \,.\, (C \parallel n\langle \mathsf{let}\ x : T = e\ \mathsf{in}\ t\rangle)$$

The interesting case is when:

$$\begin{aligned} C_1 &\equiv n\langle \mathsf{let}\ x_1 : T_1 = \mathsf{block}\ \mathsf{in}\ t_1\rangle \\ C_2 &\equiv n\langle \mathsf{let}\ x_2 : T_2 = \mathsf{return}\,(v : T_1)\ \mathsf{in}\ t_2\rangle \end{aligned}$$

and:

$$n\langle t_1[v/x]\rangle \between n\langle \mathsf{let}\ x_2 : T_2 = \mathsf{block}\ \mathsf{in}\ t_2\rangle \equiv \nu(\vec{n} : \vec{T}) \,.\, (C \parallel n\langle \mathsf{let}\ x : T = e\ \mathsf{in}\ t\rangle)$$

so by defi nition of the lts, and by induction we have:

$$(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xrightarrow{n\langle \mathsf{return}\, v\rangle ?} (\Delta, \Phi \vdash n\langle t_1[v/x]\rangle : \Theta, \Sigma)$$
$$(\Delta, \Phi \vdash n\langle t_1[v/x]\rangle : \Theta, \Sigma) \stackrel{s}{\Longrightarrow} (\Delta', \Phi \vdash \nu(\vec{n}_1 : \vec{T}_1) \,.\, (C_1' \parallel n\langle \mathsf{let}\ x : T = e\ \mathsf{in}\ t_1\rangle) : \Theta', \Sigma')$$

and

$$(\Delta, \Phi \vdash C_2 : \Theta, \Sigma) \xrightarrow{n\langle \mathsf{return}\, v\rangle !} (\Theta, \Phi \vdash n\langle \mathsf{let}\ x_2 : T_2 = \mathsf{block}\ \mathsf{in}\ t_2\rangle : \Delta, \Sigma)$$
$$(\Theta, \Phi \vdash n\langle \mathsf{let}\ x_2 : T_2 = \mathsf{block}\ \mathsf{in}\ t_2\rangle : \Delta, \Sigma) \stackrel{\bar{s}}{\Longrightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$$

where

$$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, \nu(\vec{n}_1 : \vec{T}_1) \,.\, (C_1' \parallel n\langle t_1\rangle) \between C_2' \equiv \nu(\vec{n} : \vec{T}) \,.\, (C \parallel n\langle t\rangle)$$

or symmetrically, as required. □

Lemma A.6 *If* $C_1 \between C_2 \equiv C$ *and* $C \xrightarrow{\beta} C'$ *then there exists some trace s such that* $(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \stackrel{s}{=\!\Rightarrow} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$ *and* $(\Theta, \Phi \vdash C_2 : \Delta, \Sigma) \stackrel{\bar{s}}{=\!\Rightarrow} (\Theta', \Phi \vdash C_2' : \Delta', \Sigma')$ *where* $\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma) \,.\, (C_1' \between C_2') \equiv C'$.

Proof: We must have that $C \xrightarrow{\beta} C'$ from:

$$\begin{array}{rcl} C & \equiv & \nu(\vec{n}:\vec{T}).(D \parallel n\langle \mathsf{let}\ x:T=e\ \mathsf{in}\ t\rangle) \\ C' & \equiv & \nu(\vec{n}:\vec{T},\vec{n}':\vec{T}').(D \parallel E \parallel n\langle \mathsf{let}\ \vec{x}:\vec{T}=\vec{e}\ \mathsf{in}\ t\rangle) \end{array}$$

where we have an axiom:

$$n\langle \mathsf{let}\ x:T=e\ \mathsf{in}\ t\rangle \xrightarrow{\beta} \nu(\vec{n}':\vec{T}').(E \parallel n\langle \mathsf{let}\ \vec{x}:\vec{T}=\vec{e}\ \mathsf{in}\ t\rangle$$

We then use Lemma A.5 to get (wlog):

$$\begin{array}{l} (\Delta,\Phi\vdash C_1:\Theta,\Sigma) \xrightarrow{s} (\Delta',\Phi\vdash \nu(\vec{n}_1:\vec{T}_1).(C_1'' \parallel n\langle \mathsf{let}\ x:T=e\ \mathsf{in}\ t_1\rangle):\Theta',\Sigma') \\ (\Theta,\Phi\vdash C_2:\Delta,\Sigma) \xrightarrow{\bar{s}} (\Theta',\Phi\vdash C_2':\Delta',\Sigma') \end{array}$$

where

$$\nu(\Delta',\Theta',\Sigma'\setminus\Delta,\Theta,\Sigma).\nu(\vec{n}_1:\vec{T}_1).(C_1'' \parallel n\langle t_1\rangle) \between C_2' \equiv \nu(\vec{n}:\vec{T}).(D \parallel n\langle t\rangle)$$

and so we use the axiom to get:

$$(\Delta,\Phi\vdash C_1:\Theta,\Sigma) \xrightarrow{s} (\Delta',\Phi\vdash C_1':\Theta',\Sigma')$$

where we defi ne:

$$C_1' \equiv \nu(\vec{n}_1:\vec{T}_1,\vec{n}':\vec{T}').(C_1'' \parallel E \parallel n\langle \mathsf{let}\ \vec{x}:\vec{T}=\vec{e}\ \mathsf{in}\ t_1\rangle)$$

and then verify that:

