A Foundation for Actor Computation

Gul Agha
University of Illinois
agha@cs.uiuc.edu

Ian A. Mason
Stanford University
iam@cs.stanford.edu

Scott Smith
Johns Hopkins University
scott@cs.jhu.edu

Carolyn Talcott
Stanford University
clt@sail.stanford.edu

Abstract

We present an actor language which is an extension of a simple functional language, and provide a precise operational semantics for this extension. Actor configurations are open distributed systems, meaning we explicitly take into account the interface with external components in the specification of an actor system. We define and study various notions of equivalence on actor expressions and configurations.

1. Introduction

The modern computing environment is becoming increasingly distributed. The semantics of components of open distributed systems is in its infancy. The main features of an open distributed system are that new components can be added, existing components can be replaced, and interconnections can be changed, largely without disturbing the functioning of the system. Components have no control over the components with which they might be connected. The behavior of a component is locally determined by its initial state and the history of its interactions with the environment through its interface. The internal state of a component must only be accessible through operations provided by the interface.

The actor model of computation has a built-in notion of local component and interface, and thus it is a natural model to use as a basis for a theory of open distributed computation. The actor model was originally proposed by Hewitt [13]. Actors are self-contained, concurrently interacting entities of a computing system. They communicate via message passing which is asynchronous and fair. Actors can be dynamically created and the topology of actor systems can change dynamically. The actor model is a primitive model of computation, but nonetheless easily expresses a wide range of computation paradigms. It directly supports encapsulation and sharing, and provides a natural extension of both functional programming and object style data abstraction to concurrent open systems [1, 2].

1.1. Overview

In this paper we present a study of a particular actor language. Our actor language is an extension of the call-by-value lambda calculus that includes (in addition to arithmetic primitives and structure constructors, recognizers, and destructors) primitives for creating and manipulating actors. Our approach is motivated by a desire to bridge the gap between theory and practice. The semantic theory we develop is intended to be useful for justifying program transformations for real languages, and to formalize intuitive arguments and properties used by programmers.

In our model we make explicit the notion of open system component through the notion of an *actor configuration*. An actor configuration is a collection of individually named concurrently executing actors, plus two collections of actor names that explicitly define the interface to the environment. Following the tradition of [22, 23, 18, 10, 11] we develop the semantics in two stages. The first stage consists in giving an operational semantics for actor configurations. In the second stage various notions of equivalence are investigated, both of expressions and of configurations. The operational semantics extends that of the embedded functional language in such a way that the equational theory of the functional language is preserved, giving rise to a rich set of equational reasoning principles. There are also numerous equational laws that relate to actor computations, for instance allowing two adjacent message send operations to be permuted.

The operational semantics of actor configurations is defined by a transition relation on configurations. An important aspect of the actor model is the fairness requirement: message delivery is guaranteed, and individual actor computations are guaranteed to progress. We make the fairness requirement explicit in our semantics by requiring infinite sequences of transitions on actor configurations to be fair.

We study two notions of equivalence: observational and interaction. Two expressions/configurations are said to be observationally equivalent if they give rise to the same observations, suitably defined, inside all observing contexts. This notion is closely related to testing equivalence [8]. Observational equivalence provides a semantic basis for developing sound transformation rules for expressions. Although fairness makes some aspects of reasoning more complicated, it simplifies others. Many intuitively correct equations fail in the absence of fairness. We prove that in the presence of fairness, the three standard notions of observational equivalence collapse to two. Two configurations are said to be interaction equivalent if they behave the same considered as black box devices. Interaction equivalence implies observational equivalence. The converse is an open problem.

As a first step towards an algebra of operations on actor configurations, we define a composition operator on configurations. Composition is associative, commutative, has a unit, and is and interaction equivalence is a congruence with respect to composition. This allows large complex configurations to be studied in parts and composed to form the full system. Unlike most notions of modularity and composability, which are static, the notions we define are fundamentally dynamic ones that allow for the interface between components to evolve over time.

Outline

The remainder of this paper is organized as follows. In the remainder of this section we describe the actor model of computation in more detail. §2 gives the syntax and operational semantics of our actor language. In §3 we study the notion of observational equivalence for actor expressions. In §4 we state a variety of basic equational laws along with an intuitive explanation of how these laws are established. In §5 we consider actor configurations. We establish a composability result, define notions of equivalence, relate these to expression equivalence. In §6 we give some simple examples of equivalences between configurations and show how these may be established. In §7 we develop methods for establishing expression equivalence, and use these methods to prove the laws of §4. §8 summarizes the highlights of this paper, and discusses related and future work.

1.2. The Actor Model of Computation

In this subsection we discuss the principles underlying the actor model of computation, compare it to other models, and illustrate programming in our actor language. See [1] for more discussion of the actor model, and for many examples of programming using them.

1.2.1. The Actor Philosophy

The early work on Actors [12, 13, 14, 3, 4, 5, 16, 17] proposed some fundamental principles:

- Every object is an actor; this includes messages and numbers.
- No actor can be operated on, looked at, taken apart or modified in anyway except by sending a message to that actor requesting it perform the operation itself.
- The only thing that happens in an actor system is an event, which consists of an actor receiving a message.

- An actor can send many messages at the same time but can only receive them one at a time. In other words the arrival order is linear; while the structure of events is only partially ordered by the notion of one event preceding another.
- Actors have a set of acquaintances, other actors it knows about and can send messages to. This set can increase in time since the actor may create new acquaintances, and it may also hear about them in messages it receives. These are the only ways the set of acquaintances can increase.

The key features of actor based computation, that have evolved from these guiding principles are:

- Message passing is fundamental.
- Mail delivery is guaranteed, i.e. fairness.
- There is arrival-order non-determinism in message delivery.
- Actors are history sensitive shared objects.
- Resource allocation and interconnection topology are dynamic.

1.2.2. Related Models of Concurrency

We compare the actor model with three related models of concurrency: CSP [15], the π-calculus [20], and Concurrent ML [25, 6]. All of these models have synchronous communication as primitive instead of asynchronous communication. Synchronous communication can be implemented with asynchronous primitives and vice-versa, and forms of both are required in a realistic setting. A more important general distinction is that none of these models incorporate fairness. Without fairness, specifications fail to compose — a process may behave correctly in isolation, but may fail to do so in the presence of other processes (even if it is not interacting with them).

The work on process algebras and the π-calculus has focused on understanding elementary communications between processes, abstracting away other programming language issues. The CSP model assumes a fixed interconnection topology of processes, supports only static storage allocation, and disallows recursive procedures. The π-calculus is to concurrent programming as the pure λ-calculus is to functional programming in the sense that it provides a bare minimum of primitive notions: functions and function application in the case of the λ-calculus; and channels and communication in the case of the π-calculus. Like actors, channels are dynamically created and channel names can be communicated. Unlike actors, channels do not have state, and the π-calculus provides no form of object identity.

Recently there have been two efforts [24, 6] to combine the work on ML with the work on process algebras to obtain a concurrent version of ML. As in process algebras communication channels and concurrency are the fundamental primitive for concurrent versions of ML. Processes can be created dynamically. One process can communicate with another only if they happen to share a communication channel upon which they agree to communicate. In general any number of different processes can send or receive on a given channel, thus they suffer some of the same lack of integrity as π-calculus. In CML [24] this problem may be overcome with the use of ML data abstraction mechanisms.

1.2.3. Actor Primitives

Our actor language is an extension of the call-by-value lambda calculus that includes primitives for creating and manipulating actors. An actor's behavior is described by a lambda-abstraction which embodies the code to be executed when a message is received. The actor primitives are: **send**; **become**; **newadr**; and **initbeh**.

send is for sending messages; **send**(a, v) creates a new message with receiver a and contents v and puts the message into the message delivery system.

become is for changing behavior; **become**(b) creates an anonymous actor to carry out the rest of the current computation, alters the behavior of the actor executing the **become** to be b, and frees that actor to accept another message. This provides additional parallelism. The anonymous actor may send messages or create new actors in the process of completing its computation, but will never receive any messages as its address can never be known.

newadr and **initbeh** are for actor creation. **newadr**() creates a new (uninitialized) actor and returns its address. **initbeh**(a, b) initializes the behavior of a newly created actor with address a to be b. An uninitialized actor can only be initialized by the actor which created it. Without this restriction composability of actor configurations is problematic, as it would permit an external actor to initialize an internally created actor. The allocation of a new address and initialization of the actor's behavior have been separated in order to allow an actor to learn its own address upon initialization. An alternative would be to have a **letactor** construct similar the Scheme **letrec** construct.

1.2.4. Trivial Examples

A simple actor behavior b that expects its message to be an actor address, sends the message 5 to that address, and becomes the same behavior, may be expressed as follows.

$$b = \mathbf{rec}(\lambda y.\lambda x.\mathbf{seq}(\mathbf{send}(x, 5), \mathbf{become}(y)))$$

where **rec** is a definable call-by-value fixed-point combinator (cf. [18]). An equivalent expression of this behavior is:

$$b = \mathbf{rec}(\lambda y.\lambda x.\mathbf{seq}(\mathbf{become}(y), \mathbf{send}(x, 5)))$$

since the order of executing the **become** and the **send** cannot be observed. An expression that would create an actor with this behavior and send it some other actor address a is

$$e = \mathbf{let}\{x := \mathbf{newadr}()\}\mathbf{seq}(\mathbf{initbeh}(x, b), \mathbf{send}(x, a)).$$

The behavior of a sink, an actor that ignores its messages and becomes this same behavior, is defined by

$$\mathbf{sink} = \mathbf{rec}(\lambda b.\lambda m.\mathbf{become}(b)).$$

1.2.5. Actor Cells

It is easy to represent objects with local state in our language. As an example of this we describe an actor akin to a ML reference. The actor responds to two sorts of messages. The first is a **get** message, whose second component is an address to which the response (i.e the current contents of the cell) should be sent. The second message is a **set** message, the second component of this message should be the desired new contents of the cell-actor. We use abstract syntax to hide the gory details of the structure of messages. The desired behavior is:

$$
\begin{aligned}
B_{\text{cell}} = \texttt{rec}(&\lambda b.\lambda c.\lambda m.\\
&\texttt{if}(\texttt{get?}(m),\\
&\quad \texttt{seq}(\texttt{become}(b(c)), \texttt{send}(\texttt{cust}(m), c))\\
&\quad \texttt{if}(\texttt{set?}(m),\\
&\quad\quad \texttt{become}(b(\texttt{contents}(m))),\\
&\quad\quad \texttt{become}(b(c)))))
\end{aligned}
$$

So evaluating

$$\texttt{let}\{a := \texttt{newadr}()\}\texttt{seq}(\texttt{initbeh}(a, B_{\text{cell}}(0)), e) \quad \text{where}$$

$$e = \texttt{seq}(\texttt{send}(a, \texttt{<set}, 3\texttt{>}), \texttt{send}(a, \texttt{<set}, 4\texttt{>}), \texttt{send}(a, \texttt{<get}, b\texttt{>}))$$

will result in the actor b receiving a message containing either 0, 3 or 4, depending on the arrival order. Similarly accumulators, counters and gensym actors are easily constructed, and used.

1.2.6. Join Continuations

Simple forms of recursion are often amenable to concurrent execution. A typical example is tree recursion. Consider the problem of determining the product of the leaves of a tree (whose leaves are numbers). The problem can be recursively subdivided into the problem of computing the result for the two subtrees, and multiplying the results. The product is then returned.

$$
\begin{aligned}
\texttt{treeprod} = \texttt{rec}(&\lambda f.\lambda \mathit{tree}.\\
&\texttt{if}(\texttt{isnat}(\mathit{tree}),\\
&\quad \mathit{tree},\\
&\quad f(\texttt{left}(\mathit{tree})) * f(\texttt{right}(\mathit{tree}))))
\end{aligned}
$$

In the above code, a *tree* is passed to **treeprod** which tests to see if the *tree* is a number (i.e. a leaf). If so it returns the tree, otherwise it subdivides the problem into two recursive calls. **left** and **right** are functions which pick off the left and right branches of the tree. It is clear that the arguments to $*$ may be evaluated concurrently. It is also clear that if a zero is encountered then the computation can terminate. In this example we only deal with the former optimization The latter optimization, made using continuations, is treated in detail in [31, 27].

Such concurrency can be implemented by using a *join continuation* which synchronizes the evaluation of the different arguments. For example, the **treeprod** program given above can be expressed in terms of actor primitives as:

$$
\begin{aligned}
B_{\texttt{treeprod}} = \texttt{rec}(&\lambda b.\lambda \mathit{self}.\lambda m.\\
&\texttt{seq}(\texttt{become}(b),\\
&\quad \texttt{if}(\texttt{isnat}(\texttt{tree}(m)),\\
&\quad\quad \texttt{send}(\texttt{cust}(m), \texttt{tree}(m)),\\
&\quad\quad \texttt{let}\{\mathit{newcust} := \texttt{newadr}()\}\\
&\quad\quad\quad \texttt{seq}(\texttt{initbeh}(\mathit{newcust}, B_{\texttt{joincont}}\,(\texttt{cust}(m), 0, \texttt{nil}))\\
&\quad\quad\quad\quad \texttt{send}(\mathit{self}, \texttt{<left}(\texttt{tree}(m)), \mathit{newcust}\texttt{>})\\
&\quad\quad\quad\quad \texttt{send}(\mathit{self}, \texttt{<right}(\texttt{tree}(m)), \mathit{newcust}\texttt{>})))))\\
B_{\texttt{joincont}} = \texttt{rec}(&\lambda b.\lambda \mathit{cust}.\lambda \mathit{nargs}.\lambda \mathit{firstnum}.\lambda \mathit{num}\\
&\texttt{if}(\texttt{eq}(\mathit{nargs}, 0),\\
&\quad \texttt{become}(b(\mathit{cust}, 1, \mathit{num})),\\
&\quad \texttt{seq}(\texttt{become}(\texttt{sink}),\\
&\quad\quad \texttt{send}(\mathit{cust}, \mathit{firstnum} * \mathit{num}))))
\end{aligned}
$$

When an actor with the **treeprod** behavior, $B_{\texttt{treeprod}}$, receives a list represented as a tree it creates a customer, called a *join continuation*, and sends two messages to tree-product to evaluate the two halfs of the tree. The join continuation (with behavior $B_{\texttt{joincont}}$) receives two numbers representing the computation of the products of each of the two subtrees. The join continuation then proceeds to multiply the two numbers and send the result to the original requester. Because multiplication is commutative, we need not be concerned about matching the responses to the order of the parameters. If we were dealing with an operator which was not commutative, we would need to tag the message corresponding to each argument and this tag would be returned with the response from the corresponding subcomputation. The replacement behavior of the join continuation would then depend on the order in which the evaluation of arguments was completed.

An advantage of explicit join continuations is that they provide considerable flexibility – they can be used to control the evaluation order, to do partial computations, and to do dynamic error handling. For example, if the number 0 is encountered, the join continuation can immediately return a 0 – in some cases without waiting for the results of evaluating the other subtree.

The above program may not be optimal in other respects. For example **send**s may be quite expensive. Consequently it may be prudent to check that the subcomputation is worth dispatching. This observation together with the fact, proved in §4, that **send**s of

values commute leads to the possibly faster version:

$$
\begin{array}{l}
B_{\texttt{treeproduct}} = \texttt{rec}(\lambda b.\lambda self.\lambda m. \\
\quad \texttt{seq}(\texttt{become}(b), \\
\quad\quad \texttt{if}(\texttt{isnat}(\texttt{tree}(m)), \\
\quad\quad\quad \texttt{send}(\texttt{cust}(m), \texttt{tree}(m)), \\
\quad\quad\quad \texttt{let}\{newcust := \texttt{newadr}()\} \\
\quad\quad\quad\quad \texttt{seq}(\texttt{initbeh}(newcust, B_{\texttt{joincont}}\,(\texttt{cust}(m), 0, \texttt{nil})) \\
\quad\quad\quad\quad\quad \texttt{let}\{l := \texttt{left}(\texttt{tree}(m)), r := \texttt{right}(\texttt{tree}(m))\} \\
\quad\quad\quad\quad\quad\quad \texttt{if}(\texttt{or}(\texttt{isnat}(l), \texttt{isnat}(r)), \\
\quad\quad\quad\quad\quad\quad\quad \texttt{send}(\texttt{cust}(m), \texttt{treeprod}(l) * \texttt{treeprod}(r)), \\
\quad\quad\quad\quad\quad\quad\quad \texttt{seq}(\texttt{send}(self, \texttt{<}l, newcust\texttt{>}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{send}(self, \texttt{<}r, newcust\texttt{>})))))))
\end{array}
$$

1.3. Notation.

We use the usual notation for set membership and function application. Let Y, Y_0, Y_1 be sets. We specify meta-variable conventions in the form: let y range over Y, which should be read as: the meta-variable y and decorated variants such as y', y_0, ..., range over the set Y. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length $\mathrm{Len}(\bar{y}) = n$ with ith element y_i. (Thus $[\,]$ is the empty sequence.) $u * v$ denotes the concatenation of the sequences u and v. If u is a non-empty sequence, then $\mathrm{Last}(u)$ is the last element of u. $\mathbf{P}_\omega[Y]$ is the set of finite subsets of Y. $\mathbf{M}_\omega[Y]$ is the set of (finite) multi-sets with elements in Y. $Y_0 \xrightarrow{\mathrm{f}} Y_1$ is the set of finite maps from Y_0 to Y_1. $[Y_0 \to Y_1]$ is the set of total functions, f, with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f: $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$; and $f\rfloor Y$ is the restriction of f to the set Y.

2. A Simple Lambda Based Actor Language

In this section we give the syntax and operational semantics of our actor language.

2.1. Syntax

We take as given countable sets $\mathbb{X}$ (variables) and $\mathbb{A}\mathrm{t}$ (atoms). $\mathbb{F}_n$ is the set of primitive operations of arity n and $\mathbb{F} = \bigcup_{n \in \mathbb{N}} \mathbb{F}_n$. We assume $\mathbb{A}\mathrm{t}$ contains $\texttt{t}, \texttt{nil}$ for booleans, as well as constants for natural numbers, $\mathbb{N}$. $\mathbb{F}$ contains arithmetic operations, recognizers `isatom` for atoms, and `isnat` for numbers (arities 1, 1), branching `br` (arity 3), pairing $\texttt{ispr}, \texttt{pr}, \texttt{1}^{\texttt{st}}, \texttt{2}^{\texttt{nd}}$ (arities 1, 2, 1, 1), and actor primitives `newadr`, `initbeh`, `send`, and `become` (arities 0, 2, 2, 1). The sets of expressions, $\mathbb{E}$, value expressions (or just values), $\mathbb{V}$, and contexts (expressions with holes), $\mathbb{C}$, are defined inductively as follows.

Definition ($\mathbb{E}$ $\mathbb{V}$ $\mathbb{C}$):

$$\mathbb{V} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup \mathtt{pr}(\mathbb{V}, \mathbb{V})$$

$$\mathbb{E} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup \mathtt{app}(\mathbb{E}, \mathbb{E}) \cup \mathbb{F}_n(\mathbb{E}^n)$$

$$\mathbb{C} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{C} \cup \mathtt{app}(\mathbb{C}, \mathbb{C}) \cup \mathbb{F}_n(\mathbb{C}^n) \cup \{\bullet\}$$

We let x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, e range over $\mathbb{E}$, and C range over $\mathbb{C}$. Since we are working with a syntactic reduction semantics, there is no distinction between a value expression and the value it denotes. Hence we use the terms *value* and *value expression* interchangeably. $\lambda x.e$ binds the variable x in the expression e. We write $\mathrm{FV}(e)$ for the set of free variables of e. We write $e[x := e']$ to denote the expression obtained from e by replacing all free occurrences of x by e', avoiding the capture of free variables in e'. Contexts are expressions with holes. We use $\bullet$ to denote a hole. $C[e]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process. **let**, **if**, and **seq** are the usual syntactic sugar, **seq** being a sequencing primitive.

$\mathtt{let}\{x := e_0\}e_1$ abbreviates $\mathtt{app}(\lambda x.e_1, e_0)$

$\mathtt{seq}(e_0, e_1)$ abbreviates $\mathtt{app}(\mathtt{app}(\lambda z.\lambda x.x, e_0), e_1)$

$\mathtt{if}(e_0, e_1, e_2)$ abbreviates $\mathtt{app}(\mathtt{br}(e_0, \lambda z.e_1, \lambda z.e_2), \mathtt{nil})$ for z fresh

2.2. Reduction Semantics for Actor Configurations

We give the semantics of actor expressions by defining a transition relation on actor configurations. Actor configurations have some (but not necessarily all) of the addresses of internal actors known to the outside world. These known actors are called *receptionists* because they serve as the receiving interface with the environment. An open configuration may also know addresses of some actors in the outside world. These actors are called *external actors*. The sets of receptionists and external actors are the interface of an actor configuration to its environment. They specify what actors are visible and what actor connections must be provided for the configuration to function. Both the set of receptionists and the set of external actors may grow as the configuration evolves. In addition, an actor configuration contains an actor map and a multi-set of pending messages. An actor map is a finite map from actor addresses to actor states. Each actor state is one of

$(?_a)$ uninitialized, having been newly created by an actor named a;

(b) ready to accept a message, where b is its behavior, a lambda abstraction; or

$[e]$ busy executing e, here e represents the actor's current (local) processing state.

A message m contains the address of the actor to whom it is sent and the message contents, $\texttt{<}a \Leftarrow v\texttt{>}$. The contents v can be any value constructed from atoms and actor addresses using constructors. Lambda abstractions and constructions containing lambda abstractions are not allowed to be communicated in messages. There are three reasons for this restriction. Firstly, allowing lambda abstractions to be communicated in values violates the actor principle that only an actor can change its own behavior, because a **become** in a lambda message may change the receiving actor behavior. Secondly, if lambda abstractions are communicated to external actors, there is no reasonable way to control what actor addresses are actually exported. This has unpleasant consequences in reasoning

about equivalence, amongst other things. This restriction is not a serious limitation since the address of an actor whose behavior is the desired lambda abstraction can be passed in a message. Thirdly, if lambda abstractions can be communicated in messages then syntactic extensions to the language that involve transformations such as CPS can not be done on a per actor basis, since this would require transformation of code that might arrive in a message.

We classify actor configuration transitions as internal or external to the configuration. The internal transitions are:

`rcv` receipt of a message by an actor not currently busy computing; and

`exec` an actor executing a step of its current computation.

The internal transitions involve a single active actor, which we call the *focus actor* for the transition. `exec` transitions may be purely local (a λ-transition), or messages may be sent, or a new actor may be created, or a newly created actor may be initialized. `rcv` transitions consume a message and the focus actor becomes busy.

In addition to the internal transitions of a configuration, there are `i/o` transitions that correspond to interactions with external agents:

`in` arrival of a message to a receptionist from the outside; and

`out` passing a message out to an external actor.

2.2.1. Actor Configurations

We assume that we are given a countable set $\mathbb{A}\mathrm{d}$ of actor addresses. To simplify notation, we identify $\mathbb{A}\mathrm{d}$ with $\mathbb{X}$. This pun is useful for two reasons: it allows us to use expressions to describe actor states and message contents; and it allows us to avoid problems of choice of names for newly created actors by appealing to an extended form of alpha conversion. (See [18, 10] for use of this pun to represent reference cells.)

Definition ($\mathrm{c}\mathbb{V}$ $\mathbb{A}\mathrm{s}$ $\mathbb{M}$): The set of communicable values, $\mathrm{c}\mathbb{V}$, the set of actor states, $\mathbb{A}\mathrm{s}$, and the set of messages, $\mathbb{M}$, are defined as follows.

$$\mathrm{c}\mathbb{V} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \mathbf{pr}(\mathrm{c}\mathbb{V}, \mathrm{c}\mathbb{V}) \qquad \mathbb{A}\mathrm{s} = (?_{\mathbb{X}}) \cup (\mathbb{V}) \cup [\mathbb{E}] \qquad \mathbb{M} = \langle \mathbb{V} \Leftarrow \mathbb{V} \rangle$$

We let cv range over $\mathrm{c}\mathbb{V}$ and m range over $\mathbb{M}$.

Definition (Actor Configurations): An actor configuration with actor map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written

$$\left\langle\!\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$$

where $\rho, \chi \in \mathbf{P}_{\omega}[\mathbb{X}]$, $\alpha \in \mathbb{X} \xrightarrow{\mathrm{f}} \mathbb{A}\mathrm{s}$, and $\mu \in \mathbf{M}_{\omega}[\mathbb{M}]$. Further, it is required that, letting $A = \mathrm{Dom}(\alpha)$, the following constraints are satisfied:

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$ (receptionists are internal and external actors are external),

(1) if $\alpha(a) = (?_{a'})$, then $a' \in A$ (uninitialized actors created are locally),

(2) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, and if $\langle v_0 \Leftarrow v_1 \rangle \in \mu$, then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$ (χ set is complete).

We let $\mathbb{K}$ denote the set of actor configurations and let κ range over $\mathbb{K}$. The *receptionists* ρ are names of actors within the configuration that external components may freely interact with; all other actors in the (actor) configuration are local and thus inaccessible from the outside. The *external* actors χ are names of actors that are outside this configuration but to which messages may be directed. A configuration in which both the receptionist and external actor sets are empty is said to be *closed*. For closed configurations we may omit explicit mention of the empty ρ and χ sets. The actor map portion of a configuration is presented as a list of actor states each subscripted by the actor address which is mapped to this state. If $\alpha'(a) = (b)$, and α is α' with a omitted from its domain, we write α' as $(\alpha, (b)_a)$ to focus attention on a. Similarly for other states subscripted with addresses.

The set of possible computations of an actor configuration is defined in terms of the labelled transition relation $\mapsto$ on configurations. Although this is, on the surface an interleaving semantics, it is easy to modify our transition system to obtain a truly concurrent semantics either by forming a labelled transition system with independence [26], or by using concurrent rewriting [19].

2.2.2. Decomposition and Reduction

To describe the internal transitions other than message receipt, a non-value expression is decomposed uniquely into a reduction context filled with a redex. Reduction contexts are expressions with a unique hole that play the role of continuations in abstract machine models of sequential computation. In order to distinguish holes used for different purposes, we use the sign $\Box$ for the hole occurring in a reduction context, and call such holes redex holes (although they may in fact be filled with non redex expressions). We have defined the decomposition to correspond to a left-most, outer-most, call-by-value evaluation order, thus preserving the semantics of the embedded functional language.

Definition ($\mathbb{E}_{\mathrm{rdx}}$ $\mathbb{R}$): The set of redexes, $\mathbb{E}_{\mathrm{rdx}}$, and the set of reduction contexts, $\mathbb{R}$, are defined by

$$\mathbb{E}_{\mathrm{rdx}} = \mathbf{app}(\mathbb{V}, \mathbb{V}) \cup (\mathbb{F}_n(\mathbb{V}^n) - \mathbf{pr}(\mathbb{V}, \mathbb{V}))$$

$$\mathbb{R} = \{\Box\} \cup \mathbf{app}(\mathbb{R}, \mathbb{E}) \cup \mathbf{app}(\mathbb{V}, \mathbb{R}) \cup \mathbb{F}_{n+m+1}(\mathbb{V}^n, \mathbb{R}, \mathbb{E}^m)$$

We let R range over $\mathbb{R}$.

An expression e is either a value or it can be decomposed uniquely into a reduction context filled with a redex. Thus, local actor computation is deterministic.

Lemma (Unique decomposition):

(0) $e \in \mathbb{V}$, or

(1) $(\exists! R, r)(e = R[r])$

Proof : An easy induction on the structure of e.$\Box$

Redexes can be split into two classes, purely functional and actor redexes. The actor redexes are: $\mathtt{newadr}()$, $\mathtt{initbeh}(v_0, v_1)$, $\mathtt{become}(v)$, and $\mathtt{send}(v_0, v_1)$. Reduction rules for the purely functional case are given by a relation $\overset{\lambda}{\mapsto}$ on *expressions*.

Definition ($\overset{\lambda}{\mapsto}$):

(beta-v) $R[\mathtt{app}(\lambda x.e, v)] \overset{\lambda}{\mapsto} R[e[x := v]]$

(delta) $R[\delta(v_1, \ldots, v_n)] \overset{\lambda}{\mapsto} R[v']$

where δ is an n-ary algebraic operation, $v_1, \ldots, v_n \in \mathbb{A}\mathrm{t}^n$, and $\delta(v_1, \ldots, v_n) = v'$.

(br) $R[\mathtt{br}(v_1, v_2, v)] \overset{\lambda}{\mapsto} \begin{cases} R[v_1] & \text{if } v \in \mathbb{V} - \{\mathtt{nil}\} \\ R[v_2] & \text{if } v = \mathtt{nil} \end{cases}$

(ispr) $R[\mathtt{ispr}(v)] \overset{\lambda}{\mapsto} \begin{cases} R[\mathtt{t}] & \text{if } v \in \mathtt{pr}(\mathbb{V}, \mathbb{V}) \\ R[\mathtt{nil}] & \text{if } v \in \mathbb{V} - \mathtt{pr}(\mathbb{V}, \mathbb{V}) \end{cases}$

(fst) $R[\mathtt{1^{st}}(\mathtt{pr}(v_0, v_1))] \overset{\lambda}{\mapsto} R[v_0]$

(snd) $R[\mathtt{2^{nd}}(\mathtt{pr}(v_0, v_1))] \overset{\lambda}{\mapsto} R[v_1]$

(eq) $R[\mathtt{eq}(v_0, v_1)] \overset{\lambda}{\mapsto} \begin{cases} R[\mathtt{t}] & \text{if } v_0 = v_1 \in \mathbb{A}\mathrm{t} \\ R[\mathtt{nil}] & \text{if } v_0, v_1 \in \mathbb{A}\mathrm{t} \text{ and } v_0 \neq v_1 \end{cases}$

The single-step transition relation $\mapsto$ on actor configurations is generated by the following rules. Each rule is given a label l consisting of a tag indicating the primitive instruction, and additional parameters. In all cases other than `i/o` transitions (with tags `in`, `out`) the first parameter names the *focus* actor of the transition.

Definition ($\mapsto$):

<fun : a> $\quad e \overset{\lambda}{\mapsto} e' \Rightarrow \left\langle\!\!\left\langle \alpha, [e]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [e']_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

<new : a, a'> $\quad \left\langle\!\!\left\langle \alpha, [R[\mathtt{newadr}()]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[a']]_a, (?_a)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$ $\quad a'$ fresh

<init : a, a'> $\quad \left\langle\!\!\left\langle \alpha, [R[\mathtt{initbeh}(a', v)]]_a, (?_a)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\mathtt{nil}]]_a, (v)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

<bec : a, a'> $\quad \left\langle\!\!\left\langle \alpha, [R[\mathtt{become}(v)]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\mathtt{nil}]]_{a'}, (v)_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$ $\quad a'$ fresh

<send : a, m> $\quad \left\langle\!\!\left\langle \alpha, [R[\mathtt{send}(v_0, v_1)]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\mathtt{nil}]]_a \mid \mu, m \right\rangle\!\!\right\rangle^{\rho}_{\chi}$ $\quad m = \langle v_0 \Leftarrow v_1 \rangle$

<rcv : a, cv> $\quad \left\langle\!\!\left\langle \alpha, (v)_a \mid \langle a \Leftarrow cv \rangle, \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [\mathtt{app}(v, cv)]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$

<out : m> $\quad \left\langle\!\!\left\langle \alpha \mid \mu, m \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle^{\rho'}_{\chi}$

if $m = \langle a \Leftarrow cv \rangle$, $a \in \chi$, and $\rho' = \rho \cup (\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha))$

<in : m> $\quad \left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha \mid \mu, m \right\rangle\!\!\right\rangle^{\rho}_{\chi \cup (\mathrm{FV}(cv) - \mathrm{Dom}(\alpha))}$

if $m = \langle a \Leftarrow cv \rangle$, $a \in \rho$ and $\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha) \subseteq \rho$

We write $\kappa_0 \xrightarrow{l} \kappa_1$ if $\kappa_0 \mapsto \kappa_1$ according to the rule labelled by l. We call this triple a labelled transition. We say l is *enabled in configuration* κ if there is some κ' such that

$\kappa \xrightarrow{l} \kappa'$. The transitions are labelled to allow us to reason about sequences of transitions in terms of the rules applied, and to allow for alternative representation of computations, including: sequences of configurations; sequences of labeled transitions; and sequences of labels. Note that we have chosen the labels to include sufficient information that κ' is uniquely determined by κ and l.

`i/o` transitions are transitions with tags `in` or `out` and `rcv` transitions are transitions with tag `rcv`. The remaining transitions are called `exec` transitions. The `exec` transitions correspond to the execution of functional or actor redexes.

Subtilties We allow ill-formed messages to be created, but such messages can never be delivered. The last three rules assure this by restricting the form of the message: the target must be an actor and the contents must be a communicable value. In the case of input, the actor is further restricted to be a receptionist. We could easily prevent the formation of ill-formed messages, and actor states that have non-behaviors.

A clone produced to carry on after a become is not allowed to initialize an actor created by its cloner. This is a technical simplification, with some additional bookkeeping we could keep track of cloners and allow clones to initialize. Alternatively, this technical detail would disappear if we used the `letactor` construct for actor creation.

