TOWARDS AN ALGEBRAIC FORMULATION OF ACTORS

BY

PRASANNAA THATI

B.Tech., Indian Institute of Technology at Kanpur, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

ABSTRACT

We present a fair asynchronous name-passing calculus for the Actor Model called System A. System A is based on the primitive notion of naming and is equipped with combinators which are standard in process algebra literature, such as parallel composition, restriction, input and output prefixes, and matching. We capture features specific to the Actor Model through a type system which enforces a discipline on the usage of names. We define the semantics of System A using a labeled transition system and by eliminating infinite sequences of transitions that are unfair. We believe an algebraic formulation of actors will lead to many conceptual and practical benefits. Mapping the Actor Model to a framework shared by other highly successful formalisms for concurrency such as the π-calculus and its asynchronous variants, will help us better understand the similarities and differences between the various formalisms, find connections between their theories, and apply concepts and techniques developed for one to the other. We illustrate our calculus through some examples, and identify the current and future research directions.

To my grandfather P.Laxmanna

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Gul Agha, for his support and intellectual guidance. A special thanks to him for fostering a flexible research environment which helped me discover my true interests.

I am very fortunate to have received the guidance of Dr. Carolyn Talcott at Stanford University, where a significant portion of this research was conducted. My warmest thanks to her for making the visit possible, and for the extensive discussions and comments on the initial drafts of this dissertation.

I gratefully thank Reza Ziaei for suggesting the idea of an algebraic formulation of actors. The initial discussions with him have been of great help and have significantly influenced my research. I am greatly indebted to him. I have also benefited from discussions with other former and current members of the Open Systems Lab. In particular, I would like to thank Mark Astley, Nadeem Jamali, Wooyoung Kim, Yusuke Tada, Abhay Vardhan, Carlos Varela, Nalini Venkatasubramanian, James Waldby, and Joonkyoo Yoo.

This dissertation would not have been possible without the never-ending patience and encouragement from my parents and brother. My heartfelt thanks to them. Finally, I cannot even begin to express my gratitude to my grandfather. He has greatly influenced my philosophy of life. This thesis is dedicated to him.

TABLE OF CONTENTS

Chapter

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The Actor Model was originally proposed by Hewitt [19] and its meaning has evolved over years [17, 20, 7, 12] to that described in [1]. Since its conception it has profoundly influenced research in a number of areas. The concept of continuation passing now common in functional programming was first demonstrated in [20] and carried over into Scheme. A number of concurrent object oriented languages [2, 52, 28, 27, 14, 56] have been designed based on the Actor Model. The model has been the basis for a variety of projects on high performance computing [37, 24, 36, 25]. It has also served as the foundation for dozens of projects on abstractions for simplifying the task of developing and maintaining open distributed systems [5, 11, 15, 54, 43, 48, 53].

In this dissertation we describe our attempt to capture the Actor Model in a framework that is shared by highly successful formalisms such as the π-calculus [34, 35] and its asynchronous variants [22, 10], and is very similar to those used by CCS [31] and CSP [21]. Specifically, we present a formal system called System A, which is based on the primitive notion of naming, and is equipped with standard combinators such as parallel composition, restriction, input and output prefixes, and matching. Features specific to the Actor Model are captured using a type system which enforces certain discipline on the usage of names. The semantics of System A is defined using labeled transitions and by eliminating infinite sequences of transitions that are unfair.

Research on actor semantics has seen important advances through [12, 3, 50, 49, 16]. So why yet another formalism for actors? There are two main reasons. First, investigations such as [3, 16] are not about the Actor Model per se. They do not faithfully capture all features of the Actor Model and are also loaded with extraneous details which complicate their theories. Second, while [12, 50, 49] are accurate formulations, little is known about their relation to similar semantical investigations in other well-understood and widely accepted process algebras. We will examine these shortcomings in greater detail after we briefly discuss the Actor Model. Besides being accurate and simple, our algebraic formulation of actors will lead to many conceptual and practical benefits. It would help us identify the similarities and differences between the Actor Model and other process algebras such as π-calculus, find connections between their theories, and apply concepts and techniques developed for one to the other.

In Chapter 2, we first briefly describe the Actor Model. We then examine the most prominent works on actor semantics and identify their inadequacies. In Chapter 3, we present the syntax and type system of System A and compare it with π-calculus and its variants. System A was originally inspired by π-calculus, and is close to being a typed version of polyadic π-calculus [32]. In Chapter 4, we present an operational semantics for System A. In Chapter 5, we illustrate our formalism through several examples. This dissertation is best viewed as a starting point for further research rather than a final product. We are currently investigating equivalence notions, their denotational characterizations, and algebraic theories. In Chapter 6, we identify some directions for further research.

Chapter 2

The Actor Model

In this chapter we give a brief and informal discussion of the Actor Model. We then briefly discuss event diagrams which serve as an abstract visualization tool for computation in actor systems. Finally, we discuss some contemporary formulations of the Actor Model and their shortcomings. Discussions in this chapter necessarily omit some of the significant work in actor research. However, they are sufficient for the scope of this thesis.

2.1 The Model

2.1.1 What are Actors?

A computational system in the Actor Model, also called a *configuration*, consists of a collection of concurrently executing actors and a collection of messages in transit, and has an interface to its external environment. Actors in the configuration and the environment have a unique name and a behavior, and communicate via asynchronous messages. A message contains values targeted to a single actor called the target actor. A message is consumed once it is received by the target, and therefore cannot be delivered again. Actors are reactive in nature; they execute only in response to messages received. The

interface of a configuration imposes restrictions on communication between actors in the configuration and the environment (see Section 2.1.3).

On receiving a message, an actor can perform three basic actions:

1. Create a finite number of actors with universally fresh names.
2. Send a finite number of messages to other actors.
3. Assume a new behavior with the same name.

Several observations are in order here. First, all the actions performed on receiving a message are concurrent[1]; there is no sequential ordering between any two of them. Second, actors do not have shared state; information flow in the Actor Model happens strictly by means of messages. An actor behavior can be viewed as a function from messages (received) to a collection of messages (sent), actors (created), and a (replacement) behavior. Specifically, delivery of a message effects the behavior of only the target actor. Third, actors persist in that they do not disappear after processing a message; they assume a new behavior with the same name. Since the new behavior may depend on the message received, an actor's behavior can be history sensitive. Finally, actors are created with universally fresh names; an actor can not create new actors with names received in a message or with names already known to other actors.

2.1.2 Locality Laws

The following restrictions, known as the locality laws, apply to all actor systems:

1. An actor knows the addresses of only a finite number of other actors known as its *acquaintances*. These acquaintances are a part of its behavior.

[1]Although, semantically, the actions an actor performs on receiving a message are concurrent, an implementation of the actions will have some sequentiality. For instance, since newly created actors can refer to each other, all their names have to be known before any of the actors can be created. Moreover, the behavior of an actor can be conditional on matching of names received, further introducing sequentiality.

2. Messages can carry only a finite number of names.

3. Actors cannot guess names. Specifically, when an actor a receives a message m: the acquaintances of a after it assumes the new behavior, the acquaintances of newly created actors, and the target and names in contents of messages sent by a, are all either acquaintances of a before receiving m, or contents of m, or names of the newly created actors.

Several observations can be made from the last postulate. First, an actor can send messages to only those actors it knows after receiving a message. Second, the acquaintances of an actor may evolve as computation proceeds; the actor may remember new names received in the message and forget names it knew earlier. This captures the dynamic nature of communication topology in an actor system. Finally, although actor creations on receiving a message are concurrent, the newly created actors may know each other.

2.1.3 Encapsulation

The description of a configuration defines not only the actors and messages present in the configuration, but also an interface to the environment. Actors inside the configuration, called internal actors for brevity, may send messages to and receive messages from actors outside the configuration. The interface of a configuration imposes restrictions on these communications with the environment.

The interface of a configuration consists of two components: a finite set ρ of *receptionist* names, and finite set χ of *external* names. Receptionists are actors in the configuration which are visible from outside the configuration. Therefore, if $dom(C)$ is the set of names of all actors in C then $\rho \subset dom(C)$. Only receptionists may receive messages from the environment. Names of other internal actors are unknown to the environment. This also implies that messages from the environment cannot contain names of these hidden actors. Thus, the set ρ imposes a constraint on the both the target and contents of messages that the configuration may receive from its environment. Names in a configuration which

designate actors outside the configuration constitute the set of external names. Therefore, $\chi \cap dom(C) = \emptyset$. These names may be present in the configuration either as acquaintances of internal actors or in messages inside the configuration.

Messages in a configuration that are targeted to an external actor are visible to and consumed by the environment. However, messages that are targeted to internal actors (including the receptionists), are not observable externally. Thus, strictly speaking, it is not the visibility of internal actor names, but communication to internal actors that is restricted by the interface. Readers familiar with π-calculus [34, 35] may note the difference from the restriction operation in it which controls the visibility of names.

The interface of a configuration is dynamic. As the configuration evolves, new receptionists may be added and new external actors may become known. In a communication from the environment, an internal actor may receive new names of actors outside the configuration, thus expanding the set χ. Similarly, an internal actor may send a message containing the name of a non-receptionist to an external actor, thereby expanding ρ.

2.1.4 Fairness

Message delivery in the Actor Model is fair - the delivery of a message can not be infinitely delayed. However, it can be delayed for an arbitrary but finite time. Fairness guarantees that an actor makes progress independent of other actors. Without this feature certain intuitively obvious equivalences between actor configurations do not hold under many notions of equivalences. For example, consider a notion of equivalence in which configurations which can engage in the same set of interactions with their environment are equated. This description of the equivalence notion is admittedly vague, and several details need to be filled in. Nevertheless, it should suffice for the following example which shows that in the absence of fairness collection of active garbage does not preserve semantics.

Let C_1 be a configuration with a single internal actor a, an empty message to a, and an interface with no receptionists and external names. On receiving the message, actor

a simply resends the message to itself and retains the same behavior. Now, consider a configuration C_2 with a single internal actor b, an empty message to b, and no receptionists or external names. On receiving the message, actor b sends a single empty message to an external actor c and retains the same behavior. Now, consider the configuration C_3 which is obtained by composing C_1 and C_2 (see the definition of composition below). Intuitively, our notion of equivalence should equate C_2 and C_3. The expected interaction with the environment in both cases consists of a single message to c. But, this is true only in the presence of fairness. In the absence of fairness, in C_3, the internal actor a may starve the actor b by not allowing it to receive messages.

Note that the above equivalence will hold even if a weaker notion of fairness is guaranteed, namely that no actor is ever starved, i.e. if there are messages that can be delivered to an actor the actor will eventually receive one. But the stonger notion of fairness is required to guarantee more general eventuality properties.

2.1.5 Composition

There are several ways to combine actor configurations to obtain new configurations. The configuration interfaces and properties of actor systems, such as uniqueness of actor names and persistence of actors, naturally impose constraints on these operations. In this section, we discuss only the composition operation and the constraints associated with it. In Chapter 3, we discuss all the basic combinators from which all other combinations can be derived.