$$\nu(\Delta',\Theta',\Sigma'\setminus\Delta,\Theta,\Sigma).(C_1' \between C_2') \equiv C'$$

as required. □

Lemma A.7 *If $C_1 \between C_2 \equiv C$ and $C \xrightarrow{\tau} C'$ then there exists some trace s such that $(\Delta,\Phi\vdash C_1 : \Theta,\Sigma) \stackrel{s}{\Rightarrow} (\Delta',\Phi\vdash C_1':\Theta',\Sigma')$ and $(\Theta,\Phi\vdash C_2:\Delta,\Sigma) \stackrel{\bar{s}}{\Rightarrow} (\Theta',\Phi\vdash C_2':\Delta',\Sigma')$ where $\nu(\Delta',\Theta',\Sigma'\setminus\Delta,\Theta,\Sigma).(C_1' \between C_2') \equiv C'$.*

Proof: We must have that:

$$\begin{array}{rcl} C & \equiv & \nu(\vec{n}:\vec{T}).(D \parallel p[O] \parallel n\langle \mathsf{let}\ x:T=e\ \mathsf{in}\ t\rangle) \\ C' & \equiv & \nu(\vec{n}:\vec{T}).(D \parallel p[O'] \parallel n\langle \mathsf{let}\ x:T=e'\ \mathsf{in}\ t\rangle) \end{array}$$

where we have an axiom:

$$p[O] \parallel n\langle \mathsf{let}\ x:T=e\ \mathsf{in}\ t\rangle \xrightarrow{\tau} p[O'] \parallel n\langle \mathsf{let}\ x:T=e'\ \mathsf{in}\ t\rangle$$

We then use Lemma A.5 to get (wlog):

$$\begin{array}{l} (\Delta,\Phi\vdash C_1:\Theta,\Sigma) \xrightarrow{s} (\Delta',\Phi\vdash \nu(\vec{n}_1:\vec{T}_1).(C_1'' \parallel n\langle \mathsf{let}\ x:T=e\ \mathsf{in}\ t_1\rangle):\Theta',\Sigma') \\ (\Theta,\Phi\vdash C_2:\Delta,\Sigma) \xrightarrow{\bar{s}} (\Theta',\Phi\vdash C_2'':\Delta',\Sigma') \end{array}$$

where:

$$\nu(\Delta',\Theta',\Sigma'\setminus\Delta,\Theta,\Sigma).\nu(\vec{n}_1:\vec{T}_1).(C_1'' \parallel n\langle t_1\rangle) \between C_2'' \equiv \nu(\vec{n}:\vec{T}).(D \parallel p[O] \parallel n\langle t\rangle)$$

We now have three cases:

- **Case** $(p \in \mathsf{dom}\,(C_1''))$.

  We must have that:

  $$C_1'' \equiv \nu(\vec{p} : \vec{U})\,.\,(C_1''' \parallel p[O])$$

  and so we use the axiom to get:

  $$(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xRightarrow{s} (\Delta', \Phi \vdash C_1' : \Theta', \Sigma')$$

  where we define:

  $$C_1' \equiv \nu(\vec{n}_1 : \vec{T}_1, \vec{p} : \vec{U})\,.\,(C_1''' \parallel p[O'] \parallel n\langle \mathsf{let}\ x : T = e'\ \mathsf{in}\ t_1\rangle)$$

  and then verify that:

  $$\nu(\Delta', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma)\,.\,(C_1' \between C_2'') \equiv C'$$

  as required.

- **Case** $(p \notin \mathsf{dom}\,(C_1''), n \in \mathsf{dom}\,(C_2''))$.

  We must have that:

  $$C_2'' \equiv \nu(\vec{p} : \vec{U})\,.\,(C_2''' \parallel p[O] \parallel n\langle \mathsf{let}\ y : U = \mathsf{block}\ \mathsf{in}\ t_2\rangle)$$

  Moreover, since C_1 is write-closed we must have that the axiom is:

  $$p[O] \parallel n\langle \mathsf{let}\ x : T = p.l(\vec{v})\ \mathsf{in}\ t\rangle \xrightarrow{\tau} p[O] \parallel n\langle \mathsf{let}\ x : T = O.l(p)(\vec{v})\ \mathsf{in}\ t\rangle$$

  in which case:

  $$(\Delta, \Phi \vdash C_1 : \Theta, \Sigma) \xRightarrow{s\,\nu(\vec{n}_1' : \vec{T}_1').n\langle \mathsf{call}\, p.l(\vec{v})\rangle!} (\Delta, \Phi \vdash C_1' : \Theta', \vec{n}_1' : \vec{T}_1', \Sigma')$$

  where we define:

  $$C_1' \equiv \nu(\vec{n}_1'' : \vec{T}_1'')\,.\,(C_1'' \parallel n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t_1\rangle)$$

  and we partition $\{\vec{n}_1 : \vec{T}_1\}$ into $\{\vec{n}_1' : \vec{T}_1', \vec{n}_1'' : \vec{T}_1''\}$ such that $\{\vec{n}_1'\} \subseteq \mathsf{fn}\,(p.l(\vec{v}))$ and $\{\vec{n}_1''\} \cap \mathsf{fn}\,(p.l(\vec{v})) = \emptyset$.