These choices affect the details of expression equivalence, but not the basic properties. Such choices will become more important if we want to model an implemented language and consider matters such as signaling of exceptions.

2.2.3. Computation Sequences and Paths

Definition (Computation trees): If κ is a configuration, then we define the *computation tree* for κ, $\mathcal{T}(\kappa)$, to be the set of all finite sequences of labeled transitions of the form $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < n]$ for some $n \in \mathbb{N}$. We call such sequences *computation sequences* and let ν range over them.

Lemma (Anonymity): If $\kappa \xrightarrow{\texttt{<bec}\,:\,a,\,a'\texttt{>}} \kappa'$ and ν is any computation sequence in the computation tree, $\mathcal{T}(\kappa)$, then ν contains no transitions with label of the form `<send` $: a', v$`>`.

Definition (Computation paths): The sequences of a computation tree are partially ordered by the intial segment relation. A *computation path* from κ is a maximal linearly ordered set of computation sequences in the computation tree, $\mathcal{T}(\kappa)$. Note that a path can also be regarded as a (possibly infinite) sequence of labelled transitions. We use $\mathcal{T}^{\infty}(\kappa)$ to denote the set of all paths from κ, and let π range over $\mathcal{T}^{\infty}(\mathbb{K})$. When thinking a path as a possibly infinite sequence we write $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ where $\bowtie \in \mathbb{N} \cup \{\omega\}$ is the length of the sequence.

Since the result of a transition is uniquely determined by the starting configuration and the transition label, computation sequences and paths can be equally represented by their initial configuration and the sequence of transitions labels. The sequence of configurations can be computed by induction on the index of occurrence.

Definition (Cfig)**:** Let κ be a configuration and let $L = [l_i \mid i < \bowtie]$ be a sequence of labels corresponding to a computation from κ. The ith configuration of the computation from κ determined by L, $\mathrm{Cfig}(\kappa, L, i)$, is defined by induction on i as follows.

(0) $\mathrm{Cfig}(\kappa, L, 0) = \kappa$

(1) $\mathrm{Cfig}(\kappa, L * [l_i], i+1) = \kappa'$ where $\mathrm{Cfig}(\kappa, L, i) \xrightarrow{l_i} \kappa'$.

Thus, the path π determined by κ, L is the sequence

$$[\mathrm{Cfig}(\kappa, L, i) \xrightarrow{l_i} \mathrm{Cfig}(\kappa, L, i+1) \mid i < \bowtie]$$

This notation has the advantage that when an initial starting configuration is fixed, either implicitly or explicitly, computation sequences in the computation tree can be identified with sequences of labels. When the sequence L is finite we let $\mathrm{Cfig}(\kappa, L)$ denote the final configuration: $\mathrm{Cfig}(\kappa, L) = \mathrm{Cfig}(\kappa, L, \mathrm{Len}(L))$.

Definition (multi-step transition): Let $L = [l_j \mid j < n]$ be a finite sequence of transition labels (possibly empty). L is a *multi-step transition* $\kappa \xrightarrow{L} \kappa'$ just if $\mathrm{Cfig}(\kappa, L) = \mathrm{Cfig}(\kappa, L, \mathrm{Len}(L)) = \kappa'$. Or in other words, we can find a $[\kappa_j \mid j \leq n]$ such that $\kappa = \kappa_0$, $\kappa' = \kappa_n$, and $[\kappa_j \xrightarrow{l_j} \kappa_{j+1} \mid j < n]$.

2.2.4. Fairness

We do not consider all paths admissible. We rule out those computations that are unfair, i.e. those in which there is some transition that should eventually happen and does not.

Definition (Fair paths): A path $\pi = [\nu_i \xrightarrow{l_i} \nu_{i+1} \mid i < \bowtie]$ in the computation tree $\mathcal{T}(\kappa)$ is fair if each enabled transition eventually happens or becomes permanently disabled. That is, if l is enabled in κ_i and is not of the form `<in : m>`, then $\kappa_j \xrightarrow{l} \kappa_{j+1}$ for some $j \geq i$, or l has the form `<rcv : a, cv>` and for some $j \geq i$ a is busy and never again becomes ready to accept a message. For a configuration κ we define $\mathcal{F}(\kappa)$ to be the subset of $\mathcal{T}^{\infty}(\kappa)$ that are fair.

Note that finite computation paths are fair, since by maximality all of the enabled transitions must have happened.

3. Equivalence of Expressions

In this section we study equivalence of expressions of our actor language. Our notion of equivalence is a combination of the now classic *operational equivalence* of [23] and *testing equivalence* of [8]. For the deterministic functional languages of the sort Plotkin studied, this equivalence is defined as follows. Two program expressions are said to be equivalent if they behave the same when placed in any observing context. An observing context is some complete program with a hole, such that all of the free variables in the expressions being observed are captured when the expressions are placed in the hole. The notion of "behave the same" is (for deterministic functional languages) typically equi-termination, i.e. either both converge or both diverge, but is usually insensitive to minor variations.

3.1. Events

The first step is to find proper notions of "observing context" and "behave the same" in an actor setting. The analogue of an observing context is an observing actor configuration: a configuration that contains an actor state with a hole. Since termination is not relevant for actor configurations, we instead introduce an observer primitive, `event` and observe whether or not in a given computation, `event` is executed. This approach is similar to that used in defining testing equivalence for CCS [8]. Since the language is nondeterministic, three different observations may be made instead of two: either `event` occurs for all possible executions, it occurs in some executions but not others, or it never occurs.

Definition (`event`): Formally, the language of observing contexts is obtained by introducing a new 0-ary primitive operator, `event`. We extend the reduction relation $\mapsto$ by adding the following rule.

$$\texttt{<e : } a\texttt{>} \quad \left\langle\!\!\left\langle\, \alpha, \texttt{[}R[\texttt{event()}]\texttt{]}_a \;\middle|\; \mu \,\right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle\, \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\middle|\; \mu \,\right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

Definition ($\mathbb{O}$): For an expression e, the observing configurations are configurations over the extended language of the form $\left\langle\!\!\left\langle\, \alpha, \texttt{[}C\texttt{]}_a \;\middle|\; \mu \,\right\rangle\!\!\right\rangle$, such that filling the hole in C with e results in a closed configuration. We use $\mathbb{O}$ to denote the set of observing configurations, and let O range over $\mathbb{O}$.

In our definition of observing configuration, the holes appear in the current state of an single executing actor. It is not hard to see that allowing holes in any actor state does not change the resulting notion of equivalence. A generalization of this fact, (**ocx**) is proved in §7.

We observe `event` transitions in the fair paths. We say that a path succeeds, **s**, if an `event` transition occurs in it, otherwise it fails, **f**. $obs(\pi)$ is the **s**/**f** observation of a single complete computation π, and $Obs(\kappa)$ is the set of observations possible for all paths of a closed actor configuration.

Definition (observations): Let κ be a configuration of the extended language, and let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ be a fair path, i.e. $\pi \in \mathcal{F}(\kappa)$. Define

$$obs(\pi) = \begin{cases} \mathbf{s} & \text{if } (\exists i < \bowtie, a)(l_i = \texttt{<e : } a\texttt{>}) \\ \mathbf{f} & \text{otherwise} \end{cases}$$

$$Obs(\kappa) = \begin{cases} \mathbf{s} & \text{if } (\forall \pi \in \mathcal{F}(\kappa))(obs(\pi) = \mathbf{s}) \\ \mathbf{sf} & \text{if } (\exists \pi \in \mathcal{F}(\kappa))(obs(\pi) = \mathbf{s}) \text{ and } (\exists \pi \in \mathcal{F}(\kappa))(obs(\pi) = \mathbf{f}) \\ \mathbf{f} & \text{if } (\forall \pi \in \mathcal{F}(\kappa))(obs(\pi) = \mathbf{f}) \end{cases}$$

3.2. Three Equivalences

The natural notion of observational equivalence is that equal observations are made in all closing configuration contexts. However, it may be possible in some cases to use a weaker equivalence. An **sf** observation may be considered as good as an **s** observation, and a new equivalence arises if these observations are equated. Similarly, an **sf** observation may be as bad as an **f** observation. We may define the following three equivalences.

Definition ($\cong_{1,2,3}$):

(1) (testing or convex or Plotkin or Egli-Milner)

$e_0 \cong_1 e_1$ iff $Obs(O[e_0]) = Obs(O[e_1])$ for all observing contexts $O \in \mathbb{O}$

(2) (must or upper or Smyth)

$e_0 \cong_2 e_1$ iff $Obs(O[e_0]) = \mathbf{s} \Leftrightarrow Obs(O[e_1]) = \mathbf{s}$ for all observing contexts $O \in \mathbb{O}$

(3) (may or lower or Hoare)

$e_0 \cong_3 e_1$ iff $Obs(O[e_0]) = \mathbf{f} \Leftrightarrow Obs(O[e_1]) = \mathbf{f}$ for all observing contexts $O \in \mathbb{O}$

By construction each of these equivalence relations is a congruence.

Theorem (congruence):

$$e_0 \cong_j e_1 \Rightarrow C[e_0] \cong_j C[e_1] \qquad \text{for } j \in \{1,2,3\}$$

3.3. Partial Collapse

Note that may-equivalence ($\cong_3$) is determined by computation trees. This is because all `events` are observed after some finite amount of time. This means the definition is independent of whether or not fairness is required. Since fairness sometimes makes proving equivalences more difficult, it is useful that may-equivalence can always be proved ignoring the fairness assumption. The other two equivalences are sensitive to choice of paths admitted as computations. In particular when fairness is required, as in our model, $\cong_2$ is in fact the same as $\cong_1$. In models without the fairness requirement, they are distinct. In either case, $\cong_3$ is distinct from $\cong_1$ and $\cong_2$.

Theorem (partial collapse):

$(1 = 2)$ $e_0 \cong_2 e_1$ iff $e_0 \cong_1 e_1$

$(1 \Rightarrow 3)$ $e_0 \cong_1 e_1$ implies $e_0 \cong_3 e_1$

$(3 \not\Rightarrow 1)$ $e_0 \cong_3 e_1$ does not imply $e_0 \cong_1 e_1$

To demonstrate $(1 = 2)$ we consider a fixed pair of expressions e_0, e_1 and categorize their closing configuration contexts O according to what observations are made by $O[e_0]$ and $O[e_1]$.

$$O \in o : o' \Leftrightarrow Obs(O[e_0]) = o \wedge Obs(O[e_1]) = o'$$

for $o, o' \in \{\mathbf{s}, \mathbf{sf}, \mathbf{f}\}$. This partitions the observing configuration contexts of e_0 and e_1 into nine sets labeled $o : o'$ for $o, o' \in \{\mathbf{s}, \mathbf{sf}, \mathbf{f}\}$. We can characterize the various possibilities for equivalence in terms of which sets must be empty as follows.

Lemma (sets):

- $e_0 \cong_1 e_1$ iff at most the sets labeled $\mathbf{s} : \mathbf{s}$, $\mathbf{sf} : \mathbf{sf}$, and $\mathbf{f} : \mathbf{f}$ are non-empty.
- $e_0 \cong_2 e_1$ iff the sets labeled $\mathbf{s} : \mathbf{sf}$, $\mathbf{sf} : \mathbf{s}$, $\mathbf{s} : \mathbf{f}$, $\mathbf{f} : \mathbf{s}$ are all empty.
- $e_0 \cong_3 e_1$ iff the sets labeled $\mathbf{s} : \mathbf{f}$, $\mathbf{f} : \mathbf{s}$, $\mathbf{sf} : \mathbf{f}$, $\mathbf{f} : \mathbf{sf}$ are all empty.

The key to collapsing $\cong_2$ into $\cong_1$ is the observation that if $Obs(O[e_0]) = \mathbf{f}$ and $Obs(O[e_1]) = \mathbf{sf}$ it is always possible to construct a O^* such that $Obs(O^*[e_0]) = \mathbf{s}$, and $Obs(O^*[e_1]) = \mathbf{sf}$.

Lemma (f.sf): For some e_0, e_1, if the set labeled $\mathbf{f} : \mathbf{sf}$ is non-empty then the set labeled $\mathbf{s} : \mathbf{sf}$ is non-empty. Symmetrically, if the set labeled $\mathbf{sf} : \mathbf{f}$ is non-empty then the set labeled $\mathbf{sf} : \mathbf{s}$ is non-empty.

Proof (f.sf): Let $O \in \mathbf{f} : \mathbf{sf}$. Form O' by replacing all occurrences of `event`() in O by `send`$(a, \mathtt{nil})$ for some fresh variable a. Let O^* be obtained by adding to O' a message $\mathtt{<}a \Leftarrow \mathtt{t>}$ and an actor a where a has the following behavior: If a receives the message `t`, it executes `event`() and becomes a sink, and if a receives the message `nil`, it just becomes a sink. Recall that a sink is an actor that ignores its message and becomes a sink. We claim $O^* \in \mathbf{s} : \mathbf{sf}$. If $O[e_0]$ never executes `event`(), then in any fair complete computation, the `t` message will be received by a, so $O^*[e_0]$ will always execute `event`(). If $O[e_1]$ executes `event`() in some computation, then in the corresponding computations for $O^*[e_1]$, sometimes `nil` will be received by a before `t` is received and sometimes it won't, hence $O^*[e_1]$ will execute `event`() in some computations, but not in all. □

Proof (partial collapse):

$1 = 2$ Assume $e_0 \cong_2 e_1$. Then by (**sets**) the sets labeled $\mathbf{s} : \mathbf{sf}$, $\mathbf{sf} : \mathbf{s}$, $\mathbf{s} : \mathbf{f}$, $\mathbf{f} : \mathbf{s}$ are all empty. By (**f.sf**) $\mathbf{f} : \mathbf{sf}$ and $\mathbf{sf} : \mathbf{f}$ must also be empty. Hence by (**sets**), $e_0 \cong_1 e_1$. $\square_{1 = 2}$

$1 \Rightarrow 3$ By (**sets**) $\square_{1 \Rightarrow 3}$

$3 \not\Rightarrow 1$ We construct expressions e_0, e_1 such that $e_0 \cong_3 e_1$, but $\neg(e_0 \cong_2 e_1)$. Let e_0 create an actor that sends a message (say `nil`) to an external actor a and becomes a sink, and let e_1 create an actor that may or may not send a message `nil` to a depending on a coin flip (there are numerous methods of constructing coin flipping actors), and also then becomes a sink. Let O be an observing configuration context that with an actor a that executes `event` just if `nil` is received. Then $Obs(O[e_0]) = \mathbf{s}$ but $Obs(O[e_1]) = \mathbf{sf}$, so $\neg(e_0 \cong_2 e_1)$. To show that $e_0 \cong_3 e_1$, show for arbitrary O that some path in the computation of $O[e_0]$ contains an event iff some path in the computation of $O[e_1]$ contains an event. This is easy, because when e_1's coin flip indicates `nil` is sent, the computation proceeds identically to e_0's computation. $\square_{3 \not\Rightarrow 1}$

□

Hereafter, $\cong$ (observational equivalence) will be used as shorthand for either $\cong_1$ or $\cong_2$.

The fairness requirement is critical in the proof of $(1 = 2)$. For example in CCS, where fairness is not assumed, no such collapse of $\cong_2$ to $\cong_1$ occurs. So, although fairness complicates some aspects of the theory, it simplifies others. If we omitted the fairness requirement we could make more $\cong$-distinctions between actors. For example, let a_0 be a sink. Let a_1 be an actor that also ignores its messages and becomes the same behavior, but it continues executing an infinite loop. The infinite looping actor could prevent the rest of the configuration's computation from progressing. In the presence of fairness no such starvation can occur, so the two are equivalent. Thus fairness allows modular reasoning about liveness properties: one can reason about the behavior of individual actors (and configurations) without worrying about whether composition with another would cause that actor to starve out.

4. Laws of Expression Equivalence

With a notion of equivalence on actor expressions defined, a library of useful equivalences can be established. The first part of this section contains a collection of purely functional laws that continue to hold in the actor setting. The second part contains a collection of laws for manipulating expressions that involve actor primitives. These laws are proved in §7.

4.1. Functional Laws

Since our reduction rules preserve the evaluation semantics of the embedded functional language, many of the equational laws for this language (cf. [29]) continue to hold in the full actor language. A first simple observation is two communicable values are observationally equivalent iff they are the same value expression.

Lemma (cv):

$$cv_0 \cong cv_1 \;\Leftrightarrow\; cv_0 = cv_1$$

Proof : The if direction is trivial. The only-if direction, is proved by exhibiting an observing context that distinguishes expressions that are not equal. Clearly both must be atoms, or variables, or pairs, otherwise they can be distinguished using `eq` and `ispr`. For example,

$$O = \langle\!\langle\ [\mathtt{if}(\mathtt{eq}(cv_0, \bullet), \mathtt{event}, \mathtt{nil})]_a \mid \rangle\!\rangle$$

distinguishes the atom cv_0 from any non-atom (and any other atom). Similarly,

$$O = \langle\!\langle\ [\mathtt{let}\{x := 0\}\mathtt{let}\{y := 1\}\mathtt{if}(\mathtt{eq}(x, \bullet), \mathtt{event}, \mathtt{nil})]_a \mid \rangle\!\rangle$$

distinguishes the variables x, y. Similarly, if both are pairs, we can construct contexts to distinguish differences in the components. □

Beta-v conversion and other laws of the lambda-c calculus [21], and the laws for conditional and pairing continue to hold in the actor setting.

Theorem (functional laws):

(beta-v) $\mathtt{app}(\lambda x.e, v) \cong e\{x := v\}$

(ift) $\mathtt{if}(v, e_1, e_2) \cong e_1$ if $v \in (\mathbb{A}\mathrm{t} - \{\mathtt{nil}\}) \cup \mathbb{L} \cup \mathtt{pr}(\mathbb{V}, \mathbb{V})$

(ifn) $\mathtt{if}(\mathtt{nil}, e_1, e_2) \cong e_2$

(ifelim) $\mathtt{if}(v, e, e) \cong e$

(iflam) $\lambda x.\mathtt{if}(v, e_1, e_2) \cong \mathtt{if}(v, \lambda x.e_1, \lambda x.e_2)$ $\quad x \notin \mathrm{FV}(v)$

(isprt) $\mathtt{ispr}(\mathtt{pr}(v_0, v_1)) \cong \mathtt{t}$,

(isprn) $\mathtt{ispr}(v) \cong \mathtt{nil}$ $\quad v \in \mathbb{A}\mathrm{t} \cup \mathbb{L}$

(fst) $\mathtt{1^{st}}(\mathtt{pr}(v_0, v_1)) \cong v_0$

(snd) $\mathtt{2^{nd}}(\mathtt{pr}(v_0, v_1)) \cong v_1$

Each of these laws (except for (**iflam**)) is a consequence of the following operational law.

Theorem (red-exp):

$e_0 \stackrel{\lambda}{\mapsto} e_1 \Rightarrow e_0 \cong e_1$

The theorem (**rcx**) is a special case of a theorem proved in [28].

Theorem (rcx): If R is a reduction context, then

(letx) $\texttt{let}\{x := e\}R[x] \cong R[e]$

(let.dist) $R[\texttt{let}\{x := e\}e_0] \cong \texttt{let}\{x := e\}R[e_0]$

(if.dist) $R[\texttt{if}(e, e_1, e_2)] \cong \texttt{if}(e, R[e_1], R[e_2])$

In fact these laws can be derived from (**beta-v**), the `if` laws and the following special instances.

(app) $e_0(e_1) \cong (\lambda f.f(e_1))(e_0)$

(cmps) $f(g(e)) \cong (\lambda x.f(g(x)))(e)$

(id) $\texttt{app}(\lambda x.x, e) \cong e$

Some useful corollaries of (**rcx**) are the following.

Corollary (uni-rcx):

(let.arg) $v(\texttt{let}\{x := e_0\}e_1) \cong \texttt{let}\{x := e_0\}v(e_1)$

(if.if) $\texttt{if}(\texttt{if}(e_0, e_1, e_2), e_a, e_b) \cong \texttt{if}(e_0, \texttt{if}(e_1, e_a, e_b), \texttt{if}(e_2, e_a, e_b))$

The above laws are really about equivalence of reduction contexts. They are instances of the following operational law.

Theorem (red-rcx):

$R_0[x] \stackrel{\lambda}{\mapsto} e' \wedge R_1[x] \stackrel{\lambda}{\mapsto} e' \Rightarrow R_0[e] \cong R_1[e]$

where x is a fresh variable.

We also note that any expressions that hang (reduce in a finite number of lambda steps to a stuck state) or have infinite lambda computations are observationally equivalent. If the reductions could involve non-lambda steps the result clearly fails since they could have different effects such as message sends that other actors in the configuration could observe. We let $\texttt{stuck} \in \mathit{Hang}$ be a prototypical stuck expression, for example $\texttt{app}(0, 0)$, and let $\texttt{bot} \in \mathit{Infin}$ be a prototypical expression with infinite computation, for example $\texttt{app}(\lambda x.\texttt{app}(x, x), \lambda x.\texttt{app}(x, x))$.

To make these ideas more precise we define *Hang* and *Infin* as follows.

Definition (*Hang*): Let *Hang* be the set of non-value expressions such that (every closed instance) lambda reduces (i.e. $\stackrel{\lambda}{\mapsto}$ in possibly 0 steps) to a stuck state – an expression e' that decomposes as $R[r]$ where r is a functional redex (i.e any non-actor redex) that does not reduce.

Definition (*Infin*): Let *Infin* the the set of (non-value) expressions e such that (every closed instance) has an infinite lambda reduction sequence. Thus $e \in \mathit{Infin}$ just if we can find e_j for $j \in \mathbb{N}$ such that $e_0 = e$ and $e_j \stackrel{\lambda}{\mapsto} e_{j+1}$.

Theorem (hang-infin): If $e_0, e_1 \in \mathit{Hang} \cup \mathit{Infin}$, then $e_0 \cong e_1$.

4.2. Basic Laws for Actor Primitives

Now we consider the equational properties of the actor primitives, **send**, **become**, **newadr**, and **initbeh**. As is the case for a language with state operations, $\mathtt{seq}(e, e) \cong e$ fails to hold because the computation e could have effects such as message sends. A stronger analogy exists between the actor primitives and the reference primitives $\{\mathtt{mk}, \mathtt{get}, \mathtt{set}\}$: **newadr** is an allocation primitive analogous to mk; **initbeh** and **become** update or alter state; while the effect of **send** depends on the state in a way analogous to **get**. There are limits to this analogy, for example **send** does not return anything of interest. In fact **send**, **become**, and **initbeh** all return **nil** as values. Thus

(triv) $\quad \vartheta(\bar{x}) \cong \mathtt{seq}(\vartheta(\bar{x}), \mathtt{nil}) \qquad$ for $\vartheta \in \{\mathtt{send}, \mathtt{become}, \mathtt{initbeh}\}$

That **newadr** is an allocation primitive analogous to mk manifests itself in its properties. However, because we have not allowed clones to initialize newly spawned actors, one characteristic property of allocation fails to holds. Namely,

(non-delay) $\quad \mathtt{let}\{y := e_0\}\mathtt{let}\{x := \mathtt{newadr}()\}e_1 \ncong \mathtt{let}\{x := \mathtt{newadr}()\}\mathtt{let}\{y := e_0\}e_1$

where x is not free in e_0.

Once allocated, an actor behavior is updated by either **become** or **initbeh**. In analogy with **set** both **become** and **initbeh** satisfy certain, slightly different, cancellation laws:

(can-b) $\quad \mathtt{seq}(\mathtt{become}(v_0), \mathtt{become}(v_1)) \cong \mathtt{seq}(\mathtt{become}(v_0), \mathtt{nil}) \cong \mathtt{become}(v_0)$

(can-i) $\quad \mathtt{seq}(\mathtt{initbeh}(v, v_0), \mathtt{initbeh}(v, v_1)) \cong \mathtt{seq}(\mathtt{initbeh}(v, v_0), \mathtt{stuck})$
$\cong \mathtt{seq}(\mathtt{initbeh}(v, v_0), \mathtt{bot})$

Note the difference between these two principles. In the case of **become** the second call is equivalent to **nil**, while in the case of **initbeh** it is stuck (which is equivalent to diverging).

4.2.1. Commuting Operations

How the effects of the actor primitives interact with one another is of paramount importance. We have seen some aspects of this interaction above. We now study the interactions more systematically.

Definition (commutes): We say two operations ϑ_0 and ϑ_1 commute if

$$\mathtt{let}\{x_0 := \vartheta_0(\bar{y})\}\mathtt{let}\{x_1 := \vartheta_1(\bar{z})\}e \cong \mathtt{let}\{x_1 := \vartheta_1(\bar{z})\}\mathtt{let}\{x_0 := \vartheta_0(\bar{y})\}e$$

for all $e \in \mathbb{E}$, $x_0 \notin \bar{z}$, $x_1 \notin \bar{y}$ and x_0 distinct from x_1. Similarly we say two expresions e_0 and e_1 commute iff

$$\mathtt{let}\{x_0 := e_0\}\mathtt{let}\{x_1 := e_1\}e \cong \mathtt{let}\{x_1 := e_1\}\mathtt{let}\{x_0 := e_0\}e$$

if the obvious hygene conditions hold.

newadr commutes with every operation except **become**. For example the expressions

$$e_0 = \mathtt{let}\{y := \mathtt{newadr}()\}\mathtt{let}\{z := \mathtt{become}(b)\}\mathtt{initbeh}(y, b')$$

$$e_1 = \mathtt{let}\{z := \mathtt{become}(b)\}\mathtt{let}\{y := \mathtt{newadr}()\}\mathtt{initbeh}(y, b')$$

are not equivalent, since the first will always fail to execute the initialization and the second will always succeed. A distinguishing context is

$$\langle\!\langle\ (\lambda x.\mathtt{event}())_{a_0}, [\mathtt{seq}(\bullet, \mathtt{send}(a_0, 0))]_a \mid \emptyset\ \rangle\!\rangle$$

If we allowed clones to initialize, then **newadr** would also commute with become. On the other hand, by (**can-b**) and (**can-1**) neither **become** nor **initbeh** commute with themselves, since this amounts to equivalence of two becomes (or initializations) with different behaviors. The remaining operation **send**, like **newadr**, does commute with itself:

(com-ss) $\quad \mathtt{seq}(\mathtt{send}(v_0, v_1), \mathtt{send}(v_2, v_3)) \cong \mathtt{seq}(\mathtt{send}(v_2, v_3), \mathtt{send}(v_0, v_1))$

send also commutes with **become**

(com-sb) $\quad \mathtt{seq}(\mathtt{send}(a_0, v_0), \mathtt{become}(v_1)) \cong \mathtt{seq}(\mathtt{become}(v_1), \mathtt{send}(a_0, v_0))$

The question of whether or not two distinct operations commute is simplified by the observation, captured in (**partial**), that a computation may have observable effects even if it diverges. This is in contrast to the sequential case, where an effect of a subcomputation is only observable if the computation completes.

Lemma (partial): If ϑ is not a total operation, then ϑ does not commute with **send**, **become** or **initbeh**.

Proof (partial): If $\vartheta(\bar{y})$ diverges, then

$$\mathtt{let}\{x := \vartheta(\bar{y})\}\mathtt{let}\{x_1 := \mathtt{send}(a, v)\}e$$

will not execute the **send**, whereas

$$\mathtt{let}\{x_1 := \mathtt{send}(a, v)\}\mathtt{let}\{x := \vartheta(\bar{y})\}e$$

will execute the **send**. Consequently the two expressions are easily distinguished. Similarly with the two operations **become** and **initbeh**. □

Since **initbeh** is partial, it does not commute with either **send** or **become**. For example

$$e_0 = \mathtt{seq}(\mathtt{initbeh}(a_0, b_0), \mathtt{send}(a_1, 0))$$
$$e_1 = \mathtt{seq}(\mathtt{send}(a_1, 0), \mathtt{initbeh}(a_0, b_0))$$

are distinguished by

$$O_0 = \langle\!\langle\ (\lambda x.\mathtt{event}())_{a_1}, (?_{a_1})_{a_0}, [\bullet]_a \mid \mu\ \rangle\!\rangle$$

for $a \neq a_1$, or by

$$O_1 = \langle\!\langle\ (\lambda x.\mathtt{event}())_{a_1}, (b)_{a_0}, [\bullet]_a \mid \mu\ \rangle\!\rangle$$

Also,

$$e_0 = \texttt{seq}(\texttt{initbeh}(a_0, b_0), \texttt{become}(\lambda x.\texttt{send}(a_1, 0)))$$

$$e_1 = \texttt{seq}(\texttt{become}(\lambda x.\texttt{send}(a_1, 0)), \texttt{initbeh}(a_0, b_0))$$

are distinguished by O_0, O_1 if μ contains $\texttt{<}a \Leftarrow 0\texttt{>}$.

(**partial**) emphasizes that the valid equations for actor primitives is sensitive to the details of when we check for ill-formed redexes. For example if we restricted the `send` redex to avoid ill-formed messages (**com-ss,com-sb**) would no longer hold.

We summarize these results in the following:

Lemma (commutes):

	n	s	i	b
n	+	+	+	-
s	+	+	-	+
i	+	-	-	-
b	-	+	-	-

(n) `newadr` commutes with `send`, `newadr`, and `initbeh`, but not with `become`.

(s) `send` commutes with `send`, `become`, and `newadr`, but not with `initbeh`.

(i) `initbeh` commutes with `newadr`, but not with `send`, `become`, and `initbeh`.

(b) `become` commutes with `send`, but not with `initbeh`, `newadr`, or `become`.

Note that the remaining operations in $\mathbb{F}$ (i.e. the arithmetic operations, branching `br`, and the pairing $\texttt{ispr}, \texttt{pr}, \texttt{1}^{\texttt{st}}, \texttt{2}^{\texttt{nd}}$) are all context insensitive, and thus those that are total commute with all other operations. In the case `if` this is perhaps worth pointing out:

Lemma (commutes-if): If ϑ commutes with e_0 and e_1, then it also commutes with $\texttt{if}(z, e_0, e_1)$

Proof : This follows from (**if-lam,if-dist**). □

Using these basic principles we can prove more complex properties, the following theorem being the most obvious.

Theorem (commutes): Suppose that e_0 and e_1 are built up from $\mathbb{V}$ using only the operations, `if` and `let`. Furthermore suppose every operation occurring in e_0 commutes with every operation occurring in e_1. Then

$$\texttt{let}\{x_0 := e_0\}\texttt{let}\{x_1 := e_1\}e \cong \texttt{let}\{x_1 := e_1\}\texttt{let}\{x_0 := e_0\}e$$

provided x_j is not free in e_{1-j} for $j < 2$,

Proof (commutes): The proof is by induction on the complexity of e_0. We sketch the induction step. We may assume, without loss of generality, that e_0 decomposes into $R[\vartheta(\bar{y})]$ with R being non-trivial. Then

$\texttt{let}\{x_0 := e_0\}\texttt{let}\{x_1 := e_1\}e \cong$

$\cong \texttt{let}\{x_0 := R[\vartheta(\bar{y})]\}\texttt{let}\{x_1 := e_1\}e$

by hypothesis

$\cong \texttt{let}\{x_0 := \texttt{let}\{z := \vartheta(\bar{y})\}R[z]\}\texttt{let}\{x_1 := e_1\}e$

by (**cong**) and (**letx**)

$\cong \texttt{let}\{z := \vartheta(\bar{y})\}\texttt{let}\{x_0 := R[z]\}\texttt{let}\{x_1 := e_1\}e$

by (**let.dis**)

$\cong \texttt{let}\{z := \vartheta(\bar{y})\}\texttt{let}\{x_1 := e_1\}\texttt{let}\{x_0 := R[z]\}e$

by the induction hypothesis and (**cong**)

$\cong \texttt{let}\{x_1 := e_1\}\texttt{let}\{z := \vartheta(\bar{y})\}\texttt{let}\{x_0 := R[z]\}e$

by the induction hypothesis and (**cong**)

$\cong \texttt{let}\{x_1 := e_1\}\texttt{let}\{x_0 := \texttt{let}\{z := \vartheta(\bar{y})\}R[z]\}e$

by (**let.dis**)

$\cong \texttt{let}\{x_1 := e_1\}\texttt{let}\{x_0 := e_0\}e$

by (**cong**) and (**letx**)

□

Note that the theorem fails in the case when the expressions contain **app** and λ due to the possibility of divergence.

To prove other properties involving the actor primitives we need to consider the whole context in which the effect occurs, which is at least one other actor and in general is an actor configuration.

5. Manipulation of Actor Configurations

In this section we establish some fundamental results concerning actor configurations. We show how configurations may be composed or decomposed into larger or smaller configurations. We define two notions of equivalence between configurations: observational and interaction equivalence. We also develop some basic tools for establishing these equivalences.

For simplicity we have treated the collection of messages in a configuration as a multiset. In this section we make the identity of different messages with the same target and contents explicit, by assuming that each message in the computation is uniquely identified as being there initially, or as being generated by a particular send or input transition. We write m rather than $\texttt{<}a \Leftarrow cv\texttt{>}$ and assume that m contains this identifying information.

5.1. Composition of Actor Configurations

Actor configurations can be composed to form new actor configurations. Furthermore, the computations of the composed configuration can be obtained by merging computations of the component configurations. This composition operation is commutative, associative, and has the empty configuration as unit. This is made precise by the following definitions and lemmas.