Consider the configurations C_1 and C_2 with interfaces $[\rho_1, \chi_1]$ and $[\rho_2, \chi_2]$ respectively. Then C_1 and C_2 are composable only if:

1. No two actors in the configurations have the same name:

$$dom(C_1) \cap dom(C_2) \quad = \quad \emptyset \tag{2.1}$$

   This is required to guarantee uniqueness of actor names.

2. The configurations respect each others interface:

$$\chi_1 \cap dom(C_2) \subset \rho_2 \tag{2.2}$$

$$\chi_2 \cap dom(C_1) \subset \rho_1 \tag{2.3}$$

These equations reflect the fact that for any configuration, names of internal actors which are not receptionists are unknown to the environment.

The configuration C obtained by composing C_1 and C_2 contains all the actors and messages in the two configurations, and its interface $[\rho, \chi]$ is given by

$$\rho = \rho_1 \cup \rho_2 \tag{2.4}$$

$$\chi = (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2) \tag{2.5}$$

Note that a configuration can be decomposed into sub-configurations in several ways.

2.2 Event Diagrams

Computations in actor systems can be visualized using abstract pictures called event diagrams (Figure 2.2). An event diagram depicts message deliveries during a computation and the orderings between them in time. Message deliveries are called events, and the target actor of a delivery is called the target of the event.

Since an actor processes only one message at a time, all events with the same target are totally ordered. This order, called the arrival order, is abstractly represented by life-lines in an event diagram. A life-line lists all the events at an actor according to the arrival order - further down the line, later the arrival.

In response to a received message an actor may send messages to its acquaintances. This induces yet another order on events called the activation order. This order is represented in event diagrams by arrows between events, with the direction of an arrow indicating

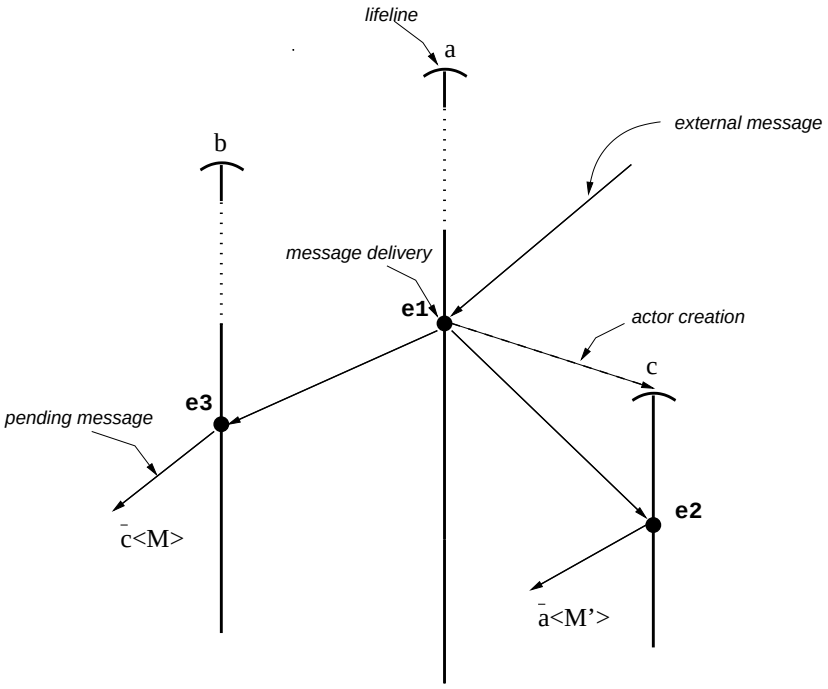

Figure 2.1: An event diagram depicting a computation involving actors a, b and c. Each vertical line represents the arrival order between events with the same target. Arrows between events, such as $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$, represent causal order. Arrows without an event at their head represent pending message deliveries, and are annotated with the corresponding message target and contents. An annotation $\overline{x}\langle y \rangle$ denotes a message with target x and contents y. Actor c is created by a during event e_1.

the causal direction. Arrows corresponding to messages received from the external world do not have an event at their tail. An actor computation in progress may have messages that have not yet been delivered. Such pending messages, also called pending events, are represented by arrows without an event at their head. These arrows are also annotated with the corresponding target and message contents.

Event diagrams lend themselves to a rigorous mathematical formulation. In fact, the power domain of event diagrams is used in [12] as the semantic domain for a simple actor-based language. However, in our discussions we will be using event diagrams only for visualizing computations. We will not be dealing with any of their formal aspects.

2.3 Current Research on Actor Semantics

We now motivate our effort towards an algebraic formulation of actors by examining the current research on actor semantics and identifying some deficiencies.

In [3], the authors extend a functional language with actor coordination primitives. The underlying functional language is the call-by-value lambda calculus with arithmetic and branching primitives, and structure constructors, recognizers and destructors. The actor primitives incorporated are, `send` for sending messages, `letactor` for creating actors, and `become` for changing the local behavior. An actor's behavior is described by a lambda abstraction which is invoked with arguments received in a message. A program is a collection of actors and messages, along with an explicit interface which identifies the receptionists and external names. The extended language is given an operational semantics by extending reductions in the embedded functional language to labeled transitions between actor configurations. The authors then investigate an equational theory based on the notion of observational equivalence which is closely related to testing equivalence [18], and present methods for establishing equivalences.

We note right away that the work in [3] is not about Actor Model per se, but about a concurrent extension of a functional language. The author's view of an actor as a sequential component is inaccurate. In particular, delivery of a message in their model initiates a sequential computation which is modeled by lambda reductions. This is not in agreement with the description in Section 2.1.1. Specifically, all actions an actor performs on receiving a message are concurrent, and features such as function abstractions and applications are extraneous to the basic Actor Model. The ad hoc approach of extending a core functional language also introduces several complexities. Distinguishing between lambda expressions and configurations entails a study of two equivalences, one on expressions and the other on configurations. Besides, retrieving concurrency related laws from the sequential setting is fairly complex. For instance, proving basic expression equivalences such as two adjacent `sends` commute, or an adjacent `send` and `become` commute is quite involved.

In contrast, our calculus accurately captures the Actor Model. We do away with expressions and work just with configurations. Expressions can be encoded as configurations, and their evaluations as computations in configurations. Basic facts related to actor primitives such as two adjacent `sends` commute are directly (and quite naturally) represented as simple structural laws that equate configurations.

In [50, 49], a semantic framework for actors systems which abstracts away from a choice of a specific programming language is defined by introducing the notion of an Abstract Actor Structure (AAS). An ASS provides an abstract set of states of individual actors, an abstract set of values that may be communicated via messages, functions that determine the local transitions of individual actors, and axioms that should be met by these entities. The axioms basically characterize the minimal semantic requirements which should be met by any actor language. Using techniques of concurrent rewriting semantics [30] the semantics of configurations is derived from transitions of individual actors. Specifically, a rewrite theory which is parametric on the chosen AAS is defined. The terms for the rewrite theory are configurations. Its equational theory extends the AAS equational theory, and the rewrite rules capture internal transitions of individual actors and exchange of messages with the environment. The rewrite theory gives rise to a variety of path-semantics which associate configurations with a set of computations with the configuration as the source. The initial model construction gives a set of finite computations. This is extended to infinite computations (called paths), fair paths, and interaction paths (which project away internal transitions from fair paths).

However, the relationship between path-semantics and closely related notions in process algebra literature, such as trace semantics [6] has not been investigated. Further, neither algebraic nor logical characterizations of equivalences have been developed. Numerous other well-understood equivalence notions such as testing equivalence [18], asynchronous bisimulation [4] and barbed congruence [29] are yet to be defined and studied in the proposed framework. In contrast our formulation uses the very algebraic framework in which these notions were proposed and investigated. We are currently investigating

how the features specific to Actor Model such as encapsulation and fairness effect these notions and their algebraic theories.

In [16], the authors present a simple calculus equipped with concurrency primitives common in process algebras. But the calculus in not an accurate formulation of the Actor Model. The authors retain the view of actors as sequential components. They do not model persistence of actors and fairness in message deliveries. Further, the calculus is not strictly asynchronous. Messages received from the environment are synchronous; but those sent by internal actors are asynchronous. Besides being inaccurate the calculus has a few other unsatisfactory features. The authors allow the use of summation operator in describing actor behaviors. But as described in [4], the meaning of summations other than those that are input guarded is unclear in the presence of asynchronous communications. Apart from the combinators standard in process algebras, the calculus also assumes the operators `send`, `create` and `become` as primitives. As we will show, these are not essential as they can be represented quite naturally using the other combinators. This gives us a calculus that is not only simpler, but also one that is close to being a typed version of π-calculus. This hopefully will help us relate directly to the rich algebraic theories available on π-calculus.

Chapter 3

System A: Syntax and Type System

In this chapter we present the syntax and type system of our calculus for actors called System A. Configurations are represented by well typed (pre)terms in System A, and computations are modeled by labeled transitions between terms. In Section 3.1, we present the syntax of preterms and informally discuss all the basic combinators. Only well typed preterms represent configurations. We present the type system in Section 3.2. In Section 3.3, we prove some properties related to soundness of the type system. In Section 3.4, we compare System A with π-calculus and its variants.

3.1 Syntax

In Chapter 2, we did not specify the primitive values assumed in an actor system. There are a number of alternatives. Formulations in [16, 3] assume, apart from names, values such as booleans and integers, constructors such as lists, and various operations on them as basic. In the spirit of π-calculus, we assume only names as primitive, and communication as the sole mechanism of computation in our calculus. Data can be represented as configurations, and the receptionists of these configurations would serve as handles to the data. Data can be communicated by passing receptionist names. We will demonstrate this via several examples in Chapter 5.

We now present the syntax of our calculus. We assume an infinite set of names $\mathcal{N}$, and a set $\mathcal{B}$ of behavior identifiers. Each behavior identifier has an associated positive integer, called its arity. We let u, v, w, x, y, z range over $\mathcal{N}$, and A, B range over $\mathcal{B}$. The set of preterms $\mathcal{C}$, is defined by the following context-free grammar. We let C range over $\mathcal{C}$.

$$\begin{aligned} C &:= 0 \\ &\mid x(\tilde{y}).C \\ &\mid \overline{x}\langle\tilde{y}\rangle \\ &\mid (\nu x)(C) \\ &\mid [x = y](C_1, C_2) \\ &\mid C_1 \mid C_2 \\ &\mid B\langle x, \tilde{y}\rangle \end{aligned}$$

In the above $\tilde{x}$ represents a finite (possibly empty) tuple of names. The tuple $\tilde{x}, \tilde{y}$ is the result of appending $\tilde{y}$ to $\tilde{x}$. Here 0 and $\overline{x}\langle\tilde{y}\rangle$ are nullary combinators, $x(\tilde{y}).$ and (νx) are unary combinators, and $\mid$ and $[x = y]$ are binary combinators. The order of precedence among combinators is the order listed above. These basic combinators are sufficient to formalize the Actor Model.