  We also have:

  $$(\Delta, \Phi \vdash C_2 : \Theta, \Sigma) \xRightarrow{s\,\nu(\vec{n}_1' : \vec{T}_1').n\langle \mathsf{call}\, p.l(\vec{v})\rangle?} (\Delta, \vec{n}_1' : \vec{T}_1', \Phi \vdash C_2' : \Theta', \Sigma')$$

  where we define:

  $$C_2' \equiv \nu(\vec{p} : \vec{U})\,.\,(C_2''' \parallel p[O] \parallel n\langle \mathsf{let}\ x : T = O.l(p)(\vec{v})\ \mathsf{in}\ \mathsf{let}\ y : U = \mathsf{return}\,(x : T)\ \mathsf{in}\ t_2\rangle)$$

  and then verify that:

  $$\nu(\Delta', \vec{n}_1' : \vec{T}_1', \Theta', \Sigma' \setminus \Delta, \Theta, \Sigma)\,.\,(C_1' \between C_2') \equiv C'$$

  as required.

- **Case** $(p \notin \mathsf{dom}\,(C_1''), n \notin \mathsf{dom}\,(C_2''))$.

  Similar to the above. □

Decomposition now follows by induction on the number of reductions in $C_1 \between C_2 \Rightarrow C'$ and makes use of Lemmas A.6 and A.7.

B Proof of definability

We have to show that for any $\Delta \vdash s : \mathsf{trace}\ \Theta$ we have $(\Delta \vdash \mathsf{Comp}\ (\Delta \vdash s : \mathsf{trace}\ \Theta) : \Theta) \stackrel{r}{\Longrightarrow}$ if and only if $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\ \Theta$.

There are two parts to this proof: 'if' and 'only if', which we will detail in the following sections. First though, for technical reasons, we extend the notion of β-reduction.

B.1 Technical preliminaries

In a component $\nu(\Delta)\,.\,(p[O] \parallel C)$, the object name p is *immutable* if:

- There are no occurrences of $p.l \Leftarrow M$ in O or C.
- In each method $\varsigma(n : T)\,.\,\lambda(\vec{x} : \vec{T})\,.\,\langle t \rangle$ in O, there are no occurrences of $n.l \Leftarrow M$ in t.

Note that since method update is only allowed on names and not variables we do not need to consider aliasing in this definition. We can now extend the notion of β-reduction to include method calls on immutable objects:

$$p[O] \parallel n\langle \mathsf{let}\ x : T = p.l(\vec{v})\ \mathsf{in}\ t \rangle \xrightarrow{\beta} p[O] \parallel n\langle \mathsf{let}\ x : T = O.l(p)(\vec{v})\ \mathsf{in}\ t \rangle \qquad \text{(when } p \text{ is immutable)}(\dagger)$$

The important property of β-reductions is that they are confluent with all other transitions:

Proposition B.1 *If*

$$\begin{array}{ccc} (\Delta \vdash C : \Theta) & \xrightarrow{\beta} & (\Delta \vdash C' : \Theta) \\ \alpha \downarrow & & \\ (\Delta' \vdash C'' : \Theta') & & \end{array}$$

then either $\alpha = \beta$ *and* $C' \equiv C''$ *or*

$$\begin{array}{ccc} (\Delta \vdash C : \Theta) & \xrightarrow{\beta} & (\Delta \vdash C' : \Theta) \\ \alpha \downarrow & & \alpha \downarrow \\ (\Delta' \vdash C'' : \Theta') & \xrightarrow{\beta} & (\Delta' \vdash C''' : \Theta') \end{array}$$

Proof: A case analysis of the possible reductions of C. □

Corollary B.2 *If*

$$\begin{array}{ccc} (\Delta \vdash C : \Theta) & \xrightarrow{\beta}{}^{*} & (\Delta \vdash C' : \Theta) \\ s \Downarrow & & \\ (\Delta' \vdash C'' : \Theta') & & \end{array} \qquad \textit{then} \qquad \begin{array}{ccc} (\Delta \vdash C : \Theta) & \xrightarrow{\beta}{}^{*} & (\Delta \vdash C' : \Theta) \\ s \Downarrow & & s \Downarrow \\ (\Delta' \vdash C'' : \Theta') & \xrightarrow{\beta}{}^{*} & (\Delta' \vdash C''' : \Theta') \end{array}$$

B.2 The 'if' direction

We suppose that $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\,\Theta$. We note that, due to Proposition 5.4, it suffices to show that: $(\Delta \vdash \mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta) : \Theta) \stackrel{s}{\Longrightarrow}$. We proceed by describing the different components which may be reached from $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ after performing each visible action in s. We do this by giving in Figure 22 a definition for *a component for* $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$. The intended meaning is that a component for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ has already performed the prefix r of s and is still able to perform the remaining actions in s. Note that in any component for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$, the only mutable object is ref: all other objects are immutable. This allows us to use the extended notion of β-reduction given by (†) above.

Lemma B.3 *For any* $\Delta \vdash s : \mathsf{trace}\,\Theta$ *we have* $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$ *is a component for* $\Delta \vdash \varepsilon \leq s : \mathsf{trace}\,\Theta$.

Proof: An inspection of the definition of $\mathsf{Comp}\,(\Delta \vdash s : \mathsf{trace}\,\Theta)$. □

Lemma B.4 *If* $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$ *and* $\Delta' \vdash C : \Theta'$ *is a component for* $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ *then* $(\Delta' \vdash C : \Theta') \stackrel{a}{\Longrightarrow} (\Delta'' \vdash C' : \Theta'')$ *where* C' *is a component for* $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$.