Definition (Composable): Two configurations $\kappa_i = \left\langle\!\!\left\langle\ \alpha_i \mid \mu_i\ \right\rangle\!\!\right\rangle_{\chi_i}^{\rho_i}$, $i < 2$ are composable if $\mathrm{Dom}(\alpha_0) \cap \mathrm{Dom}(\alpha_1) = \emptyset$, $\chi_0 \cap \mathrm{Dom}(\alpha_1) \subseteq \rho_1$, and $\chi_1 \cap \mathrm{Dom}(\alpha_0) \subseteq \rho_0$.

Definition (Composition, decomposition): The composition $\kappa_0 \parallel \kappa_1$ of composable configurations $\kappa_i = \left\langle\!\!\left\langle\ \alpha_i \mid \mu_i\ \right\rangle\!\!\right\rangle_{\chi_i}^{\rho_i}$, $i < 2$ is defined by

$$\kappa_0 \parallel \kappa_1 = \left\langle\!\!\left\langle\ \alpha_0 \cup \alpha_1 \mid \mu_0 \cup \mu_1\ \right\rangle\!\!\right\rangle_{(\chi_0 \cup \chi_1) - (\rho_0 \cup \rho_1)}^{\rho_0 \cup \rho_1}$$

(κ_0, κ_1) is a decomposition of κ if κ_i, $i < 2$ are composable configurations, and $\kappa = \kappa_0 \parallel \kappa_1$.

Lemma (AC): Let $\kappa_i = \left\langle\!\!\left\langle\ \alpha_i \mid \mu_i\ \right\rangle\!\!\right\rangle_{\chi_i}^{\rho_i}$, $i < 3$ be pairwise composable configurations. And let $\kappa_\emptyset = \left\langle\!\!\left\langle\ \emptyset \mid \emptyset\ \right\rangle\!\!\right\rangle$ be the empty configuration. Then

$$\kappa_0 \parallel \kappa_1 = \kappa_1 \parallel \kappa_0$$
$$\kappa_0 \parallel \kappa_\emptyset = \kappa_0$$
$$(\kappa_0 \parallel \kappa_1) \parallel \kappa_2 = \kappa_0 \parallel (\kappa_1 \parallel \kappa_2)$$

Proof : Using the AC properties of set union, the only thing to check is the equality of external actors for the two associations. It is easy to see that

$$\begin{aligned}
&(((\chi_0 \cup \chi_1) - (\rho_0 \cup \rho_1)) \cup \chi_2) - (\rho_0 \cup \rho_1 \cup \rho_2) \\
&\quad = (\chi_0 \cup \chi_1 \cup \chi_2) - (\rho_0 \cup \rho_1 \cup \rho_2) \\
&\quad = (\chi_0 \cup ((\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2))) - (\rho_0 \cup \rho_1 \cup \rho_2)
\end{aligned}$$

□

5.1.1. Projection and Merging of Sequences

We define projection and merging operations to relate computations of composite configurations with those of their components. In the remainder of this subsection we relax our notation convention slightly. We say that π *is computation for* κ if π is a computation sequence or path with first element κ. Let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ be a computation for κ and let κ be decomposed as $\kappa^0 \parallel \kappa^1$. Then we can decompose each $\kappa_i = \kappa_i^0 \parallel \kappa_i^1$ keeping newly created actors/receptionists and messages in the component that generated them. Given such a decomposition, we can choose points to move messages sent across the internal boundary from the sender to reciever message pool such that receives are always from the local pool. We call such a choice of points an `i/o`-*marking* relative to the decomposition (κ^0, κ^1). Given a decomposition and `i/o`-marking we can project π to obtain computations

for κ^j for $j < 2$. This projection preserves fairness of paths if the i/o-marking moves all messages sent from one component to the other, not just those that are eventually received. Such i/o-markings are called *fair*.

Definition (sequence/path decomposition): Let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ be a computation sequence or path for κ. A decomposition (κ^0, κ^1) of κ naturally induces a decomposition of π – a sequence of pairs $[(\kappa_i^0, \kappa_i^1) \mid i < \bowtie]$ such that (κ_i^0, κ_i^1) is a decomposition of κ_i for $i < \bowtie$. The decomposition of π is defined by induction on i as follows. For $i = 0$ we take the given decomposition of κ. The decomposition of κ_{i+1} is obtained from the decomposition (κ_i^0, κ_i^1) of κ_i in one of two ways according to the form of l_i.

(1) l_i is a **exec** transition with focus $a \in \mathrm{Dom}(\alpha_i^j)$, a **rcv** transition with focus $a \in \mathrm{Dom}(\alpha_i^j)$ of $m \in \mu_i^j$, an **input** to a in j, or an **output** from j. In this case, κ_{i+i}^j is the unique configuration such that $\kappa_i^j \xrightarrow{l_i} \kappa_{i+i}^j$, and $\kappa_{i+1}^{1-j} = \kappa_i^{1-j}$.

(2) l_i is a **rcv** transition with focus $a \in \mathrm{Dom}(\alpha_i^j)$ of $m \in \mu_i^{1-j}$. In this case, κ_{i+i}^j is the unique configuration such that $\kappa_i^j \xrightarrow{L_i} \kappa_{i+i}^j$, where $L_i = [\texttt{<in}: m\texttt{>}, l_i]$, and κ_{i+i}^{j-1} is the unique configuration such that $\kappa_i^{j-1} \xrightarrow{\texttt{<out}: m\texttt{>}} \kappa_{i+i}^{j-1}$.

Definition (i/o-marking): Let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ be a computation sequence or path for κ, and let (κ^0, κ^1) be a decomposition of κ. An i/o-marking of π relative to (κ^0, κ^1) is a set, IO, of pairs (k, m) such that the conditions (1-3) below hold.

(1) If $(k, m) \in IO$, then $k < \bowtie$, and m is generated by l_i for some $i \leq k$ such that the focus of l_i and the target of the message are in different components of the path decomposition.

(2) If $(k, m) \in IO$ and $(k', m) \in IO$, then $k = k'$

(3) If l_i is a case 2 **rcv** of m generated in step $i' < i$, then there is some k with $i' \leq k < i$ such that $(k, m) \in IO$

IO is said to be *fair* if the converse to (1) also holds, i.e. if condition (4) below also holds.

(4) If m is generated by l_i for some $i < \bowtie$ such that the focus of l_i and the target of the message are in different components of the path decomposition, then $(k, m) \in IO$, for some k such that $i \leq k$.

An element (k, m) of IO is called a mark at point k in π. There may be more than one such mark at any given point k. A *disambiguation* of IO is a family $\ll$ of total orderings $\ll_k$ on the marks at point k in IO. We will also call the pair $IO, \ll$ an i/o-marking.

Definition (i/o-restriction): Let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ be a computation sequence or path for κ, and let $IO, \ll$ be an i/o-marking of π relative to decomposition (κ^0, κ^1) of κ. Let ν be an initial segment of π. $IO', \ll'$ is an i/o-restriction (or, just restriction) of $IO, \ll$ to ν if $IO' \subseteq IO$, $\ll'$ is the restriction of $\ll$ to IO', and if $(k, m) \in IO'$, then

- $k \leq \mathrm{Len}(\nu)$
- if $(k', m') \in IO$ and $k' < k$ or $k' = k$ and $m' \ll_k m$, then $(k', m') \in IO'$

Definition (projection): If π is a computation for κ and $IO, \ll$ is an i/o-marking for π relative to the decomposition $\kappa = \kappa^0 \parallel \kappa^1$, then the projection $\pi\rfloor_j^{IO,\ll}$ onto component

j for $j < 2$ is defined as follows. Let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ with decomposition $[(\kappa_i^0, \kappa_i^1) \mid i < \bowtie]$ induced by $\kappa = \kappa^0 \parallel \kappa^1$. Then

$$\pi\rfloor_j^{IO,\ll} = [\kappa_i^j \xrightarrow{L_i^j * M_i^j} \kappa_{i+1}^j \mid i < \bowtie]$$

where

- L_i^j is $[l_i]$ if l_i located in j, and $[\,]$ if l_i located in $1-j$, and
- M_i^j is a sequence of `i/o` transition labels corresponding to the marks at point i in IO. Let (i, m_k) be the k-th element in the ordering $\ll_i$. Then the k element of M_i^j is given by
  - `<in : `m_k`>` if the target of m_k is in j
  - `<out : `m_k`>` if the target of m_k is in $1-j$

Lemma (project-computation): If π is a computation for κ, and $IO, \ll$ is an `i/o`-marking for π relative to the decomposition $\kappa = \kappa^0 \parallel \kappa^1$, then the projection $\pi\rfloor_j^{IO,\ll}$ is a computation for κ^j, for $j < 2$. Furthermore if $\pi \in \mathcal{F}(\kappa)$ and IO is a fair marking, then $\kappa^j \in \mathcal{F}(\kappa^j)$, for $j < 2$.

Proof : Let

$$\pi\rfloor_j^{IO,\ll} = [\kappa_i^j \xrightarrow{L_i^j * M_i^j} \kappa_{i+1}^j \mid i < \bowtie]$$

be the projection of π onto j relative to the decomposition $\kappa = \kappa^0 \parallel \kappa^1$ and the marking $IO, \ll$. Then by the construction of the decomposition and the definitions of M_i^j, L_i^j $\kappa_i^j \xrightarrow{L_i^j * M_i^j} \kappa_{i+1}^j$ is a (multi-step) labelled transition for $i < \bowtie$. Thus the projections are (possibly infinite) labelled computation sequences.

Assume $\pi \in \mathcal{F}(\kappa)$ and IO is a fair marking. Suppose l is enabled in κ_i^j, i.e. $\kappa_i^j \xrightarrow{l} \kappa'$ for some κ'. If l is an internal transition, or a `out` transition to an actor external to the composite system, then l is enabled in κ_i, and by fairness of π, either l is l_k for some $k \geq i$, or l is a `rcv` by a and at some $k \geq i$, `rcv`'s by a are permanently disabled in π. In the first case l_k will be a transition in $\pi\rfloor_j^{IO,\ll}$, and in the second case `rcv`'s by a are permanently disabled in $\pi\rfloor_j^{IO,\ll}$ at the projection of k. If $l =$ `<out : `m`>` with the target of m a receptionist of $1-j$, then $(k, m) \in IO$ for some k by fairness of IO, and by the definition of projection l will occur in the projection of l_k. □

Lemma (project-restrict): Let π be a computation for κ, and let $IO, \ll$ be an `i/o`-marking for π relative to the decomposition $\kappa = \kappa^0 \parallel \kappa^1$.

- If ν is an initial segment of π, then any restriction, $IO', \ll'$, of $IO, \ll$ to ν is an `i/o`-marking for ν relative to the decomposition $\kappa = \kappa^0 \parallel \kappa^1$, and $\nu\rfloor_j^{IO',\ll'}$ is an initial segment of $\pi\rfloor_j^{IO,\ll}$ for $j < 2$.
- If ν_0 is an initial segment of $\pi\rfloor_0^{IO,\ll}$ there there is an initial segment, ν, of π and a the restriction $IO', \ll'$ of $IO, \ll$ to ν such that $\nu_0 = \nu\rfloor_0^{IO',\ll'}$.

Definition (complimentary): We say l_0, l_1 are complimentary if $l_j = \texttt{<in}:m\texttt{>}$ and $l_{1-j} = \texttt{<out}:m\texttt{>}$ for some $j < 2$.

Definition (merged): The relation $sMerged(\nu^0, \nu^1, \nu)$ on computation sequences is defined by induction on the combined lengths of ν^0, ν^1. In this definition we use the notation $\nu \xrightarrow{l} \kappa$ to stand for the sequence $[\kappa_0 \xrightarrow{l_0} \kappa_1 \xrightarrow{l_1} \ldots \xrightarrow{l_n} \kappa_n \xrightarrow{l} \kappa]$ where $\nu = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < n]$.

(0) $sMerged([\kappa_0], [\kappa_1], [\kappa_0 \parallel \kappa_1])$ if κ_0, κ_1 are composable.

(1) $sMerged(\nu_0, \nu_1, \nu) \Rightarrow sMerged(\nu_0 \xrightarrow{l} \kappa_0, \nu_1, \nu \xrightarrow{l} \kappa_0 \parallel \kappa_1)$

if Last$(\nu_1) = \kappa_1$, κ_0, κ_1 are composable, and l is not output to receptionist of κ_1

(2) $sMerged(\nu_0, \nu_1, \nu) \Rightarrow sMerged(\nu_0, \nu_1 \xrightarrow{l} \kappa_1, \nu \xrightarrow{l} \kappa_0 \parallel \kappa_1)$

if Last$(\nu_0) = \kappa_0$, κ_0, κ_1 are composable, and l is not output to receptionist of κ_0

(3) $sMerged(\nu_0, \nu_1, \nu) \Rightarrow sMerged(\nu_0, \xrightarrow{l_0} \kappa_0, \nu_1 \xrightarrow{l_1} \kappa_1, \nu)$

if l_0, l_1 are complimentary

Lemma (merged-cseq): If $\nu_j \in \mathcal{T}(\kappa_j)$ for $j < 2$ and $sMerged(\nu_0, \nu_1, \nu)$, then

(1) $\nu \in \mathcal{T}(\kappa_0 \parallel \kappa_1)$, and

(2) there exists a a unique i/o-marking, IO, and disambiguation $\ll$, such that $\nu_j = \nu\rfloor_j^{IO,\ll}$ for $j < 2$.

Proof : (1) follows by induction on the definition of *sMerged*. It is easy to check that in each case ν is a well-formed computation sequence.

An i/o-marking, IO, and disambiguation $\ll$ is constructed by induction on the combined lengths of ν_0, ν_1. If $\nu_j = [\kappa_j]$, then $\nu = [\kappa_0 \parallel \kappa_1]$. In this case, $IO = \emptyset = \ll$, uniquely. Otherwise there are two cases to consider according to whether or not the final transition of ν_j is an output to a receptionist of ν_{1-j} for some $j < 2$.

For the first case, suppose $\nu_0 = \nu_0' \xrightarrow{l_0} \kappa_0'$ and l_0 is an output to a receptionist of ν_1. Then by definition of *sMerged*, we must have $\nu_1 = \nu_1' \xrightarrow{l_1} \kappa_1'$, l_0, l_1 are complimentary, and $sMerged(\nu_0', \nu_1', \nu)$. By induction let $IO', \ll'$ be the unique disambiguated i/o-marking such that $\nu_0' = \nu\rfloor_0^{IO',\ll'}$, and $\nu_1' = \nu\rfloor_1^{IO',\ll'}$. In this case, $IO = IO' \cup \{(k, m)\}$ where $k = \text{Len}(\nu)$ and m is the message transmitted by l_0, l_1. $<$ is $<'$ with $<_k$ extended to make m last in the ordering of marks at point k. The case when the final transition of ν_1 is an output to ν_0 is analogous.

For the second case, by definition of *sMerged*, we must have $\nu = \nu' \xrightarrow{l} \kappa'$, where $\nu_j = \nu_j' \xrightarrow{l} \kappa_j'$, $\nu_{1-j}' = \nu_{1-j}$, and $\kappa' = \kappa_j' \parallel \text{Last}(\nu_{1-j})$, for some $j < 2$, and $sMerged(\nu_0', \nu_1', \nu')$. By induction let $IO', \ll'$ be the unique disambiguated i/o-marking such that $\nu_0' = \nu'\rfloor_0^{IO',\ll'}$, and $\nu_1 = \nu'\rfloor_1^{IO',\ll'}$. In this case, take $IO = IO'$ and $\ll = \ll'$.

□

Lemma (project-merged): If κ_0 and κ_1 are composable, $\nu \in \mathcal{T}(\kappa_0 \parallel \kappa_1)$, and IO is an i/o-marking of ν relative to the decomposition (κ_0, κ_1), then $sMerged(\nu\rfloor_0^{IO,\ll}, \nu'\rfloor_1^{IO,\ll}, \nu)$ for any disambiguation $\ll$ of IO.

Proof : The proof is by induction on the length of ν plus the size of IO. If $\nu = [\kappa_0 \parallel \kappa_1]$ then we are done by clause (0) of the definition of *sMerged*. Suppose $\nu = \nu' \xrightarrow{l_n} \kappa_{n+1}$ has $n+1$ transitions. If $(n,m) \in IO$ then let $IO' = IO - \{(n,m)\}$, and let $\ll'$ be the restriction of $\ll$ to IO'. Suppose the target of m is in component 1, (the other case is similar). Thus $\nu\rfloor_0^{IO,\ll} = \nu\rfloor_0^{IO',\ll'} \xrightarrow{\texttt{<out : } m\texttt{>}} \kappa_{n+1}^0$, $\nu\rfloor_1^{IO,\ll} = \nu\rfloor_1^{IO',\ll'} \xrightarrow{\texttt{<in : } m\texttt{>}} \kappa_{n+1}^1$, and by induction $sMerged(\nu\rfloor_0^{IO',\ll'}, \nu\rfloor_1^{IO',\ll'}, \nu)$. Hence by clause (3) of the definition $sMerged(\nu\rfloor_0^{IO}, \nu\rfloor_1^{IO}, \nu)$. If there is no m such that $(n,m) \in IO$, then suppose the focus of l_n is in component 0, (the other case is similar). Thus $\nu\rfloor_0^{IO,\ll} = \nu'\rfloor_0^{IO,\ll} \xrightarrow{l_n} \kappa_{n+1}^0$, $\nu\rfloor_1^{IO,\ll} = \nu'\rfloor_1^{IO,\ll}$, and by induction $sMerged(\nu'\rfloor_0^{IO,\ll}, \nu'\rfloor_1^{IO,\ll}, \nu')$. Hence by clause (1) of the definition $sMerged(\nu\rfloor_0^{IO,\ll}, \nu\rfloor_1^{IO,\ll}, \nu)$.

□

Definition ($\mathcal{T}(\kappa_0) \parallel \mathcal{T}(\kappa_1)$): The composition of the computation trees of composable configurations κ_0, κ_1 is defined as follows.

$$\mathcal{T}(\kappa_0) \parallel \mathcal{T}(\kappa_1) = \{\nu \mid (\exists \nu_0 \in \mathcal{T}(\kappa_0), \nu_1 \in \mathcal{T}(\kappa_1))(sMerged(\nu_0, \nu_1, \nu))\}$$

Theorem (Composition of Computation Trees): The computation tree of the composition of actor configurations is the composition of the computation trees of the components.

$$\mathcal{T}(\kappa_0 \parallel \kappa_1) = \mathcal{T}(\kappa_0) \parallel \mathcal{T}(\kappa_1)$$

Proof : by (**merged-cseq**) and (**project-merged**). □

5.1.2. The Merging of Paths

Now we extend the composition result to computation paths. We first give two definitions of merged paths and show they are equivalent.

Definition (path-merge-1): $pMerged_1(\pi_0, \pi_1, \pi)$ iff the initial configurations of π_0, π_1 induce a decomposition of π and there is a fair, disambiguated i/o-marking $IO, \ll$ of π for this decomposition such that π_j is the projection of π determined by $IO, \ll$ onto component j for $j < 2$.

Definition (path-merge-2): $pMerged_2(\pi_0, \pi_1, \pi)$ iff for each initial sequence ν of π there are initial sequences ν_j of π_j for $j < 2$ such that $sMerged(\nu_0, \nu_1, \nu)$, and for each initial sequence ν_j of π_j (for $j < 2$) there are initial sequences ν of π and ν_{1-j} of π_{1-j} such that $sMerged(\nu_0, \nu_1, \nu)$.

Lemma (pm12): $pMerged_1(\pi_0, \pi_1, \pi)$ iff $pMerged_2(\pi_0, \pi_1, \pi)$.

Proof (pm12): For the only if direction, assume $pMerged_1(\pi_0, \pi_1, \pi)$. Let $[(\kappa_i^0, \kappa_i^1) \mid i < \bowtie]$ be the decomposition of π induced by the initial configurations of π_0, π_1, let $IO, \ll$ be the fair, disambiguated i/o-marking of π for this decomposition with projections π_0, π_1. Suppose ν is an initial segment of π, then by (**project-restrict**) the restriction of $IO, \ll$

to ν is a i/o-marking for ν with corresponding projections ν_j, being initial segments of the projections π_j of π and by (**project-merged**) $sMerged(\nu_0, \nu_1, \nu)$. Suppose ν_0 is an initial segment of π_0. Then by (**project-restrict**) ν_0 is the projection onto component 0 determined by some restriction of $IO, \ll$ to some initial segment ν of π. Take ν_1 to be the projection of ν onto component 1 as above. $\Box_{\mathbf{onlyif}}$

For the if direction, assume $pMerged_2(\pi_0, \pi_1, \pi)$ and let $\kappa_0, \kappa_1, \kappa$ be the corresponding initial configurations. Then (κ_0, κ_1) is a decomposition of κ. For each initial segment ν of π, let $IO_\nu, \ll_\nu$ be the maximal i/o-marking such that $\nu\rfloor_j^{IO_\nu, \ll_\nu}$ is an initial segment of π_j for $j < 2$. Note that these marking form a chain. Let $IO, \ll$ be the union of this chain of markings. We claim (i) that $IO, \ll$ is a fair i/o-marking for π, relative to the decomposition (κ_0, κ_1) of κ, and (ii) that $\pi_j = \pi\rfloor_j^{IO, \ll}$. (ii) is easy. To see (i) we show that each of the conditions of the definition of and i/o-marking hold. (1) and (2) follow from the fact that if $(k, m), (k', m) \in IO$, then $(k, m), (k', m) \in IO_\nu$ for some initial segment ν of π. (3) follows from the fact that if l_i is a case 2 `rcv` of π then it is a case 2 `rcv` of some initial segment ν. For (4), suppose that m is generated by l_i in component 0 with target in component 1. Then by fairness of π_0 there is some initial segment ν_0 that contains the generation and output of m. Choose ν_1, ν initial segments of π_1, π such that $pMerged(\nu_0, \nu_1, \nu)$. Then by maximality IO_ν contains (k, m) for some k. $\Box_{\mathbf{if}}$

$\Box_{\mathbf{pm12}}$

Henceforth we will write *pMerged* without a subscript to mean the relation given by either of the above definitions. Note that, as suggested by the if-direction argument, it is necessary to require that each initial segment of π_j be mergable with some initial segment of π_{1-j} in (**path-merge-2**). Otherwise the situation could arise where $[\kappa_0^1 \stackrel{l}{\longrightarrow} \kappa_1^1]$ is an initial sequence of π_1, κ_0^1 is quiescent, and l is an input. Then π could just be π_0 with the extra actors tacked on, without violating any of the other conditions.

Using the definition of path merging and the facts about projections we can extend the tree composition results to fair computation sets.

Definition ($\mathcal{F}(\kappa_0) \parallel \mathcal{F}(\kappa_1)$): The composition of the sets of fair computation paths of composable configurations κ_0, κ_1 is defined as follows.

$$\mathcal{F}(\kappa_0) \parallel \mathcal{F}(\kappa_1) = \{\pi \mid (\exists \pi_0 \in \mathcal{F}(\kappa_0), \pi_1 \in \mathcal{F}(\kappa_1))(pMerged(\pi_0, \pi_1, \pi))\}$$

Theorem (Composition of Fair Computations): The set of fair computation paths of the composition of actor configurations is the composition of the fair computation paths of the components.

$$\mathcal{F}(\kappa_0 \parallel \kappa_1) = \mathcal{F}(\kappa_0) \parallel \mathcal{F}(\kappa_1)$$

Proof : For the left-to-right containment, assume $\pi \in \mathcal{F}(\kappa_0 \parallel \kappa_1)$. Let $IO, \ll$ be any fair i/o-marking of π for the decomposition $(\kappa_0 \parallel \kappa_1)$ of κ. Then by (**project-computation**) $\pi\rfloor_j^{IO, \ll} \in \mathcal{F}(\kappa_j)$ for $j < 2$ and hence $\pi \in \mathcal{F}(\kappa_0) \parallel \mathcal{F}(\kappa_1)$.

For the right-to-left containment, assume $\pi \in \mathcal{F}(\kappa_0) \parallel \mathcal{F}(\kappa_1)$ and let $\pi_j \in \mathcal{F}(\kappa_j)$ be such that $pMerged(\pi_0, \pi_1, \pi)$. Thus $\pi_j = \pi\rfloor_j^{IO, \ll}$ for $j < 2$ for some fair i/o-marking of π for the decomposition $(\kappa_0 \parallel \kappa_1)$ of κ. We only need to show that π is fair. Suppose l is

enabled at stage i in π then it is enabled at the corresponding point in one of the component paths, and by fairness of the component paths either the transition occurs at some point in the component or it becomes permanently disabled at some point in the component. The same will be true in π. □

We give one further characterization of merging of computation paths in terms of their input, output transitions. Informally, π_0 and π_1 are mergable just if they have disjoint sets of actors, and if there are complimentary subsequences of their `i/o` transitions that include all output from one path to actors of the other path.

Definition (mergable): Let $\pi_j = [\kappa_i^j \xrightarrow{l_i} \kappa_{i+1}^j \mid i < \bowtie^j]$ be computation paths with $\kappa_i^j = \left\langle\!\!\left\langle\, \alpha_i^j \mid \mu_i^j \,\right\rangle\!\!\right\rangle_{\chi_i^j}^{\rho_i^j}$ for $i < \bowtie^j$ and $j < 2$. π_0 and π_1 are mergable just if the following conditions hold.

(1) $\bigcup_{i<\bowtie^0} \mathrm{Dom}(\alpha_i^0) \cap \bigcup_{i<\bowtie^1} \mathrm{Dom}(\alpha_i^1) = \emptyset$, and

(2) if $a \in \chi_i^j \cap \mathrm{Dom}(\alpha_k^{1-j})$ then $a \in \rho_{k'}^{1-j}$ for some $k' \geq k$.

(3) There are subsequences, $\xi^j = [\xi_i^j \mid i < \bowtie]$,of $\bowtie^j$ such that

(3.1) $l_{\xi_i^0}^0$ and $l_{\xi_i^1}^1$ are complimentary, for $i < \bowtie$.

(3.2) if $l_i^j =$ `<out : m>` with the target of m a receptionist of π^{1-j} then i is in the range of ξ^j.

We call ξ^0, ξ^1 a *synchronization* for π_0, π_1.

A synchronization for a mergable pair of paths determines a set of mergings of the paths in which complimentary `i/o` transitions become internal boundary crossings.

Definition (*Mergings*): Let $\pi_j = [\kappa_i^j \xrightarrow{l_i} \kappa_{i+1}^j \mid i < \bowtie^j]$ for $j < 2$ be mergable with synchronization ξ^0, ξ^1. The $\mathit{Mergings}(\pi_0, \xi^0, \pi_1, \xi^1)$ is the set of computation paths $\pi = (\kappa_0^0 \parallel \kappa_0^1; \zeta)$ where $\zeta = [L_i \mid i < \bowtie]$, L_i is any interleaving of L_i^0, L_i^1, and $L_i^j = [l_m^j \mid \xi_i^j < m < \xi_{i+1}^j]$. (In the case $\bowtie$ is finite we define $\xi_\bowtie^j$ to be $\bowtie^j$.)

Lemma (mergings-merged):

(1) If $\mathit{pMerged}(\pi_0, \pi_1, \pi)$ then π_j for $j < 2$ are mergable.

(2) If $\pi_j = [\kappa_i^j \xrightarrow{l_i} \kappa_{i+1}^j \mid i < \bowtie^j] \in \mathcal{F}(\kappa_0^j)$ for $j < 2$ are mergable with synchronization ξ^0, ξ^1, then $\mathit{pMerged}(\pi_0, \pi_1, \pi)$ for $\pi \in \mathit{Mergings}(\pi_0, \xi^0, \pi_1, \xi^1)$.

Proof : For (1) it is easy to see that the hygiene conditions (1-2) of the definition of mergable are satisfied. Let $IO, \ll$ be a fair `i/o`-marking for π with projections π_j for $j < 2$, given by the definition of *pMerged*. This naturally induces a synchronization pair.

For (2) we need to find a fair `i/o`-marking $IO, \ll$ for π relative to the decomposition (κ_0^0, κ_0^1) such that $\pi_j = \pi\rfloor_j^{IO,\ll}$ for $j < 2$. Let $\pi = (\kappa_0^0 \parallel \kappa_0^1; [L_i \mid i < \bowtie])$ Intuitively we mark each $i < \bowtie$ with the message m exchanged at stage i (i.e. the message part of ξ_i^j). Flattening the sequence of multi-steps we obtain a corresponding marking IO of the single step representation of the path. Because some of the L^i may be empty, some points may have multiple marks. These ordering in the multi-step sequence induces an ordering $\ll$ of the set of marks at each point in the single step sequence. Since the π_j are fair, any message

generated in π_j to an actor in π_{1-j} will be output, and hence have a mark in IO. Thus IO is fair.

□

5.2. Equivalence of Actor Configurations

We first define *interaction equivalence* for actor configurations. This is a natural extension of the notion of trace equivalence to actor configurations. Traditionally, traces are defined to be finite sequences of events or actions. Two objects are trace equivalent if they have the same set of traces. Three points distinguish interaction equivalence from the traditional trace equivalence.

(1) Since trace equivalence is defined for finite traces only, it cannot account for liveness properties such as fairness. Thus we must also consider infinite traces. Related notions in the literature include failure sets and completed traces [7, 15, 9].

(2) Only fair traces should be considered.

(3) A trace semantics should more directly reflect the behavior of an actor configuration considered as a component. In particular, they should allow for the dynamic addition of external actors to the environment as the configuration evolves. To our knowledge the present work is the first attempt to seriously address this problem.

We then extend the observational equivalence on expressions to actor configurations. As with expressions, this is accomplished by considering closed configurations obtained by adding observers. We also extend interaction equivalence to expressions. We show that interaction equivalence implies observational equivalence, both for expressions and configurations. The converse is an open problem.

5.2.1. Interaction Equivalence

Informally, two actor configurations are interaction equivalent, written $\kappa_0 \cong_{\mathtt{i}} \kappa_1$, if their fair computations give rise to the same set of `i/o` transition sequences. To make this precise we first define the `i/o` projection operations on transition labels, computation sequences, paths, and trees.

Definition (`i/o`):

$$\mathtt{i/o}(l) = \begin{cases} [\texttt{<out}: a, m\texttt{>}] & \text{if } l = \texttt{<out}: a, m\texttt{>} \\ [\texttt{<in}: a, m\texttt{>}] & \text{if } l = \texttt{<in}: a, m\texttt{>} \\ [\,] & \text{otherwise} \end{cases}$$

$$\mathtt{i/o}([\kappa_i \xrightarrow{l_i} \kappa_i' \;\big|\; i < \bowtie]) = *[\mathtt{i/o}(l_i) \;\big|\; i < \bowtie] = \mathtt{i/o}(l_0) * \mathtt{i/o}(l_1) * \ldots * \mathtt{i/o}(l_i) \ldots$$

$$\mathtt{i/o}(T) = \{\mathtt{i/o}(\pi) \;\big|\; \pi \in T\} \qquad \text{for any set, } T, \text{ of paths}$$

Definition ($\kappa_0 \cong_{\mathtt{i}} \kappa_1$): Let κ_0, κ_1 be actor configurations with the same receptionists. Then

$$\kappa_0 \cong_{\mathtt{i}} \kappa_1 \;\Leftrightarrow\; \mathtt{i/o}(\mathcal{F}(\kappa_0)) = \mathtt{i/o}(\mathcal{F}(\kappa_1))$$

Lemma ($\cong_{\mathtt{i}}$-equiv): The relation $\cong_{\mathtt{i}}$ is an equivalence relation on configurations (transitive, reflexive, symmetric).

Lemma ($\cong_{\mathtt{i}}$-compose): If $\kappa_0 \cong_{\mathtt{i}} \kappa_1$ and κ is composable with κ_j for $j < 2$ then $\kappa_0 \parallel \kappa \cong_{\mathtt{i}} \kappa_1 \parallel \kappa$.

Proof : It suffices to show that for each $\pi_0' \in \mathcal{F}(\kappa_0 \parallel \kappa)$, we can find $\pi_1' \in \mathcal{F}(\kappa_1 \parallel \kappa)$, such that $\mathtt{i/o}(\pi_0') = \mathtt{i/o}(\pi_1')$. Assume $\pi_0' \in \mathcal{F}(\kappa_0 \parallel \kappa)$, and find $\pi_0 \in \mathcal{F}(\kappa_0)$, $\pi \in \mathcal{F}(\kappa)$ such that $pMerged(\pi_0, \pi, \pi_0')$ (pick a fair $\mathtt{i/o}$-marking and project). By (**mergings-merged**) π_0 is mergable with π. Since $\kappa_0 \cong_{\mathtt{i}} \kappa_1$ we can find $\pi_1 \in \mathcal{F}(\kappa_1)$ such that $\mathtt{i/o}(\pi_0) = \mathtt{i/o}(\pi_1)$, and π_1 satisfies the conditions (1-2) for mergability with π. Thus π_1 is mergable with π and any such merging π_1' is a fair computation for $\kappa_1 \parallel \kappa$. By careful bookkeeping we can choose and interleaving of segments between synchronization points we can find a merging $\pi_1' \in \mathcal{F}(\kappa_1 \parallel \kappa)$ such that $\mathtt{i/o}(\pi_0') = \mathtt{i/o}(\pi_1')$. □

Interaction equivalence naturally extends to expressions as follows. Two expressions are interaction equivalent just if replacing some occurrences of one by the other in a configuration results in interaction equivalent configurations.