Not all preterms represent actor configurations. The syntax rules are not stringent enough to ensure all actor properties. In Section 3.2, we show a few preterms which are not actor configurations, and present a type system which guarantees that every well typed preterm is an actor configuration. Well typed preterms are called *terms*. Following is an informal description of each kind of (pre)term.

1. *Nil Process*, 0: This represents a configuration with no actors and messages.

2. *Output*, $\overline{x}\langle\tilde{y}\rangle$: This represents a configuration containing a single message. The message is targeted to an external actor x and contains the tuple of names $\tilde{y}$.

3. *Input*, $x(\tilde{y}).C$: This represents a configuration containing a single actor named x. The actor is also a receptionist and may receive messages from the environment.

All names in $\tilde{y}$ are distinct and are bound by the input $x(\tilde{y})$. The actor receives a message containing a tuple $\tilde{z}$ of the same arity as $\tilde{y}$, and replaces itself with the configuration $C\{\tilde{z}/\tilde{y}\}$ (see the definition of substitution below). The configuration $C\{\tilde{z}/y\}$ in turn should contain the actor x with a new behavior (this is guaranteed by the type system we impose), and (possibly) some freshly created actors and messages. The abstraction $(\tilde{y})C$ is therefore the actor's behavior.

4. *Composition*, $C_1 \mid C_2$: This represents a composition of two actor configurations. We discussed composition in Section 2.1.5.

5. *If-then-else*, $[x = y](C_1, C_2)$: This preterm is C_1 if x and y are the same names. Otherwise it is C_2.

6. *Restriction*, $(\nu x)C$: This represents the configuration C, but with actor x no longer a receptionist. The name x is private to the configuration. It is bound by restriction (νx).

7. *Behavior Instantiation*, $B\langle x, \tilde{y}\rangle$. Each behavior identifier B has a single defining equation of the form $B \stackrel{def}{=} (u, \tilde{v})u(\tilde{w}).C$, where $u \notin \tilde{v}$, and $\tilde{v}$ and $\tilde{w}$ are finite tuples of distinct names. The definition provides a template for an actor behavior. An instantiation $B\langle x, \tilde{y}\rangle$ represents the configuration $(u(\tilde{w}).C)\{(x, \tilde{y})/(u, \tilde{v})\}$. The tuple $u, \tilde{v}$ contains *exactly* (all and only) the free names in $u(\tilde{w}).C$, and is called the acquaintance list. The arity of B is equal to arity of the acquaintance list. We assume that for any occurrence of $B\langle\tilde{x}\rangle$, B and $\tilde{x}$ are of the same arity. Note that since actors are persistent, every behavior definition always refers to a behavior definition (possibly itself). Every configuration comes along with a *finite* number of behavior definitions which define behavior identifiers occurring in it. We assume that while composing two such configurations, their behavior definitions are consistent, i.e. they both have identical definitions for any common behavior identifier.

The interface of a term is implicit in its syntax. The type system in Section 3.2 derives the interface of a term. The fact that a configuration's interface may evolve dynamically is captured by the labeled transition system presented in Chapter 4.

Readers familiar with π-calculus may note that the syntax of System A and π-calculus are closely related. All combinators in System A are from π-calculus. So, at least syntactically, every preterm is a π-calculus term. But the semantics of System A differs from π-calculus. We present a detailed comparison in Section 3.4.

Finally, note that names are the only communicable values in System A. Thus, names in terms are always substituted by names and not by configurations. A higher order version of the Actor Model where configurations can be passed in messages is left for future research.

Before we present the type system, a few definitions and notational conventions are in order.

Notation 1 *1. We use $\equiv$ to denote syntactic identity. For example, $C_1 \equiv C_2$ means that C_1 and C_2 are syntactically identical.*

2. We write $(\nu x_1...x_n)C$ instead of $(\nu x_1)...(\nu x_n)C$. Further, if $\tilde{x} = (x_1...x_n)$, we may also write the same as $(\nu\tilde{x})C$. If $\tilde{x}$ is the empty tuple then $(\nu\tilde{x})C$ just means C.

3. For every tuple $\tilde{x}$, we will denote the set of names occurring in $\tilde{x}$ by $\{\tilde{x}\}$, and the length (arity) of $\tilde{x}$ by $len(\tilde{x})$.

Definition 1 *The set of free names and bound names in a preterm C, $fn(C)$ and $bn(C)$ respectively, have the usual definitions. Note that $fn(B\langle\tilde{x}\rangle) = \{\tilde{x}\}$ and $bn(B\langle\tilde{x}\rangle) = \emptyset$. The set of names that occur in a preterm is defined as $n(C) = fn(C) \cup bn(C)$.*

Definition 2 *The binary operation '-' on finite tuples is defined as follows. The empty tuple is denoted by nil.*

$$nil - \tilde{w} \;=\; nil$$

$$(u, \tilde{v}) - \tilde{w} \;\; = \;\; \begin{cases} \tilde{v} - \tilde{w} & \textit{if } u \in \{\tilde{w}\} \\ u, (\tilde{v} - \tilde{w}) & \textit{otherwise} \end{cases}$$

Definition 3 *Two preterms are said to be alpha equivalent if one can be transformed into the other by using the congruence laws and the following alpha conversion laws*

$$\begin{array}{ll} (\nu y)C \equiv_\alpha (\nu z)C[z/y], & \textit{where } z \notin n(C) \\ x(\tilde{y}).C \equiv_\alpha x(\tilde{z}).C[\tilde{z}/\tilde{y}] & \textit{where } \{\tilde{z}\} \cap n(C) = \emptyset \textit{ and } z_i = z_j \Leftrightarrow i = j \end{array}$$

The notation $C[\tilde{z}/\tilde{y}]$ denotes the term obtained by replacing all free occurrences of y_i by z_i. Alpha equivalent terms represent the same actor configuration; $\tilde{y}$ in $x(\tilde{y}).C$ is just a place holder for a tuple that will be received in a message, and x in $(\nu x)C$ is a private name that can not be identified with any other name. In the following, we do not distinguish between alpha equivalent terms and work modulo alpha equivalence. For preterms C_1, C_2 we may write $C_1 = C_2$ to mean that $C_1 \equiv_\alpha C_2$.

The following lemmas are routine to prove.

Lemma 1 *Let $\tilde{u}$ and $\tilde{v}$ each be a tuples of distinct names, $len(\tilde{u}) = len(\tilde{v})$, $\{\tilde{v}\} \cap n(C) = \emptyset$, and $\sigma = \{\tilde{v}/\tilde{u}\}$. Then $fn(C[\tilde{v}/\tilde{u}]) = \sigma(fn(C))$.*

Lemma 2 *The relation $\equiv_\alpha$ is an equivalence relation.*

Definition 4 *A name substitution is a function from names to names which is almost everywhere the identity. We let σ range over name substitutions. Suppose $\tilde{x}$ and $\tilde{y}$ are tuples of same length, and $x_i = x_j$ iff $i = j$. We write $\{\tilde{y}/\tilde{x}\}$ for the substitution which maps x_i to y_i, and is identity on all other names. We write $\sigma(x)$ for the name to which x is mapped by σ. Similarly, $\sigma(\tilde{x})$ and $\sigma(fn(C))$ denote the vector and set obtained by applying σ to each element of $\tilde{x}$ and $fn(C)$ respectively.*

The reader may verify the following identities which we will be using later. For any sets of names R and S, and name substitution σ

$$\sigma(R) - \sigma(S) = \sigma(R - S) - \sigma(S) \tag{3.1}$$

$$\sigma(R) - \sigma(S) = \sigma(R - S) \qquad \texttt{if } \forall x, y \in R \cup S.\ \sigma(x) = \sigma(y) \Leftrightarrow x = y \tag{3.2}$$

Definition 5 *Substitution on preterms is defined modulo alpha equivalence. We denote a preterm obtained by applying the substitution σ to the preterm C by $C\sigma$. An inductive definition of substitution is shown below. We assume for all $x \in bn(C)$, $\sigma(x) = x$. To ensure that a name free in C does not become bound in $C\sigma$, we assume $bn(C) \cap \sigma(fn(C)) = \emptyset$. If not, we can pick $C' \equiv_\alpha C$ such that $bn(C') \cap fn(\sigma) = \emptyset$.*

$$\begin{aligned}
&0\sigma = 0\\
&(\overline{x}\langle\tilde{y}\rangle)\sigma = \overline{\sigma(x)}\langle\sigma(\tilde{y})\rangle\\
&(x(\tilde{y}).C)\sigma = \sigma(x)(\tilde{y}).(C\sigma)\\
&((\nu x)C)\sigma = (\nu x)(C\sigma)\\
&(C_1 \mid C_2)\sigma = C_1\sigma \mid C_2\sigma\\
&[x = y](C_1, C_2)\sigma = [\sigma(x) = \sigma(y)](C_1\sigma, C_2\sigma)\\
&B\langle\tilde{x}\rangle\sigma = B\langle\sigma(\tilde{x})\rangle
\end{aligned}$$

The following lemma is routine to prove.

Lemma 3 *If $C_1 \equiv_\alpha C_2$ then $C_1\sigma \equiv_\alpha C_2\sigma$.*

3.2 Type System for Configurations

Not all preterms represent configurations. With the syntax rules presented in Section 3.1 one can construct preterms which violate properties which every actor system should satisfy. In this section we present a type system on preterms to guarantee uniqueness

of actor names, persistence of actors, and anonymity of new actors. Every well typed preterm, also called a term, will be a configuration.

The locality laws are automatically guaranteed by the syntax rules. We will show this later in this section when we have defined the notion of actor acquaintances in our formal system. Encapsulation and fairness are dynamic properties which constrain the ways a configuration may evolve and interact with its environment. Naturally, these are to be expressed in a framework which captures computation in actor systems. We will return to these when we present an operational semantics for System A in Chapter 4.

The properties we try to guarantee by type checking a term should hold even as the term evolves and interacts with its environments. When we present the operational semantics in Chapter 4 we will prove that this is indeed the case.

Before we discuss the type system, let us examine a few preterms which are not actor configurations.

1. The preterm $x(y).C_1 \mid x(y).C_2$ contains two actors with name x and hence violates uniqueness of actor names.

2. The preterm $x(y).0$ violates the persistence of actors. The actor x disappears after receiving a message, i.e. it does not assume a replacement behavior.

3. The preterm $x(y).(x(z).C_1 \mid y(z).C_2)$ violates anonymity of newly created actors. An actor can create new actors only with fresh names; these names should be private to the creating actor and not be acquaintances of other actors. Therefore, creation of actors with well-known names or names received in a message is ruled out. In the preterm above, actor x creates an actor with a name it receives in a message. Similarly, the preterm $x(y).(x(z).C_1 \mid u(z).C_2)$ is not valid as the actor x creates an actor with a well known name u; the actor u is a receptionist of the resulting configuration and can receive messages from the environment.