Proof: By considering the definition of $\Delta \vdash r : \mathsf{trace}\,\Theta$ we see that the following cases are exhaustive:

1. **Case** $a = \nu(\Theta''').\,n\langle \mathsf{return}\,v\rangle!$ and $C \equiv \nu(\Theta''').\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel n\langle \mathsf{let}\,y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ \mathsf{let}\,x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle]$

   We have:

   $$\begin{aligned}
   &(\Delta' \vdash C : \Theta')\\
   &\quad \stackrel{\tau}{\longrightarrow} (\Delta' \vdash \nu(\Theta''').\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel\\
   &\qquad\quad n\langle \mathsf{let}\,y : U = \mathsf{state}_r.\mathsf{out}_U()\ \mathsf{in}\ \mathsf{let}\,x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta')\\
   &\quad \stackrel{\beta}{\longrightarrow}{}^{*} (\Delta' \vdash \nu(\Theta''').\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel\\
   &\qquad\quad n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)]; \mathsf{let}\,y : U = v\ \mathsf{in}\ \mathsf{let}\,x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta')\\
   &\quad \stackrel{\tau}{\longrightarrow} (\Delta' \vdash \nu(\Theta''', \mathsf{state}_{ra} : \mathsf{State}).\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)] \parallel\\
   &\qquad\quad n\langle \mathsf{let}\,y : U = v\ \mathsf{in}\ \mathsf{let}\,x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta')\\
   &\quad \stackrel{\beta}{\longrightarrow}{}^{*} (\Delta' \vdash \nu(\Theta''', \mathsf{state}_{ra} : \mathsf{State}).\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)] \parallel\\
   &\qquad\quad n\langle \mathsf{let}\,x : T = \mathsf{return}\,(v : U)\ \mathsf{in}\ t\rangle] : \Theta')\\
   &\quad \stackrel{a}{\longrightarrow} (\Delta' \vdash \nu(\mathsf{state}_{ra} : \mathsf{State}).\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)] \parallel\\
   &\qquad\quad n\langle \mathsf{let}\,x : T = \mathsf{block}\ \mathsf{in}\ t\rangle] : \Theta', \Theta''')
   \end{aligned}$$

   which is a component for $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$ as required.

2. **Case** $a = \nu(\Theta''').\,n\langle \mathsf{call}\,p.l(\vec{v})\rangle!$ and $C \equiv \nu(\Theta''').\,C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel n\langle \mathsf{let}\,y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ t\rangle]$

A *component for* $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ (resp. for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$) is one of the form:

$$
\begin{array}{l}
\nu(\Theta(s) \setminus \Theta(q)) \,.\, \nu(\mathsf{ref} : \mathsf{Ref}) \,.\, \nu(\mathsf{state}_{r'} : \mathsf{State} \mid \Delta \vdash r' \leq r : \mathsf{trace}\,\Theta) \,.\, (\\
\quad \mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\quad \prod\{\mathsf{state}_{r'}[\mathsf{State}(\Delta \vdash r' \leq s : \mathsf{trace}\,\Theta)] \mid \Delta \vdash r' \leq r : \mathsf{trace}\,\Theta\} \parallel \\
\quad \prod\{p[l_i = \mathsf{ref.val.inCall}_{p.l_i:L_i} \mid i = 1 \ldots n] \mid p : [l_i : L_i \mid i = 1 \ldots n] \in \Theta, \Theta(s)\} \parallel \\
\quad \prod\{n\langle t_n \rangle \mid n : \mathsf{thread} \in \Theta, \Theta(s)\} \parallel \\
\quad \prod\{n\langle t_n \rangle \mid n : \mathsf{thread} \in \Delta, \Delta(s) \text{ and } n \in \mathsf{threads}\,(q)\} \\
)
\end{array}
$$

where t_n is a thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ (resp. for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$).

A *thread at n for* $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ is one of the following:

1. $\mathsf{let}\, x : T = \mathsf{ref.val.out}_T()\, \mathsf{in}\, t$
   where n is output-enabled in $\Delta \vdash r : \mathsf{trace}\,\Theta$ and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.
2. $\mathsf{let}\, x : T = \mathsf{block}\, \mathsf{in}\, t$
   where n is input-enabled in $\Delta \vdash r : \mathsf{trace}\,\Theta$ and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.

A $\mathsf{return}\,(v : T)$ *thread at n for* $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ is one of the following:

1. v
   where n is balanced in r.
2. $\mathsf{ref.val.inReturn}_T(v); t$
   where $r = r_1\, a\, r_2$, $a = \nu(\Theta') \,.\, n\langle \mathsf{call}\, p.l(\vec{v}) \rangle !$, n is balanced in r_2, and t is a thread at n for $\Delta \vdash r_1 \leq s : \mathsf{trace}\,\Theta$.
3. $\mathsf{let}\, y : U = \mathsf{return}\,(v : T)\, \mathsf{in}\, t$
   where $r = r_1\, a\, r_2$, $a = \nu(\Theta') \,.\, n\langle \mathsf{call}\, p.l(\vec{v}) \rangle ?$, n is balanced in r_2, and t is a $\mathsf{return}\,(y : U)$ thread at n for $\Delta \vdash r_1 \leq s : \mathsf{trace}\,\Theta$.