Definition ($e_0 \cong_{\mathtt{i}} e_1$): $e_0 \cong_{\mathtt{i}} e_1$ just if $\kappa_0 \cong_{\mathtt{i}} \kappa_1$ for all κ_i of the form

$$\left\langle\!\!\left\langle\, \alpha, \llbracket C[e_i] \rrbracket_a \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$$

Using interaction equivalence we can state a conjecture concerning the connection between certain configurations and expressions that construct them. This strengthens the intuition that, at least statically, there is a strong connection between expressions and configurations. For simplicity we consider the case of a single actor configuration.

Conjecture (io-exp): Let $\kappa_j = \left\langle\!\!\left\langle\, (b_j(a))_a \mid \emptyset \,\right\rangle\!\!\right\rangle_{\chi}^{a}$, and let e_j be expressions constructing κ_j.

$$e_j = \mathtt{let}\{a := \mathtt{newadr}()\}\mathtt{let}\{z := \mathtt{initbeh}(a, b_j(a))\}a$$

Then

$$\kappa_0 \cong_{\mathtt{i}} \kappa_1 \;\Rightarrow\; e_0 \cong_{\mathtt{i}} e_1$$

We note that it is not necessarily the case that $b_0(x) \cong b_1(x)$. We will see an example in the next section. The intuition behind the conjecture is the following. Let O be an observing context for e_j, and let $\pi \in \mathcal{F}(O[e_0])$. Without loss we treat executions of e_0 as single steps. We annotate each configuration in π to mark the remaining occurrences of e_0 (in descendants of holes that remain untouched). We also list actors a_i that have been created by executions of e_0 and associated with each a_i we list any actors created as a result of a_i receiving a message. We consider each a_i and its associates as a subconfiguration. We mark for each a_i the subpath π_i^0 corresponding to transitions internal to the subconfiguration, inserting phantom $\mathtt{i/o}$ messages to represent the crossing of internal boundaries. We claim that $\pi_i^0 \in \mathcal{F}(\kappa_0)$ and hence we can find $\pi_i^1 \in \mathcal{F}(\kappa_1)$ such that $\mathtt{i/o}(\pi_i^0) = \mathtt{i/o}(\pi_i^1)$. We further claim that each π_i^0 can be replaced by the corresponding π_i^1 and that the result will be a fair computation for $O[e_1]$. The hard part is getting the path into canonical form so that the joint replacement can be done without getting confused.

5.2.2. Observational Equivalence

Now we extend the notion of observational equivalence to configurations.

Definition (Observing Configurations): For an actor configuration $\kappa = \left\langle\!\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$ the observing configurations are configurations over the extended language of the form $\kappa' = \left\langle\!\!\left\langle\, \alpha' \mid \mu' \,\right\rangle\!\!\right\rangle_{\rho}^{\chi}$ such that κ' is composable (in the sense of §5.1) with κ.

We are interested in observing internal **event** transitions rather than interactions with the environment. Thus we define an operation Hide(κ) hiding all the receptionists of a configuration.

Definition (Hide(κ)): $\text{Hide}(\left\langle\!\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}) = \left\langle\!\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\emptyset}$

Definition ($\kappa_0 \cong \kappa_1$): For $\kappa_0 = \left\langle\!\!\left\langle\, \alpha_0 \mid \mu_0 \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$ and $\kappa_1 = \left\langle\!\!\left\langle\, \alpha_1 \mid \mu_1 \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$, $\kappa_0 \cong \kappa_1$ iff $Obs(\text{Hide}((\kappa_0 \parallel \kappa'))) = Obs(\text{Hide}((\kappa_1 \parallel \kappa')))$ for all observing configurations κ' for κ_j, $j < 2$.

Note the following. Firstly, Hide($(\kappa \parallel \kappa')$) is a closed configuration for observer κ' for κ. Secondly, no two closed configurations can be distinguished by an external observer.

Using observational equivalence configurations we can extend the property of congruence with respect to expression construction to congruence with respect to configuration construction. Namely, replacing an expression occurring in a configuration by an observationally equivalent one yields an equivalent configuration.

Theorem (exp-cfig): If $e_0 \cong e_1$ then

(i) $\kappa_0 = \left\langle\!\!\left\langle\, \alpha, \llbracket C[e_0] \rrbracket_a \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \cong \left\langle\!\!\left\langle\, \alpha, \llbracket C[e_1] \rrbracket_a \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} = \kappa_1$

(ii) $\kappa_0' = \left\langle\!\!\left\langle\, \alpha, (\lambda x.C[e_0])_a \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} \cong \left\langle\!\!\left\langle\, \alpha, (\lambda x.C[e_1])_a \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho} = \kappa_1'$

Proof : (i) We need to show that $Obs(\text{Hide}(\kappa_0 \parallel \kappa')) = Obs(\text{Hide}(\kappa_1 \parallel \kappa'))$ for any observing κ'. Note however that $\text{Hide}(\kappa_0 \parallel \kappa') = O[e_0]$ and $\text{Hide}(\kappa_1 \parallel \kappa') = O[e_1])$ for some $O \in \mathbb{O}$, so the result follows directly from the definition of $\cong$.

(ii) We need to show that $Obs(\text{Hide}(\kappa_0' \parallel \kappa')) = Obs(\text{Hide}(\kappa_1' \parallel \kappa'))$ for any observing κ'. Pick any $\pi_0 \in \mathcal{F}(\text{Hide}(\kappa_0' \parallel \kappa'))$, then we must find $\pi_1 \in \mathcal{F}(\text{Hide}(\kappa_1' \parallel \kappa'))$ such that $obs(\pi_0) = obs(\pi_1)$. There are two cases to consider. Either actor a never becomes active, or it becomes active first after k steps of computation. In the first case, the e_i are never touched, so both computations proceed uniformly, thus their observation and fairness behavior both correspond. In the second case, consider the step where a receives its first message:

$$\left\langle\!\!\left\langle\, \alpha', (\lambda x.C[e_0])_a \mid \mu, \texttt{<}a \Leftarrow cv\texttt{>} \,\right\rangle\!\!\right\rangle \xrightarrow{\texttt{<rcv : }a, cv\texttt{>}} \left\langle\!\!\left\langle\, \alpha', \llbracket \textbf{app}(\lambda x.C[e_0], cv) \rrbracket_a \mid \mu \,\right\rangle\!\!\right\rangle = O[e_0]$$

Factor $\pi_0 = \nu[e_0] * \pi_0'$, where $\pi_0' \in \mathcal{F}(O[e_0])$ and $\nu[]$ denotes a sequence where each configuration in the sequence contains a hole that computes uniformly in the hole. Thus, $Obs(O[e_0]) = Obs(O[e_1])$ because $e_0 \cong e_1$ and O is a configuration context. This means

by the definition of Obs there is a path $\pi_1' \in \mathcal{F}(O[e_1])$, such that $obs(\pi_0') = obs(\pi_1')$. Let $\pi_1 = \nu[e_1] * \pi_1'$. Then, $\pi_1 \in \mathcal{F}(\text{Hide}(\kappa_1 \parallel \kappa'))$, since by construction it is a computation for $(\text{Hide}(\kappa_1 \parallel \kappa'))$, and because π fair implies $\nu * \pi$ is fair for any ν such that $\nu * \pi$ is a computation path. Moreover, $obs(\pi_0) = obs(\pi_1)$ because any **event** transitions in $\nu[e_0]$ also occur in $\nu[e_1]$, and $obs(\pi_0') = obs(\pi_1')$ by hypothesis.

□

Theorem (io-op): If two configurations are interaction equivalent, then they are observationally equivalent. Similarly for expressions.

(1) $\kappa_0 \cong_{\mathtt{i}} \kappa_1 \Rightarrow \kappa_0 \cong \kappa_1$

(2) $e_0 \cong_{\mathtt{i}} e_1 \Rightarrow e_0 \cong e_1$

Proof (1): Let κ be any observing configuration for κ_0, κ_1, and let $\pi_0' \in \mathcal{F}(\text{Hide}(\kappa_0 \parallel \kappa))$. Then $\pi_0' \in \mathcal{F}(\kappa_0 \parallel \kappa)$ and by (**composition of fair computations**) $\pi_0' \in \mathcal{F}(\kappa_0) \parallel \mathcal{F}(\kappa)$. Consequently $(\exists \pi_0 \in \mathcal{F}(\kappa_0), \pi \in \mathcal{F}(\kappa))(pMerged(\pi_0, \pi, \pi_0'))$, so by (**mergings-merged**) the π_0 and π are mergable, and since all **event**s take place in π we have that $obs(\pi_0') = obs(\pi)$. Since $\kappa_0 \cong_{\mathtt{i}} \kappa_1$ we can find $\pi_1 \in \mathcal{F}(\kappa_1)$ such that $\mathtt{i/o}(\pi_0) = \mathtt{i/o}(\pi_1)$. Hence π_1 is mergable with π and, by reasoning similar to the above, for any merging π_1' we have $obs(\pi_0') = obs(\pi_1')$. Also π_1' has no input transitions, since the only inputs of π_1, π are synchronized with outputs from the opposite component. Thus $\pi_1' \in \mathcal{F}(\text{Hide}(\kappa_1 \parallel \kappa))$. $\square_1$

Proof (2): Assume $e_0 \cong_{\mathtt{i}} e_1$ and let O be an observing context for e_0, e_1. Let a^* be and actor not in the domain of O and let O^* be obtained from O by replacing all occurrences of **event**() by **send**$(a^*, \mathtt{nil})$. Then $\kappa_j = O^*[e_j]$ is a configuration with no receptionists and one external actor, a^* and $\kappa_0 \cong_{\mathtt{i}} \kappa_1$. Let $\kappa^* = \left\langle\!\!\left\langle\, (\lambda m.\mathtt{event}())_{a^\bullet} \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle_{\emptyset}^{a^\bullet}$. Then κ^* is an observing configuration for κ_j for $j < 2$. Furthermore, $Obs(\text{Hide}(\kappa_j \parallel \kappa^*)) = Obs(O[e_j])$ for $j < 2$. By (1) $Obs(\text{Hide}(\kappa_0 \parallel \kappa^*)) = Obs(\text{Hide}(\kappa_1 \parallel \kappa^*))$. $\square_2$

$\square_{\mathbf{io-op}}$

The converse of (**io-op**) is an open problem.

5.3. Transforming Paths

Now we develop some basic tools for manipulating paths. The main objective is to be able to treat finite sequences of transition labels as defining single multi-step transitions, and to establish conditions under which we may assume that a path has been put into suitable canonical form.

We let **<exec** : a**>** stand for any transition (label) with focus a, other than **rcv**. This implicitly excludes **in** and **out**, as they are not considered to have focus actors.

Definition (Independence –): Two transitions are independent if whenever they can occur consecutively, they can occur in either order, and both orders define the same multi-step transition. Formally, $l_0 - l_1$ just if

$$\kappa \xrightarrow{[l_0, l_1]} \kappa' \;\Leftrightarrow\; \kappa \xrightarrow{[l_1, l_0]} \kappa'$$

for all configurations κ, κ'.

Note that two transitions are (vacuously) considered independent if they can not occur adjacently. For example `rcv` is independent of `send`, `newadr`, `initbeh` at the same location because the transition immediately following a `rcv` must be `beta-v`, and no `exec` other than a `become` can precede an `rcv`.

Theorem (Independence): The following pairs of transitions are independent:

$$\texttt{<in}: m\texttt{>} - \begin{cases} \texttt{<rcv}: a, cv\texttt{>} & \text{if } m \neq \texttt{<}a \Leftarrow cv\texttt{>} \\ \texttt{<exec}: a\texttt{>} \end{cases}$$

$$\texttt{<out}: m\texttt{>} - \begin{cases} \texttt{<rcv}: a, cv\texttt{>} \\ \texttt{<exec}: a\texttt{>} & \text{except } \texttt{<send}: a, m\texttt{>} \end{cases}$$

$$\texttt{<rcv}: a, cv\texttt{>} - \begin{cases} \texttt{<in}: m'\texttt{>} & \text{if } m' \neq m \\ \texttt{<out}: m'\texttt{>} \\ \texttt{<exec}: a'\texttt{>} & \text{except } \texttt{<send}: a', \texttt{<}a \Leftarrow cv\texttt{>>} \end{cases}$$

$$\texttt{<init}: a, a'\texttt{>} - \begin{cases} \texttt{<in/out}: m\texttt{>} \\ \texttt{<rcv}: a_0, cv\texttt{>} & \text{if } a_0 \neq a' \\ \texttt{<exec}: a_0\texttt{>} & \text{if } a_0 \neq a' \end{cases}$$

$$\texttt{<bec}: a, a'\texttt{>} - \begin{cases} \texttt{<in/out}: m\texttt{>} \\ \texttt{<rcv}: a_0, cv\texttt{>} & \text{if } a_0 \neq a \\ \texttt{<exec}: a_0\texttt{>} & \text{if } a_0 \neq a \text{ and } a_0 \neq a' \end{cases}$$

$$\texttt{<send}: a, m\texttt{>} - \begin{cases} \texttt{<in}: m'\texttt{>} \\ \texttt{<out}: m'\texttt{>} & \text{if } m' \neq m \\ \texttt{<rcv}: a_0, cv\texttt{>} & \text{if } m \neq \texttt{<}a_0 \Leftarrow cv\texttt{>} \\ \texttt{<exec}: a'\texttt{>} & \text{if } a \neq a' \end{cases}$$

$$\texttt{<new}: a, a'\texttt{>} - \begin{cases} \texttt{<in/out}: m\texttt{>} \\ \texttt{<rcv}: a_0, cv\texttt{>} \\ \texttt{<exec}: a_0\texttt{>} & \text{if } a_0 \neq a \end{cases}$$

$$\texttt{<fun}: a\texttt{>} - \begin{cases} \texttt{<in/out}: m\texttt{>} \\ \texttt{<rcv}: a', cv\texttt{>} & \text{if } a' \neq a \\ \texttt{<exec}: a'\texttt{>} & \text{if } a' \neq a \end{cases}$$

Proof (Independence): Each case must be checked. Consider the first case, <in : m>. This step depends on nothing since it is an input transition, so it only must be checked that it is not permuted to be before a step that depends directly on it. The only step that directly depends on it is a receive of the same message, by inspection of the transition rules. Thus, all other transitions are independent of <in : m>. The other cases may be argued similarly. □

An event thread, I, at a in π is a finite subsequence of transitions of π directly initiated by receipt of a message by a and executed by a and its clones. In other words an event thread at a in π is a sequence in the computation of the configuration obtained by restricting the configuration of π at the point of the message receipt to the focus actor a. We further require that there are no receive transitions, other than the initial one, and no input or output transitions.

Definition (event thread): Let

$\mathcal{F}(\kappa) = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$, and $\pi \in \mathcal{F}(\kappa)$,

$I = [i_j \mid j < n]$ such that $j < j' < n \Rightarrow i_j < i_{j'}$,

$L(I, \pi) = [l_{i_j} \mid j < n]$, the transition sequence corresponding to I in π.

Then I is an *event thread* at a in π (initiated at i_0) if

(1) $l_{i_0} = \mathtt{rcv}(a, cv)$ for some cv,

(2) $L(I, \pi)$ contains no `rcv`, `in` or `out` transitions after the first `rcv`, and

(3) $L(I, \pi)$ is a computation for $\left\langle\!\!\left\langle\, \alpha_{i_0} \rfloor \{a\} \mid \texttt{<}a \Leftarrow cv\texttt{>} \,\right\rangle\!\!\right\rangle$.

Condition (**3**) makes explicit the structure of an event thread, running the actor in a configuration with only itself and no receives can only compute itself and its clones.

Definition (macro-step): Let $\pi \in \mathcal{F}(\kappa)$. A macro step at a in π is a tail of an event thread at a in π, i.e. an event thread with some of the initial transitions possibly omitted.

Definition (permutation): Let $\pi = \kappa; \zeta \in \mathcal{F}(\kappa)$. $\pi' = \kappa; \zeta' \in \mathcal{F}(\kappa)$ is a permutation of π if there is a bijection p on $\{0 \leq i < \bowtie\}$ where $\bowtie$ is the length (domain) of ζ and ζ' such that $\zeta' = \zeta \circ p$, i.e. $\zeta'(i) = \zeta(p(i))$ for $0 \leq i < \bowtie$.

Theorem (macro-step): Let $\pi \in \mathcal{F}(\kappa)$, and let I be a macro step in π. Then we can find $\pi' \in \mathcal{F}(\kappa)$ and a bijection p such that π' is a permutation of π via p, and letting I' be the image of I under p, $L(I', \pi')$ is a multi-step of π': $I' = [k + i \mid 0 \leq i < \mathrm{Len}(I)]$ for some k.

Corollary (macro-step): The π' given by (**macro-step**) also satisfies:

(io) $\mathtt{i/o}(\pi) = \mathtt{i/o}(\pi')$

(ev) $obs(\pi) = obs(\pi')$

Proof (macro-step): This is just iterated application of (**independence**) to bubble up steps to occur immediately after the initial `rcv`. □

Theorem (infinite-macro-steps): Let $\pi \in \mathcal{F}(\kappa)$, and let I_j for $j \in J$ and index set J be a family (possibly infinite) of pairwise disjoint macro steps in π. (Two macro steps I, I' in π are disjoint if they have an empty intersection.) Then we can find $\pi' \in \mathcal{F}(\kappa)$ and a bijection p such that

(1) π' is a permutation of π via p;

(2) letting I'_j be the image of I_j under p, then $L(I'_j, \pi')$ is a multi-step transition of π' for $j \in J$

Corollary (infinite-macro-steps): Let π' be as in (**infinite-macro-steps**), then

(io) $\mathtt{i/o}(\pi) = \mathtt{i/o}(\pi')$

(ev) $obs(\pi) = obs(\pi')$

Proof : p is computed by induction on the index set. It will be the identity up to the point where the first macro step begins. The next group of transitions will be those of the macro step. Then add the length of the macro step to transitions not included until the next macro step begins and so on. It is easy to see that the result is a computation. Enabledness of transitions is not changed, and every transition of π occurs either as part of a macro step or delayed by a finite amount corresponding to the sum of the lengths of the preceding (finite number of) macro steps. Thus fairness is also preserved. (1) is ensured by construction, and (2) is ensured by the fact that the two computations are related by a bijection on the sequence of transition labels. □

6. Examples of Configuration Equivalence

In this section we give two examples that illustrate a method for establishing equivalence of configurations. This first example is an instance of a simple transformation that eliminates indirection in returning a reply to a message. The second example involves a variety of ways in which recursion can be expressed. Equivalence is established by proving interaction equivalence. This is accomplished by establishing a causality relation between messages input from the environment and messages output to the environment. Then we show that for each configuration and each possible pattern of input/output augmented with causality there is a fair computation path with that pattern. The work of establishing the causality relation is made easier by showing that computations can be put into a simple canonical form.

6.1. Indirection Elimination

We begin with a two actor system with one receptionist and one external actor. The receptionist takes requests for transforming data, applies one operation, and sends the result to the other internal actor. The second actor applies a second operation and sends the result back to the receptionist, who returns it to the external actor. This is transformed into a system in which the second actor is modified to return its results directly to the external actor. The receptionist is unchanged. We show that this transformation is sound, i.e. that the two systems are equivalent, using the ideas sketched above. This approach also works for transformations such as fusion or splitting of internal actors [33]. We assume that `do`?, `ret`? are true on disjoint sets of messages, and that `ret`?(`mkret`(v)) is true. We also assume that f and g are functions that, when applied to communicable values, have finite

computations, with no observable effects, returning communicable values. Let procedures **b0**, **b1** be defined by

$$b_0 = \lambda x.\lambda a.\mathtt{rec}(\lambda b.\lambda m.\mathtt{seq}(\mathtt{become}(b), \mathtt{if}(\mathtt{do?}(m), \mathtt{send}(a, f(m)), \mathtt{if}(\mathtt{ret?}(m), \mathtt{send}(x, m), \mathtt{nil}))))$$

$$b_1 = \lambda c.\mathtt{rec}(\lambda b.\lambda m.\mathtt{seq}(\mathtt{become}(b), \mathtt{send}(c, \mathtt{mkret}(g, m))))$$

Now we construct the two systems, each with receptionist a_0 and external actor x. They differ only in whether the parameter c in the internal actors behavior is instantiated to a_0 (indirect reply) or to x (direct reply). Thus we define behaviors B_0 and $B_1(c)$, and $\kappa^{(c)}$, for $c \in \{a_0, x\}$.

$$B_0 = \mathtt{b0}(x, a_1), \qquad B_1(c) = \mathtt{b1}(c), \qquad \kappa^{(c)} = \left\langle\!\!\left\langle\ (B_0)_{a_0}, (B_1(c))_{a_1} \ \middle|\ \emptyset \right\rangle\!\!\right\rangle_x^{a_0}$$

The soundness of the transformation from $\kappa^{(a_0)}$ to $\kappa^{(x)}$ is expressed by the following theorem.

Theorem (rem.indir): $\kappa^{(a_0)} \cong_{\mathtt{i}} \kappa^{(x)}$

Corollary (rem.indir): $\kappa^{(a_0)} \cong \kappa^{(x)}$

The proof uses two key lemmas. The first lemma (**ri-1**) says that if we are given a (possibly infinite) sequence ζ of input and output transitions for $\kappa^{(c)}$, and a map giving for each occurrence of an output transition in ζ the (unique) input transition occurrence to which it is a reply, then we can find $\pi \in \mathcal{F}(\kappa^{(c)})$ such that $\mathtt{i/o}(\pi) = \zeta$, for $c \in \{a_0, x\}$. The second lemma (**ri-2**) says that if $\pi \in \mathcal{F}(\kappa^{(c)})$, then we can find a map γ, such that $\mathtt{i/o}(\pi), \gamma$ satisfy the conditions of (**ri-1**).

Lemma (ri-1): Let ζ be a (possibly infinite) sequence of input and output transitions for $\kappa^{(c)}$

$$\zeta = [l_i \mid i < \bowtie] \qquad \text{where } l_i \in \texttt{<in : <}a_0 \Leftarrow \mathrm{c}\mathbb{V}\texttt{>>} \cup \texttt{<out : <}x \Leftarrow \mathrm{c}\mathbb{V}\texttt{>>} \text{ for } i < \bowtie$$

and define

$$\mathtt{in}_\zeta = \{i \mid \zeta(i) \in \texttt{<in : <}a_0 \Leftarrow \mathrm{c}\mathbb{V}\texttt{>>}\}$$

$$\mathtt{out}_\zeta = \{i \mid \zeta(i) \in \texttt{<out : <}x \Leftarrow \mathrm{c}\mathbb{V}\texttt{>>}\}.$$

Suppose γ maps each output to its causing input, and each input that satisfies **ret**? or **do**? causes an output with appropriate contents. I.e., $\gamma \in [\mathtt{out}_\zeta \to \mathtt{in}_\zeta]$ is an injection such that

(1) $\gamma(i) < i$ for $i \in \mathtt{out}_\zeta$

(2) if $\zeta(j) = \texttt{<in : <}a_0 \Leftarrow cv\texttt{>>}$ then $\gamma(i) = j$ for some i with $\zeta(i) = \texttt{<out : <}x \Leftarrow cv'\texttt{>>}$, and

(do) if $\mathtt{do?}(cv)$, then $cv' = \mathtt{mkret}(g(f(cv)))$

(ret) if $\mathtt{ret?}(cv)$, then $cv' = cv$.

Then we can find $\pi \in \mathcal{F}(\kappa^{(c)})$ such that $\mathtt{i/o}(\pi) = \zeta$, for $c \in \{a_0, x\}$.

Lemma (ri-2): For $c \in \{a_0, x\}$, if $\pi \in \mathcal{F}(\kappa^{(c)})$, then we can find γ, such that $\mathtt{i/o}(\pi), \gamma$ satisfy the conditions of (**ri-1**).

Before we prove these lemmas we show how they are used to prove the soundness theorem.

Proof (rem.indir): It suffices to show that, for $c, c' \in \{a_0, x\}$, if $\pi \in \mathcal{F}(\kappa^{(c)})$ then we can find $\pi' \in \mathcal{F}(\kappa^{(c')})$ such that $\mathtt{i/o}(\pi) = \mathtt{i/o}(\pi')$. Assume $\pi \in \mathcal{F}(\kappa^{(c)})$. Then by (**ri-2**) we obtain γ such that $\mathtt{i/o}(\pi_c), \gamma$ satisfy the conditions of (**ri-1**). Then by (**ri-1**) we obtain $\pi' \in \mathcal{F}(\kappa^{(c')})$ such that $\mathtt{i/o}(\pi') = \mathtt{i/o}(\pi)$, as required. $\square_{\mathbf{rem.indir}}$

The proof of the lemmas is greatly simplified by showing that any fair computation path for one of the initial configurations can be replaced (preserving fairness and observable trace) by another computation path (for the same initial configuration) that is in a simple canonical form in which the processing of each message received by an internal actor can be thought of as a single multi-step. Thus there are only input, output, and receive transitions to consider. To describe the canonical form, we first define the multi-step transitions $R_0(cv)$ and $R_1(cv, c)$.

Definition (R_0): Let π be a computation for $\kappa^{(c)}$ in which a message $\mathtt{<}a_0 \Leftarrow cv\mathtt{>}$ occurs at some stage. Let I be the maximal event thread at a_0 in π beginning with $\mathtt{<rcv}: a_0, cv\mathtt{>}$. Then $R_0(cv) = [l_i \mid i \in I]$. Thus, if

$$\kappa = \left\langle\!\!\left\langle \, (B_0)_{a_0}, (B_1(c))_{a_1} \mid \{\mathtt{<}a_0 \Leftarrow cv\mathtt{>}\} \, \right\rangle\!\!\right\rangle_x^{a_0}$$

then $\kappa \xrightarrow{R_0(cv)} \kappa'$ by execution transitions focused at a_0 (no input, output, or receive transitions), no execution transitions are enabled in κ'. From the definition of B_0 we can see that

$$\kappa' = \begin{cases} \left\langle\!\!\left\langle \, (B_0)_{a_0}, (B_1(c))_{a_1} \mid \{\mathtt{<}a_1 \Leftarrow cv'\mathtt{>}\} \, \right\rangle\!\!\right\rangle_x^{a_0} & \text{if } \mathtt{do?}(cv) \text{ and } cv' = f(cv) \\ \left\langle\!\!\left\langle \, (B_0)_{a_0}, (B_1(c))_{a_1} \mid \{\mathtt{<}x \Leftarrow cv'\mathtt{>}\} \, \right\rangle\!\!\right\rangle_x^{a_0} & \text{if } \mathtt{ret?}(cv) \text{ and } cv' = cv \\ \left\langle\!\!\left\langle \, (B_0)_{a_0}, (B_1(c))_{a_1} \mid \emptyset \, \right\rangle\!\!\right\rangle_x^{a_0} & \text{if neither } \mathtt{ret?}(cv) \text{ nor } \mathtt{do?}(cv). \end{cases}$$

Note that the definition of R_0 is independent of the particular computation, since the behavior of a_0 does not change.

Definition (R_1): $R_1(cv)$ is obtained from the maximal event thread beginning with $\mathtt{<rcv}: a_1, cv\mathtt{>}$ as in the definition of R_0. Thus, if

$$\kappa = \left\langle\!\!\left\langle \, (B_0)_{a_0}, (B_1(c))_{a_1} \mid \{\mathtt{<}a_1 \Leftarrow cv\mathtt{>}\} \, \right\rangle\!\!\right\rangle_x^{a_0}$$

then $\kappa \xrightarrow{R_1(cv)} \kappa'$ by execution transitions focused at a_1 (no input, output, or receive transitions), and no execution transitions remain enabled in κ'. From the definition of $B_1(c)$ we can see that

$$\kappa' = \left\langle\!\!\left\langle \, (B_0)_{a_0}, (B_1(c))_{a_1} \mid \{\mathtt{<}c \Leftarrow \mathtt{mkret}(g(cv))\mathtt{>}\} \, \right\rangle\!\!\right\rangle_x^{a_0}.$$

Lemma (canon-ri): Let $\pi \in \mathcal{F}(\kappa^{(c)})$. Then we can find $\pi' \in \mathcal{F}(\kappa^{(c)})$ such that $\mathtt{i/o}(\pi) = \mathtt{i/o}(\pi')$ and $\pi' = [\kappa_i \xrightarrow{L_i} \kappa_{i+1} \mid i \in \bowtie]$ where L_i is one of the following:

- an input of the form $\mathtt{<in} : \mathtt{<}a_0 \Leftarrow cv\mathtt{>>}$;
- an output of the form $\mathtt{<out} : \mathtt{<}x \Leftarrow cv\mathtt{>>}$; or
- one of the multi-step transitions $R_0(cv)$ or $R_1(cv)$.

Proof : By (**infinite macro-steps**). □

Proof (ri-1): Assume the conditions of (**ri-1**) hold for ζ, γ. Define π by inserting multi-step transitions: (1) just after each input of a no-op message; and (2) just before each output transition. Suppose $\zeta(j) = \mathtt{<in} : \mathtt{<}a_0 \Leftarrow cv\mathtt{>>}$ where neither $\mathtt{do?}(cv)$ nor $\mathtt{ret?}(cv)$ hold. Then insert $R_0(cv)$ just after j. Suppose $i \in \mathtt{out}_\zeta$, $\gamma(i) = j$, and $\zeta(j) = \mathtt{<in} : \mathtt{<}a_0 \Leftarrow cv\mathtt{>>}$. Then we define the sequence L_i to insert as follows. $L_i = [R_0(cv)] \circ L'$ where L' is empty unless $\mathtt{do?}(cv)$, in which case

$$L' = [R_1(f(cv)), R_0(\mathtt{mkret}(g(f(cv))))], \quad \text{if } c = a_0, \text{ and}$$

$$L = [R_1(f(cv))], \quad \text{if } c = x.$$

It is easy to see that the result of the insertions π is a computation path for $\kappa^{(c)}$ with $\mathtt{i/o}(\pi) = \zeta$. To see that π is fair, note that the only transitions that are enabled and not taken immediately are receives of input messages that cause outputs. This is because any input that causes no output is received immediately, and any message to x that appears in the message pool is output immediately following the multi-step transition that enables it. For the remaining input messages there is a corresponding output by the conditions on γ and its receipt occurs in the multi-step □

Proof (ri-2): Suppose $\pi \in \mathcal{F}(\kappa^{(c)})$ and let $\zeta = \mathtt{i/o}(\pi)$. By (**cannon-ri**) we may assume $\pi = [\kappa_i \xrightarrow{L_i} \kappa_{i+1} \mid i \in \bowtie]$ where L_i is either an input, an output, or one of the multi-step transitions $R_j(cv)$. Thus the interesting configurations in π only differ in the message component. We annotate messages with the index of the input event of the causing message. An input message causes itself, and the cause annotation of the message, if any, generated by $R_j(cv)$ is the same as the annotation of the message processed. Thus, ignoring the unchanging parts of the configurations:

$$\mu_i \xrightarrow{\mathtt{<in} : \mathtt{<}a_0 \Leftarrow cv\mathtt{>>}} \mu_i \cup \{\mathtt{<}a_0 \Leftarrow cv\mathtt{>}^i\}$$

$$\mu_i\{\mathtt{<}a_0 \Leftarrow cv\mathtt{>}^j\} \xrightarrow{R_0(cv)} \mu_i \cup \begin{cases} \{\mathtt{<}a_1 \Leftarrow f(cv)\mathtt{>}^j\} & \text{if } \mathtt{do?}(cv) \\ \{\mathtt{<}x \Leftarrow cv\mathtt{>}^j\} & \text{if } \mathtt{ret?}(cv) \\ \{\} & \text{otherwise} \end{cases}$$

$$\mu_i\{\mathtt{<}a_1 \Leftarrow cv\mathtt{>}^j\} \xrightarrow{R_1(cv)} \mu_i \cup \{\mathtt{<}c \Leftarrow g(cv)\mathtt{>}^j\}$$

For $i \in \mathtt{out}_\varphi$, let i' be the index of the output transition in π and let j' be the annotation of the message output at i. Let j be the index of the input transition j' in ζ and define $\gamma(i) = j$. Then it is easy to see from the definitions of the multi-step transitions that ζ, γ satisfies the conditions of (**ri-1**). □

6.2. Varieties of Recursion as Patterns of Message Passing

We define a trivial recursive procedure, **cd**, which takes a number, counts down to 0 and returns 0. We then define various actors that have the count-down behavior. An actor with count-down behavior responds to a message containing a number and a caller, by counting from that number down to 0 and sending a message containing 0 to the caller. The behaviors B_0 and B_1 do this by simply applying **cd** to the number. The difference is that B_0 sends the reply before becoming ready for the next message, while B_1 becomes ready for the next message immediately, leaving the work of the current message to a clone. This increases the opportunity for parallelism – i.e the number of actors with work to do. B_2 implements the count-down recursion by decrementing the number and sending itself a message with the decremented number and the caller. When the number reaches 0, B_2 replys to the caller. In this way an actor with behavior B_2 may interleave working on requests from various callers. B_3 creates a new actor to handle each message. Again, this is a way of increasing parallelism. The new actor has behavior B_2. It could equally well have been given behavior B_0 or B_1. Finally B_4 also creates a new actor for each message. This new actor implements a loop with loop variable i (the current value of the number counting down). It goes once around the loop each time it receives a 'tick' (any message at all).