4. The preterm $x(y).(\nu z)(x(y).C \mid \overline{u}\langle z\rangle)$ where $z \notin fn(C)$ is not an actor configuration because on receiving a message it generates a private name z without associating

it with a behavior (an observant reader may note that the preterm is invalid even without the side condition $z \notin fn(C)$). An implicit feature of the Actor Model is that every name generated during a computation is associated with a behavior. This follows from postulate 3 of the locality laws described in Section 2.1.2. Names that are initially present in the configuration and that do not denote internal actors are assumed to denote actors in the environment. The preterm above, in fact eventually exports the name z to the external actor u even though there is no internal actor named z. For the same reason, the preterm $(\nu x)C$ where $x \notin fn(C)$ is not an actor configuration.

The moral of these examples is, we have to impose some constraints on the ways configurations may be used to built new configurations. This is not a new idea. In Section 2.1.5, we already discussed the constraints which accompany composition of actor configurations. Recall that the constraints were expressed on the internal actor names and configuration interfaces. Since all properties of our concern here are related to actor names, we would expect such constraints on interfaces and internal actor names to suffice for the other combinators. It is indeed the case. In fact, all the constraints can be expressed by only referring to the receptionists of the configurations being combined. This is possible because of the restriction operator. A name bound by a restriction cannot be identified with any other name. Since the names of non receptionists are bound by restrictions, they are automatically guaranteed to be unique. In the case of composition, for example, conditions 2.2 and 2.3 are automatically satisfied, and equation 2.1 is equivalent to $\rho_1 \cap \rho_2 = \emptyset$.

A judgment in our type system is of form $C : [\rho, \chi]$, and is interpreted as, C is a configuration with interface $[\rho, \chi]$. Since ρ and χ are sets the order in which names are listed in them is ignored. The type rules are shown in Table 3.2. The set of all typing judgments is the smallest set that is closed under these rules. It is clear that for any term C, if $C : [\rho_1, \chi_1]$ and $C : [\rho_2, \chi_2]$ are typing judgements then $\rho_1 = \rho_2$ and $\chi_1 = \chi_2$.

$NIL: \dfrac{}{0 : [\emptyset, \emptyset]} \qquad MSG: \dfrac{}{\overline{x}\langle\tilde{y}\rangle : [\emptyset, \{x, \tilde{y}\}]}$
$ACT: \dfrac{C : [\{x\}, \chi]}{x(\tilde{y}).C : [\{x\}, fn(C) - \{x, \tilde{y}\}]} \quad x \notin \{\tilde{y}\}$
$RES: \dfrac{C : [\rho \cup \{x\}, \chi]}{(\nu x)C : [\rho, \chi]} \quad x \notin \rho$
$COMP: \dfrac{C_1 : [\rho_1, \chi_1] \quad C_2 : [\rho_2, \chi_2]}{C_1 \mid C_2 : [\rho_1 \cup \rho_2, (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)]} \quad \rho_1 \cap \rho_2 = \phi$
$COND: \dfrac{C_1 : [\rho, \chi_1] \quad C_2 : [\rho, \chi_2]}{[x = y](C_1,\ C_2) : [\rho, \chi]} \quad \chi = \begin{cases} \chi_1 & \text{if } x \equiv y \\ \chi_2 & \text{otherwise} \end{cases}$
$BINST: \dfrac{}{B\langle x, \tilde{y}\rangle : [\{x\}, \{\tilde{y}\} - \{x\}]}$

Table 3.1: Type rules for System A

An output term represents a configuration with no actors and a single message targeted to an external actor. The configuration, therefore, has no receptionists, and the external names are all those that occur in the message, either as the target or the contents. Hence the rule MSG.

Rule ACT reflects the fact that an input term represents a configuration with a single actor which is also a receptionist. The rule also guarantees persistence of actors by ensuring that the configuration with which an actor x replaces itself on receiving a message, always contains x with a replacement behavior. Further, it stipulates that x is the only receptionist of the resulting configuration. This is because new actors, if any, are created with private names. External names of the configuration are the acquaintances of actor x, except itself (see below for a definition and discussion of acquaintances). Names in the tuple $\tilde{y}$ are bound names which will be substituted by names received in a message. Therefore, all the names in $\tilde{y}$ should be distinct and $x \notin \{\tilde{y}\}$.

Rule RES reflects the fact that the outermost restriction in $(\nu x)C$ makes the name x private to C. Actor x is no longer a receptionist in $(\nu x)C$. The side condition makes sure that there in fact is an actor with name x in C. This guarantees the feature that names in an actor system are always associated with a behavior. This in turn justifies the fact that the restriction leaves the set of external names unchanged.

We discussed the details of rule $COMP$ in Section 2.1.5 and earlier in this section.

Rule $COND$ stipulates that both the sub-configurations in a conditional construct have the same receptionist sets. This restriction is more stringent than necessary. It has been formulated this way to keep the type system simple. Suppose one of the names being matched is bound by an input prefix, i.e. the condition can be tested only after the receipt of a message. Then to ensure uniqueness of actor names, we would have to take the union of the receptionist sets of the sub-configurations to be the receptionist set of the entire construct. On the other hand, to ensure that all names are associated with a behavior, we would have to take the intersection. The easiest way to resolve this is to presume both the sets to be equal. This assumption does not result in a loss of expressive

power. The examples in Chapter 5 demonstrate this fact. The set of external names is chosen according to the outcome of the test.

Rule $BINST$ states that $B\langle x, \tilde{y}\rangle$ is a configuration with interface $[\{x\}, \{\tilde{y}\} - \{x\}]$. As discussed in Section 3.1, given a behavior definition $B \stackrel{def}{=} (u, \tilde{v})u(\tilde{w}).C$, the instantiation $B\langle x, \tilde{y}\rangle$ represents the term $(u(\tilde{w}).C)\{(x, \tilde{y})/(u, \tilde{v})\}$. This is captured in the formalisms we present in Chapter 4. As a consequence of the rule, the judgement $(u(\tilde{w}).C)\{(x, \tilde{y})/(u, \tilde{v})\} : [\{x\}, \{\tilde{y}\} - \{x\}]$ should be derivable. There is some ambiguity in the statement above. Recall that substitution is defined only modulo alpha equivalence. So, more precisely, for every possible result C' of the substitution, the judgment $C' : [\{x\}, \{\tilde{y}\} - \{x\}]$ should be derivable. In section 3.3 we will also show that all these constraints are satisfied just if $u(\tilde{w}).C : [\{u\}, \{v\}]$. Thus, type checking a configuration involves checking all the accompanying behavior definitions (which we require to be finite in number); for every definition $B \stackrel{def}{=} (u, \tilde{v})u(\tilde{w}).C$, we require $u(\tilde{w}).C : [\{u\}, \{v\}]$.

At this point, the reader is urged to verify that the preterms discussed earlier in this section which violated actor properties are not well typed. We show this only for $x(\tilde{y}).C_1 \mid x(\tilde{z}).C_2$. The key to the argument is, given a term, there is only one type rule that can be used in the last step of a derivation (if any) for a judgement involving the term. Suppose $x(\tilde{y}).C_1 \mid x(\tilde{z}).C_2 : [\rho, \chi]$. Then the last step in the derivation of this judgement has to be an application of the $COMP$ rule. So, $x(\tilde{y}).C_1 : [\rho_1, \chi_1]$ and $x(\tilde{z}).C_2 : [\rho_2, \chi_2]$ for some ρ_1, ρ_2 such that $\rho_1 \cap \rho_2 = \emptyset$. The last derivation step of $x(\tilde{y}).C_1 : [\rho_1, \chi_1]$ has to be an application of the ACT rule. This implies $x \in \rho_1$. By a similar argument, $x \in \rho_2$. This implies $\rho_1 \cap \rho_2 \neq \emptyset$, which is a contradiction.

We now show that the locality laws (Section 2.1.2) hold in our system. Before we proceed, the definition of acquaintances of an actor is in order. We define the acquaintances of an actor $x(y).C$ to be all the free names in $(y)C$. The only actions an actor may perform with the names of its acquaintances is to send messages with these names as target or contents, or create actors which have these acquaintances. Specifically, an actor can not create actors with any of these names except its own. In fact, the type system ensures

that new actors are always created with private names (bound by a restriction). So our definition of acquaintances is sound.

Locality laws 1 and 2 hold because $fn(C)$ is finite for every term C. An actor $x(\tilde{y}).C$ on receiving a message $\overline{x}\langle\tilde{z}\rangle$ replaces itself with the configuration $C\{\tilde{z}/\tilde{y}\}$. This is captured in the formalisms we present in Chapter 4. Input and output subterms in $C\{\tilde{z}/\tilde{y}\}$ that do not occur inside an actor behavior are the freshly created actors and messages. Names that occur free in these substerms are either also free in $C\{\tilde{z}/\tilde{y}\}$, or are bound by restrictions. By the RES rule, the ones bound by restriction denote newly created actors. Now, locality law 3 follows from the fact that $fn(C\{\tilde{z}/\tilde{y}\}) \subset fn((\tilde{y})C) \cup \{\tilde{z}\}$.

There could be a minor complaint against our definition of acquaintances. According to the definition, an actor always has itself as an acquaintance even though it may not 'know' its own name. For example, an actor may neither send messages using its name, or create actors which have it as an acquaintance. An easy remedy would be to include x as an acquaintance of $x(\tilde{y}).C$ only if there is an output term or a behavior instantiation in C which contains a free occurrence of x. In any case, this minor inaccuracy in the definition, does not have any undesirable consequences. Hence we retain the definition as it is. Finally, it is easy to see that the acquaintances of an actor may dynamically evolve as the actor assumes new behaviors.

3.3 Some Soundness Properties of the Type System

If $C : [\rho, \chi]$ is a typing judgement then we expect $\rho \cap \chi = \emptyset$. By induction on the structure of C we can show that this is true. The key to this property is equation 2.5 (Section 2.1.3). We omit the proof as it is simple.

Theorem 1 *Let* $C : [\rho, \chi]$. *Then* $\rho \cap \chi = \emptyset$.

Proof: By structural induction on C. $\square$

Alpha equivalent terms represent the same configuration. Therefore, the type system should respect alpha equivalence. Theorem 2 states this formally. Before the theorem a few lemmas are in order.

Lemma 4 *If* $C : [\rho, \chi]$ *then* $\rho \cup \chi \subset fn(C)$.

Proof: By structural induction on C. $\square$

Lemma 4 is not a surprise. It simply states that the names in a term's interface occur free in the term.

The proof of the following lemma is a bit tedious, although conceptually very simple. Uninterested readers may skip the proof.

Lemma 5 *Let* $\tilde{u}$ *and* $\tilde{v}$ *each be a tuples of distinct names,* $len(\tilde{u}) = len(\tilde{v})$, $\{\tilde{v}\} \cap n(C) = \emptyset$, *and* $\sigma = \{\tilde{v}/\tilde{u}\}$. *Then* $C : [\rho, \chi]$ *implies* $C[\tilde{v}/\tilde{u}] : [\sigma(\rho), \sigma(\chi)]$.

Proof: We prove this by structural induction on C.