Figure 22: Definition of a component for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ and for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$

A *thread at n for* $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ is one of the following:

1. stop
2. a thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$
   where $\mathsf{proj}\,n\,(q) = \mathsf{proj}\,n\,(r)$.
3. $\mathsf{let}\,x : T = p.l(\vec{v})\,\mathsf{in}\,t$
   where $\mathsf{proj}\,n\,(q\,a) = \mathsf{proj}\,n\,(r)$, $a = \nu(\Theta')\,.\,n\langle \mathsf{call}\,p.l(\vec{v})\rangle!$, and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.
4. $\mathsf{let}\,x : T = \mathsf{return}\,(v : U)\,\mathsf{in}\,t$
   where $\mathsf{proj}\,n\,(q\,a) = \mathsf{proj}\,n\,(r)$, $a = \nu(\Theta')\,.\,n\langle \mathsf{return}\,v\rangle!$, and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.
5. $\mathsf{let}\,y : U = \mathsf{ref.val.in}\mathsf{Call}_{p.l:L}(\vec{v})\,\mathsf{in}\,\mathsf{let}\,x : T = \mathsf{return}\,(y : U)\,\mathsf{in}\,t$
   where $\mathsf{proj}\,n\,(q) = \mathsf{proj}\,n\,(r\,a)$, $a = \nu(\Delta')\,.\,n\langle \mathsf{call}\,p.l(\vec{v})\rangle?$, and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.
6. t
   where $\mathsf{proj}\,n\,(q) = \mathsf{proj}\,n\,(r\,a)$, $a = \nu(\Delta')\,.\,n\langle \mathsf{return}\,v\rangle?$, and t is a $\mathsf{return}\,(v : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ for some T.
7. $\mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)];t$
   where $\mathsf{proj}\,n\,(q) = \mathsf{proj}\,n\,(r\,a)$, and t is a thread at n for $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$.
8. t
   where $n\langle t\rangle \xrightarrow{\beta} n\langle t'\rangle$ and t' is a thread at n for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$

Figure 23: Definition of a thread for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$

We have:

$$
\begin{array}{l}
(\Delta' \vdash C : \Theta') \\
\quad \xrightarrow{\tau} (\Delta' \vdash \nu(\Theta''').\, C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\quad n\langle \mathsf{let}\ y : U = \mathsf{state}_r.\mathsf{out}_U()\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{\beta}{}^{*} (\Delta' \vdash \nu(\Theta''').\, C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\quad n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)]; \\
\qquad\qquad \mathsf{let}\ x : T = p.l(\vec{v})\ \mathsf{in}\ \mathsf{ref.val.inReturn}_T(x); \mathsf{let}\ y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{\tau} (\Delta' \vdash \nu(\Theta''', \mathsf{state}_{ra} : \mathsf{State}).\, C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)] \parallel \\
\qquad\quad n\langle \mathsf{let}\ x : T = p.l(\vec{v})\ \mathsf{in}\ \mathsf{ref.val.inReturn}_T(x); \mathsf{let}\ y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{a} (\Delta' \vdash \nu(\mathsf{state}_{ra} : \mathsf{State}).\, C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)] \parallel \\
\qquad\quad n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ \mathsf{ref.val.inReturn}_T(x); \mathsf{let}\ y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ t\rangle] : \Theta', \Theta''')
\end{array}
$$

which is a component for $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$ as required.

3. **Case** $a = \nu(\Delta''').\, n\langle \mathsf{return}\, v\rangle?$ and $C \equiv C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ \mathsf{ref.val.inReturn}_T(x); t\rangle]$

   We have:

$$
\begin{array}{l}
(\Delta' \vdash C : \Theta') \\
\quad \xrightarrow{a} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\quad n\langle \mathsf{let}\ x : T = v\ \mathsf{in}\ \mathsf{ref.val.inReturn}_T(x); t\rangle] : \Theta') \\
\quad \xrightarrow{\beta}{}^{*} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\quad n\langle \mathsf{ref.val.inReturn}_T(v); t\rangle] : \Theta') \\
\quad \xrightarrow{\tau} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\quad n\langle \mathsf{state}_r.\mathsf{inReturn}_T(v); t\rangle] : \Theta') \\
\quad \xrightarrow{\beta}{}^{*} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\quad n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)]; t\rangle] : \Theta') \\
\quad \xrightarrow{\tau} (\Delta', \Delta''' \vdash C[\nu(\mathsf{state}_{ra} : \mathsf{State}).\, \mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)] \parallel \\
\qquad\quad n\langle t\rangle] : \Theta')
\end{array}
$$

   which is a component for $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$ as required.

4. **Case** $a = \nu(\Delta''').\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle?$ and $C \equiv C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t\rangle]$

We have:

$$
\begin{array}{l}
(\Delta' \vdash C : \Theta') \\
\quad \xrightarrow{a} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\qquad n\langle \mathsf{let}\ y : U = p.l(\vec{v})\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{\beta}{}^{*} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\qquad n\langle \mathsf{let}\ y : U = \mathsf{ref.val.inCall}_{p.l:L}(\vec{v})\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{\tau} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\qquad n\langle \mathsf{let}\ y : U = \mathsf{state}_r.\mathsf{inCall}_{p.l:L}(\vec{v})\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{\beta}{}^{*} (\Delta', \Delta''' \vdash C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel \\
\qquad\qquad n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)]; \\
\qquad\qquad \mathsf{let}\ y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta') \\
\quad \xrightarrow{\tau} (\Delta', \Delta''' \vdash C[\nu()\,.\,\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)] \parallel \\
\qquad\qquad n\langle \mathsf{let}\ y : U = \mathsf{ref.val.out}_U()\ \mathsf{in}\ \mathsf{let}\ x : T = \mathsf{return}\,(y : U)\ \mathsf{in}\ t\rangle] : \Theta')
\end{array}
$$

which is a component for $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$ as required.