In fact, these actors are all equivalent to one that, for messages of the appropriate form, returns 0 to the caller immediately. The point is not to analyze the functional behavior, but to analyze the message patterns used to express the recursion. The examples can easily be elaborated to behaviors that compute more interesting functions using the same recursion patterns. We chose this simple case to focus on methods for proving equivalence of the various recursion patterns.

To avoid anomalies, we choose behaviors which ignore messages that are not of the expected form, i.e. messages that are not pairs whose first component is a number. This is done using the test **bad?**.

$$\mathtt{bad?} = \lambda x.\mathtt{if}(\mathtt{pr}(x), \mathtt{if}(\mathtt{isnat}(1^{\mathtt{st}}(x)), \mathtt{nil}, \mathtt{t}), \mathtt{t})$$

$$\mathtt{cd} = \mathtt{rec}(\lambda d \lambda x.\mathtt{if}(\mathtt{eq}(x, 0), 0, d(x-1)))$$

$$B_0 = \mathtt{rec}(\lambda b.\lambda m.\ \mathtt{if}(\mathtt{bad?}(m),\ \mathtt{become}(b),\ \mathtt{let}\{n := 1^{\mathtt{st}}(m), c := 2^{\mathtt{nd}}(m)\}\mathtt{seq}(\mathtt{send}(c, \mathtt{cd}(n)), \mathtt{become}(b))))$$

$$B_1 = \mathtt{rec}(\lambda b.\lambda m.\ \mathtt{if}(\mathtt{bad?}(m),\ \mathtt{become}(b),\ \mathtt{let}\{n := 1^{\mathtt{st}}(m), c := 2^{\mathtt{nd}}(m)\}\mathtt{seq}(\mathtt{become}(b), \mathtt{send}(c, \mathtt{cd}(n)))))$$

$$B_2 = \lambda a.\mathtt{rec}(\lambda b.\lambda m.$$
$$\mathtt{if}(\mathtt{bad?}(m),$$
$$\mathtt{become}(b),$$
$$\mathtt{let}\{n := \mathtt{1^{st}}(m), c := \mathtt{2^{nd}}(m)\}$$
$$\mathtt{seq}(\mathtt{become}(b),$$
$$\mathtt{if}(\mathtt{eq}(n,0), \mathtt{send}(c,n), \mathtt{send}(a, \mathtt{pr}(n-1,c))))))$$

$$B_3 = \mathtt{rec}(\lambda b.\lambda m.$$
$$\mathtt{if}(\mathtt{bad?}(m),$$
$$\mathtt{become}(b),$$
$$\mathtt{let}\{n := \mathtt{1^{st}}(m), c := \mathtt{2^{nd}}(m)\}$$
$$\mathtt{seq}(\mathtt{become}(b),$$
$$\mathtt{let}\{a := \mathtt{newadr}()\}$$
$$\mathtt{seq}(\mathtt{initbeh}(a, B_2(a)), \mathtt{send}(a, \mathtt{pr}(n,c))))))$$

$$B_{4x} = \lambda a.\lambda c.\mathtt{rec}(\lambda b.\lambda i.\lambda m.$$
$$\mathtt{if}(\mathtt{eq}(i,0),$$
$$\mathtt{send}(c,0),$$
$$\mathtt{seq}(\mathtt{become}(b(i-1)), \mathtt{send}(a, \mathtt{nil}))))$$

$$B_4 = \mathtt{rec}(\lambda b.\lambda m.$$
$$\mathtt{if}(\mathtt{bad?}(m),$$
$$\mathtt{become}(b),$$
$$\mathtt{let}\{n := \mathtt{1^{st}}(m), c := \mathtt{2^{nd}}(m)\}$$
$$\mathtt{seq}(\mathtt{become}(b),$$
$$\mathtt{let}\{a := \mathtt{newadr}()\}$$
$$\mathtt{seq}(\mathtt{initbeh}(a, B_{4x}(a,c,n)), \mathtt{send}(a, \mathtt{nil})))))$$

Define the configurations $\kappa^{(j)}$ for $j \leq 4$ by

$$\kappa^{(j)} = \left\langle\!\!\left\langle\ (B_j)_a \ \middle|\ \emptyset \right\rangle\!\!\right\rangle_{\emptyset}^{\{a\}} \quad \text{for} \quad j \in \{0,1,3,4\}$$

$$\kappa^{(2)} = \left\langle\!\!\left\langle\ (B_2(a))_a \ \middle|\ \emptyset \right\rangle\!\!\right\rangle_{\emptyset}^{\{a\}}$$

Theorem (cd-c): The configurations $\kappa^{(j)}$ for $j \leq 4$ are interaction equivalent (and hence observationally equivalent).

Define the expressions $e^{(j)}$ for $j \leq 4$ by

$$e^{(j)} = \mathtt{let}\{a := \mathtt{newadr}\}\mathtt{initbeh}(a, B_j) \quad \text{for} \quad j \in \{0,1,3,4\}$$

$$e^{(2)} = \mathtt{let}\{a := \mathtt{newadr}\}\mathtt{initbeh}(a, B_2(a))$$

By (**io-exp**) we have the following corollary, expressing the equivalences in terms of expressions constructing the various actor systems.

Corollary (cd-e): The expressions $e^{(j)}$ for $j \leq 4$ are observationally equivalent.

The proof of (**cd-c**) is similar to that for the indirection removal example. It relies on two lemmas that show that the observable behavior is characterized by the possible $\mathtt{i/o}$ paths and maps from output to causing input.

Lemma (cd-1): Let ζ be a (possibly infinite) sequence of input/output transitions:

$$\zeta = [l_i \mid i < \bowtie] \qquad \text{where } l_i \in \mathtt{<in} : \mathtt{<}a \Leftarrow \mathtt{pr}(\mathbb{N}, \mathbb{A}\mathrm{d})\mathtt{>>} \cup \mathtt{<out} : \mathtt{<}\mathbb{A}\mathrm{d} \Leftarrow 0\mathtt{>>} \text{ for } i < \bowtie$$

and define

$$\mathtt{in}_\zeta = \{i \mid \zeta(i) \in \mathtt{<in} : \mathtt{<}a \Leftarrow \mathtt{pr}(\mathbb{N}, \mathbb{A}\mathrm{d})\mathtt{>>}\}$$
$$\mathtt{out}_\zeta = \{i \mid \zeta(i) \in \mathtt{<out} : \mathtt{<}\mathbb{A}\mathrm{d} \Leftarrow 0\mathtt{>>}\}$$

Suppose $\gamma \in [\mathtt{out}_\zeta \to \mathtt{in}_\zeta]$ is an injection such that

(1) $\gamma(i) < i$ with $i \in \mathtt{out}_\zeta$, and

(2) if $\zeta(j) = \mathtt{<in} : \mathtt{<}a \Leftarrow \mathtt{pr}(n, c)\mathtt{>>}$ with $n \in \mathbb{N}$, then $j = \gamma(i)$ for some i with $\zeta(i) = \mathtt{<out} : \mathtt{<}c \Leftarrow 0\mathtt{>>}$.

Then we can find $\pi \in \mathcal{F}(\kappa^{(j)})$ such that $\mathtt{i/o}(\pi) = \zeta$, for each case $j \leq 4$.

We interpret (1) to mean that γ maps each output to its causing input, and (2) to mean that each input of the correct form causes an output.

Lemma (cd-2): For each case $j \leq 4$, if $\pi \in \mathcal{F}(\kappa^{(j)})$, then we can find γ, such that $\mathtt{i/o}(\pi), \gamma$ satisfy the conditions of (**cd-1**).

The equivalence of configurations $\kappa^{(j)}$ for $j \leq 4$, follows easily from (**cd-1,cd-2**) as in the previous example.

Proof (cd-c): It suffices to show that for any $i, j \leq 4$, if $\pi^i \in \mathcal{F}(\kappa^{(i)})$ then we can find $\pi^j \in \mathcal{F}(\kappa^{(j)})$ such that $\mathtt{i/o}(\pi^i) = \mathtt{i/o}(\pi^j)$. Assume $\pi^i \in \mathcal{F}(\kappa^{(i)})$. Then by (**cd-2**) we obtain γ such that $\mathtt{i/o}(\pi^i), \gamma$ satisfy the conditions of (**cd-1**). Then by (**cd-1**) we obtain $\pi^j \in \mathcal{F}(\kappa^{(j)})$ such that $\mathtt{i/o}(\pi^j) = \mathtt{i/o}(\pi^i)$, as required. $\square_{\mathbf{cd-c}}$

Again, the key to proving the lemmas is to find a suitable canonical form for the fair computations of each initial configuration. The points of interest are input, output, receive, and count-down steps. We define the multi-step transitions $R_j(a, m, a')$ and $Cd_j(a', c, n)$. The parameter a in R_j and a' in Cd_j are the focus of the multi-step transitions. The parameter a' in R_j is the actor that will carry out the count down (a when j is 0 or 2 and a clone or newly created actor when j is 1, 3, or 4.

Definition (R_j, Cd_j): If $\mathtt{bad?}(cv)$, then

$$\left\langle\!\left\langle\, \alpha, (B_j)_a \mid \mu, \mathtt{<}a \Leftarrow cv\mathtt{>} \,\right\rangle\!\right\rangle^a_\chi \xrightarrow{R_j(a,cv,a)} \left\langle\!\left\langle\, \alpha, (B_j)_a \mid \mu \,\right\rangle\!\right\rangle^a_\chi$$

Otherwise the multi-step transitions are defined by rules of the form

$$\alpha_0 \mid \mu_0 \xrightarrow{L} \alpha_1 \mid \mu_1$$

where α_0 will specify the state of a single actor, the focus of L, and μ_0 specifies the messages consumed, if any. α_1 will specify the resulting state of the focus actor, as well as any newly created/cloned actors. μ_1 will be the messages generated, if any. Each such rule abbreviates a transition

$$\left\langle\!\!\left\langle\, \alpha \cup \alpha_0 \;\middle|\; \mu \cup \mu_0 \,\right\rangle\!\!\right\rangle^{a}_{\chi} \xrightarrow{L} \left\langle\!\!\left\langle\, \alpha \cup \alpha_1 \;\middle|\; \mu \cup \mu_1 \,\right\rangle\!\!\right\rangle^{a}_{\chi}$$

where $\mathrm{Dom}(\alpha) \cap \mathrm{Dom}(\alpha_j) = \emptyset$ and $\cup$ on messages is multiset union. L is defined as the least multi-step transition leading from the left hand configuration of the rule to the right hand configuration.

$E_0(c,n) = \mathtt{seq}(\mathtt{send}(c, \mathtt{cd}(n)), \mathtt{become}(B_0))$

$(B_0)_a \mid \mathtt{<}a \Leftarrow \mathtt{pr}(n,c)\mathtt{>} \quad \xrightarrow{R_0(a,\mathtt{pr}(n,c),a)} \quad [E_0(c,n)]_a \mid \emptyset$

$[E_0(c,n)]_a \mid \emptyset \quad \xrightarrow{Cd_0(a,c,n)} \quad \begin{cases} (B_0)_a \mid \mathtt{<}c \Leftarrow 0\mathtt{>} & \text{if } n = 0 \\ [E_0(c,n-1)]_a \mid \emptyset & \text{if } n > 0 \end{cases}$

$E_1(c,n) = \mathtt{send}(c, \mathtt{cd}(n))$

$(B_1)_a \mid \mathtt{<}a \Leftarrow \mathtt{pr}(n,c)\mathtt{>} \quad \xrightarrow{R_1(a,\mathtt{pr}(n,c),a')} \quad (B_1)_a, [E_1(c,n)]_{a'} \mid \emptyset$

a' fresh

$[E_1(c,n)]_{a'} \mid \emptyset \quad \xrightarrow{Cd_1(a',c,n)} \quad \begin{cases} [\mathtt{nil}]_{a'} \mid \mathtt{<}c \Leftarrow 0\mathtt{>} & \text{if } n = 0 \\ [E_1(c,n-1)]_{a'} \mid \emptyset & \text{if } n > 0 \end{cases}$

$(B_2(a))_a \mid \mathtt{<}a \Leftarrow \mathtt{pr}(n,c)\mathtt{>} \quad \xrightarrow{R_2(a,\mathtt{pr}(n,c),a)} \quad (B_2)_a \mid m$

where $m = \mathtt{<}c \Leftarrow 0\mathtt{>}$ if $n = 0$, and $m = \mathtt{<}a \Leftarrow \mathtt{pr}(n-1,c)\mathtt{>}$ if $n > 0$

$Cd_2(a,c,n) = R_2(a, \mathtt{pr}(c,n), a)$

$(B_3)_a \mid \mathtt{<}a \Leftarrow \mathtt{pr}(n,c)\mathtt{>} \quad \xrightarrow{R_3(a,\mathtt{pr}(n,c),a)} \quad (B_3)_a, (B_2(a'))_{a'} \mid \mathtt{<}a' \Leftarrow \mathtt{pr}(n,c)\mathtt{>}$

a' fresh

$Cd_3(a',c,n) = R_2(a', \mathtt{pr}(c,n), a')$

$(B_4)_a \mid \mathtt{<}a \Leftarrow \mathtt{pr}(n,c)\mathtt{>} \quad \xrightarrow{R_4(a,\mathtt{pr}(n,c),a')} \quad (B_4)_a, (B_{4x}(a',c,n))_{a'} \mid \mathtt{<}a' \Leftarrow \mathtt{nil}\mathtt{>}$

a' fresh

$(B_{4x}(a',c,0))_{a'} \mid \mathtt{<}a' \Leftarrow \mathtt{nil}\mathtt{>} \quad \xrightarrow{Cd_4(a',c,0)} \quad [\mathtt{nil}]_{a'} \mid \mathtt{<}c \Leftarrow 0\mathtt{>}$

$(B_{4x}(a',c,n+1))_{a'} \mid \mathtt{<}a' \Leftarrow \mathtt{nil}\mathtt{>} \quad \xrightarrow{Cd_4(a',c,n+1)} \quad (B_{4x}(a',c,n))_{a'} \mid \mathtt{<}a' \Leftarrow \mathtt{nil}\mathtt{>}$

Lemma (canon-cd): Let $\pi \in \mathcal{F}(\kappa^{(j)})$. Then we can find $\pi' \in \mathcal{F}(\kappa^{(j)})$ such that $\mathtt{i/o}(\pi) = \mathtt{i/o}(\pi')$ and $\pi' = [\kappa_i \xrightarrow{L_i} \kappa_{i+1} \mid i \in \bowtie]$ where L_i is one of the following:

- an input of the form $\texttt{<in}:\texttt{<}a \Leftarrow cv\texttt{>>}$;
- an output of the form $\texttt{<out}:\texttt{<}c \Leftarrow 0\texttt{>>}$; or
- one of the multi-step transitions $R_j(a, n, a')$ or $Cd_j(a', c, n)$).

Proof : By (**infinite macro-steps**). □

Proof (cd-1):

Assume ζ, γ satisfy the conditions of (**cd-1**). For $k \leq 4$ we construct $\pi^{(k)}$ by inserting multi-step transitions as follows. Suppose $\zeta(j) = \texttt{<in}:\texttt{<}a_0 \Leftarrow cv\texttt{>>}$ where $\texttt{bad?}(cv)$ holds. Then insert $R_j(a, cv, a)$ just after this transition. Suppose $i \in \texttt{out}_\zeta$, $\gamma(i) = j$, and $\zeta(j) = \texttt{<in}:\texttt{<}a \Leftarrow \texttt{pr}(n, c)\texttt{>>}$. Then for $\pi^{(k)}$ insert L_k just before $\zeta(i)$ where L_k is defined as follows.

$$L_0 = [R_0(a, \texttt{pr}(n, c), a), Cd_0(a, c, n-k) \mid k \leq n]$$
$$L_2 = [R_2(a, \texttt{pr}(n-k, c), a) \mid k \leq n]$$
$$L_j = [R_j(a, \texttt{pr}(n, c), a'), Cd_j(a', c, n-k) \mid k \leq n] \qquad \text{for } j \in \{1, 3, 4\}, \text{ where } a' \text{ is fresh}$$

Clearly $\pi^{(k)}$ is a complete computation from $\kappa^{(k)}$. So we need only show fairness. From the definitions of R_k, Cd_k it is easy to see that any enabled transition other than receipt of an external message by a or output of a message sent to an external actor is enabled and taken during L_k. At any point there is at most one message waiting to be sent to an external actor. The output transition occurs just after the generating L_k. Finally for any $j \in \texttt{in}_\zeta$ either $\zeta(i)$ is bad and is received (discarded) immediately, or is some $i \in \texttt{out}_\zeta$ such that $\gamma(i) = j$ and hence the receipt of that message by a initiates the L_k inserted just before $\zeta(i)$. $\square_{\mathbf{cd-1}}$

Proof (cd-2): Assume $\pi \in \mathcal{F}(\kappa^{(k)})$, and let $\zeta = \texttt{i/o}(\pi)$. By (**cannon-cd**) we may assume $\pi = [\kappa_i \xrightarrow{L_i} \kappa_{i+1} \mid i \in \bowtie]$ where $\kappa_i = \left\langle\!\!\left\langle\, \alpha_i \mid \mu_i \,\right\rangle\!\!\right\rangle^a$, and L_i is either an input, an output, or one of the multi-step transitions $R_j(a, cv, a')$, or $Cd_j(a', c, n)$. We want to construct γ to satisfy the conditions of (**cd-1**). To do this we form an annotated version, π', of π in which each message (occurrence) and each executing actor state, in each configuration is tagged with the index of the input event causing this occurrence or state (via a chain of receives and sends). This is done by elaborating the (multi-step) transition rules as follows.

$$\left\langle\!\!\left\langle\, \alpha_i \mid \mu_i \,\right\rangle\!\!\right\rangle^a_\chi \xrightarrow{\texttt{<in}:m\texttt{>}} \left\langle\!\!\left\langle\, \alpha_i \mid \mu_i, m^i \,\right\rangle\!\!\right\rangle^a_{\chi'}$$

$$(B_0)_a \mid \texttt{<}a \Leftarrow \texttt{pr}(n, c)\texttt{>}^i \xrightarrow{R_0(a, \texttt{pr}(n,c), a)} [E_0(c, n)]^i_a \mid \emptyset$$

$$[E_0(c, n)]^i_a \mid \xrightarrow{Cd_0(a,c,n)} \begin{cases} (B_0)_a \mid \texttt{<}c \Leftarrow 0\texttt{>}^i & \text{if } n = 0 \\ [E_0(c, n-1)]^i_a \mid \emptyset & \text{if } n > 0 \end{cases}$$

$$(B_1)_a \mid \texttt{<}a \Leftarrow \texttt{pr}(n, c)\texttt{>}^i \xrightarrow{R_1(a, \texttt{pr}(n,c), a')} (B_1)_a, [E_1(c, n)]^i_{a'} \mid \emptyset$$

$$[E_1(c, n)]^i_{a'} \mid \emptyset \xrightarrow{Cd_1(a', c, n)} \begin{cases} [\texttt{nil}]_{a'} \mid \texttt{<}c \Leftarrow 0\texttt{>}^i & \text{if } n = 0 \\ [E_1(c, n-1)]^i_{a'} \mid \emptyset & \text{if } n > 0 \end{cases}$$

$$(B_2(a))_a \mid \texttt{<}a \Leftarrow \texttt{pr}(n, c)\texttt{>}^i \xrightarrow{R_2(a, \texttt{pr}(n,c), a)} (B_2)_a \mid m^i$$

$$(B_3)_a \mid \texttt{<}a \Leftarrow \mathbf{pr}(n,c)\texttt{>}^i \xrightarrow{R_3(a,\mathbf{pr}(n,c),a)} (B_3)_a, (B_2(a'))_{a'} \mid \texttt{<}a' \Leftarrow \mathbf{pr}(n,c)\texttt{>}^i$$

$$(B_4)_a \mid \texttt{<}a \Leftarrow \mathbf{pr}(n,c)\texttt{>}^i \xrightarrow{R_1(a,\mathbf{pr}(n,c),a')} (B_4)_a, (B_{4x}(a',c,n))_{a'} \mid \texttt{<}a' \Leftarrow \mathbf{nil}\texttt{>}^i$$

$$(B_{4x}(a',c,0))_{a'} \mid \texttt{<}a' \Leftarrow \mathbf{nil}\texttt{>}^i \xrightarrow{Cd_4(a',c,0)} [\mathbf{nil}]_{a'} \mid \texttt{<}c \Leftarrow 0\texttt{>}^i$$

$$(B_{4x}(a',c,n+1))_{a'} \mid \texttt{<}a' \Leftarrow \mathbf{nil}\texttt{>}^i \xrightarrow{Cd_4(a',c,n+1)} (B_{4x}(a',c,n))_{a'} \mid \texttt{<}a' \Leftarrow \mathbf{nil}\texttt{>}^i$$

For $i \in \mathbf{out}_\varphi$, let i' be the index of the output transition in π and let j' be the annotation of the message output at i. Let j be the index of the input transition j' in ζ and define $\gamma(i) = j$. Then it is easy to see from the definitions of the multi-step transitions that ζ, γ satisfies the conditions of (**ri-1**). $\square_{\mathbf{cd-2}}$

Using the analysis of the count-down behaviors, and the methods for establishing expression equivalence we can in fact show that the behaviors (lambda abstractions) B_j for $j \in \{0,1,3,4\}$ are equivalent. We can not directly compare B_2 as it requires an additional argument. But, for example it is easy to see that $\lambda x.B_0$ is not equivalent to B_2, because the equivalence of $\kappa^{(2)}$ to the others depended on applying B_2 to the address of the actor in which it occurs. An observing context that distinguishes between $\lambda x.B_0$ and B_2 is

$$\left\langle\!\!\left\langle (\lambda x.\mathbf{event}())_{a_0}, (\lambda x.\mathbf{sink})_{a_1}, [\mathbf{app}(\mathbf{app}(\bullet, a_0), \mathbf{pr}(1, a_1))] \mid \emptyset \right\rangle\!\!\right\rangle$$

An alternative to filtering out messages of the wrong form by explicit tests in the behaviors B_i is to use a filter actor as receptionist. For this we would obtain B_i' from B_i by omitting the **bad**? test; replacing $\mathbf{if}(\mathbf{bad?}(m), \mathbf{become}(b), b_i)$ by b_i in each case. In the system configuration we would replace the working receptionist actor by a filter actor that does the checking and forwards messages that pass the test to the working actor. For example, define

$$\mathbf{cdin} = \lambda z.\mathbf{rec}(\lambda b.\lambda m.\mathbf{seq}(\mathbf{become}(b), \mathbf{if}(\mathbf{bad?}(m), \mathbf{nil}, \mathbf{send}(z, m))))$$

and replace the κ^j definitions by

$$\kappa^{(j)} = \left\langle\!\!\left\langle (B_j')_{a'}, (\mathbf{cdin}(a'))_a \mid \emptyset \right\rangle\!\!\right\rangle_{\emptyset}^{\{a\}}$$

(with additional arg of a' to B_j' when $j = 2$). This has the advantage that the test is only done once per message and is part of the interface to the system rather than the working part. However the behaviors B_j' are not going to be equivalent in a general context.

7. Proving Expression Equivalence

In this section we develop three instances of a general method for proving expressions observationally equivalent. The method is a minor generalization of the techniques used to prove the (**ciu**) theorem (a form of context lemma) in [18].

(1) The first instance of the method treats equivalence of expressions that have a common reduct – i.e. expressions that step in 0 or more steps to the same expression having the same effects (sends, becomes, creation of new actors, initializing new actors). This is called the *common reduct case.*

(2) The second instance of the method is an elaboration of the first, treating expressions that reduce to lambda abstractions that are application equivalent – i.e. have a common reduct when applied to any value. This is called the *two stage reduction case.*

(3) The third instance of the method treats equivalence of reduction contexts. This is called the *equivalence of reduction contexts case.*

We provide examples of the use of these techniques by using them to establish the equational laws stated in §4. We note that these methods and proofs can easily be modified to prove interaction equivalence, instead of observational equivalence of expressions. For simplicity we have only treated observational equivalence here.

7.1. The General Method

Each of these three methods is based on the idea of using *configuration templates* to establish a correspondence between the fair computations of configurations containing the entities to be proved equivalent. A configuration template is simply a configuration with holes, i.e. schematic variables, that may be instantiated by various sorts of syntactic entities. Observing contexts correspond to a special case of configuration templates.

The first step then is to choose a class of configuration templates CT such that $e_0 \cong e_1$ if $Obs(ct[e_0]) = Obs(ct[e_1])$ for all templates $ct \in CT$. To establish the equality of observations, it is sufficient to construct, for each $\pi_0 \in \mathcal{F}(ct[e_0])$, a $\pi_1 \in \mathcal{F}(ct[e_1])$ such that $obs(\pi_0) = obs(\pi_1)$ and conversely. We call such a construction a *path correspondence.* The crucial fact concerning configuration templates is that one can compute symbolically with them in the sense that computation is parametric in the holes. Perhaps an appropriate name for this form of computation is *uniform computation or reduction.*

A suitable class of configuration templates is obtained by extending each syntactic class to allow holes and defining appropriate notions of hole filling. Decomposition theorems and schematic reduction rules are then developed. In each of the three methods the only essential difference is the type and number of holes needed:

(1) For the common reduct case we define templates by adding holes for expressions. We call these holes *expression holes*, and they are denoted by $\circ$.

(2) For the two stage reduction case we need not only expression holes but also a countable family of holes for lambda abstractions. We call these holes *abstraction holes* and they are denoted by $\triangleright_j$ for $j \in \mathbb{N}$. Note that these holes are filled by values, specifically by lambda abstractions, not simply by expressions.

(3) For the equivalence of reduction contexts we need an entirely new class of holes. These holes will be for reduction contexts. We call them *reduction context holes*, and denote them by $\diamond$. Note that these holes will be filled by reduction contexts and are *not* to be confused with redex holes. As far as we are aware the introduction of holes that are filled by contexts is completely novel.

For each case, syntactic classes X are annotated with the signs of the sorts of holes they contain: ${}^{\circ}X$ for expression holes; ${}^{\circ\triangleright}X$ for expression and lambda abstraction holes; and ${}^{\diamond}X$ for reduction context holes. We prefix the names of these classes by E-, LE-, or R- respectiveley. Thus E-expressions are expression templates with holes for expressions, ${}^{\circ}\mathbb{E}$ is the set of E-expressions, and we let ${}^{\circ}e$ range over ${}^{\circ}\mathbb{E}$. Similarly for other syntactic classes and hole types.

The idea underlying the construction of a path correspondence to establish equivalence is the same for each of the three cases. It relies on the ability to localize differences in computations as multi-steps, and to use holes to formalize the aspects of computation that are independent of the local differences. Consider the case of proving expressions equivalent using templates with expression holes. We consider fair computation paths starting from an E-configuration with holes filled by one of the expressions, say e_0. For each such path, π_0, we show how to obtain a sequence of E-configurations satisfying two conditions. The first is that filling the holes in the sequence of E-configurations with e_0 (and filling in transition labels) yields π_0. The second is that filling the holes in the sequence of E-configurations with e_1 (and filling in transition labels) yields a fair computation path with the same observation. The other two cases are simple variations on this idea.

One of the keys ideas in uniform computation is to insure that transitions commute with hole filling; except of course when the *hole is touched*, i.e. information about the contents of the hole is required to carry out the step. Consider the schematic redex $\mathtt{app}(\lambda x.\bullet, v)$. We need a notation that allows us to carry out this reduction in such a way that filling the hole and the reducing gives the same result as reducing and then filling the hole. For this purpose we associate with each hole a substitution to be applied when the hole is filled. The domain of the substitution also determines the variables of an expression that are trapped at the hole. This localizes trapping and allows renaming of lambda-variables even in the presence of holes (which is not the case for traditional notions of expression context). A detailed development of this notation can be found in [30, 32]. We use $\circ[{}^{\circ}\sigma]$ to denote an expression hole with associated substitution ${}^{\circ}\sigma$ (which may in turn have expressions holes in its range), a similar notation holds for the other classes of holes: $\triangleright_j[{}^{\circ\triangleright}\sigma]$ for abstraction holes, and $\diamond[{}^{\diamond}\sigma]$ for reduction holes.

To simplify definitions of syntactic classes we treat `app` on a par with elements of $\mathbb{F}_2$. We use Θ_n for syntactic operations of arity n, and Θ^e_n to indicate the operations of the extended language (i.e. Θ_0 extended to include `event`). Thus:

Definition (Θ_n Θ^e_n):

$$\Theta_2 = \mathbb{F}_2 \cup \{\mathtt{app}\} \qquad \Theta_n = \mathbb{F}_n \qquad \text{for } n \neq 2$$
$$\Theta^e_0 = \Theta_0 \cup \{\mathtt{event}\} \qquad \Theta^e_n = \Theta_n \qquad \text{for } n \neq 0$$

7.2. Common Reduct Case

We now treat the common reduct case in depth. The other two cases follow in the same manner and we allow ourselves to be a little more terse.

7.2.1. E-Syntax

As mentioned above, syntactic classes, X, with expression holes are indicated by the mark ${}^\circ X$. Metavariables ranging over these classes are indicated by the same mark, and we prefix the names of these classes by E-. Thus we have E-expressions where ${}^\circ e$ ranges over ${}^\circ\mathbb{E}$, E-configurations where ${}^\circ\kappa$ ranges over ${}^\circ\mathbb{K}$, etc. We first define the E- analogs of expression, value expression, and value substitution.

Definition (${}^\circ\mathbb{E}$ ${}^\circ\mathbb{V}$ ${}^\circ\mathbb{S}$)**:**

$$
\begin{aligned}
{}^\circ\mathbb{V} &= \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.{}^\circ\mathbb{E} \cup \mathbf{pr}({}^\circ\mathbb{V}, {}^\circ\mathbb{V}) \\
{}^\circ\mathbb{E} &= {}^\circ\mathbb{V} \cup \Theta^e_n({}^\circ\mathbb{E}^n) \cup \circ[{}^\circ\mathbb{S}] \\
{}^\circ\mathbb{S} &= \mathbb{X} \xrightarrow{\mathrm{f}} {}^\circ\mathbb{V}
\end{aligned}
$$

λ is the only binding operator, and free variables of E-expressions are defined as follows:

Definition (FV(${}^\circ e$) FV(${}^\circ\sigma$))**:**

$$
\mathrm{FV}({}^\circ e) = \begin{cases} \mathrm{FV}({}^\circ\sigma) & \text{if } {}^\circ e = \circ[{}^\circ\sigma] \\ \{{}^\circ e\} & \text{if } {}^\circ e \in \mathbb{X} \\ \mathrm{FV}({}^\circ e_0) - \{z\} & \text{if } {}^\circ e = \lambda z.{}^\circ e_0 \\ \mathrm{FV}({}^\circ e_0) \cup \ldots \cup \mathrm{FV}({}^\circ e_n) & \text{if } {}^\circ e = \theta({}^\circ e_1, \ldots, {}^\circ e_n) \text{ and } \theta \in \Theta^e_n \end{cases}
$$

$$
\mathrm{FV}({}^\circ\sigma) = \bigcup_{x \in \mathrm{Dom}({}^\circ\sigma)} \mathrm{FV}({}^\circ\sigma(x))
$$

The variables in the domain of occurrences of ${}^\circ\sigma$ are neither free or bound. In particular, renaming of bound variables only applies to the range of a substitution associated with a hole, not to its domain.

Definition (${}^\circ e[{}^\circ\sigma]$ ${}^\circ\sigma_1 \odot {}^\circ\sigma_2$)**:** Substitution is extended to E-expressions as follows:

$$
{}^\circ e[{}^\circ\sigma] = \begin{cases} \circ[{}^\circ\sigma \odot {}^\circ\sigma'] & \text{if } {}^\circ e = \circ[{}^\circ\sigma'] \\ {}^\circ e & \text{if } {}^\circ e \in \mathbb{X} - \mathrm{Dom}({}^\circ\sigma) \\ {}^\circ\sigma({}^\circ e) & \text{if } {}^\circ e \in \mathrm{Dom}({}^\circ\sigma) \\ \lambda z.{}^\circ e_0[{}^\circ\sigma\rfloor(\mathrm{Dom}({}^\circ\sigma) - \{z\})] & \text{if } {}^\circ e = \lambda z.{}^\circ e_0 \text{ and } z \notin \mathrm{FV}({}^\circ\sigma) \\ \theta({}^\circ e_0[{}^\circ\sigma], \ldots, {}^\circ e_n[{}^\circ\sigma]) & \text{if } {}^\circ e = \theta({}^\circ e_1, \ldots, {}^\circ e_n) \text{ and } \theta \in \Theta^e_n \end{cases}
$$

$$
{}^\circ\sigma_1 \odot {}^\circ\sigma_2 = \lambda x \in \mathrm{Dom}({}^\circ\sigma_2).{}^\circ\sigma_2(x)[{}^\circ\sigma_1]
$$

As defined here substitution is a partial operation. Using renaming substitutions we can define α renaming in the usual way. We consider E-expressions (and entities containing them) to be equivalent if the differ only by α renaming. Thus, for any substitution we can alway choose an α variant so that substitution is defined.