Base Case: C is either the nil term, an output term, or a behavior instantiation. It is easy to verify that the hypothesis is true for these cases.

Induction Step: We have to consider four cases.

1. $C \equiv x(\tilde{y}).C_1$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\{x\}, \chi']}{x(\tilde{y}).C_1 : [\{x\}, fn(C_1) - \{x, \tilde{y}\}]} \qquad x \notin \{\tilde{y}\}$$

   $C[\tilde{v}/\tilde{u}] \equiv \sigma(x)(\tilde{y}).(C_1[\tilde{v'}/\tilde{u'}])$, where $\tilde{u'} = \tilde{u} - \tilde{y}$ and $\tilde{v'} = \sigma(\tilde{u'})$. Let $\sigma' = \{\tilde{v'}/\tilde{u'}\}$. By induction hypothesis, $C_1[\tilde{v'}/\tilde{u'}] : [\{\sigma'(x)\}, \sigma'(\chi')]$. Since $x \notin \{\tilde{y}\}$, we have

$\sigma'(x) = \sigma(x)$. Since $\{\tilde{v}\} \cap n(C) = \emptyset$, we have $\sigma(x) \notin \{\tilde{y}\}$. Then by rule ACT, $C[\tilde{v}/\tilde{u}] : [\{\sigma(x)\}, fn(C_1[\tilde{u'}/\tilde{v'}]) - \{\sigma(x), \tilde{y}\}]$. We are done if we show $fn(C_1[\tilde{u'}/\tilde{v'}]) - \{\sigma(x), \tilde{y}\} = \sigma(fn(C_1) - \{x, \tilde{y}\})$. By Lemma 1, $fn(C_1[\tilde{v'}/\tilde{u'}]) = \sigma'(fn(C_1))$. Also, since σ' is identity on $\tilde{y}$, $\sigma'(fn(C_1)) = \sigma(fn(C_1) - \{\tilde{y}\}) \cup \{fn(C_1) \cap \{\tilde{y}\}\}$. Now, using this and the identity $(R \cup S) - T = (R - T) \cup (S - T)$, we deduce $fn(C_1[\tilde{u'}/\tilde{v'}]) - \{\sigma(x), \tilde{y}\} = \sigma(fn(C_1) - \{\tilde{y}\}) - \{\sigma(x), \tilde{y}\}$. Now, $\sigma(fn(C_1) - \{\tilde{y}\}) - \{\sigma(x), \tilde{y}\} = \sigma(fn(C_1) - \{\tilde{y}\}) - \{\sigma(x)\} - \{\tilde{y}\} = \sigma(fn(C_1) - \{x, \tilde{y}\}) - \{\tilde{y}\}$ (using 3.2) $= \sigma(fn(C_1) - \{x, \tilde{y}\})$ (since $\{\tilde{v}\} \cap \{\tilde{y}\} = \emptyset$).

2. $C \equiv (\nu x)C_1$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\rho \cup \{x\}, \chi]}{(\nu x)C_1 : [\rho, \chi]} \; x \notin \rho$$

$C[\tilde{v}/\tilde{u}] \equiv (\nu x)C_1[\tilde{v'}/\tilde{u'}]$, where $\tilde{u'} = \tilde{u} - x$ and $\tilde{v'} = \sigma(\tilde{u'})$. Let $\sigma' = \{\tilde{v'}/\tilde{u'}\}$. Since $\sigma'(x) = x$, by induction hypothesis, $C_1[\tilde{v'}/\tilde{u'}] : [\sigma'(\rho) \cup \{x\}, \sigma'(\chi)]$. By Theorem 1, $x \notin \chi$; so $\sigma'(\chi) = \sigma(\chi)$. Since $x \notin \rho$, $\sigma'(\rho) = \sigma(\rho)$. Therefore, $C_1[\tilde{v'}/\tilde{u'}] : [\sigma(\rho) \cup \{x\}, \sigma(\chi)]$. Since $\{\tilde{v}\} \cap n(C) = \emptyset$, $x \notin \sigma(\rho)$. Then by rule RES, $C[\tilde{v}/\tilde{u}] : [\sigma(\rho), \sigma(\chi)]$.

3. $C \equiv C_1 \mid C_2$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\rho_1, \chi_1] \quad C_2 : [\rho_2, \chi_2]}{C_1 \mid C_2 : [\rho_1 \cup \rho_2, (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)]} \; \rho_1 \cap \rho_2 = \emptyset$$

$C[\tilde{v}/\tilde{u}] \equiv C_1[\tilde{v}/\tilde{u}] \mid C_2[\tilde{v}/\tilde{u}]$. By induction hypothesis, $C_1[\tilde{v}/\tilde{u}] : [\sigma(\rho_1), \sigma(\chi_1)]$ and $C_2[\tilde{v}/\tilde{u}] : [\sigma(\rho_2), \sigma(\chi_2)]$. Since σ is one-to-one on $fn(C)$, by Lemma 4, it is also one-to-one on $\rho_1 \cup \rho_2 \cup \chi_1 \cup \chi_2$. Then, since $\rho_1 \cap \rho_2 = \emptyset$, we have $\sigma(\rho_1) \cap \sigma(\rho_2) = \emptyset$. Then by rule $COMP$, $C : [\sigma(\rho_1 \cup \rho_2), \sigma(\chi_1 \cup \chi_2) - \sigma(\rho_1 \cup \rho_2)]$. Now, using identity 3.2, we get $C : [\sigma(\rho_1 \cup \rho_2), \sigma((\chi_1 \; cup \chi_2) - (\rho_1 \cup \rho_2))]$.

4. $C \equiv [x = y](C_1, C_2)$: The argument for this case is simple and left to the reader.

□

We are now ready to show that the type system respects alpha equivalence.

Theorem 2 *If $C_1 \equiv_\alpha C_2$ then $C_1 : [\rho, \chi]$ if and only if $C_2 : [\rho, \chi]$.*

Proof: We only prove $C_1 : [\rho, \chi]$ implies $C_2 : [\rho, \chi]$. The other direction follows from Lemma 2. The proof is by induction on the length of derivation of $C_1 \equiv_\alpha C_2$.

Base Case: The derivation is by a direct application of the alpha conversion laws (see Section 3.1). We have two cases to consider:

1. $C_1 \equiv (\nu y)C$ and $C_2 \equiv (\nu z)C[z/y]$, where $z \notin n(C)$. Then by rule RES, $C : [\rho \cup \{y\}, \chi]$ and $y \notin \rho$. Further, by Theorem 1, $y \notin \chi$. Then by Lemma 5, $C[z/y] : [\rho \cup \{z\}, \chi]$. By Lemma 4, $\rho \subset fn(C)$. So $z \notin \rho$. Then by rule RES, $C_2 : [\rho, \chi]$.

2. $C_1 \equiv x(\tilde{y}).C$ and $C_2 \equiv x(\tilde{z}).C[\tilde{z}/\tilde{y}]$, where $\tilde{z}$ is a tuple of distinct names and $\{\tilde{z}\} \cap n(C) = \emptyset$. Let $\sigma = \{\tilde{z}/\tilde{y}\}$. By rule ACT, $C : [\{x\}, \chi']$ for some χ', and $x \notin \{\tilde{y}\}$. Then by Lemma 5, $C[\tilde{z}/\tilde{y}] : [\{x\}, \sigma(\chi')]$. By Lemma 4, $x \in fn(C)$. Hence $x \notin \{\tilde{z}\}$. Then by rule ACT, $C_2 : [\{x\}, fn(C[\tilde{z}/\tilde{y}]) - \{x, \tilde{z}\}]$. We are done if we show $fn(C[\tilde{z}/\tilde{y}]) - \{x, \tilde{z}\} = fn(C) - \{x, \tilde{y}\}$. Now, $fn(C[\tilde{z}/\tilde{y}]) - \{x, \tilde{z}\} = \sigma(fn(C)) - \{x, \tilde{z}\}$ (by Lemma 1) $= \sigma(fn(C)) - \sigma(\{x, \tilde{y}\})$ (since $x \notin \{\tilde{y}\}$) $= \sigma(fn(C) - \{x, \tilde{y}\})$ (by identity 3.2) $= fn(C) - \{x, \tilde{y}\}$ (since σ is identity on $fn(C) - \{x, \tilde{y}\}$).

Induction Step: The last step of an arbitrary derivation can use either the alpha conversion laws or the congruence laws. We have already considered the former. The arguments for congruence laws are straight forward and left to the reader. □

Not all substitutions on a term C yield terms. A substitution σ may identify names of two distinct actors in C and violate uniqueness of actor names. But, if σ renames

different actors in C to different names then $C\sigma$ should be a term. Further, if $C : [\rho, \chi]$ and $C\sigma : [\rho', \chi']$ then we expect $\rho' = \sigma(\rho)$. On the other hand, a similar relationship between χ' and χ need not hold. This is because of two reasons. First, external names in C can be renamed to internal actor names. Second, renaming can change the outcome of conditions which are 'ready to be tested', i.e. which do not occur as part of an actor behavior. For example, the external names in $[x = y](C_1, C_2)$ are external names in C_1. But the external names after the substitution $\{y/x\}$ are those in C_2. If σ does not identify any two names used in testing conditions which are not part of actor behaviors then we would expect $\chi' = \sigma(\chi) - \sigma(\rho)$. Theorem 3 formally states these properties related to substitutions. Before the theorem, a few definitions are in order.

Definition 6 *A substitution σ is said to be a proper renaming of a set of names L if for all $x, y \in L$, $\sigma(x) = \sigma(y) \Leftrightarrow x = y$.*

Definition 7 *For a preterm C, we define $tn(C)$ inductively as follows:*

$$
\begin{aligned}
&tn(0) = \emptyset \\
&tn(\overline{x}\langle\tilde{y}\rangle) = \emptyset \\
&tn(x(\tilde{y}).C) = \emptyset \\
&tn((\nu x)C) = tn(C) - \{x\} \\
&tn(C_1|C_2) = tn(C_1) \cup tn(C_2) \\
&tn([x = y](C_1, C_2)) = \{x, y\} \cup tn(C_1) \cup tn(C_2) \\
&tn(B\langle\tilde{x}\rangle) = \emptyset
\end{aligned}
$$

Clearly, $tn(C) \subset fn(C)$. Roughly speaking, $tn(C)$ is the set of all names, free occurrences of which are used in a condition that is ready to be tested in C

Uninterested readers may skip the rather tedious proof of the following theorem.

Theorem 3 *If $C : [\rho, \chi]$ and σ is a proper renaming of ρ then $C\sigma : [\sigma(\rho), \chi']$, for some χ'. Further, if σ is a proper renaming of $tn(C)$ then $\chi' = \sigma(\chi) - \sigma(\rho)$.*

Proof: The proof is by structural induction on C.

Base Case: It is easy to verify that the statement is true if C is the nil term, an output term, or a behavior instantiation.