5. **Case** $a = \nu(\Delta''')\,.\,n\langle \mathsf{call}\,p.l(\vec{v})\rangle?$ and $C \equiv C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r]]$ where $n \notin \Theta'$.
   Similar to the previous case. □

The 'if' half of definability now follows, by induction on Lemma B.4, with Lemma B.3 as the base case.

B.3 The 'only if' direction

We suppose that $\Delta \vdash s : \mathsf{trace}\,\Theta$ and that $(\Delta \vdash \mathsf{Comp}\ (\Delta \vdash s : \mathsf{trace}\,\Theta) : \Theta) \stackrel{r}{\Longrightarrow}$ so we must demonstrate that $\Delta \vdash r \sqsubseteq s : \mathsf{trace}\,\Theta$. As above we make an auxiliary definition of *a component for* $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ in Figures 22 and 23 with the intended meaning that a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ has performed the trace q and this is $\sqsubseteq$ related to some prefix of s. Note that, as prefix ordering $\leq$ on traces is contained in $\sqsubseteq$ and $\sqsubseteq$ is transitive, then we also have $q \sqsubseteq s$ for such components. Again, in any component for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$, the only mutable object is ref: all other objects are immutable. This allows us to use the extended notion of β-reduction given by (†) above.

Lemma B.5 *For any* $\Delta \vdash s : \mathsf{trace}\,\Theta$ *we have* $\mathsf{Comp}\ (\Delta \vdash s : \mathsf{trace}\,\Theta)$ *is a component for* $\Delta \vdash \varepsilon \sqsubseteq \varepsilon \leq s : \mathsf{trace}\,\Theta$.

Proof: An inspection of the definition of $\mathsf{Comp}\ (\Delta \vdash s : \mathsf{trace}\,\Theta)$. □

Lemma B.6 *If* C *is a component for* $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ *and* $C \xrightarrow{\beta} C'$ *then* C' *is a component for* $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$.

Proof: An inspection of the definition of a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$. □

Lemma B.7 *If* C *is a component for* $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ *and* $C \xrightarrow{\tau} C'$ *then* $C' \xrightarrow{\beta}{}^{*} C''$ *where* C'' *is a component for* $\Delta \vdash q \sqsubseteq r' \leq s : \mathsf{trace}\,\Theta$.

Proof: The following cases are exhaustive:

1. **Case** $C \equiv C[n\langle \mathsf{let}\, x : T = \mathsf{ref.val.inCall}_{p.l:L}(\vec{v})\ \mathsf{in}\ t\rangle] \xrightarrow{\tau} C[n\langle \mathsf{let}\, x : T = \mathsf{state}_r.\mathsf{inCall}_{p.l:L}(\vec{v})\ \mathsf{in}\ t\rangle] \equiv C'$
where $\mathsf{proj}\, n\ (q) = \mathsf{proj}\, n\ (r\, a)$, $a = \nu(\Delta')\,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle?$, and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\ \Theta$.

   If (up to α-converting Θ') $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$ then we have:

$$C' \xrightarrow{\beta}{}^{*} C[n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)]; \mathsf{let}\, x : T = \mathsf{ref.val.out}_U()\ \mathsf{in}\ t\rangle]$$

   which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required.

   If $\Delta \vdash r\, a \not\leq s : \mathsf{trace}\ \Theta$ then we have:

$$C' \xrightarrow{\beta}{}^{*} C[n\langle \mathsf{stop}\rangle]$$

   which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required.

2. **Case** $C \equiv C[n\langle \mathsf{ref.val.inReturn}_T(v); t\rangle] \xrightarrow{\tau} C[n\langle \mathsf{state}_r.\mathsf{inReturn}_T(v); t\rangle] \equiv C'$
where $\mathsf{proj}\, n\ (q) = \mathsf{proj}\, n\ (r\, a)$, $a = \nu(\Delta')\,.\, n\langle \mathsf{return}\, v\rangle?$, and t is a thread at n for $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$.

   If (up to α-converting Θ') $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$ then we have:

$$C' \xrightarrow{\beta}{}^{*} C[n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)]; t\rangle]$$

   which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required.

   If $\Delta \vdash r\, a \not\leq s : \mathsf{trace}\ \Theta$ then we have:

$$C' \xrightarrow{\beta}{}^{*} C[n\langle \mathsf{stop}\rangle]$$

   which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required.

3. **Case** $C \equiv C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_r] \parallel n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)]; t\rangle]$
$\xrightarrow{\tau} \nu(\mathsf{state}_{ra} : \mathsf{State})\,.\, C[\mathsf{ref}[\mathsf{val} = \mathsf{state}_{ra}] \parallel \mathsf{state}_{ra}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)] \parallel n\langle t\rangle] \equiv C'$
where t is a thread at n for $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$.

   By definition, C' is a component for $\Delta \vdash q \sqsubseteq r\, a \leq s : \mathsf{trace}\ \Theta$.