Expression hole filling is defined by induction on the structure of ${}^\circ e$. Like substitution, we avoid capture of free variables in e by lambda binding. We let ${}^\circ e[\circ := e]$ be the result of filling expression holes in ${}^\circ e$ with e. All capture is done at hole occurrences.

Definition (${}^\circ e[\circ := e]$)**:**

$${}^\circ e[\circ := e] = \begin{cases} e[\lambda x \in \mathrm{Dom}({}^\circ\sigma).{}^\circ\sigma(x)[\circ := e]] & \text{if } {}^\circ e = \circ[{}^\circ\sigma] \\ \theta({}^\circ e_1[\circ := e], \ldots, {}^\circ e_n[\circ := e]) & \text{if } {}^\circ e = \theta({}^\circ e_1, \ldots, {}^\circ e_n) \text{ and } \theta \in \Theta^e_n \\ {}^\circ v & \text{if } {}^\circ e = {}^\circ v \in \mathbb{A}\mathrm{t} \cup \mathbb{X} \\ \lambda x.{}^\circ e[\circ := e] & \text{if } {}^\circ e = \lambda x.{}^\circ e \text{ and } x \text{ not free in } e \end{cases}$$

Lemma (fil-subst): Hole filling and substitution commute.

$${}^\circ e[{}^\circ\sigma][\circ := e'] = {}^\circ e[\circ := e'][{}^\circ\sigma[\circ := e']]$$

if $\mathrm{Dom}({}^\circ\sigma) \cap \mathrm{FV}(e') = \emptyset$.

Proof : By induction on the structure of ${}^\circ e$. We assume the names of bound variables in ${}^\circ e$ have been chosen not to conflict with any free variables in e', or the range of ${}^\circ\sigma$, or the domain of ${}^\circ\sigma$. As examples, we consider the cases where ${}^\circ e$ is a lambda abstraction or a hole. If ${}^\circ e = \lambda z.{}^\circ e_0$ then

$$\begin{aligned} &{}^\circ e[{}^\circ\sigma][\circ := e'] \\ &\quad = (\lambda z.{}^\circ e_0[{}^\circ\sigma])[\circ := e'] \\ &\quad = \lambda z.({}^\circ e_0[{}^\circ\sigma][\circ := e']) && \text{by hygiene assumptions} \\ &\quad = \lambda z.({}^\circ e_0[\circ := e'][{}^\circ\sigma[\circ := e']]) && \text{by the Induction Hypothesis} \\ &\quad = (\lambda z.{}^\circ e_0[\circ := e'])[{}^\circ\sigma[\circ := e']] && \text{by hygiene assumptions} \\ &\quad = {}^\circ e[\circ := e'][{}^\circ\sigma[\circ := e']] \end{aligned}$$

If ${}^\circ e = \circ[{}^\circ\sigma']$ then

$$\begin{aligned} &{}^\circ e[{}^\circ\sigma][\circ := e'] \\ &\quad = \circ[{}^\circ\sigma'][{}^\circ\sigma][\circ := e'] \\ &\quad = (\circ[\lambda z \in \mathrm{Dom}({}^\circ\sigma').{}^\circ\sigma'(z)[{}^\circ\sigma]])[\circ := e'] \\ &\quad = e'[\lambda z \in \mathrm{Dom}({}^\circ\sigma').{}^\circ\sigma'(z)[{}^\circ\sigma][\circ := e']] \\ &\quad = e'[\lambda z \in \mathrm{Dom}({}^\circ\sigma').{}^\circ\sigma'(z)[\circ := e'][{}^\circ\sigma[\circ := e']]] && \text{by the Induction Hypothesis} \\ &\quad = e'[{}^\circ\sigma'[\circ := e'] \odot {}^\circ\sigma[\circ := e']] \\ &\quad = \circ[{}^\circ\sigma'][\circ := e'][{}^\circ\sigma[\circ := e']] && \text{by hygiene assumptions} \\ &\quad = {}^\circ e[\circ := e'][{}^\circ\sigma[\circ := e']] \end{aligned}$$

□

Next we define analogs of redex and reduction context.

Definition (${}^\circ\mathbb{R}$ ${}^\circ\mathbb{E}_{\mathrm{rdx}}$)**:**

$${}^\circ\mathbb{R} = \{\Box\} \cup \Theta_{m+n+1}({}^\circ\mathbb{V}^m, {}^\circ\mathbb{R}, {}^\circ\mathbb{E}^n)$$

$${}^\circ\mathbb{E}_{\mathrm{rdx}} = \Theta^e_n({}^\circ\mathbb{V}^n)$$

Note that E-reduction contexts possess two types of holes, consequently we must disambiguate the process of hole filling. Note that the unique occurrence of a redex hole is not adorned with a substitution, consequently the process of filling the redex hole, $\Box$, with the E-expression, ${}^\circ e$, remains unchanged, and we denote it by ${}^\circ R[\Box := {}^\circ e]$.

In the case of multiple hole filling we write ${}^\circ R[\circ := e_0][\Box := e]$ for the result of filling the expression holes with e_0, and the redex hole with e.

Lemma (E-properties):

(1) ${}^\circ R[\circ := e_0][\Box := e] = {}^\circ R[\Box := e][\circ := e_0]$

(2) Filling an E-expression, E-reduction context, or E-redex with an expression yields an expression, reduction context, or redex, respectively.

7.2.2. E-Expression Decomposition

We now give a decomposition lemma for E-expressions: An E-expression ${}^\circ e$ is either an E-value (element of ${}^\circ\mathbb{V}$) or it can be decomposed uniquely into an E-reduction context filled with either an E-redex or with an expression hole.

Lemma (E-expression decomposition):

(0) ${}^\circ e \in {}^\circ\mathbb{V}$, or

(1) $(\exists!\, {}^\circ R, {}^\circ r)({}^\circ e = {}^\circ R[\Box := {}^\circ r])$, or

(2) $(\exists!\, {}^\circ R, {}^\circ \sigma)({}^\circ e = {}^\circ R[\Box := \circ[{}^\circ \sigma]])$

Proof : An easy induction on the structure of ${}^\circ e$. We consider two example cases. First, suppose ${}^\circ e = \circ[{}^\circ \sigma]$. Then we have case (2) with ${}^\circ R = \Box$. Second, suppose ${}^\circ e = \theta({}^\circ e_1, \ldots, {}^\circ e_n)$. If ${}^\circ e_i \in {}^\circ\mathbb{V}$ for $1 \le i \le n$, then we have case (1) with ${}^\circ R = \Box$ (and ${}^\circ r = {}^\circ e$). If ${}^\circ e_i \notin {}^\circ\mathbb{V}$ for some $1 \le i \le n$, assume k to be the least such i. Then by the induction hypothesis, ${}^\circ e_k$ decomposes either as (i) ${}^\circ R'[\Box := {}^\circ r]$, or as (ii) ${}^\circ R'[\Box := \circ[{}^\circ \sigma]]$. Taking ${}^\circ R = \theta({}^\circ e_1, \ldots, {}^\circ e_{k-1}, {}^\circ R', {}^\circ e_{k+1}, \ldots, {}^\circ e_n)$ we obtain the desired decomposition of ${}^\circ e$. $\Box$

7.2.3. E-Configurations

An E-configuration ${}^\circ\kappa$ is formed like a configuration, using E-expressions and E-values instead of simple expressions and values.

Definition (${}^\circ\mathbb{K}$):

$${}^\circ\mathbb{K} = \left\langle\!\!\left\langle\, {}^\circ\mathbb{A}\mathrm{c} \;\middle|\; {}^\circ\mathbb{M} \,\right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

where

${}^\circ\mathbb{A}\mathrm{c} = \mathbb{A}\mathrm{d} \xrightarrow{\mathrm{f}} {}^\circ\mathbb{A}\mathrm{s}$

${}^\circ\mathbb{A}\mathrm{s} = (\lambda x.{}^\circ\mathbb{E}) \cup [{}^\circ\mathbb{E}] \cup \{(?_{\mathbb{A}\mathrm{d}})\}$

${}^\circ\mathbb{M} = \langle {}^\circ\mathbb{V} \Leftarrow {}^\circ\mathbb{V} \rangle$

and the constraints specified in the definition of actor configurations in §2 are satisfied.[1] We let ${}^\circ\kappa$ range over ${}^\circ\mathbb{K}$, and ${}^\circ\alpha$ range over ${}^\circ\mathbb{A}\mathrm{c}$. Filling expression holes of an E-configuration, E-actor map, E-actor state, E-multiset of messages, and E-messages is defined in the obvious manner. Let ${}^\circ X$ stand generically for an element of one of these E-syntactic classes, then we define ${}^\circ X[\circ := e]$ as follows:

Definition (${}^\circ X[\circ := e]$):

$$
{}^\circ X[\circ := e] = \begin{cases}
\left\langle\!\!\left\langle\, {}^\circ\alpha[\circ := e] \;\middle|\; {}^\circ\mu[\circ := e] \,\right\rangle\!\!\right\rangle^{\rho}_{\chi} & \text{if } {}^\circ X = \left\langle\!\!\left\langle\, {}^\circ\alpha \mid {}^\circ\mu \,\right\rangle\!\!\right\rangle^{\rho}_{\chi} \\
\lambda x \in \mathrm{Dom}({}^\circ\alpha).{}^\circ\alpha(x)[\circ := e] & \text{if } {}^\circ X = {}^\circ\alpha \\
((\lambda x.{}^\circ e)[\circ := e]) & \text{if } {}^\circ X = (\lambda x.{}^\circ e) \\
[\![{}^\circ e[\circ := e]]\!] & \text{if } {}^\circ X = [\![{}^\circ e]\!] \\
(?_a) & \text{if } {}^\circ X = (?_a) \\
\{{}^\circ m[\circ := e] \;|\; {}^\circ m \in {}^\circ\mu\} & \text{if } {}^\circ X = {}^\circ\mu \\
\mathtt{<}{}^\circ v_0[\circ := e] \Leftarrow {}^\circ v_1[\circ := e]\mathtt{>} & \text{if } {}^\circ X = \mathtt{<}{}^\circ v_0 \Leftarrow {}^\circ v_1\mathtt{>}
\end{cases}
$$

An E-configuration, ${}^\circ\kappa$, is *closing* for e if ${}^\circ\kappa[\circ := e]$ is a closed configuration. Dually an expression e is a *valid filling* for an E-configuration, ${}^\circ\kappa$, if ${}^\circ\kappa[\circ := e]$ is a configuration. As for atoms and variables, the notion of communicable value remains unchanged and we do not introduce new notation for these. In particular, although messages may have holes, a message with a hole can effectively be ignored. This is because holes in E-values must occur inside λ's and hence filling these holes cannot yield communicable values or actor addresses. Thus a message with a hole can never be delivered or output. The next lemma expresses the fact that closing E-configurations make just the same observations as simple observing contexts.

Lemma (ocx): $e_0 \cong e_1$ iff $Obs({}^\circ\kappa[e_0]) = Obs({}^\circ\kappa[e_1])$ for all ${}^\circ\kappa$ that close e_0, e_1.

Proof : The backward implication is easy to see, since $\mathbb{O}$ is (with suitable translation to account for trapping at holes rather than at lambdas) a subset of ${}^\circ\mathbb{K}$. The idea for the proof of the forward implication is to define for each configuration context ${}^\circ\kappa$, an observing context O whose computations give rise to the same set of observations. In fact O evolves to ${}^\circ\kappa$ in a finite number of steps. For an E-expression ${}^\circ e$ we define ${}^\circ e^\dagger$ to be the result of recursively replacing decorated holes $\circ[{}^\circ\sigma]$ by applications $\mathtt{app}(\ldots\mathtt{app}(\lambda x_1.\ldots\lambda x_n.\bullet, {}^\circ\sigma(x_1)^\dagger), \ldots, {}^\circ\sigma(x_n)^\dagger)$ where $\{x_1, \ldots, x_n\} = \mathrm{Dom}({}^\circ\sigma)$. Let ${}^\circ\kappa = \left\langle\!\!\left\langle\, {}^\circ\alpha \mid \mu \,\right\rangle\!\!\right\rangle$, let $A = [a_i \mid i < n] = \mathrm{Dom}({}^\circ\alpha)$, and define $O = \left\langle\!\!\left\langle\, [\![e_{\circ_\kappa}]\!]_{\hat{a}} \mid \emptyset \,\right\rangle\!\!\right\rangle$ where $\hat{a} \notin A$, and $e_{\circ_\kappa}$ is constructed as follows. Let

$E = \{i < n \mid (\exists {}^\circ e_i)({}^\circ\alpha(a_i) = [\![{}^\circ e_i]\!])\}$, and let n_E be the cardinality of E.

$I = \{i < n \mid (\exists {}^\circ v_i)({}^\circ\alpha(a_i) = (\,{}^\circ v_i))\}$.

$B_i(a_0, \ldots, a_{n-1}) = {}^\circ v_i^\dagger$, if ${}^\circ\alpha(a_i) = (\,{}^\circ v_i)$.

$B_i(a_0, \ldots, a_{n-1}) = \lambda a.\mathtt{seq}(\mathtt{send}(a, 0), {}^\circ e_i^\dagger)$, if ${}^\circ\alpha(a_i) = [\![{}^\circ e_i]\!]$.

$\mu = \{\mathtt{<}z_j \Leftarrow {}^\circ v_j\mathtt{>} \mid j < m\}$

---

[1] The only condition whose meaning is altered in this general setting is (2), where the free variables of any holes must be taken into consideration.

Define

$$W_{\circ\kappa} = \mathtt{rec}(\lambda b.\lambda k.\lambda m.\mathtt{if}(\mathtt{eq}(k,0),\mathtt{seq}(\mathtt{send}(z_j,{}^\circ v_j^\dagger)_{j<m}),\mathtt{become}(b(k-1))))$$

$$\begin{aligned} e_{\circ\kappa} = {} & \mathtt{let}\{a_i := \mathtt{newadr}()\}_{i<n} \\ & \quad \mathtt{seq}(\mathtt{initbeh}(a_i, B_i(a_0,\ldots,a_{n-1}))_{i\in I\cup E}, \\ & \qquad \mathtt{send}(a_i,\hat{a})_{i\in E}, \\ & \qquad \mathtt{become}(W_{\circ\kappa}(n_E))) \end{aligned}$$

Now, we claim that any computation of ${}^\circ\kappa[\circ := e]$ has a corresponding computation (with same observations) of $O[e]$ obtained by accepting all the startup messages, sending and accepting the acks, and completing the computation of the initializing actor (which can then be ignored). Conversely any computation of $O[e]$ has a corresponding computation of ${}^\circ\kappa[\circ := e]$ obtained by ignoring the finite amount of initializing activity. This argument can be made more rigorous by decomposing paths in either system and then replacing hole contents and making local transformation of non-uniform transition steps. This process is explained in more detail in the proof of the theorem (**fun-red-eq**) below. □

7.2.4. E-Reduction

The reduction relations $\overset{\lambda}{\mapsto}$ and $\mapsto$ are extended to the generalized domains in the obvious fashion, simply by liberally annotating metavariables with °'s, modulo the extension of substitution to E-expressions. As examples, we give the (**beta-v**), (**br**), and (**eq**) clauses of $\overset{\lambda}{\mapsto}$ and the internal transitions for $\mapsto$ on closed E-configurations.

Definition ($\overset{\lambda}{\mapsto}$):

(beta-v) $${}^\circ R[\square := \mathtt{app}(\lambda x.{}^\circ e, {}^\circ v)] \overset{\lambda}{\mapsto} {}^\circ R[\square := {}^\circ e[x := {}^\circ v]]$$

(br) $${}^\circ R[\square := \mathtt{br}({}^\circ v_1, {}^\circ v_2, {}^\circ v)] \overset{\lambda}{\mapsto} \begin{cases} {}^\circ R[\square := {}^\circ v_1] & \text{if } {}^\circ v \in {}^\circ\mathbb{V} - \{\mathtt{nil}\} \\ {}^\circ R[\square := {}^\circ v_2] & \text{if } {}^\circ v = \mathtt{nil} \end{cases}$$

(eq) $${}^\circ R[\square := \mathtt{eq}({}^\circ v_0, {}^\circ v_1)] \overset{\lambda}{\mapsto} \begin{cases} {}^\circ R[\square := \mathtt{t}] & \text{if } {}^\circ v_0 = {}^\circ v_1 \in \mathbb{A}\mathrm{t} \\ {}^\circ R[\square := \mathtt{nil}] & \text{if } {}^\circ v_0, {}^\circ v_1 \in \mathbb{A}\mathrm{t} \text{ and } {}^\circ v_0 \neq {}^\circ v_1 \end{cases}$$

Definition ($\mapsto$):

<fun : a> $${}^\circ e \overset{\lambda}{\mapsto} {}^\circ e' \Rightarrow \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ e]_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ e']_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle$$

<new : a, a'> $$\left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{newadr}()]]_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := a']]_a, (?_a)_{a'} \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \qquad a' \text{ fresh}$$

<init : a, a'> $$\left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{initbeh}(a', {}^\circ v)]]_a, (?_a)_{a'} \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{nil}]]_a, ({}^\circ v)_{a'} \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle$$

<bec : a, a'> $$\left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{become}({}^\circ v)]]_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{nil}]]_{a'}, ({}^\circ v)_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \qquad a' \text{ fresh}$$

<send : a, m> $$\left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{send}({}^\circ v_0, {}^\circ v_1)]]_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{nil}]]_a \;\middle|\; {}^\circ\mu, \mathtt{<}{}^\circ v_0 \Leftarrow {}^\circ v_1\mathtt{>} \right\rangle\!\!\right\rangle$$

<rcv : a, cv> $$\left\langle\!\!\left\langle {}^\circ\alpha, ({}^\circ v)_a \;\middle|\; \mathtt{<}a \Leftarrow cv\mathtt{>}, {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [\mathtt{app}({}^\circ v, cv)]_a \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle$$

7.2.5. E-Uniform Computation

The notion of E-uniformity of computation is made precise in the following definitions and lemmas. The basic idea is that given a decomposition of a configuration as an E-configuration with holes filled by a given expression, then any transition step leading from that configuration is either independent of what appears in the holes, or it explicitly uses information about the contents of some hole occurrence.

Definition (E-hole touching): Let ${}^\circ\kappa = \left\langle\!\!\left\langle\, {}^\circ\alpha \mid {}^\circ\mu \,\right\rangle\!\!\right\rangle$. We say that ${}^\circ\kappa$ touches a hole at a if ${}^\circ\alpha(a) = \texttt{[}{}^\circ R[\Box := \circ[{}^\circ\sigma]]\texttt{]}$ for some ${}^\circ R$, ${}^\circ\sigma$.

We say that a transition $\kappa \xrightarrow{l} \kappa'$ touches a hole relative to a decomposition $\kappa = {}^\circ\kappa[\circ := e]$ if l has focus a and ${}^\circ\kappa$ touches a hole at a.

Lemma (E-Uniform Computation):

(1) If ${}^\circ\kappa \xrightarrow{l} {}^\circ\kappa'$, then ${}^\circ\kappa[\circ := e] \xrightarrow{l} {}^\circ\kappa'[\circ := e]$ for any valid filling expression e.

(2) If ${}^\circ\kappa$ has no transition with focus a (and a is an actor of ${}^\circ\kappa$), then either ${}^\circ\kappa$ touches a hole at a or ${}^\circ\kappa[\circ := e]$ has no transition with focus a for any valid filling expression e.

(3) If $\kappa \xrightarrow{l} \kappa'$ and $\kappa = {}^\circ\kappa[\circ := e]$, then either the transition touches a hole or we can find ${}^\circ\kappa'$ such that $\kappa' = {}^\circ\kappa'[\circ := e]$ and ${}^\circ\kappa \xrightarrow{l} {}^\circ\kappa'$.

Proof (1): This is proved by considering cases on the transition rule applied. The only interesting case is (**beta-v**). This follows from (**fil-subst**) $\Box_1$

Proof (2): Assume ${}^\circ\kappa = \left\langle\!\!\left\langle\, {}^\circ\alpha \mid {}^\circ\mu \,\right\rangle\!\!\right\rangle$ has no transition with focus a, and ${}^\circ\kappa$ does not touch a hole at a. Then one of the following holds:

(i) ${}^\circ\alpha(a) = (?'_a)$

(ii) ${}^\circ\alpha(a) = (\,{}^\circ v)$ and ${}^\circ\mu$ contains no messages deliverable to a

(iii) ${}^\circ\alpha(a) = \texttt{[}{}^\circ v\texttt{]}$

(iv) ${}^\circ\alpha(a) = \texttt{[}{}^\circ R[\Box := \texttt{initbeh}({}^\circ v_0, {}^\circ v_1)]\texttt{]}$ where ${}^\circ v_0$ is not the address of an uninitialized actor created by a

(v) ${}^\circ\alpha(a) = \texttt{[}{}^\circ R[\Box := {}^\circ R]\texttt{]}$ where ${}^\circ R$ is a non-actor redex that is stuck.

In each of these cases, it easy to see that there will be no transition with focus a enabled when the expressions holes are filled. $\Box_2$

Proof (3): Assume $\kappa = \left\langle\!\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\!\right\rangle \xrightarrow{l} \kappa' = \left\langle\!\!\left\langle\, \alpha' \mid \mu' \,\right\rangle\!\!\right\rangle$ and $\kappa = {}^\circ\kappa[\circ := e]$. Thus ${}^\circ\kappa = \left\langle\!\!\left\langle\, {}^\circ\alpha \mid {}^\circ\mu \,\right\rangle\!\!\right\rangle$ where $\alpha = {}^\circ\alpha[\circ := e]$ and $\mu = {}^\circ\mu[\circ := e]$. We want to find ${}^\circ\alpha'$, ${}^\circ\mu'$ such that ${}^\circ\kappa \xrightarrow{l} {}^\circ\kappa' = \left\langle\!\!\left\langle\, {}^\circ\alpha' \mid {}^\circ\mu' \,\right\rangle\!\!\right\rangle$, $\alpha' = {}^\circ\alpha'[\circ := e]$, and $\mu' = {}^\circ\mu'[\circ := e]$. Since we are considering closed configurations there are no `i/o` transitions. Thus, we need to consider only two cases `rcv` transitions and `exec` transitions. We split the `exec` transitions into functional and actor primitives.

Receive: $l =$ `<rcv : a, cv>`, `<a` $\Leftarrow$ `cv>` $\in \mu$, and $\alpha(a) = (v)$. Thus ${}^\circ\alpha(a) = (\,{}^\circ v)$ with $v = {}^\circ v[\circ := e]$. Thus we let ${}^\circ\alpha' = {}^\circ\alpha\{a := \texttt{[app}({}^\circ v, cv)\texttt{]}\}$, and ${}^\circ\mu = {}^\circ\mu' \cup \{\texttt{<}a \Leftarrow cv\texttt{>}\}$.

Execution-lambda: $l = \texttt{<fun}:a\texttt{>}$, $\alpha(a) = [R[\Box := r]]$ and $r \stackrel{\lambda}{\mapsto} e'$. Thus ${}^\circ\alpha(a) = [{}^\circ R[\Box := {}^\circ r]]$ with $R = {}^\circ R[\circ := e]$, and $r = {}^\circ r[\circ := e]$. Thus we want to find ${}^\circ e'$ such that ${}^\circ r \stackrel{\lambda}{\mapsto} {}^\circ e'$. Then ${}^\circ\alpha' = {}^\circ\alpha\{a := [{}^\circ R[\Box := {}^\circ e']]\}$ and ${}^\circ\mu' = {}^\circ\mu$. If $r = \texttt{app}(\lambda z.e_0, v)$ (z chosen fresh), then $e' = e_0[z := v]$, and ${}^\circ r = \texttt{app}(\lambda z.{}^\circ e_0, {}^\circ v)$ where $e_0 = {}^\circ e_0[\circ := e]$ and $v = {}^\circ v[\circ := e]$. Take ${}^\circ e' = {}^\circ e_0[z := {}^\circ v]$ and use (**fil-subst**). If $r = \texttt{eq}(v_0, v_1)$, then ${}^\circ r = \texttt{eq}({}^\circ v_0, {}^\circ v_1)$ where $v_j = {}^\circ v_j[\circ := e]$ for $j < 2$. e' is `t` or `nil` and we may take ${}^\circ e' = e'$. The remaining cases are similar.

Execution-actor: If $l = \texttt{<send}:a\texttt{>}$, then $\alpha(a) = [R[\Box := \texttt{send}(v_0, v_1)]]$ $\alpha'(a) = [R[\Box := \texttt{nil}]]$, $\mu' = \mu \cup \{\texttt{<}v_0 \Leftarrow v_1\texttt{>}\}$. Also ${}^\circ\alpha(a) = [{}^\circ R[\Box := \texttt{send}({}^\circ v_0, {}^\circ v_1)]]$ where $R = {}^\circ R[\circ := e]$, and $v_j = {}^\circ v_j[\circ := e]$ for $j < 2$. Take ${}^\circ\alpha' = {}^\circ\alpha\{a := [{}^\circ R[\Box := \texttt{nil}]]\}$ and ${}^\circ\mu' = {}^\circ\mu \cup \{\texttt{<}{}^\circ v_0 \Leftarrow {}^\circ v_1\texttt{>}\}$.

If $l = \texttt{<become}:a, a'\texttt{>}$, then a' is fresh, $\alpha(a) = [R[\Box := \texttt{become}(v)]]$ $\alpha'(a) = (v)$, $\alpha'(a') = [R[\Box := \texttt{nil}]]$, and $\mu' = \mu$. Also ${}^\circ\alpha(a) = [{}^\circ R[\Box := \texttt{become}({}^\circ v)]]$ where $R = {}^\circ R[\circ := e]$, and $v = {}^\circ v[\circ := e]$. Take ${}^\circ\alpha' = {}^\circ\alpha\{a := ({}^\circ v), a' := [{}^\circ R[\Box := \texttt{nil}]]\}$ and ${}^\circ\mu' = {}^\circ\mu$.

If $l = \texttt{<new}:a, a'\texttt{>}$, then a' is fresh, $\alpha(a) = [R[\Box := \texttt{newadr}()]]$ $\alpha'(a) = [R[\Box := \texttt{nil}]]$, $\alpha'(a') = (?_a)$, and $\mu' = \mu$. Also ${}^\circ\alpha(a) = [{}^\circ R[\Box := \texttt{newadr}()]]$ where $R = {}^\circ R[\circ := e]$. Take ${}^\circ\alpha' = {}^\circ\alpha\{a := [{}^\circ R[\Box := \texttt{nil}]], a' := (?_a)\}$ and ${}^\circ\mu' = {}^\circ\mu$.

If $l = \texttt{<init}:a, a'\texttt{>}$, then $\alpha(a) = [R[\Box := \texttt{initbeh}(a', v)]]$, $\alpha(a') = (?_a)$, $\alpha'(a) = [R[\Box := \texttt{nil}]]$, $\alpha'(a') = (v)$, and $\mu' = \mu$. Also ${}^\circ\alpha(a) = [{}^\circ R[\Box := \texttt{initbeh}(a', {}^\circ v)]]$, where $R = {}^\circ R[\circ := e]$, and $v = {}^\circ v[\circ := e]$, and ${}^\circ\alpha(a') = (?_a)$.. Take ${}^\circ\alpha' = {}^\circ\alpha\{a := [{}^\circ R[\Box := \texttt{nil}]], a' := ({}^\circ v)\}$ and ${}^\circ\mu' = {}^\circ\mu$.

$\Box_3$ $\Box_{\mathbf{E-uniform}}$

7.2.6. The (fun-red-eq) Theorem

Now we have enough notation to describe the construction of path correspondences for expressions with uniform common reducts. We first consider the case of expressions that reduce via purely functional reductions. Then we show how this construction can be modified to allow for reduction of actor primitives.

Theorem (fun-red-eq): If for each ${}^\circ\sigma$ with domain containing the free variables of e_0, e_1, either $e_j[{}^\circ\sigma]$ hangs for $j < 2$, or there is some ${}^\circ e_c$ such that $e_j[{}^\circ\sigma]$ reduces in 0 or more $\stackrel{\lambda}{\mapsto}$ steps to ${}^\circ e_c$ uniformly, then $e_0 \cong e_1$.

Corollary (fun-red-eq): The following laws are instances of (**fun-red-eq**): (**red-exp**), (**beta-v**), (**ift**), (**ifn**), (**ifelim**), (**isprt**), (**isprn**), (**fst**), and (**snd**).

Proof: Assume that for each closing ${}^\circ\sigma$ there is ${}^\circ e_{c,j}$ such that $e_j[{}^\circ\sigma] \stackrel{\lambda}{\mapsto} \ldots \stackrel{\lambda}{\mapsto} {}^\circ e_{c,j}$ $j < 2$, uniformly, and either ${}^\circ e_{c,j}$ is (uniformly) stuck for $j < 2$, or ${}^\circ e_{c,0} = {}^\circ e_{c,1}$. In either case we call ${}^\circ e_{c,j}$ the common reduct. We want to show that $e_0 \cong e_1$. By (**ocx**) it is sufficient to show that $Obs({}^\circ\kappa[\circ := e_0]) = Obs({}^\circ\kappa[\circ := e_1])$ for any ${}^\circ\kappa$ that is a closing E-configuration for e_0 and e_1. To do this, we show that for any $\pi_0 \in \mathcal{F}({}^\circ\kappa[\circ := e_0]) = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i \in \bowtie]$ we can find $\pi_1 \in \mathcal{F}({}^\circ\kappa[\circ := e_1])$ such that $obs(\pi_0) = obs(\pi_1)$. (The case with 0 and 1 interchanged is symmetric.)

Informally, by the uniformity property of computations, we see that replacing occurrences of e_0 by e_1 has no effect on a computation except where a hole is touched. Using (**infinite macro-steps**) we can localize non-uniform steps so that when a hole is touched,

reduction to a common reduct occurs in a single multi-step (which involves no event transitions). Thus we may obtain a computation for ${}^\circ\kappa[\circ := e_1]$ by replacing occurrences of e_0 by e_1 and replacing multi-step transitions reducing $e_0[{}^\circ\sigma]$ to its common reduct by multi-step transitions reducing $e_1[{}^\circ\sigma]$ to its common reduct. To insure completeness/fairness of the result, we need to take account of the case where a hole $\circ[{}^\circ\sigma]$ is exposed, but the multi-step for $e_0[{}^\circ\sigma]$ is trivial and hence does not appear as a transition. We do this by inserting the corresponding multi-step for $e_1[{}^\circ\sigma]$ at the point where the hole is first exposed. Such holes then effectively disappear, since they are either filled with a stuck expression or with the same expression. Below we make this informal argument more rigorous, by the following steps:

(1) We analyze the configurations occurring in π_0 and record occurrences of e_0 in holes descending from ${}^\circ\kappa$. This gives us decompositions ${}^\circ\kappa_i[\circ := e_0]$ of κ_i. In the cases where a hole is touched such that e_0 is its common reduct, we fill that hole with e_0 giving a new E-configuration ${}^\circ\kappa_i'$ with one less hole, such that ${}^\circ\kappa_i'[\circ := e_0]$ is κ_i. This process of filling holes with common reducts continues until the transition l_i is either uniform or touches a hole in which e_0 is not its common reduct. We also record subsequences of transitions corresponding to uniform reduction of such occurrences of e_0 to its common reduct.

(2) Using (**infinite-macro-steps**) we may assume that the path is expressed in terms of multi-step transitions such that the recorded subsequences of transitions corresponding to non-trivial reduction to a common reduct are single multi-steps. We also insert copies of κ_i for each hole that is filled with a common reduct, remembering the corresponding decomposition, and insert empty multi-steps between these copies. We also insert a copy of κ_i and a connecting empty multi-step for each hole that occurs in a reduction context that is not touched – because the occurrence of e_0 is stuck, or because it is a value and placing it in the redex hole produces either a value or a stuck state.

(3) Form π_1 by filling the holes of ${}^\circ\kappa_i$ (and remaining holes is additional copies) with e_1 and replacing multi-steps for e_0 by corresponding multi-steps for e_1 (empty multi-steps may expand to non-trivial reductions of occurrences of e_1 to its common reduct.

It is easy to see that π_1 is a computation path. The argument that it is complete and fair relies on the insertion of multi-steps, and uses the same case analysis that was used in the uniform computation lemma.

Step (1) We analyze and decompose π_0 to obtain

(i) for each $i < \bowtie$ a series of decompositions ${}^\circ\kappa_{i,j}$ for $j \leq n_i$ such that $\kappa_i = {}^\circ\kappa_{i,j}[\circ := e_0]$ for $j \leq n_i$ and such that ${}^\circ\kappa_{i,n_i} \xrightarrow{l_i} {}^\circ\kappa_{i+1,0}$ uniformly, or the transition touches a hole in which e_0 has non-trivial reduction to its common reduct. (We call this entering the hole.) n_i will be 0 except in the case of a hole touched in which e_0 is its own common reduct. Then we fill that hole with the common reduct and redecompose.