Induction Step: Using Lemma 3 and Theorem 2, we may assume (with appropriate alpha conversions) $\sigma(x) = x$ for all $x \in bn(C)$, and $bn(C) \cap \sigma(fn(C)) = \emptyset$.

We have to consider four kinds of terms.

1. $C \equiv (\nu x)C_1$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\rho \cup \{x\}, \chi]}{(\nu x)C_1 : [\rho, \chi]} \quad x \notin \rho$$

   By our assumption, $C\sigma = (\nu x)C_1\sigma$, $\sigma(x) = x$, and $x \notin \sigma(\rho)$ (using Lemma 4). Then σ is a proper renaming of $\rho \cup \{x\}$, and by induction hypothesis, $C_1\sigma : [\sigma(\rho) \cup \{x\}, \chi']$ for some χ'. Then by rule RES, $C\sigma : [\sigma(\rho), \chi']$. Further, suppose σ is a proper renaming of $tn(C)$. We know $tn(C_1) \subset tn(C) \cup \{x\}$. Since $tn(C) \subset fn(C)$, σ is also a proper renaming of $tn(C_1)$, and by induction hypothesis, $\chi' = \sigma(\chi) - (\sigma(\rho) \cup \{x\})$. By Lemma 4, $\chi \subset fn(C)$. Then, by our assumption $x \notin \sigma(\chi)$. Therefore $\chi' = \sigma(\chi) - \sigma(\rho)$.

2. $C \equiv x(\tilde{y}).C_1$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\{x\}, \chi']}{x(\tilde{y}).C_1 : [\{x\}, fn(C_1) - \{x, \tilde{y}\}]} \quad x \notin \{\tilde{y}\}$$

   By our assumption, $C\sigma = \sigma(x)(\tilde{y}).C_1\sigma$, $\sigma(\tilde{y}) = \tilde{y}$, and $\sigma(x) \notin \{\tilde{y}\}$. By induction hypothesis, $C_1\sigma : [\{\sigma(x)\}, \chi'']$ for some χ''. Then by rule ACT, we have $C\sigma : [\{\sigma(x)\}, fn(C_1\sigma) - \{\sigma(x), \tilde{y}\}]$. Now, $fn(C_1\sigma) - \{\sigma(x), \tilde{y}\} = \sigma(fn(C_1)) - \{\sigma(x), \sigma(\tilde{y})\}$ (since $\sigma(\tilde{y}) = \tilde{y}$) $= \sigma(fn(C_1) - \{x, \tilde{y}\}) - \{\sigma(x)\} - \{\tilde{y}\}$ (using identity 3.1) $= \sigma(fn(C_1) - \{x, \tilde{y}\}) - \{\sigma(x)\}$ (since $\sigma(fn(C)) \cap \{\tilde{y}\} = \emptyset$).

3. $C \equiv C_1 \mid C_2$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\rho_1, \chi_1] \quad C_2 : [\rho_2, \chi_2]}{C_1 \mid C_2 : [\rho_1 \cup \rho_2, (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)]} \quad \rho_1 \cap \rho_2 = \emptyset$$

We have $C\sigma = C_1\sigma | C_2\sigma$. Now, σ is also a proper renaming of ρ_1 and ρ_2. Therefore, by induction hypothesis, $C_1\sigma : [\sigma(\rho_1), \chi_1']$ and $C_2\sigma : [\sigma(\rho_2), \chi_2']$, for some χ_1', χ_2'. Since $\rho_1 \cap \rho_2 = \emptyset$ and σ is a proper renaming of $\rho_1 \cup \rho_2$, we deduce $\sigma(\rho_1) \cap \sigma(\rho_2) = \emptyset$. Then by rule $COMP$, $C\sigma : [\sigma(\rho_1 \cup \rho_2), \chi']$, where $\chi' = (\chi_1' \cup \chi_2') - \sigma(\rho_1 \cup \rho_2)$. Suppose σ is a proper renaming of $tn(C_1|C_2)$. Then σ is also a proper renaming of $tn(C_1)$ and $tn(C_2)$. Then by induction hypothesis, $\chi_1' = \sigma(\chi_1) - \sigma(\rho_1)$ and $\chi_2' = \sigma(\chi_2) - \sigma(\rho_2)$. Using the identities $((R - S) \cup (T - U)) - (S \cup U) = (R \cup T) - (S \cup U)$ and 3.1, we deduce $\chi' = \sigma((\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)) - \sigma(\rho_1 \cup \rho_2)$.

4. $C \equiv [x = y](C_1, C_2)$: The last step in a derivation of $C : [\rho, \chi]$ has to be

$$\frac{C_1 : [\rho, \chi_1] \quad C_2 : [\rho, \chi_2]}{[x = y](C_1,\ C_2) : [\rho, \chi]} \quad \chi = \begin{cases} \chi_1 & \text{if } x \equiv y \\ \chi_2 & \text{otherwise} \end{cases}$$

We have $C\sigma = [\sigma(x) = \sigma(y)](C_1\sigma, C_2\sigma)$. By induction hypothesis, $C_1\sigma : [\sigma(\rho), \chi_1']$ and $C_2\sigma : [\sigma(\rho), \chi_2']$ for some χ_1', χ_2'. Then by rule $COND$, $C\sigma : [\sigma(\rho), \chi']$ for some χ'. Further, suppose σ is a proper renaming of $tn(C)$. Then σ is also a proper renaming of $tn(C_1)$ and $tn(C_2)$. Then by induction hypothesis, $\chi_1' = \sigma(\chi_1) - \sigma(\rho)$ and $\chi_2' = \sigma(\chi_2) - \sigma(\rho)$. Since $\sigma(x) \equiv \sigma(y)$ if and only if $x \equiv y$, it follows that $\chi' = \sigma(\chi) - \sigma(\rho)$.

□

Recall our discussion on rule $BINST$ in Section 3.2. We promised that the constraints which accompany the rule are satisfied provided, for any definition $B \stackrel{def}{=} (u, \tilde{v})u(\tilde{w}).C$

we have $u(\tilde{w}).C : [\{u\}, \{v\}]$. This can be verified using Theorem 3. For any tuple $x, \tilde{y}$, $\{(x, \tilde{y})/(u, \tilde{v})\}$ is a proper renaming of $\{u\}$ and $tn(u(\tilde{w}).C)$ $(= \emptyset)$. Then, by Theorem 3, it follows that $(u(\tilde{w}).C)\{(x, \tilde{y})/(u, \tilde{v})\} : [\{x\}, \{\tilde{y}\} - \{x\}]$.

Before concluding the section, we note that the properties we have shown do not by themselves imply that the type system is sound. These are just a few properties which should be observed if the type system is sound. In fact, in Chapter 4 will prove a few more soundness properties (Sections 4.1 and 4.2.3).

3.4 A Comparison with π-Calculus and its Variants

The syntax of System A is a refinement of the syntax of (polyadic) π-calculus [34, 35, 32]. As in π-calculus, only names are assumed primitive in System A, and all combinators in System A are from π-calculus. We note that the status of conditional construct in π-calculus is unclear. Different versions of the calculus make different choices. Some versions such as [39, 45] retain it, while some such as [32] drop it. Others such as [34, 35] retain it but without the `else` part. The effect of these choices on expressiveness, and equivalences and their axiomatizations has been studied to some extent [40, 38, 9, 39]. In System A we retain the conditional construct mainly because of its convenience. We defer the study of its effect until we have investigated some notions of equivalences in our calculus. We will be using the construct extensively in our examples in Chapter 5.

The central difference between π-calculus and System A is that names in System A identify persistent agents rather than stateless channels. Of course, agents with identities can be interpreted in π-calculus as processes that listen to a single port. Accordingly, our type system can viewed as one that enforces this object paradigm on π-calculus. But we note right away that this does *not* mean that System A is a typed polyadic π-calculus. Encapsulation and fairness in actor systems change the usual operational semantics of π-calculus in a non-trivial way. While encapsulation precludes certain transitions in π-

calculus, fairness deems certain infinite sequences of transitions illegal. The semantics of System A is defined in Chapter 4 using a labeled transition system.

A number of variations of π-calculus have been investigated for specific purposes. The asynchronous π-calculus [22, 10] has served as the basis of experimental programming languages such as Pict [42] and Join [13]. Several type systems have been imposed on π-calculus to capture and study certain disciplines in usage of names that arise very often in practice. The typed versions have been used to prove correctness of program transformations [47, 45], use π-calculus as a metalanguage for semantics of typed object-oriented languages [46], and prove stronger versions of some standard theorems in π-calculus which in turn are useful in establishing certain equivalences [41]. We now briefly discuss some of the relevant variants in the context of System A.

Asynchronous variants of π-calculus such as [22, 10] model asynchronous communication by allowing only the `nil` process as the continuation of an output prefix. Specifically, messages are processes which output at a port and become inactive. We have adopted this idea in System A. Further, like these asynchronous formulations we have dropped the choice combinator. Parallel composition is sufficient to model non-determinism in actor systems. In any case, the meaning of a choice that is not input guarded is unclear in asynchronous calculi [4].

Milner's *sorting* [32] is historically the first of all type systems on π-calculus. A sorting enforces constrains on the name tuples that can be communicated over a channel. Names are partitioned into a collection of sorts and a sorting function is defined which maps sorts onto sequences of sorts. If a sort s is mapped to the sequence (t_1, t_2) then the channel s can carry only pairs of names, where the first name is in t_1 and the second is in t_2. If x is in sort s, u in t_1, and v in t_2 then prefixes $x(u, v)$ and $\overline{x}\langle u, v\rangle$ respect the sorting. A process is said to be well sorted if all prefixes in it respect the given sorting. The essential consequence of this exercise is that for a sorted process arity mismatches such as in $x(u, v).P|\overline{x}\langle y\rangle.Q$ are guaranteed not to occur during runtime. The sorting discipline plays an important role in proving certain properties of π-calculus [55, 44].

The main limitation of sorting is that sort information is completely static: it cannot describe sequencing of values communicated over a channel. For example, it cannot describe a channel which is used to carry an alternating sequence of two and three tuples, or tuples of different sort sequences which have the same length. But this is precisely what is very convenient, if not essential, in actor systems. Names in an actor system identify persistent agents which may change their behavior as they evolve and accept messages with tuples of different types. In fact, we will be using this feature extensively in our examples in Chapter 5. There seems to be no simple of way of extending the sorting discipline to take into account such dynamic constrains on communications. In any case, our aim is to capture only the essential features of the Actor Model. We therefore do not adopt the sorting discipline and leave such embellishments for future research. Readers familiar with Milner's sorting can verify that our encoding of boolean negation in Section 5.2 is not well-sorted.