4. **Case** $C \equiv C[n\langle \mathsf{let}\, x : T = \mathsf{ref.val.out}_T()\ \mathsf{in}\ t\rangle] \xrightarrow{\tau} C[n\langle \mathsf{let}\, x : T = \mathsf{state}_r.\mathsf{out}_T()\ \mathsf{in}\ t\rangle] \equiv C'$
where $\mathsf{proj}\, n\ (q) = \mathsf{proj}\, n\ (r)$, n is output-enabled in $\Delta \vdash r : \mathsf{trace}\ \Theta$ and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\ \Theta$.

   If $\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta$ and $a = \nu(\Theta')\,.\, n\langle \mathsf{call}\, p.l(\vec{v})\rangle!$ then:

$$\begin{aligned} C' \xrightarrow{\beta}{}^{*} \quad & C[n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\, a \leq s : \mathsf{trace}\ \Theta)]; \\ & \quad \mathsf{ref.val.inReturn}_U(p.l(\vec{v})); \mathsf{let}\, x : T = \mathsf{ref.val.out}_T()\ \mathsf{in}\ t\rangle] \end{aligned}$$

   which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required.

If $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$ and $a = \nu(\Theta')\,.\,n\langle \mathsf{return}\,v\rangle!$ then we must have that $r = r_1\,\nu(\Theta')\,.\,n\langle \mathsf{call}\,p.l(\vec{v})\rangle?r_2$ where n is balanced in r_2. Thus, since t is a $\mathsf{return}\,(x:T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$ we must have that:

$$t = \mathsf{let}\,y:U = \mathsf{return}\,(x:T)\,\mathsf{in}\,t'$$

where t' is a $\mathsf{return}\,(y:U)$ thread at n for $\Delta \vdash r_1 \leq s : \mathsf{trace}\,\Theta$, so t' is also a $\mathsf{return}\,(y:U)$ thread at n for $\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta$, so $\mathsf{let}\,x:T = v\,\mathsf{in}\,t$ is a thread at n for $\Delta \vdash q \sqsubseteq r\,a \leq s : \mathsf{trace}\,\Theta$. Then:

$$C' \xrightarrow{\beta}{}^{*}\, C[n\langle \mathsf{ref.val} := \mathsf{new}[\mathsf{State}(\Delta \vdash r\,a \leq s : \mathsf{trace}\,\Theta)]; \mathsf{let}\,x:T = v\,\mathsf{in}\,t\rangle]$$

which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ as required.

Otherwise:

$$C' \xrightarrow{\beta}{}^{*}\, C[n\langle \mathsf{stop}\rangle]$$

which is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ as required. □

Lemma B.8 *If $\Delta' \vdash C : \Theta'$ is a component for $\Delta \vdash q \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ and $(\Delta' \vdash C : \Theta') \xrightarrow{a} (\Delta'' \vdash C' : \Theta'')$ then $C' \xrightarrow{\beta}{}^{*}\, C''$ where C'' is a component for $\Delta \vdash q\,a \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$.*

Proof: The following cases are exhaustive:

1. **Case** $(\Delta' \vdash C : \Theta') \xrightarrow{\nu(\Delta'').n\langle \mathsf{call}\,p.l(\vec{v})\rangle?} (\Delta', \Delta'' \vdash C \parallel n\langle \mathsf{let}\,x:T = p.l(\vec{v})\,\mathsf{in}\,\mathsf{return}\,(x:T)\rangle : \Theta')$ where $n \notin \Theta'$.

   We have:

   $$C' \xrightarrow{\beta}{}^{*}\, C \parallel n\langle \mathsf{let}\,x:T = \mathsf{ref.val.inCall}_{p.l:L}(\vec{v})\,\mathsf{in}\,\mathsf{return}\,(x:T)\rangle$$

   which is a component for $\Delta \vdash q\,a \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ as required.

2. **Case** $(\Delta' \vdash C[n\langle \mathsf{let}\,x:T = \mathsf{block}\,\mathsf{in}\,t\rangle] : \Theta') \xrightarrow{\nu(\Delta'').n\langle \mathsf{call}\,p.l(\vec{v})\rangle?} (\Delta', \Delta'' \vdash C[n\langle \mathsf{let}\,y:U = p.l(\vec{v})\,\mathsf{in}\,\mathsf{let}\,x:T = \mathsf{return}\,(y:U)\,\mathsf{in}\,t\rangle] : \Theta')$
   where $\mathsf{proj}n\,(q) = \mathsf{proj}n\,(r)$, n is input-enabled in $\Delta \vdash r : \mathsf{trace}\,\Theta$ and t is a $\mathsf{return}\,(x:T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.

   We have:

   $$C' \xrightarrow{\beta}{}^{*}\, C[n\langle \mathsf{let}\,y:U = \mathsf{ref.val.inCall}_{p.l:L}(\vec{v})\,\mathsf{in}\,\mathsf{let}\,x:T = \mathsf{return}\,(y:U)\,\mathsf{in}\,t\rangle]$$

   which is a component for $\Delta \vdash q\,a \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ as required.

3. **Case** $(\Delta' \vdash C[n\langle \mathsf{let}\,x:T = \mathsf{block}\,\mathsf{in}\,t\rangle] : \Theta') \xrightarrow{\nu(\Delta'').n\langle \mathsf{return}\,v\rangle?} (\Delta', \Delta'' \vdash C[n\langle \mathsf{let}\,x:T = v\,\mathsf{in}\,t\rangle] : \Theta')$
   where $\mathsf{proj}n\,(q) = \mathsf{proj}n\,(r)$, n is input-enabled in $\Delta \vdash r : \mathsf{trace}\,\Theta$ and t is a $\mathsf{return}\,(x:T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\,\Theta$.