(ii) The set I of indices of transitions that enter holes

(iii) The map J from I to the sequence of indices of transitions corresponding to the thread of computation that carries out the reduction to the common reduct.

This is done incrementally by defining sequences I_n, J_n by induction on n and taking $I = \bigcup_{n<\bowtie} I_n$, and $J = \bigcup_{n<\bowtie} J_n$. At stage (i,j) we have defined I_i, J_i, and ${}^\circ\kappa_{i,j}$. If $j = n_i$, then the next stage is $(i+1, 0)$ otherwise it is $(i, j+1)$.

Stage $(0,0)$ $I_0 = \emptyset$, and J_0 is the empty map. ${}^\circ\kappa_{0,0} = {}^\circ\kappa$.

At stage (i,j) There are four cases to consider:

(1) l_i is execution in a hole;

(2) l_i is uniform wrt ${}^\circ\kappa_{i,j}$

(3) l_i touches a hole and

(3.1) enters the hole

(3.2) does not enter the hole

Case 1: This case occurs if i is an element of $M = J_i(m)$ for some $m \in I_i$. Thus $n_i = j$ and $l_i = \texttt{<fun}: a\texttt{>}$ for some a. We move to stage $(i+1,0)$ with $I_{i+1} = I_i$, $J_{i+1} = J_i$, ${}^\circ\mu_{i+1,0} = {}^\circ\mu_{i,j}$, and ${}^\circ\alpha_{i+1,0} = {}^\circ\alpha_{i,j}\{a := [{}^\circ R[\Box := {}^\circ e_{m,k+1}]]\}$ where ${}^\circ R$, and ${}^\circ e_{m,k+1}$ are obtained as follows. Let k be the index of i in M. The hole is entered at stage (m, n_m) with ${}^\circ\alpha_{m,n_m}(a) = [{}^\circ R[\Box := \circ[{}^\circ\sigma]]]$. Let ${}^\circ e_{m,0} = e_0[{}^\circ\sigma]$, let n be the length of M, and let $[{}^\circ e_{m,k} \overset{\lambda}{\mapsto} {}^\circ e_{m,k+1} \mid k < n]$ be the thread of computation reducing $e_0[{}^\circ\sigma]$ to its common reduct ${}^\circ e_{m,n}$. Note that ${}^\circ\alpha_{i,j}(a) = [{}^\circ R[\Box := {}^\circ e_{m,k}]]$.

Case 2: $n_i = j$ and we move to stage $(i+1,0)$ with $I_{i+1} = I_i$, $J_{i+1} = J_i$, ${}^\circ\kappa_{i+1,0}$ such that ${}^\circ\kappa_{i,j} \xrightarrow{l_i} {}^\circ\kappa_{i+1,0}$ uniformly according to the uniformity lemma.

Case 3.1: In this case $l_i = \texttt{<fun}: a\texttt{>}$ for some a. $n_i = j$ and we move to stage $(i+1,0)$ with $I_{i+1} = I_i \cup \{i\}$, $J_{i+1} = J_i\{i := M\}$, ${}^\circ\mu_{i+1} = {}^\circ\mu_i$, and ${}^\circ\alpha_{i+1} = {}^\circ\alpha_i\{a := [{}^\circ R[\Box := {}^\circ e_{i,1}]]\}$ where M, ${}^\circ R$, and ${}^\circ e_{i,1}$ are obtained as follows. Let ${}^\circ\alpha_{i,j}(a) = [{}^\circ R[\Box := \circ[{}^\circ\sigma]]]$, let ${}^\circ e_{i,0} = e_0[{}^\circ\sigma]$, and let $[{}^\circ e_{i,k}[{}^\circ\sigma] \overset{\lambda}{\mapsto} {}^\circ e_{i,k+1} \mid k < n+1]$ be the thread of computation reducing $e_0[{}^\circ\sigma]$ to its common reduct ${}^\circ e_{i,n+1}$. By fairness, there is a sequence of indices $M = [i_k \mid k < n+1]$ with $i_0 = i$, $i_k < i_{k+1}$ for $k < n+1$ such than M is the macro step corresponding to the above lambda reduction.

Case 3.2: We move to stage $(i, j+1)$ with $I_{i+1} = I_i$, $J_{i+1} = J_i$, ${}^\circ\mu_{i,j+1} = {}^\circ\mu_{i,j}$, and ${}^\circ\alpha_{i,j+1} = {}^\circ\alpha_{i,j}\{a := [{}^\circ R[\Box := e_0[{}^\circ\sigma]]]\}$ where a, ${}^\circ R$, ${}^\circ\sigma$ are obtained as follows. a is the focus of l_i, and ${}^\circ\alpha_{i,j}(a) = [{}^\circ R[\Box := \circ[{}^\circ\sigma]]]$, with $e_0[{}^\circ\sigma]$ equal to its common reduct.

Step (2) The family $J(i)$ for $i \in I$ satisfies the conditions of (**infinite-macro-step**). Hence we may assume that π_0 has the form

$$[[{}^\circ\kappa_{i,0}[\circ := e_0] \xrightarrow{[\,]} \ldots \xrightarrow{[\,]} {}^\circ\kappa_{i,n_i} \xrightarrow{L_i} {}^\circ\kappa_{i+1,0}[\circ := e_0]] \mid i < \bowtie]$$

where each L_i is either a single (uniform) transition, or a multi-step reduction of an occurrence of e_0 to its common reduct, and the E-configurations obtained by the above decomposition method. For each i, if ${}^\circ\alpha_{i,0}(a) = [{}^\circ R[\Box := \circ[{}^\circ\sigma]]]$, and and there are no transitions L_j for $i \le j$ with focus a, and i is the least such index, we insert before each transition leading from ${}^\circ\alpha_{i,0}$ an an empty transition ${}^\circ\kappa_{i,0}[\circ := e_0] \xrightarrow{[\,]} {}^\circ\kappa_{i,0}[\circ := e_0]$.

Step (3) We let π_1 be the path

$$[[{}^\circ\kappa_{i,0}[\circ := e_1] \xrightarrow{L_{i,0}} \ldots \xrightarrow{L_{i,n_i-1}} {}^\circ\kappa_{i,n_i}[\circ := e_1] \xrightarrow{L'_i} {}^\circ\kappa_{i+1,0}[\circ := e_1]] \mid i < \bowtie]$$

where L_i' is L_i if L_i is a single (uniform) transition; L_i' is the corresponding macro-step reduction of the occurrence of e_1 to its common reduct, if L_i is a macro-step or an empty transition.

Clearly π_1 is a complete computation path. Also the transitions are the same except for points where holes are touched, but these differences are not observable. Thus $obs(\pi_0) = obs(\pi_1)$.

It remains to check that fairness has been preserved. Suppose some transition l is enabled at stage i in π_1. We have three cases:

Receive: Suppose l is receipt of $\texttt{<}a \Leftarrow cv\texttt{>}$. Then ${}^\circ\alpha_i(a) = (\lambda x.{}^\circ e)$ and hence l is enabled in π_0 at stage i. If l fires in π_0 at stage $i' \geq i$, then it also fires at this stage in π_1. Suppose l never fires in π_0. Then by fairness, there is some $i' > i$ such that ${}^\circ\alpha_j(a)$ is an executing state for $j \geq i'$. By construction l is permanently disabled at i' in π_1 as well.

Uniform Execution: Suppose l is an execution step with focus a where ${}^\circ\alpha_i(a) = [{}^\circ R[\Box := {}^\circ r]]$. Then l is enabled in π_0 at i, it can not be disabled, and must occur in π_0 at some stage $i' \geq i$ and hence will occur in π_1 at that stage.

Hole Touching: Suppose l is an execution step by a with ${}^\circ\alpha_i(a) = [{}^\circ e]$ where ${}^\circ e = {}^\circ R[\Box := \circ[{}^\circ\sigma]]$. First assume $e_1[{}^\circ\sigma]$ reduces. If ${}^\circ e[\circ := e_0]$ does not reduce, then by construction, the transition is taken in π_1 as soon as it is enabled. If ${}^\circ e[\circ := e_0]$ reduces, then a transition will eventually be taken at a in π_0, and the l will be taken at the corresponding point in π_1. Suppose $e_1[{}^\circ\sigma]$ does not reduce. Then it must be a value, hence the common reduct. Hence the reduction of e_0 is enabled in π_0 and will eventually be taken. ${}^\circ R$ has the form ${}^\circ R_0[\Box := \theta({}^\circ v^m, \circ[{}^\circ\sigma], {}^\circ e^n)]$. If all the E-expressions ${}^\circ e^n$ are E-values, then l must be reduction of the redex in ${}^\circ R_0$ and this is also enabled now, in π_0. Otherwise consider decomposition of the first non-value element of ${}^\circ e^n$ and repeat this argument. Since we are now looking at a smaller E-expression, we eventually reach the point where the step enabled in π_1 corresponds to one in π_0 and hence will occur eventually. $\Box$

7.2.7. The Proof of the (hang-infin) Theorem

We now prove (**hang-infin**), which says that any two expressions that hang or have infinite computations are observationally equivalent.

Proof (hang-infin): Assume $e_0, e_1 \in \mathit{Hang} \cup \mathit{Infin}$. We want to show that $e_0 \cong e_1$. Let ${}^\circ\kappa = \left\langle\!\!\left\langle {}^\circ\alpha \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle$ be a closing E-configuration for e_0 and e_1. Assume $\pi_0 \in \mathcal{F}({}^\circ\kappa[\circ := e_0]) = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \;\middle|\; i \in \bowtie]$. We want to find ${}^\circ\alpha_i$, and L_i, such that $\kappa_i = {}^\circ\kappa_i[\circ := e_0]$ where ${}^\circ\kappa_i = \left\langle\!\!\left\langle {}^\circ\alpha_i \;\middle|\; {}^\circ\mu_i \right\rangle\!\!\right\rangle$ and, letting $\pi_1 = [{}^\circ\kappa_i[\circ := e_1] \xrightarrow{L_i} {}^\circ\kappa_{i+1}[\circ := e_1] \;\middle|\; i \in \bowtie]$, we have $\pi_1 \in \mathcal{F}({}^\circ\kappa[\circ := e_1])$ and $obs(\pi_0) = obs(\pi_1)$. (Actually, we let holes in π_j be filled by any expression of the same class as e_j.) For the base case we have ${}^\circ\alpha_0 = {}^\circ\alpha$. Assume we have ${}^\circ\alpha_i$. Suppose $e_0 \in \mathit{Hang}$. Let a be the focus of l_i. We first consider each a' other than a such that ${}^\circ\alpha_i(a') = {}^\circ R[\Box := \circ[{}^\circ\sigma]]$. If $e_1 \in \mathit{Hang}$ then we just insert any steps needed to reach the stuck state (we assume that they are already macroized for π_0). If $e_1 \in \mathit{Infin}$, then insert the step to reach the next element of its infinite sequence. Now we consider the transition label l_i. If it does not touch a hole, then ${}^\circ\alpha_{i+1}$ is given by the uniform transition lemma. Suppose ${}^\circ\alpha_i(a) = {}^\circ R[\Box := \circ[{}^\circ\sigma]]$. Then ${}^\circ\alpha_{i+1}$ has the same decomposition, just possible different expressions (of the same class) filling the holes. $\Box_{\mathbf{h-i}}$

7.2.8. The Proofs of the Actor Primitive Laws

We show how to modify the construction for the purely functional case to establish equivalence where reductions may involve actor primitives. The notion of common reduct is a fragment of an E-configuration which must be merged with the parent configuration. In the equivalences considered below, with one exception, no extra steps need to be inserted since in every instance either both expressions step, or both hang. Thus we only need to construct ${}^{\circ}\alpha_{i+1}$ for the hole touching case. Fairness follows easily using the same argument as for the functional case. The exception is the commuting of **newadr** with **initbeh**, which will be treated in more detail below.

Proof (triv): Suppose e_0, e_1 are taken from the pair $\vartheta(\bar{x}), \mathtt{seq}(\vartheta(\bar{x}), \mathtt{nil})$ where $\vartheta \in \{\mathtt{send}, \mathtt{become}, \mathtt{initbeh}\}$. Suppose l_i is an execution by a with ${}^{\circ}\alpha_i(a) = [{}^{\circ}R[\Box := \circ[{}^{\circ}\sigma]]]$. We define ${}^{\circ}\alpha_{i+1}$ for each form $\vartheta(\bar{x})$.

($\mathtt{send}(x_0, x_1)$) ${}^{\circ}\alpha_{i+1} = {}^{\circ}\alpha_i\{a := [{}^{\circ}R[\Box := \mathtt{nil}]]\}$. Also, ${}^{\circ}\mu_{i+1} = {}^{\circ}\mu_i \cup \{\langle {}^{\circ}v_0 \Leftarrow {}^{\circ}v_1 \rangle\}$, where ${}^{\circ}v_j = x_j[{}^{\circ}\sigma]$. (Note that any holes appearing in messages must be inside lambdas and hence the message is ill-formed and can be ignored.)

($\mathtt{become}(x_0)$) ${}^{\circ}\alpha_{i+1} = {}^{\circ}\alpha_i\{a := ({}^{\circ}\sigma(x_0)), a' := [{}^{\circ}R[\Box := \mathtt{nil}]]\}$ where a' is a new (anonymous) actor identifier.

($\mathtt{initbeh}(x_0, x_1)$) ${}^{\circ}\alpha_{i+1} = {}^{\circ}\alpha_i\{a := [{}^{\circ}R[\Box := \mathtt{nil}]], {}^{\circ}\sigma(x_0) := ({}^{\circ}\sigma(x_1))\}$. Since the step occurs in π_0, we may assume $a' = {}^{\circ}\sigma(x_0) \in \mathrm{Dom}({}^{\circ}\alpha_i)$, and ${}^{\circ}\alpha_i(a') = (?_a)$.

$\Box_{\mathbf{triv}}$

Proof (can-b): Suppose e_0, e_1 are taken from the pair

$$\mathtt{seq}(\mathtt{become}(v_0), \mathtt{become}(v_1)), \qquad \mathtt{seq}(\mathtt{become}(v_0), \mathtt{nil}).$$

Assume l_i is an execution by a with ${}^{\circ}\alpha_i(a) = [{}^{\circ}R[\Box := \circ[{}^{\circ}\sigma]]]$. Define

$${}^{\circ}\alpha_{i+1} = {}^{\circ}\kappa_i\{a := (v_0[{}^{\circ}\sigma]), a' := [{}^{\circ}R[\Box := \mathtt{nil}]]\}$$

where a' is a new anonymous actor identifier. We ignore the additional anonymous actor with state $(v_1[{}^{\circ}\sigma])$ in the case of two becomes, since it is carried along unchanged in the remainder of the computation. $\Box$

Proof (can-i): Suppose e_0, e_1 are taken from the pair

$$\mathtt{seq}(\mathtt{initbeh}(v, v_0), \mathtt{initbeh}(v, v_1)), \qquad \mathtt{seq}(\mathtt{initbeh}(v, v_0), \mathtt{stuck}).$$

(The equivalence with **stuck** replaced by **bot** follows from (**hang-infin**).) Assume l_i is an execution by a with ${}^{\circ}\alpha_i(a) = [{}^{\circ}R[\Box := \circ[{}^{\circ}\sigma]]]$. Define

$${}^{\circ}\alpha_{i+1} = {}^{\circ}\alpha_i\{a := [{}^{\circ}R[\Box := \mathtt{bot}]], {}^{\circ}\sigma(v) := ({}^{\circ}\sigma(v_0))\}$$

Since the step occurs in π_0, we may assume $a' = {}^{\circ}\sigma(v) \in \mathrm{Dom}({}^{\circ}\alpha_i)$, and ${}^{\circ}\alpha_i(a') = (?_a)$. Officially, in π_1 ${}^{\circ}\alpha_{i+1}(a)$ should be $[{}^{\circ}R[\Box := \mathtt{initbeh}(v, v_1)[{}^{\circ}\sigma]]]$. But, as in the proof of (**hang-infin**) we treat stuck expressions as indistinguishable. $\Box$

Proof (commutes): Suppose e_0, e_1 are taken from one of the pairs in the commutes lemma, except (**n-i**). Assume l_i is an execution by a with ${}^{\circ}\alpha_i(a) = [{}^{\circ}R[\Box := \circ[{}^{\circ}\sigma]]]$.

(s-s)

$\texttt{let}\{x_0 := \texttt{send}(v_0, v_1)\}\texttt{let}\{x_1 := \texttt{send}(v_2, v_3)\}e$

$\texttt{let}\{x_1 := \texttt{send}(v_2, v_3)\}\texttt{let}\{x_0 := \texttt{send}(v_0, v_1)\}e$

Define

$^\circ\alpha_{i+1} = {}^\circ\alpha_i\{adr := [^\circ R[\Box := e\{x_0 := \texttt{nil}, x_1 := \texttt{nil}\}[^\circ\sigma]]]\}$

$^\circ v_j = v_j[^\circ\sigma] \quad \text{for} \quad j < 4$

$^\circ\mu_{i+1} = {}^\circ\mu_i \cup \{<{}^\circ v_0 \Leftarrow {}^\circ v_1>, <{}^\circ v_2 \Leftarrow {}^\circ v_3>\}$

$\Box_{\mathbf{s-s}}$

(s-n)

$\texttt{let}\{x_0 := \texttt{send}(v_0, v_1)\}\texttt{let}\{x_1 := \texttt{newadr}()\}e$

$\texttt{let}\{x_1 := \texttt{newadr}()\}\texttt{let}\{x_0 := \texttt{send}(v_0, v_1)\}e$

Let a_1 be fresh and define

$^\circ\alpha_{i+1} = {}^\circ\alpha_i\{a := [^\circ R[\Box := e\{x_0 := \texttt{nil}, x_1 := a_1\}[^\circ\sigma]]]a_1 := (?_a)\}$

$^\circ v_j = v_j[^\circ\sigma] \quad \text{for} \quad j < 2$

$^\circ\mu_{i+1} = {}^\circ\mu_i \cup \{<{}^\circ v_0 \Leftarrow {}^\circ v_1>\}$

$\Box_{\mathbf{s-n}}$

(n-n)

$\texttt{let}\{x_0 := \texttt{newadr}()\}\texttt{let}\{x_1 := \texttt{newadr}()\}e$

$\texttt{let}\{x_1 := \texttt{newadr}()\}\texttt{let}\{x_0 := \texttt{newadr}()\}e$

Let a_0, a_1 be fresh and define

$^\circ\alpha_{i+1} = {}^\circ\alpha_i\{a := [^\circ R[\Box := e\{x_0 := a_0, x_1 := a_1\}[^\circ\sigma]]]a_0 := (?_a)\, a_1 := (?_a)\}$

$\Box_{\mathbf{n-n}}$

(n-i) To complete the proof, we treat the case (**n-i**). Here e_0, e_1 are taken from the pair

$\texttt{let}\{x_0 := \texttt{newadr}()\}\texttt{let}\{x_1 := \texttt{initbeh}(v, v_1)\}e$

$\texttt{let}\{x_1 := \texttt{initbeh}(v, v_1)\}\texttt{let}\{x_0 := \texttt{newadr}()\}e$

We first consider possible dangling transitions. This can only happen if e_0 does the initialization first. Suppose $^\circ\alpha_i(a) = [^\circ R[\Box := \circ[^\circ\sigma]]]$, such that when $\texttt{initbeh}(v, v_1)$ is placed in the expression hole no transition is enabled. Then we insert the **newadr** transition in L_i and remove the hole, leaving some stuck expression in its place. As before we do not distinguish between different stuck expressions placed in the hole. Also, in π_1 there will be an extra uninitialized actor carried along untouched for each such insertion. These can also be ignored. Now we consider the transition l_i. Assume its focus is a and $^\circ\alpha_i(a) = [^\circ R[\Box := \circ[^\circ\sigma]]]$. Since we have removed holes with stuck initializations, we may assume that $\texttt{initbeh}(v, v_1)$ at this point. Let a_0 be fresh and define

$^\circ\alpha_{i+1} = {}^\circ\alpha_i\{a := [^\circ R[\Box := e\{x_0 := a_0, x_1 := \texttt{nil}\}[^\circ\sigma]]]a_0 := (?_a)\, a_1 := (v_1[^\circ\sigma])\}$

$\Box_{\mathbf{n-i}}$

$\Box_{\mathbf{commutes}}$

7.3. Equivalence by Two Stage Reduction

There is one remaining equivalence to establish using common reducts:

(if.lam) $\lambda x.\mathtt{if}(v, e_1, e_2) \cong \mathtt{if}(v, \lambda x.e_1, \lambda x.e_2) \quad x \notin \mathrm{FV}(v)$

The intuitive reasoning behind this equivalence is that for any closing substitution (allowing holes, and actor addresses in the range) the two expressions reduce to equivalent lambda expressions. In fact these lambda expressions have the property that when applied to any argument they reduce to a common expression.

The method developed so far requires reduction to a common local configuration in one stage. Thus we must elaborate the notion of a template to provide for two stages. Specifically, we add a family of holes for lambda-abstractions, which we denote by $\triangleright_j$ for $j \in J$ for some $J \in \mathbb{N} \cup \{\omega\}$.

7.3.1. LE-Syntax

Syntactic classes X with both expression and lambda holes are indicated by the mark ${}^{\circ\triangleright}X$, and we prefix the names of these classes by LE-, thus we have LE-expressions, LE-configurations, etc. The defining clauses are as before with two exceptions: lambda holes are added to the clause generating values; and $\mathtt{app}(\triangleright_j[{}^{\circ\triangleright}\sigma], {}^{\circ\triangleright}v)$ is omitted from the class of LE-redexes. The latter exception is made in order to preserve the property that redexes reduce uniformly.

Definition (${}^{\circ\triangleright}\mathbb{V}$, ${}^{\circ\triangleright}\mathbb{E}$, ${}^{\circ\triangleright}\mathbb{S}$, ${}^{\circ\triangleright}\mathbb{R}$, ${}^{\circ\triangleright}\mathbb{E}_{\mathrm{rdx}}$):

${}^{\circ\triangleright}\mathbb{V} = \mathbb{A}\mathrm{t} \cup \mathbb{X} \cup \lambda\mathbb{X}.{}^{\circ\triangleright}\mathbb{E} \cup \mathtt{pr}({}^{\circ\triangleright}\mathbb{V}, {}^{\circ\triangleright}\mathbb{V}) \cup \triangleright_{\mathbb{N}}[{}^{\circ\triangleright}\mathbb{S}]$

${}^{\circ\triangleright}\mathbb{E} = {}^{\circ\triangleright}\mathbb{V} \cup \Theta^{e}_{n}({}^{\circ\triangleright}\mathbb{E}^{n}) \cup \{\circ[{}^{\circ\triangleright}\mathbb{S}]\}$

${}^{\circ\triangleright}\mathbb{S} = \mathbb{X} \xrightarrow{\mathrm{f}} {}^{\circ\triangleright}\mathbb{V}$

${}^{\circ\triangleright}\mathbb{R} = \{\square\} \cup \Theta_{m+n+1}({}^{\circ\triangleright}\mathbb{V}^{m}, {}^{\circ\triangleright}\mathbb{R}, {}^{\circ\triangleright}\mathbb{E}^{n})$

${}^{\circ\triangleright}\mathbb{E}_{\mathrm{rdx}} = \Theta^{e}_{n}({}^{\circ\triangleright}\mathbb{V}^{n}) - \mathtt{app}(\triangleright_j[{}^{\circ\triangleright}\mathbb{S}], {}^{\circ\triangleright}\mathbb{V})$

Note that lambda holes can occur in the range of a value substitution, and as arguments in redices, except in the function position of an application. Using the double index convention, we write ${}^{\circ\triangleright}e[\triangleright_j := \varphi_j]$ to indicate the simultaneous filling of the holes $\triangleright_j$ with the corresponding lambdas φ_j from some previously specified family $\{\varphi_j\}_{j\in J}$ of lambda abstractions. The definitions of substitution, free variables, and hole filling are entirely analogous to the common reduct case and we omit them.

The decomposition lemma is modified as follows. An LE-expression ${}^{\circ\triangleright}e$ is either an LE-value expression (element of ${}^{\circ\triangleright}\mathbb{V}$), or it can be decomposed uniquely into an LE-reduction context with redex hole filled with either an LE-redex, an LE-expression hole, or an application of a lambda hole (to an LE-value).

Lemma (LE-expression decomposition):

(0) ${}^{\circ\triangleright}e \in {}^{\circ\triangleright}\mathbb{V}$, or

(1) $(\exists!\,{}^{\circ\triangleright}R, {}^{\circ\triangleright}r)({}^{\circ\triangleright}e = {}^{\circ\triangleright}R[\square := {}^{\circ\triangleright}r])$, or

(2) $(\exists!\,{}^{\circ\triangleright}R, {}^{\circ\triangleright}\sigma)({}^{\circ\triangleright}e = {}^{\circ\triangleright}R[\square := \circ[{}^{\circ\triangleright}\sigma]])$, or

(3) $(\exists!\,{}^{\circ\triangleright}R, {}^{\circ\triangleright}\sigma, {}^{\circ\triangleright}v)({}^{\circ\triangleright}e = {}^{\circ\triangleright}R[\square := \mathtt{app}(\triangleright_j[{}^{\circ\triangleright}\sigma], {}^{\circ\triangleright}v)])$

7.3.2. LE-computation

The definition of LE-configurations and LE-reduction are the natural extensions of E-configurations and E-reduction to the situation with lambda abstraction holes added. The definition of hole touching and the uniform computation lemmas genearalize easily to this situation.

Definition (${}^{\circ\triangleright}\mathbb{K}$):

$${}^{\circ\triangleright}\mathbb{K} = \left\langle\!\!\left\langle\, {}^{\circ\triangleright}\mathbb{A}\mathrm{c} \;\middle|\; {}^{\circ\triangleright}\mathbb{M} \,\right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

where

$${}^{\circ\triangleright}\mathbb{A}\mathrm{c} = \mathbb{A}\mathrm{d} \xrightarrow{\mathrm{f}} {}^{\circ\triangleright}\mathbb{A}\mathrm{s}$$

$${}^{\circ\triangleright}\mathbb{A}\mathrm{s} = (\lambda x.{}^{\circ\triangleright}\mathbb{E}) \cup [{}^{\circ\triangleright}\mathbb{E}] \cup \{(?_{\mathbb{A}\mathrm{d}})\}$$

$${}^{\circ\triangleright}\mathbb{M} = \mathtt{<}{}^{\circ\triangleright}\mathbb{V} \Leftarrow {}^{\circ\triangleright}\mathbb{V}\mathtt{>}$$

and the constraints specified in the definition of actor configurations in §2. are satisfied. We let ${}^{\circ\triangleright}\kappa$ range over ${}^{\circ\triangleright}\mathbb{K}$, and ${}^{\circ\triangleright}\alpha$ range over ${}^{\circ\triangleright}\mathbb{A}\mathrm{c}$ Filling expression and abstraction holes of an LE-configuration, LE-actor map, LE-actor state, LE-multiset of messages, and LE-messages is defined in the obvious manner.

An LE-configuration, ${}^{\circ}\kappa$, is closing for e and a family $\{\varphi_j\}_{j\in J}$ of lambda abstractions if $({}^{\circ\triangleright}\kappa[\circ := e])[\triangleright_j := \varphi_j]$ is a closed configuration.

Definition (LE-Reduction): The reduction relations $\overset{\lambda}{\mapsto}$ and $\mapsto$ are extended to the generalized domains in the obvious fashion, simply by liberally annotating metavariables with ${}^{\circ\triangleright}$'s. We omit the details.

Definition (LE-hole touching): If ${}^{\circ\triangleright}\kappa = \left\langle\!\!\left\langle\, {}^{\circ\triangleright}\alpha \;\middle|\; {}^{\circ\triangleright}\mu \,\right\rangle\!\!\right\rangle$, then ${}^{\circ\triangleright}\kappa$ touches a hole at a if ${}^{\circ\triangleright}\alpha(a) = [{}^{\circ\triangleright}e]$ and either ${}^{\circ\triangleright}e = {}^{\circ\triangleright}R[\square := \circ[{}^{\circ\triangleright}\sigma]]$ or ${}^{\circ\triangleright}e = {}^{\circ\triangleright}R[\square := \mathtt{app}(\triangleright_j[{}^{\circ\triangleright}\sigma], {}^{\circ\triangleright}v)]$. A transition from ${}^{\circ\triangleright}\kappa[\circ := e][\triangleright_j := \varphi_j]$ touches a hole at a if the focus actor of the transition is a and ${}^{\circ\triangleright}\kappa$ touches a hole at a.

Note that since an abstraction hole must be filled with a value, they are not touched in the same ways as arbitrary expression holes, in particular if the transition is a **<fun : a>** execution step where ${}^{\circ}\alpha(a) = [{}^{\circ}e]$ and ${}^{\circ}e = {}^{\circ\triangleright}R[\square := \mathtt{app}(\lambda x.{}^{\circ}e', \triangleright_j[{}^{\circ\triangleright}\sigma])]$, then this is not considered touching the hole, $\triangleright_j$.

The (**E-Uniform Computation**) lemma generalizes to the situation with added abstraction holes.

Lemma (LE-Uniform Computation):

(1) If ${}^{\circ\triangleright}\kappa \xrightarrow{l} {}^{\circ\triangleright}\kappa'$, then ${}^{\circ\triangleright}\kappa[\circ := e][\triangleright_j := \varphi_j] \xrightarrow{l} {}^{\circ\triangleright}\kappa'[\circ := e][\triangleright_j := \varphi_j]$ for any valid filling expression e and family of lambda abstractions φ_j.

(2) If ${}^{\circ\triangleright}\kappa$ has no transition with focus a (and a is an actor of ${}^{\circ\triangleright}\kappa$), then either ${}^{\circ\triangleright}\kappa$ touches a hole at a or ${}^{\circ\triangleright}\kappa[\circ := e][\triangleright_j := \varphi_j]$ has no transition with focus a for and valid filling expression e and family of lambda abstractions φ_j.

(3) If $\kappa \xrightarrow{l} \kappa'$ and $\kappa = {}^{\circ\triangleright}\kappa[\circ := e][\triangleright_j := \varphi_j]$, then either the transition touches a hole or we can find ${}^{\circ\triangleright}\kappa'$ such that $\kappa' = {}^{\circ\triangleright}\kappa'[\circ := e][\triangleright_j := \varphi_j]$ and ${}^{\circ\triangleright}\kappa \xrightarrow{l} {}^{\circ\triangleright}\kappa'$.

Proof : Similar to the proof of (**E-uniform computation**) Now there are two cases in which a hole is touched in the decomposition of ${}^{\circ\triangleright}e$, namely cases (2) and (3) of the decomposition lemma. □

7.3.3. LE-Main Theorem

Theorem (eq-reduct): Let $e_0, e_1, \varphi_{0,j}, \varphi_{1,j}$ for $j < J$ be such that for each ${}^{\circ\triangleright}\sigma$ we can find $j \in J$ such that $e_i[{}^{\circ\triangleright}\sigma]$ reduces uniformly via $\stackrel{\lambda}{\mapsto}$ steps to $\varphi_{i,j}[{}^{\circ\triangleright}\sigma]$ for $i < 2$, and that for each ${}^{\circ\triangleright}\sigma$, ${}^{\circ\triangleright}v$, and $j \in J$ we can find ${}^{\circ\triangleright}e_c$ such that $\mathtt{app}(\varphi_{i,j}[{}^{\circ\triangleright}\sigma], {}^{\circ\triangleright}v)$ reduces uniformly via $\stackrel{\lambda}{\mapsto}$ steps to ${}^{\circ\triangleright}e_c$ for $i < 2$. Then $e_0 \cong e_1$.

Corollary (eq-reduct): (**if.lam**) is an example. Here we take

$e_0 = (\lambda x.\mathtt{if}(v, e_a, e_b)$

$e_1 = \mathtt{if}(v, \lambda x.e_a, \lambda x.e_b))$ where $x \notin \mathrm{FV}(v)$

$J = \{a, b\}$

$\varphi_{0,j} = \lambda x.\mathtt{if}(v, e_a, e_b)$

$\varphi_{1,j} = \lambda x.e_j$ for $j \in J$

Proof : Let ${}^{\circ\triangleright}\kappa = \left\langle\!\!\left\langle\, {}^{\circ\triangleright}\alpha \;\middle|\; {}^{\circ\triangleright}\mu \,\right\rangle\!\!\right\rangle$ be a closing LE-configuration for $e_0, e_1, \varphi_{0,j}, \varphi_{1,j} \mid j \in J$. Assume $\pi_0 \in \mathcal{F}({}^{\circ\triangleright}\kappa[\circ := e_0][\triangleright_j := \varphi_{0,j}]) = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i \in \bowtie]$. We want to find ${}^{\circ\triangleright}\kappa_i$, and L_i, such that $\kappa_i = {}^{\circ\triangleright}\kappa_i[\circ := e_0]$ and, letting $\pi_1 = [{}^{\circ\triangleright}\kappa_i[\circ := e_1][\triangleright_j := \varphi_{1,j}] \xrightarrow{L_i} {}^{\circ\triangleright}\kappa_{i+1}[\circ := e_1][\triangleright_j := \varphi_{1,j}] \mid i \in \bowtie]$, we have $\pi_1 \in \mathcal{F}({}^{\circ\triangleright}\kappa[\circ := e_1])$ and $obs(\pi_0) = obs(\pi_1)$. At each stage i we first consider dangling steps. Suppose ${}^{\circ\triangleright}\alpha_i(a) = [{}^{\circ\triangleright}R[\square := \circ[{}^{\circ\triangleright}\sigma]]]$ and the reduction of e_0 is trivial, i.e. $e_0[{}^{\circ\triangleright}\sigma] = \varphi_{0,j}[{}^{\circ\triangleright}\sigma]$ for some j. Then we prefix L_i with the transitions for $e_1[{}^{\circ\triangleright}\sigma] \stackrel{\lambda}{\mapsto} \varphi_{1,j}[{}^{\circ\triangleright}\sigma]$ and convert the E-hole to $\triangleright_j$. This is done for each $a \in \mathrm{Dom}({}^{\circ}\alpha_i)$ meeting the condition.