In [41], Milner's sorting is refined by imposing further discipline on the usage of names. The ability to read from a channel, the ability to write to a channel, and the ability to both read and write are all distinguished from one another. Besides the sorting function, a function I is defined from sorts to a set of tags which indicate the operations allowed on names in a given sort. The tags are: r for read, w for write, and b for both. The fact that a channel in a sort s such that $I(s) = b$ can be used in a context which expects it with an r or w naturally gives rise to a subtype relation $\leq$ on (tagged) sorts. Now, let us reconsider the example we presented while discussing sorting. Suppose u is in sort t'_1 and v is in sort t'_2. Then the input prefix $x(u, v)$ is well sorted if $t'_i \leq t_i$ for $i = 1, 2$, and $I(s)$ is either r or b. Similarly the output prefix $\overline{x}\langle u, v\rangle$ is well typed if if $t'_i \leq t_i$ for $i = 1, 2$, and $I(s)$ is either w or b. The greater precision of this type system is used to prove correctness of certain encodings which are otherwise unsound in the untyped π-calculus. In [46] this type system is enhanced with variant types, and the resulting calculus is used as a metalanguage for interpreting Abadi and Cardelli's first order functional Object Calculus.

An analogy between our type system and the one discussed above is illuminating. First, since we have not adopted the notion of sorting let us consider a scenario where names are directly associated with tags which indicate whether they may be validly used for input and/or output. Now, we can interpret of our type system as follows. The fact that an actor cannot create new actors with names received in a message means that the bound names in an input prefix are implicitly tagged with a w. This is also reminiscent of languages such as Pict and Join where only output capabilities can be passed in messages. Similarly, since names bound by restriction denote internal actors they are assume to be tagged with b. A judgement $C : [\rho, \chi]$ can be interpreted as: C is well-typed in a context which tags names in ρ with b, and the names in χ with w.

The reader may note that in contrast to [41] the annotations of all free and bound occurrences of names are inferred by our type system; they need not be explicitly mentioned. Further, associating input/output capabilities with names only enforces one of the properties of actor systems. Our type system enforces several additional constraints which guarantee properties such as uniqueness of actor names, and persistence of actors.

Chapter 4

System A: Operational Semantics

In this chapter we define the semantics of System A using a labeled transition system. Although only terms represent configurations we define the transition system over preterms. We do this to show that our type system is sound; we will prove that the set of terms is closed under transitions. The transition system by itself does not enforce fairness on message deliveries. We capture fairness by imposing additional constraints on sequences of transitions.

In Section 4.1 we define an equivalence relation on preterms which helps simplify the definition of our labeled transition system. We also present the notion of canonical form of terms which is useful in defining fairness. In Section 4.2 we present the labeled transition system, and show how it captures the evolution of a configuration's interface and the encapsulation enforced by the interface. We define fair computations in Section 4.3.

4.1 Equivalence Relation on Preterms

To simplify the definition of our labeled transition system we identify several preterms. This approach has been adopted in the formulation of a number of calculi [32, 22], and was originally inspired by the Chemical Abstract Machine of Berry and Boudol [8].

Definition 8 *The relation $\simeq_{\mathcal{C}}$ is the smallest equivalence relation on preterms such that the following laws hold:*

(alpha) $C_1 \simeq_{\mathcal{C}} C_2$ *if* $C_1 \equiv_\alpha C_2$

(comm) $C_1|C_2 \simeq_{\mathcal{C}} C_2|C_1$

(assoc) $(C_1|C_2) \mid C_3 \simeq_{\mathcal{C}} C_1 \mid (C_2|C_3)$

(scope-nest) $(\nu x, y)C \simeq_{\mathcal{C}} (\nu y, x)C$

(scope-expn) $(\nu x)C_1|C_2 \simeq_{\mathcal{C}} (\nu x)(C_1|C_2)$ *if* $x \notin fn(C_2)$

(grbg) $C|0 \simeq_{\mathcal{C}} C$

(const) $B\langle x, \tilde{y}\rangle \simeq_{\mathcal{C}} C\{(x,\tilde{y})/(u,\tilde{v})\}$ *if* $B \stackrel{def}{=} (u,\tilde{v})u(\tilde{w}).C$

(cond) $$[x = y](C_1, C_2) \simeq_{\mathcal{C}} \begin{cases} C_1 & \text{if } x \equiv y \\ C_2 & \text{otherwise} \end{cases}$$

(cntxt) $C_1 \simeq_{\mathcal{C}} C_2$ *implies*

(a) $(\nu x)C_1 \simeq_{\mathcal{C}} (\nu x)C_2$

(b) $C|C_1 \simeq_{\mathcal{C}} C|C_2$

Note that $\simeq_{\mathcal{C}}$ relates two terms only if they represent the same configuration. Further, $\simeq_{\mathcal{C}}$ is not a congruence relation. We do not want the **(cond)** law to be applied to a conditional construct occuring in an actor behavior. The names being matched may be bound by an input prefix, in which case the condition is to be tested only after the bound names are replaced by names received in a message. Specifically, the following context law is not appropriate for our purposes.

$$C_1 \simeq_{\mathcal{C}} C_2 \quad \Rightarrow \quad x(\tilde{y}).C_1 \simeq_{\mathcal{C}} x(\tilde{y}).C_2.$$

It is easy to see that the following context law is redundant. It follows from **(cond)** and equivalence laws.

$$C_1 \simeq_{\mathcal{C}} C_1', C_2 \simeq_{\mathcal{C}} C_2' \quad \Rightarrow \quad [x = y](C_1, C_2) \simeq_{\mathcal{C}} [x = y](C_1', C_2')$$

Since $\simeq_{\mathcal{C}}$ relates only identical terms we would expect our type system to respect $\simeq_{\mathcal{C}}$. Theorem 4 formally states this property.

Theorem 4 *Let $C_1 \simeq_{\mathcal{C}} C_2$. Then $C_1 : [\rho, \chi]$ iff $C_2 : [\rho, \chi]$.*

Proof: We prove this by induction on the number of steps in a derivation of $C_1 \simeq_{\mathcal{C}} C_2$.

Base case: The relation is derived by a direct application of a law other than **(cntxt)**. We consider only the interesting cases, and leave the rest for the reader.

1. **(alpha)**: The hypothesis holds by Theorem 2.

2. **(scopeexp)**: We only prove $C_1 : [\rho, \chi]$ implies $C_2 : [\rho, \chi]$. The other direction can be proved in a similar way. For some terms C_1', C_2', we have $C_1 = (\nu x)C_1'|C_2'$, $C_2 = (\nu x)(C_1'|C_2')$ and $x \notin fn(C_2')$. For some $\rho_1, \rho_2, \chi_1, \chi_2$, we have $C_1' : [\rho_1, \chi_1]$ and $C_2' : [\rho_2, \chi_2]$. Then by rules RES and $COMP$ of table 3.2, $(\rho_1 - \{x\}) \cap \rho_2 = \emptyset$, and $\rho = (\rho_1 - \{x\}) \cup \rho_2,\ \chi = (\chi_1 \cup \chi_2) - ((\rho_1 - \{x\}) \cup \rho_2)$. By Lemma 4, $x \notin \rho_2, x \notin \chi_2$, and by Theorem 1, $x \notin \chi_1$. It follows that $\rho_1 \cap \rho_2 = \emptyset$, $\rho = (\rho_1 \cup \rho_2) - \{x\}$, and $\chi = (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)$. Then by rules $COMP$ and RES, we have $C_2 : [\rho, \chi]$.

3. **(const)**: We considered this in the discussion following Theorem 3 (Section 3.3).

Induction Step: The last step of an arbitrary derivation can use any of the laws. The only laws not considered in the base case are the **(cntxt)** laws, and symmetry and transitivity laws of equivalence relation. The arguments for all these cases are straightforward, and are left to the reader. □

Related to $\simeq_{\mathcal{C}}$ is the notion of canonical form. Although this notion can be defined on preterms, we bother to define it only on terms.

Definition 9 *A term $(\nu\tilde{x})(C_1|\cdots|C_n)$ is said to be in canonical form if each C_i is either an input term, or an output term. If $\tilde{x}$ is the empty tuple then $(C_1|\cdots|C_n)$ is said to be open.*

In a canonical term $(\nu\tilde{x})(\|_{i=1}^{i=n} C_i)$, the C_i's which are input terms are all the internal actors, and the C_i's which are output terms are all the messages in the configuration. The tuple

$\tilde{x}$ contains names of all the internal actors which are not receptionists, and no two names in it are the same. This can be deduced from rule RES of Table 3.2. If $(\nu\tilde{x})(\mid_{i=1}^{i=n} C_i) : [\rho, \chi]$ then by repeated application of the rule, it follows that $\tilde{x}$ is a tuple of distinct names and $\mid_{i=1}^{i=n} C_i : [\rho \cup \{\tilde{x}\}, \chi]$.

Terms in canonical form are handy because it is possible to unambiguously refer to an actor or a message in the term by using the actor name or the message target and contents, even if the names we use are bound. This is not always possible for arbitrary terms. For example, in a term of the form $(\nu x)x(\tilde{y}).C_1 \mid (\nu x)x(\tilde{z}).C_2$, the bound name x is used to denote two internal actors. The actors have distinct identities because the restrictions have disjoint scopes.

Readers familiar with [3] may note that the representation of actor configurations used in it closely corresponds to our notion of canonical form. However, there is some difference between the two. In [3], the interface of a configuration is explicit and remembers all the external names that were ever present in the configuration. This information is used to ensure that no internal actors are created with external names. But in our case, the interface is implicit and is derived from the syntax of a term. This implies that external names may be forgotten as a given term evolves. For example, this may happen when an internal actor forgets some of its acquaintances after receiving a message, or when a message targeted to an external actor is sent to the environment. We make up for this loss in the definition of admissible computations (Section 4.3.2).

We now show that every term is equivalent to a term in canonical form.

Lemma 6 *For every term C, there is a canonical term C' such that $C \simeq_{\mathcal{C}} C'$.*

Proof: Let C be a given term. We prove the hypothesis by structural induction on C. If C is an output or input term, it is already in canonical form. If C is a behavior instantiation, then the **(const)** law gives a canonical form. If $C \equiv (\nu x)C_1$ then a canonical form of C_1 together with the **(cntxt)**(a) law gives a canonical form of C. If $C \equiv [x = y](C_1, C_2)$ then canonical forms of C_1 and C_2 together with rule **(cond)** give

a canonical form of C. Finally, if $C \equiv C_1 \mid C_2$ then we can find canonical forms of C_1 and C_2, and pull out the outer restrictions in the canonical forms with repeated alpha-conversions and application of **(scope-exp)** and **(cntxt)**(a) laws to get a canonical form of C. $\Box$

A term may have more than one canonical form. The following lemma shows how different canonical forms of a given term are related.

Lemma 7 *Let $(\nu\tilde{x})C_1 \simeq_{\mathcal{C}} (\nu\tilde{y})C_2$ and C_1, C_2 be open. Then the following statements are true*

1. *$len(\tilde{x}) = len(\tilde{y})$*
2. *There exists a permutation $\tilde{y'}$ of names in $\tilde{y}$ such that $C_2 \simeq_{\mathcal{C}} C_1\{\tilde{y'}/\tilde{x}\}$*

Proof: We prove the statements simultaneously by induction on the maximum of $len(\tilde{x})$ and $len(\tilde{y})$. For the case where both the tuples are empty, statement 1 is trivially true, and statement 2 follows by setting $\tilde{y'} = \tilde{y}$ and using **(alpha)** and equivalence laws. In the induction step we look at a derivation of $(\nu\tilde{x})C_1 \simeq_{\mathcal{C}} (\nu\tilde{y})C_2$. We use the fact that the last derivation step has to be an application of a law, in which both the terms being related are in canonical form and at least one of them has a restriction as the outermost combinator. This narrows down the possibilities to **(alpha)**, **(scope-nest)**, **(cntxt)**(a), and the equivalence laws. Further details are left to the reader.