   We have:

   $$C' \xrightarrow{\beta}{}^{*}\, C[n\langle t[v/x]\rangle]$$

   which is a component for $\Delta \vdash q\,a \sqsubseteq r \leq s : \mathsf{trace}\,\Theta$ as required.

4. **Case** $(\Delta' \vdash \nu(\Theta''').\, C[n\langle \mathsf{let}\ x : T = p.l(\vec{v})\ \mathsf{in}\ t\rangle] : \Theta') \xrightarrow{\nu(\Theta''').n\langle \mathsf{call}\, p.l(\vec{v})\rangle!} (\Delta' \vdash C[n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t\rangle] : \Theta', \Theta''')$
   where $\mathsf{proj} n\ (q\, a) = \mathsf{proj} n\ (r)$, and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\ \Theta$.

   We have C' is a component for $\Delta \vdash q\, a \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required.

5. **Case** $(\Delta' \vdash \nu(\Theta''').\, C[n\langle \mathsf{let}\ x : T = \mathsf{return}\,(v : U)\ \mathsf{in}\ t\rangle] : \Theta') \xrightarrow{\nu(\Theta''').n\langle \mathsf{return}\, v\rangle!} (\Delta' \vdash C[n\langle \mathsf{let}\ x : T = \mathsf{block}\ \mathsf{in}\ t\rangle] : \Theta', \Theta''')$
   where $\mathsf{proj} n\ (q\, a) = \mathsf{proj} n\ (r)$, and t is a $\mathsf{return}\,(x : T)$ thread at n for $\Delta \vdash r \leq s : \mathsf{trace}\ \Theta$.

   We have C' is a component for $\Delta \vdash q\, a \sqsubseteq r \leq s : \mathsf{trace}\ \Theta$ as required. □

The 'only if' half of definability now follows, by induction on Lemmas B.6, B.7, and B.8, with Lemma B.5 as the base case, making appropriate use of Corollary B.2.

References

[1] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Declarative Programming*. Addison-Wesley, 1989.

[2] S. Abramsky. Domain theory in logical form. *Ann. Pure Appl. Logic*, 51:1–77, 1991.

[3] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[4] M. Boreale, R. de Nicola, and R. Pugliese. Trace and testing equivalences on asynchronous processes. *Information and Computation*, 172:139–164, 2002.

[5] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. Assoc. Comput. Mach.*, 31(3):560–599, 1984.

[6] L. Cardelli and M. Abadi. *A Theory of Objects*. Springer-Verlag, 1996.

[7] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *Proc. CONCUR*, pages 655–670, 1996.

[8] W. Ferreira, M. Hennessy, and A. S. A. Jeffrey. A theory of weak bisimulation for core CML. *J. Functional Programming*, 8(5):447–491, 1998.

[9] M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus. In *Proc. IEEE Conf. Logic in Computer Science*, pages 43–54. IEEE Press, 1996.

[10] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In *Proc. Int. Workshop on High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[11] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proc. ACM Symp. Principles of Programming Languages*, pages 386–395. ACM Press, 1996.

[12] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[13] M. Hennessy. A fully abstract denotational semantics for the π-calculus. *Theoretical Computer Science*, 278(1):53–89, 2002.

[14] M. Hennessy and J. Rathke. Typed behavioural equivalences for processes in the presence of subtyping. In *Proc. Computing: Australasian Theory Symposium*, volume 61 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[15] K. Honda and M. Tokoro. On asynchronous communication semantics. In *Proc. ECOOP Workshop on Object-Based Concurrent Computing*, volume 612 of *Lecture Notes in Computer Science*, pages 21–51. Springer-Verlag, 1992.

[16] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163:285–408, 2000.

[17] A. S. A. Jeffrey and J. Rathke. Towards a theory of weak bisimulation for local names. In *Proc. IEEE Logic in Computer Science*, pages 56–66. IEEE Press, 1999.

[18] A. S. A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of Concurrent ML with local names. In *Proc. IEEE Logic in Computer Science*, pages 311–321. IEEE Press, 2000.

[19] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. ACM Symp. Principles of Distributed Computing*, pages 137–151. ACM Press, 1987.

[20] R. Milner. Fully abstract semantics of typed λ-calculi. *Theoret. Comput. Sci.*, 4:1–22, 1977.

[21] R. Milner. *Communicating and Mobile Systems*. Cambridge University Press, 1999.

[22] R. Milner, J. Parrow, and D. Walker. A calculus of mobile proceses. *Inform. and Comput.*, 100(1):1–77, 1992.

[23] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. Int. Colloq. Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

[24] J.-H. Morris. Lambda calculus models of programming languages. Dissertation, M.I.T., 1968.

[25] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. Int. Symp. Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, 1993.

[26] G. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5:223–256, 1977.

[27] G. B. J. Rumbaugh and I. Jacobson. *The Unifi ed Modeling Language: User Guide*. Addison Wesley, 1999.

[28] D. S. Scott. Domains for denotational semantics. In M. Neilsen and E. M. Schmidt, editors, *Proc. ICALP 82*, pages 577–613. Springer-Verlag, 1982. LNCS 140.

[29] P. Selinger. First-order axioms for asynchrony. In *Proc. CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 376–390. Springer-Verlag, 1997.