Now we consider the decomposition of the configuration at stage $i+1$ in the case l_i touches a hole. Suppose l_i is an execution by a with ${}^{\circ\triangleright}\alpha_i(a) = [{}^{\circ\triangleright}R[\square := \circ[{}^{\circ\triangleright}\sigma]]]$. By the elimination of 'dangling steps' we may assume that e_0 is not a value expression and hence the execution occurs at the hole. Suppose also that $e_i[{}^{\circ\triangleright}\sigma] \stackrel{\lambda}{\mapsto} \varphi_{i,j}[{}^{\circ\triangleright}\sigma]$. Define ${}^{\circ\triangleright}\alpha_{i+1} = {}^{\circ\triangleright}\alpha_i\{a := [{}^{\circ\triangleright}R[\square := \triangleright_j[{}^{\circ\triangleright}\sigma]]]\}$.

Suppose l_i is an execution by a with ${}^{\circ\triangleright}\alpha_i(a) = [{}^{\circ\triangleright}R[\square := \mathtt{app}(\triangleright_j[{}^{\circ\triangleright}\sigma], {}^{\circ\triangleright}v)]]$. Suppose also that $\varphi_{i,j}[{}^{\circ\triangleright}\sigma]$ reduces to $e_c[{}^{\circ\triangleright}\sigma]$ by $\stackrel{\lambda}{\mapsto}$ steps. Define ${}^{\circ\triangleright}\alpha_{i+1} = {}^{\circ\triangleright}\alpha_i\{a := [{}^{\circ\triangleright}R[\square := e_c[{}^{\circ\triangleright}\sigma]]]\}$.

It is easy to check (as in the proof of (**fun-red-eq**)) that fairness is preserved. □

7.4. Equivalence of Reduction Contexts

To establish the equivalence of reduction contexts, we define templates for syntactic entities — expressions, reduction context, redexes, configurations — with holes to be filled by a reduction context. We then proceed as before, to show how configurations can be suitably decomposed in order to define the desired path correspondences.

7.4.1. R-Syntax

We use $\diamond$ for reduction context holes and signify the corresponding syntactic entities with a mark $\diamond$. We prefix names of templates for syntactic classes by R-, thus expression templates are called R-expressions, etc.

Definition ($^\diamond\mathbb{E}$ $^\diamond\mathbb{V}$):

$$^\diamond\mathbb{V} = \mathrm{At} \cup \mathbb{X} \cup \lambda\mathbb{X}.{}^\diamond\mathbb{E} \cup \mathbf{pr}({}^\diamond\mathbb{V}, {}^\diamond\mathbb{V})$$

$$^\diamond\mathbb{E} = {}^\diamond\mathbb{V} \cup \Theta^e_n({}^\diamond\mathbb{E}^n) \cup \diamond^{{}^\diamond\mathbb{S}}[\square := {}^\diamond\mathbb{E}]$$

$$^\diamond\mathbb{S} = \mathbb{X} \xrightarrow{\mathrm{f}} {}^\diamond\mathbb{V}$$

$^\diamond e[\diamond := R]$ is the result of filling R-holes in $^\diamond e$ with R. We give only the clause for the hole case in the recursive definition of filling.

Definition ($^\diamond e[\diamond := R]$):

$$(\diamond^{{}^\diamond\sigma}[\square := {}^\diamond e])[\diamond := R] = R[\sigma][\square := {}^\diamond e[\diamond := R]]$$

where $\sigma = {}^\diamond\sigma[\diamond := R] = \lambda x \in \mathrm{Dom}({}^\diamond\sigma).{}^\diamond\sigma(x)[\diamond := R]$

Definition ($^\diamond\mathbb{R}$ $^\diamond\mathbb{E}_{\mathrm{rdx}}$):

$$^\diamond\mathbb{R} = \{\square\} \cup \Theta_{m+n+1}({}^\diamond\mathbb{V}^m, {}^\diamond\mathbb{R}, {}^\diamond\mathbb{E}^n) \cup \diamond^{{}^\diamond\mathbb{S}}[\square := {}^\diamond\mathbb{R}]$$

$$^\diamond\mathbb{E}_{\mathrm{rdx}} = \Theta^e_n({}^\diamond\mathbb{V}^n)$$

The clauses directly involving holes in the definitions of hole filling for R-reduction contexts are:

$$\square[\diamond := R] = \square$$

$$\square[\square := {}^\diamond e] = {}^\diamond e$$

$$(\diamond^{{}^\diamond\sigma}[\square := {}^\diamond R])[\diamond := R] = R^\sigma[\square := {}^\diamond R[\diamond := R]]$$

where $\sigma = {}^\diamond\sigma[\diamond := R] = \lambda x \in \mathrm{Dom}({}^\diamond\sigma).{}^\diamond\sigma(x)[\diamond := R]$

$$(\diamond^{{}^\diamond\sigma}[\square := {}^\diamond R])[\square := {}^\diamond e] = \diamond^{{}^\diamond\sigma}[\square := {}^\diamond R[\square := {}^\diamond e]]$$

Note that filling R-holes in R-expressions, R-reduction contexts, or R-redexes with a reduction context yields an expression, a reduction context, or redex, respectively.

An R-expression $^\diamond e$ is either an R-value (element of $^\diamond\mathbb{V}$) or it can be decomposed uniquely into an R-reduction context filled with either an R-redex or an R-hole.

Lemma (R-expression decomposition):

(0) $^\diamond e \in {}^\diamond\mathbb{V}$, or

(1) $(\exists!{}^\diamond R, {}^\diamond r)({}^\diamond e = {}^\diamond R[\square := {}^\diamond r])$, or

(2) $(\exists!{}^\diamond R, {}^\diamond\sigma, {}^\diamond v)({}^\diamond e = {}^\diamond R[\square := \diamond^{{}^\diamond\sigma}[\square := {}^\diamond v]])$

Proof : An easy induction on the structure of $^\diamond e$. $\square$

7.4.2. R-Configurations

Definition ($^\diamond\mathbb{K}$): An R-configuration for reduction contexts $^\diamond\kappa$ is formed like a configuration, but using R-expressions instead of simple expressions.

$$^\diamond\mathbb{K} = \left\langle\!\!\left\langle\, ^\diamond\mathbb{A}\mathrm{c} \;\middle|\; ^\diamond\mathbb{M} \,\right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

$$^\diamond\mathbb{A}\mathrm{c} = \mathbb{A}\mathrm{d} \xrightarrow{\mathrm{f}} {}^\diamond\mathbb{A}\mathrm{s}$$

$$^\diamond\mathbb{A}\mathrm{s} = (\lambda x.{}^\diamond\mathbb{E}) \cup [{}^\diamond\mathbb{E}] \cup \{(?_{\mathbb{A}\mathrm{d}})\}$$

$$^\diamond\mathbb{M} = \mathtt{<}{}^\diamond\mathbb{V} \Leftarrow {}^\diamond\mathbb{V}\mathtt{>}$$

We let $^\diamond\kappa$ range over $^\diamond\mathbb{K}$, and $^\diamond\alpha$ range over $^\diamond\mathbb{A}\mathrm{c}$ Filling holes of an R-configuration is analogous to filling holes of an E-configuration. An R-configuration $^\diamond\kappa$ is closing for R if $^\diamond\kappa[\diamond := R]$ is a closed configuration.

7.4.3. R-Reduction

Definition (touching R-holes): A transition l from $^\diamond\kappa[\diamond := R]$ touches an R-hole at a if l is an execution transition with focus a and execution state of $^\diamond\kappa$ at a decomposes according to case (2) of the decomposition lemma.

7.4.4. R-Uniform Computation

Lemma (Uniform Computation): An R-redex reduces or hangs uniformly (for a given enabling occurrence in a configuration). Hence transitions not touching an R-hole are uniform. More precisely, if $^\diamond\kappa_i[\diamond := R] \xrightarrow{l_i} \kappa_{i+1}$, with focus a, that does not touch an R-hole at a, then l_i is either a receive or an execution transition in which the execution state of $^\diamond\kappa_i$ at a decomposes according to case (1) of the decomposition lemma. Let $^\diamond\kappa_i = \left\langle\!\!\left\langle\, ^\diamond\alpha_i \;\middle|\; ^\diamond\mu_i \,\right\rangle\!\!\right\rangle$. Then the decomposition of $\kappa_{i+1} = \left\langle\!\!\left\langle\, ^\diamond\alpha_{i+1} \;\middle|\; ^\diamond\mu_{i+1} \,\right\rangle\!\!\right\rangle$ is defined as follows.

Receive: In the receive case we must have that $^\diamond\alpha_i(a) = (\lambda x.{}^\diamond e)$. Thus $^\diamond\alpha_{i+1} = {}^\diamond\alpha_i\{a := \mathtt{[app}(\lambda x.{}^\diamond e, cv)\mathtt{]}$, and $^\diamond\mu_i = {}^\diamond\mu_{i+1} + \mathtt{<}a \Leftarrow cv\mathtt{>}$.

Uniform execution: In the uniform execution case $^\diamond\alpha_i(a) = [{}^\diamond e]$. where $^\diamond e$ has the form $^\diamond R[\square := {}^\diamond r]$. In this case the step is independent of what fills the holes. Thus we can find $^\diamond\kappa_{i+1}$ such that $^\diamond\kappa_i \xrightarrow{l_i} {}^\diamond\kappa_{i+1}$ uniformly.

Now we show how to establish equivalence of expressions of the form $R_j[\square := e]$ for $j < 2$ where the $R_j[\square := v]$ have a common reduct for any value expression.

Theorem (eq-r): If for z fresh, there is some e such that $R_j[\square := z]$ reduces uniformly via 0 or more $\xrightarrow{\lambda}$ steps to e for $j < 2$, then $R_0[\square := e] \cong R_0[\square := e]$ for any e.

Corollary (eq-r): (**app**), (**cmps**), (**id**), (**letx**), (**let.dist**), (**if.dist**) are instances of (**eq-r**).

In fact we prove a slightly more general result, since we use a weaker assumption on the reduction points: for each $^\diamond\sigma$, $^\diamond v$, we can find $^\diamond e_2$ such that $R_j^{\diamond\sigma}[\square := {}^\diamond v]$ reduces uniformly via 0 or more $\xrightarrow{\lambda}$ steps to $^\diamond e_2$ for $j < 2$.

Proof (eq-r): Suppose R_0, R_1 are reduction points that we wish to establish the observational equivalence of. I.e. we want to show $R_0[\Box := e] \cong R_1[\Box := e]$ for all expressions e. Let ${}^{\diamond}\kappa = \left\langle\!\left\langle {}^{\diamond}\alpha \;\middle|\; {}^{\diamond}\mu \right\rangle\!\right\rangle$ be a closing R-configuration for R_0, R_1. Assume $\pi_0 \in \mathcal{F}({}^{\diamond}\kappa[\diamond := R_0]) = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i \in \bowtie]$. We want to find ${}^{\diamond}\kappa_i$, and L_i, such that $\kappa_i = {}^{\diamond}\kappa_i[\diamond := R_0]$ and, letting $\pi_1 = [{}^{\diamond}\kappa_i[\diamond := R_1] \xrightarrow{L_i} {}^{\diamond}\kappa_{i+1}[\diamond := R_1] \mid i \in \bowtie]$, we have $\pi_1 \in \mathcal{F}({}^{\diamond}\kappa[\diamond := R_1])$ and $obs(\pi_0) = obs(\pi_1)$. As in the expression context case, we can focus our attention on the construction of the actor configuration part, since here also deliverable messages can not have holes. For the base case we have ${}^{\diamond}\alpha_0 = {}^{\diamond}\alpha$. Assume we have ${}^{\diamond}\alpha_i$ and consider cases on the transition label l_i. we have three cases to consider. For the base case we have ${}^{\diamond}\alpha_0 = {}^{\diamond}\alpha$. At stage i suppose ${}^{\diamond}\alpha_i(a') = {}^{\diamond}R[\Box := \diamond^{\diamond\sigma}[\Box := {}^{\diamond}v]]$. If $R_0^{\diamond\sigma}[\Box := {}^{\diamond}v]$ is the common form (i.e. the reduction is trivial), then we prefix L_i with steps for reduction of $R_1^{\diamond\sigma}[\Box := {}^{\diamond}v]$ to common form and remove this hole. This is carried out for each a' in the domain of ${}^{\diamond}\alpha_i$.

Now, suppose l_i is an execution with focus a that touches a hole. Thus ${}^{\diamond}\alpha_i(a) = [{}^{\diamond}e]$ where ${}^{\diamond}e$ has the form ${}^{\diamond}R[\Box := \diamond^{\diamond\sigma}[\Box := {}^{\diamond}v]]$. Suppose also that $R_j^{\diamond\sigma}[\Box := {}^{\diamond}v] \overset{\lambda}{\mapsto} {}^{\diamond}e_c$ for $j \in 2$. Then we define ${}^{\diamond}\alpha_{i+1} = {}^{\diamond}\alpha_i\{a := [{}^{\diamond}R[\Box := {}^{\diamond}e_2]]\}$.

Let $\pi_1 = [{}^{\diamond}\kappa_i[\diamond := e_1] \xrightarrow{L_i} {}^{\diamond}\kappa_{i+1}[\diamond := e_1] \mid i < \bowtie]$ be the constructed computation path. Note that the transitions are the same except for the points where holes are touched, but these differences are not observable. Clearly, under the assumptions on R_j, π_1 is a computation path. It remains to show that the construction preserves fairness.

Suppose some transition l is enabled at stage i in π_1. If l is a receive or uniform execution, then l is also enabled in π_0 at stage i and will eventually occur, uniformly. Suppose l is an execution step by a with ${}^{\diamond}\alpha_i(a) = [{}^{\diamond}R[\Box := \diamond^{\diamond\sigma}[\Box := {}^{\diamond}v]]]$. Either $R_0^{\diamond\sigma}[\Box := {}^{\diamond}v]$ is in common form and the transition occurs as soon as it is enabled in π_1, or a transition is enabled at this hole in π_0. This transition will eventually occur. If $R_1^{\diamond\sigma}[\Box := {}^{\diamond}v]$ is not in common form, l will happen in π_1 at the same stage. If $R_1^{\diamond\sigma}[\Box := {}^{\diamond}v]$ is in common form, then we must consider whether l occurs at the hole – i.e. whether or not the common form is an R-value expression. If l occurs at the hole, then the same transition is now enabled in π_0 and will eventually occur in both paths. Otherwise consider the decomposition after the transition in π_0. As shown before, this reduces to considering a proper subexpression of ${}^{\diamond}R[\Box := \diamond^{\diamond\sigma}[\Box := {}^{\diamond}v]]$ and the process will eventually terminate with a uniform execution.

$\Box_{\mathbf{eq-r}}$

8. Discussion

In this paper we have presented a mathematical theory of actor computation.

8.1. Highlights

The highlights of this theory include:

(1) a language for actor computation which is an extension of the call-by-value lambda calculus that includes primitives for creating and manipulating actors;

(2) the concept of an *actor configuration* that makes explicit the notion of open system component;

(3) an operational semantics defined by a labelled transition relation on configurations;

(4) a notion of computation that incorporates fairness is an essential way;

(5) a notion of observational equivalence of expressions based on traditional operational and testing equivalence;

(6) a proof that in the presence of fairness the classic three testing equivalences collapse to two;

(7) a plethora of laws of expression equivalence that incorporates the equational theory of the embedded functional language;

(8) a composition operation on actor configurations that is associative, commutative, has a unit, and is thus a first step towards an algebra of configurations;

(9) a notion of interaction equivalence of actor configurations that generalizes traditional trace equivalence;

(10) a notion of observational equivalence of actor configurations that generalizes the same notion on expressions;

(11) a collection of results that establish that interaction equivalence implies observational equivalence, and is a congruence with respect to composition;

(12) examples of interaction equivalent configurations, and methods for establishing their equivalence;

(13) methods for establishing equivalence of expressions.

8.2. Future Directions

The theory presented is perhaps best viewed as a starting point for further research rather than a final product. In particular there at least 3 directions for further research:

(a) developing an algebra of operations on configurations;

(b) a systematic study of notions of equivalence within the actor theory;

(c) developing a logic for specifying components as actor configurations, methods for verifying that programs implementing components meet their specifications, and methods for refining specifications into implementations.

and 3 open problems:

(a) does observational equivalence imply interaction equivalence?

(b) is the conjecture (**io-exp**) true?

(c) is there a context lemma (similar to (**ciu**)) for observational equivalence of expressions?

Furthermore, in hindsight, the theory as presented here might benefit from:

(a) replacing the `newadr`, `initbeh` choice of language primitives by a `letactor` primitive;

(b) making the operational semantics truly concurrent;

(c) mechanical checking of the more tedious parts of theory, for example the (**independence**) lemma.

Acknowledgements

The authors would like to thank Carl Hewitt and Richard Weyhrauch for many discussions about actor computation that served as a foundation for this work.

This research was partially supported by DARPA contract NAG2-703, NSF grants CCR-8917606, CCR-8915663, CCR-9109070 and INT-89-20626, and DARPA and NSF joint contract CCR 90-07195, ONR contract N00014-90-J-1899, and by the Digital Equipment Corporation.

9. References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Attardi and C. Hewitt. Proving properties of concurrent programs expressed as behavioral specifications. In *Summer School on Foundations of Artifical Intelligence and Computer Science (Pisa 1978)*, 1978.

[4] G. Attardi and C. Hewitt. Specifying and proving properites of guardians for distributed systems, 1978.

[5] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[6] D. Berry, R. Milner, and D.N. Turner. A semantics for ML concurrency primitives. In *Conference record of the 19th annual ACM symposium on principles of programming languages*, pages 105–129, 1992.

[7] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[8] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[9] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.

[10] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[11] M. Felleisen and A. K. Wright. A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Rice University Computer Science Department, 1991.

[12] C. Hewitt. *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot.* PhD thesis, MIT, 1971.

[13] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[14] C. Hewitt and R. Atkinson. Synchronization in actor systems. In *Conference record of ACM symposium on principles of programming languages*, pages 267–280, 1977.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[16] H. Lieberman. A preview of act i. AI-TR- 625, MIT Artificial Intelligence Laboratory, 1981.

[17] H. Lieberman. Thinking about lots of things at once without getting confused: Parallelism in act i. AI-TR- 626, MIT Artificial Intelligence Laboratory, 1981.

[18] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[19] J. Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of Concur'90*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer-Verlag, 1990.

[20] R. Milner, J. G. Parrow, and D. J. Walker. A calculus of mobile processes, parts i and ii. Technical Report ECS-LFCS-89-85, -86, Edinburgh University, 1989.

[21] E. Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988.

[22] J. H. Morris. *Lambda calculus models of programming languages.* PhD thesis, Massachusetts Institute of Technology, 1968.

[23] G. Plotkin. Call-by-name, call-by-value and the lambda-v-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[24] J. H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Cornell University, 1991.

[25] J. H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, 1992. available as Technical Report 92-1285.

[26] V. Sassone, M. Nielsen, and G. Winskel. A hierarchy of models for concurrency. In *The Fourth International Conference on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science. Springer Verlag, 1993.

[27] C. L. Talcott. *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation.* PhD thesis, Stanford University, 1985.

[28] C. L. Talcott. Programming and proving with function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department, 1989.

[29] C. L. Talcott. A theory for program and data specification. In *Design and Implementation of Symbolic Computation Systems, DISCO'90*, volume 429 of *Lecture Notes in Computer Science*, pages 91–100. Springer-Verlag, 1990. Full version to appear in TCS special issue.

[30] C. L. Talcott. Binding structures. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*. Academic Press, 1991.

[31] C. L. Talcott. A theory for program and data specification. *Theoretical Computer Science*, 104:129–159, 1993.

[32] C.L. Talcott. A theory of binding structures and its applications to rewriting. *Theoretical Computer Science*, to appear, 1991.

[33] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge Mass., 1990.

10. Index of Notations

Symbol	Description	§
$\mathbb{N}$	The natural numbers, $i, j, \ldots, n \in \mathbb{N}$	1.3
Y^n	Sequences from Y of length n, $\bar{y} = [y_1, \ldots, y_n] \in Y^n$	1.3
Y^*	Finite sequences from Y	1.3
$[]$	The empty sequence	1.3
$\mathrm{Len}(\bar{y})$	The length of the sequence $\bar{y}$	1.3
$u * v$	The concatenation of the sequences u and v	1.3
$\mathrm{Last}(u)$	The last element of the sequence u	1.3
$\mathbf{P}_\omega[Y]$	Finite subsets of Y	1.3
$\mathbf{M}_\omega[Y]$	Finite multi-sets with elements in Y	1.3
$Y_0 \xrightarrow{\mathrm{f}} Y_1$	Finite maps from Y_0 to Y_1	1.3
$Y_0 \to Y_1$	Total functions from Y_0 to Y_1	1.3
$\mathrm{Dom}(f)$	The domain of the function f	1.3
$\mathrm{Rng}(f)$	The range of the function f	1.3
$f\{y := y'\}$	An extension to, or alteration of, the function f	1.3
$f \rfloor Y$	The restriction of f to the set Y	1.3
$\mathbb{X}$	A countably infinite set of variables, $x, y, z \in \mathbb{X}$	2.1
$\mathbb{A}\mathrm{t}$	Atoms	2.1
$\mathtt{t}, \mathtt{nil}$	Atoms playing the role of booleans	2.1
$\mathbb{F}$	Operations, $\delta \in \mathbb{F}$	2.1
$\mathbb{F}_n$	n-ary operation symbols	2.1
$\mathbb{F}_0$	Zero-ary operation symbols $\supseteq \{\mathtt{newadr}\}$	2.1
$\mathbb{F}_1$	Unary operation symbols $\supseteq \{\mathtt{isatom}, \mathtt{isnat}, \mathtt{ispr}, \mathtt{1^{st}}, \mathtt{2^{nd}}, \mathtt{become}\}$	2.1
$\mathbb{F}_2$	Binary operation symbols $\supseteq \{\mathtt{pr}, \mathtt{initbeh}, \mathtt{send}\}$	2.1
$\mathbb{F}_3$	Ternary operation symbols $\supseteq \{\mathtt{br}\}$	2.1
$\mathbb{L}$	λ-abstractions, $\lambda x.e \in \mathbb{L}$	2.1
$\mathbb{V}$	Value expressions, $v \in \mathbb{V}$	2.1
$\mathbb{E}$	Expressions, $e \in \mathbb{E}$	2.1
$\lambda x.e$	Abstractions	2.1
$\mathtt{app}(e_0, e_1)$	Application	2.1
$\delta(\bar{e})$	Application of operations	2.1
$\mathtt{if}(e_0, e_1, e_2)$	Conditional branching	2.1
$\mathtt{let}\{x := e_0\}e_1$	Lexical variable binding	2.1
$\mathtt{seq}(e_1, \ldots, e_n)$	Sequencing construct	2.1
$\mathrm{FV}(e)$	The free variables of the expression e	2.1
$ex[e']$	The result of substituting e' for x in e	2.1
$\mathbb{C}$	Contexts, $C \in \mathbb{C}$	2.1
$\bullet$	The hole in contexts	2.1
$C[e]$	The result of filling the context with e	2.1

Symbol	Description	§
$\mathbb{A}\mathrm{d}$	Actor addresses, identified with $\mathbb{X}$	2.2
$\mathbb{A}\mathrm{s}$	Actor states, $(?_a)$, (b), $[e] \in \mathbb{A}\mathrm{s}$	2.2
$(?_a)$	An uninitialized actor created by a	2.2
(b)	An actor with behavior b ready to accept a message	2.2.1
$[e]$	A processing actor with current computation e	2.2.1
$\mathbb{M}$	Messages, $\texttt{<}\mathbb{V} \Leftarrow \mathbb{V}\texttt{>} \in \mathbb{M}$	2.2.1
$\mathrm{c}\mathbb{V}$	Communicable values, $cv \in \mathrm{c}\mathbb{V}$	2.2.1
$\left\langle\!\!\left\langle\, \alpha \mid \mu \,\right\rangle\!\!\right\rangle_{\chi}^{\rho}$	An actor configuration with:	
	α – an actor map	
	μ – a multi-set of messages	
	ρ – the receptionists	
	χ – the external actors	2.2.1
$\mathbb{K}$	Actor configurations, $\kappa \in \mathbb{K}$	2.2.2
$\mathbb{E}_{\mathtt{rdx}}$	The set of redexes, $r \in \mathbb{E}_{\mathtt{rdx}}$	2.2.2
$\mathbb{R}$	The set of reduction contexts, $R \in \mathbb{R}$	2.2.2
$\square$	The reduction context hole	2.2.2
$\overset{\lambda}{\mapsto}$	The reduction relation for functional redexes, $e_0 \overset{\lambda}{\mapsto} e_1$	2.2.2
$\mapsto$	The reduction relation for configurations, $\kappa_0 \mapsto \kappa_1$	2.2.2
$\kappa_0 \xrightarrow{l} \kappa_1$	$\kappa_0 \mapsto \kappa_1$ via the rule labelled by l	2.2.2
Labels :	Transition labels, $l \in$ Labels	2.2.2
<fun : a**>**	A functional transition	2.2.2
<new : a, a'**>**	**newadr** redex transition with focus a	2.2.2
<init : a, a'**>**	**initbeh** redex transition with focus a	2.2.2
<bec : a, a'**>**	**become** redex transition with focus a	2.2.2
<send : a, m**>**	**send** redex transition with focus a	2.2.2
<rcv : a, cv**>**	The receipt of a message with focus a	2.2.2
<out : m**>**	A message exiting the configuration	2.2.2
<in : m**>**	A message entering the configuration	2.2.2
$\mathcal{T}(\kappa)$	All finite sequences of labeled transitions from κ, $\nu \in \mathcal{T}(\kappa)$	2.2.3
$\mathcal{T}^{\infty}(\kappa)$	the set of all computation paths in $\mathcal{T}(\kappa)$, $\pi \in \mathcal{T}^{\infty}(\mathbb{K})$	2.2.3
$\bowtie \in \mathbb{N} \cup \{\omega\}$	The length of a finite or infinite sequence	2.2.3
$\mathrm{Cfig}(\kappa, L, i)$	The ith configuration of the computation from κ via L	2.2.3
$\kappa \xrightarrow{L} \kappa'$	A multi-step transition	2.2.3
$\mathcal{F}(\kappa)$	the fair subset of $\mathcal{T}^{\infty}(\kappa)$	2.2.4
event	A zero-ary primitive/observation	3.1
<e : a**>**	An observation transition	3.1
$\left\langle\!\!\left\langle\, \alpha, [C]_a \mid \mu \,\right\rangle\!\!\right\rangle$	An observing configuration	3.1
$\mathbb{O}$	The set of observing configurations, $O \in \mathbb{O}$	3.1

Symbol	Description	§
$\mathbf{s}$	Signifies that an `event` transition occurs	3.1
$\mathbf{f}$	Signifies that an `event` transition does not occur	3.1
$obs(\pi)$	The $\mathbf{s}/\mathbf{f}$ classification of the path π	3.1
$Obs(\kappa)$	The $\mathbf{s}/\mathbf{f}$ classification of the configuration, κ, $\in \{\mathbf{s}, \mathbf{f}, \mathbf{sf}\}$	3.1
$e_0 \cong_1 e_1$	Testing or Convex or Plotkin or Egli-Milner equivalence	3.2
$e_0 \cong_2 e_1$	Must or Upper or Smyth equivalence	3.2
$e_0 \cong_3 e_1$	May or Lower or Hoare equivalence	3.2
$e_0 \cong e_1$	Operational equivalence (either $\cong_1$ or equivalently $\cong_2$)	3.2
Hang	The set of all stuck expressions, `stuck` $\in$ *Hang*	4.1
Infin	The set of all diverging expressions, `bot` $\in$ *Infin*	4.1
$\kappa_0 \parallel \kappa_1$	The composition of the two configurations	5.1
`i/o`	The interactions (Input/Output) of a path or configuration	5.1.1
sMerged	The merge relation on finite computations	5.1.1
`<io` : a, cv`>`	A pseudo transition	5.1.1
$\mathcal{T}(\kappa_0) \parallel \mathcal{T}(\kappa_1)$	The merge operation on trees	5.1.2
$\kappa_0 \cong_{\mathtt{i}} \kappa_1$	Interaction equivalence between configurations	5.2.1
$\kappa_0 \cong \kappa_1$	Obesrvational equivalence between configurations	5.2.2
`<exec` : a`>`	Any transition with *focus* a, other than `rcv`	5.3
$l_0 - l_1$	l_0 and l_1 are independant transitions	5.3
$\circ^{\circ\sigma}$	An expression hole with substitution $^\circ\sigma$	7.1
Θ_n	Syntactic constructs of arity n	7.1
Θ_n^e	Syntactic constructs of arity n (including `event`)	7.1
$^\circ\mathbb{E}$	Expressions with expression holes, $^\circ e \in {}^\circ\mathbb{E}$	7.2.1
$^\circ\mathbb{V}$	Value expressions with expression holes, $^\circ v \in {}^\circ\mathbb{V}$	7.2.1
$^\circ\mathbb{S}$	Substitutions with expression holes, $^\circ\sigma \in {}^\circ\mathbb{S}$	7.2.1
$^\circ\mathbb{R}$	Reduction contexts with expression holes, $^\circ R \in {}^\circ\mathbb{R}$	7.2.1
$^\circ\mathbb{E}_{\mathtt{rdx}}$	Redexes with expression holes, $^\circ r \in {}^\circ\mathbb{E}_{\mathtt{rdx}}$	7.2.1
$^\circ e[\circ := e]$	Expression hole filling	7.2.1
$^\circ\mathbb{K}$	Configurations with expression holes $^\circ\kappa \in {}^\circ\mathbb{K}$	7.2.3
$^\circ\mathbb{A}\mathrm{c}$	An actor map with expression holes $^\circ\alpha \in {}^\circ\mathbb{A}\mathrm{c}$	7.2.3
$^\circ\mathbb{M}$	Messages with expression holes	7.1

Symbol	Description	§
$\triangleright_j$	A family of abstraction holes	7.3
$X^{\circ\triangleright}$	The syntactic class X with expression and abstraction holes	7.3
${}^{\circ\triangleright}\mathbb{V}$	Value expressions with expression and abstraction holes, ${}^{\circ\triangleright}v \in {}^{\circ\triangleright}\mathbb{V}$	7.3
${}^{\circ\triangleright}\mathbb{E}$	Expressions with expression and abstraction holes, ${}^{\circ\triangleright}e \in {}^{\circ\triangleright}\mathbb{E}$	7.3
$\triangleright_{\mathbb{N}}^{\circ\triangleright\mathbb{S}}$	Abstraction holes annotated with substitutions	7.3
${}^{\circ\triangleright}e[\triangleright_j := \varphi_j]$	Abstraction hole filling	7.3
$\diamond^{\diamond\sigma}$	A reduction context hole	7.4
${}^{\diamond}\mathbb{E}$	Expressions with reduction context holes, ${}^{\diamond}e \in {}^{\diamond}\mathbb{E}$	7.4
${}^{\diamond}\mathbb{V}$	Value expressions with reduction context holes, ${}^{\diamond}v \in {}^{\diamond}\mathbb{V}$	7.4
${}^{\diamond}\mathbb{S}$	Substitutions with reduction context holes ${}^{\diamond}\sigma \in {}^{\diamond}\mathbb{S}$	7.3
${}^{\diamond}e[\diamond := R]$	Reduction context hole filling	7.4
${}^{\diamond}\mathbb{R}$	Reduction contexts with reduction context holes, ${}^{\diamond}R \in {}^{\diamond}\mathbb{R}$	7.4
${}^{\diamond}\mathbb{E}_{\mathtt{rdx}}$	Redexes with reduction context holes, ${}^{\diamond}r \in {}^{\diamond}\mathbb{E}_{\mathtt{rdx}}$	7.4
${}^{\diamond}\mathbb{K}$	Configurations with reduction context holes, ${}^{\diamond}\kappa \in {}^{\diamond}\mathbb{K}$	7.4
${}^{\diamond}\mathbb{A}\mathrm{c}$	Actor maps with reduction context holes, ${}^{\diamond}\alpha \in {}^{\diamond}\mathbb{A}\mathrm{c}$	7.4
${}^{\diamond}\mathbb{M}$	Messages with reduction context holes	7.4
${}^{\diamond}\mathbb{A}\mathrm{s}$	Actor states with reduction context holes	7.4

Contents