4.2 Labeled Transition System

We model computations in actor systems using labeled transitions between preterms. A transition is of form $C_1 \xrightarrow{\alpha} C_2$, and means that C_1 can perform the action α and evolve into C_2. We call C_1 the source of the transition, and C_2 the target. Although transitions are defined on preterms for reasons mentioned earlier, in all our informal discussions we assume that the source of a transitions is a term.

Action (α)	Kind	$fn(\alpha)$	$bn(\alpha)$
τ	Silent	$\emptyset$	$\emptyset$
$\mathtt{in}(\overline{x}\langle\tilde{y}\rangle)$	Input	$\{x, \tilde{y}\}$	$\emptyset$
$\mathtt{out}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle)$	Output	$\{x, \tilde{y}\} - \{\tilde{y'}\}$	$\{\tilde{y'}\}$

Table 4.1: Free and bound names of actions

4.2.1 Actions

We denote the set of actions by $\mathcal{A}$, and let α, β range over it. There are three kinds of actions

1. Silent, τ: The transition $C_1 \xrightarrow{\tau} C_2$ represents the delivery of a single message to its target in C_1. After the delivery C_1 evolves to C_2. This action does not involve an interaction between C_1 and its environment. Specifically, the message is already present in C_1. We represent all internal deliveries uniformly by τ. This is because we wish to abstract away the details of internal communication.

2. Input, $\mathtt{in}(\overline{x}\langle\tilde{y}\rangle)$: The transition $C_1 \xrightarrow{\mathtt{in}(\overline{x}\langle\tilde{y}\rangle)} C_2$ means that C_1 may receive the message $\overline{x}\langle\tilde{y}\rangle$ from its environment and evolve into C_2. The message is targeted to a receptionist in C_1 and is not yet delivered to its target at the end of the transition. It is just added to the pool of messages in C_1.

3. Output, $\mathtt{out}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle)$: The transition $C_1 \xrightarrow{\mathtt{out}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle)} C_2$ means that C_1 may emit the message $\overline{x}\langle\tilde{y}\rangle$ to its environment and evolve into C_2. The name x is an external name in C_1. Names in $\tilde{y'}$ are all distinct, occur in $\tilde{y}$ and denote non-receptionists in C_1. It is therefore always the case that $x \notin \{\tilde{y'}\}$ and $\{\tilde{y'}\} \subset \{\tilde{y}\}$.

All the actions along with definitions of free and bound names in them are given in Table 4.1. For an action, α we define $n(\alpha) = fn(\alpha) \cup bn(\alpha)$. For actions α and β, we write $\alpha \equiv \beta$ if α and β are syntactically identical.

4.2.2 Transitions

The transition relation $\xrightarrow{\alpha}$ is the smallest relation on preterms that is closed under the rules in Table 4.2.2. The $RECV$, PAR and $HIDE$ rules are for inferring transitions that involve a silent action. The IN and OUT rules are for inferring transitions that involve input and output actions. The $EQUIV$ rule is used for both kinds of transitions.

The $RECV$ rule represents the delivery of a message to its target. It formally captures what we have been assuming all along; the actor $x(\tilde{y}).C$ may receive a message $\overline{x}\langle\tilde{z}\rangle$, where $\tilde{y}$ and $\tilde{z}$ are tuples of the same length, and replace itself with $C\{\tilde{z}/\tilde{y}\}$. Our type system guarantees that the actor x is present in $C\{\tilde{z}/\tilde{y}\}$. By the RES rule of Table 3.2, $C : [\{x\}, \chi']$ for some χ', and by Theorem 3, $C\{\tilde{z}/\tilde{y}\} : [\{x\}, \chi'\{\tilde{z}/\tilde{y}\} - \{x\}]$. Actors other than x and all messages in $C\{\tilde{z}, \tilde{y}\}$ are all created in response to the delivery. All the fresh actors are anonymous, because only x is the receptionist of $C\{\tilde{z}/\tilde{y}\}$. We already showed in Section 3.2 that the locality laws are satisfied.

The PAR and $HIDE$ rules state that internal actions can occur under composition and restriction. The PAR rule captures the fact that actors in the two configurations being composed execute concurrently. The $HIDE$ rule captures the fact that the restriction (νx) in $(\nu x)C$ hides actor x in C only from the environment; actors in C with x as an acquaintance can still send messages to x.

The IN rule states that a configuration may receive a message targeted to one of its receptionists from its environment. At the end of the transition the message is not yet delivered to its target; it is just added to the pool of messages in the configuration.

The OUT rule states that a configuration may emit a message in it that is targeted to an external actor. The message may contain names of non-receptionists which will then become receptionists in the resulting configuration. Bound names in the output action are the names of non-receptionists that are being exported; the RES rule guarantees that these names refer to internal actors. The names are no longer bound by restrictions in the transition target. Further, by Lemma 4, these names are fresh; they do not already

$EQUIV: \dfrac{C_1 \simeq_{\mathcal{C}} C_1' \quad C_1' \xrightarrow{\alpha} C_2' \quad C_2' \simeq_{\mathcal{C}} C_2}{C_1 \xrightarrow{\alpha} C_2}$
$RECV: \dfrac{}{x(\tilde{y}).C \mid \overline{x}\langle\tilde{z}\rangle \xrightarrow{\tau} C\{\tilde{z}/\tilde{y}\}} \quad len(\tilde{y}) = len(\tilde{z})$
$PAR: \dfrac{C_1 \xrightarrow{\tau} C_1'}{C_1 \mid C_2 \xrightarrow{\tau} C_1' \mid C_2} \qquad HIDE: \dfrac{C \xrightarrow{\tau} C'}{(\nu x)C \xrightarrow{\tau} (\nu x)C'}$
$IN: \dfrac{}{C \xrightarrow{\mathtt{in}(\overline{x}\langle\tilde{y}\rangle)} C \mid \overline{x}\langle\tilde{y}\rangle} \quad C : [\rho, \chi],\ x \in \rho$
$OUT: \dfrac{}{(\nu\tilde{y'})(C \mid \overline{x}\langle\tilde{y}\rangle) \xrightarrow{\mathtt{out}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle)} C} \quad C : [\rho, \chi],\ x \notin \rho,\ \{\tilde{y'}\} \subset \{\tilde{y}\}$

Table 4.2: Transition rules for System A

denote a receptionist or external name in the transition source. Theorem 6 formally characterizes the way a configuration's interface may change during a transition.

The $EQUIV$ rules states that equivalent terms have the same transitions. This enables us to use the laws for $\simeq_{\mathcal{C}}$ to juxtapose a message and its target so that the $RECV, PAR$ and $HIDE$ laws can be applied to derive a transition that corresponds to the delivery of the message. Similarly a configuration with a message targeted to an external actor can be transformed to a form suitable for the OUT rule.

Recall that encapsulation laws constrain the interactions between a configuration and its environment based on the configuration's interface (Section 2.1.3). We now show that the transition relation satisfies these laws.

Theorem 5 *Let* $C : [\rho, \chi]$. *Then*

1. $C \xrightarrow{\mathtt{in}(\overline{x}\langle\tilde{y}\rangle)} C'$ *implies* $x \in \rho$.
2. $C \xrightarrow{\mathtt{out}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle)} C'$ *implies* $x \in \chi$.

Proof: We prove this by induction on the number of steps in a derivation of a transition.

Base Case: If the transition is derived by a direct application of IN rule then the statements clearly hold. If the derivation is a direct application of OUT rule then $C \equiv (\nu\tilde{y})(C'|\overline{x}\langle y\rangle)$, and for some ρ', χ', $C' : [\rho', \chi']$ then $x \notin \rho'$. Then by $COMP$ rule of Table 3.2, $x \in \chi$.

Induction Step: The last derivation step has to be an application of the $EQUIV$ rule. Then, from induction hypothesis and Theorem 4, it follows that the statements hold. □

The encapsulation laws also stipulate that messages received by a configuration from its environment cannot contain names of actors in the configuration which are not receptionists. This is automatically guaranteed in our system because names of internal actors are bound by restrictions and hence cannot be identified with names received in such messages. Note that in the IN rule the message received is placed outside the scope of all restrictions.

4.2.3 Soundness

If our type system is sound the set of terms should be closed under transitions. Specifically, for a term C if $C \xrightarrow{\alpha} C'$ then C' should also be a term. The following theorem says that this is indeed the case. The theorem also characterizes the way configuration

interfaces change during transitions (see Section 2.1.3). During output actions names of non-receptionists may be exported to the environment, thus creating new receptionists. Messages received during input actions may contain external names that the configuration does not yet know. During internal message deliveries or output actions the configuration may loose some external names.

Theorem 6 *Let* $C_1 : [\rho_1, \chi_1]$ *and* $C_1 \xrightarrow{\alpha} C_2$. *Then* $C_2 : [\rho_2, \chi_2]$ *where*

$$\rho_2 = \begin{cases} \rho_1 \cup \{\tilde{y'}\} & \textit{if } \alpha = \mathtt{out}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle) \\ \rho_1 & \textit{otherwise} \end{cases}$$

$$\chi_2 = \begin{cases} \chi_1 \cup (\{\tilde{y}\} - \rho_1) & \textit{if } \alpha = \mathtt{in}(\overline{x}\langle\tilde{y}\rangle) \\ \chi', \quad \textit{where } \chi' \subset \chi_1 & \textit{otherwise} \end{cases}$$

Proof: We prove this by induction on the number of steps in a derivation of $C_1 \xrightarrow{\alpha} C_2$.

Base case: The transition is derived by a direct application of $RECV, IN$ or OUT rules of Table 4.2.2. We show that the hypothesis holds in each case. Variables that occur in the following arguments are from the rules in Table 4.2.2.

1. $RECV$: By the ACT rule of Table 3.2 we deduce $C : [\{x\}, \chi']$ for some χ'. By rules ACT and $COMP$ of Table 3.2, $\rho_1 = \{x\}$ and $\chi_1 = (fn(C) - \{x, \tilde{y}\}) \cup (\{\tilde{z}\} - \{x\})$. Trivially, $\{\tilde{z}/\tilde{y}\}$ is a proper renaming of $\{x\}$. By the ACT rule, we have $x \notin \{\tilde{y}\}$. Then, by Theorem 3, $C\{\tilde{z}/\tilde{y}\} : [\{x\}, \chi_2]$ for some χ_2. By Lemma 4 and Theorem 1, we have $\chi_2 \subset fn(C\{\tilde{z}/\tilde{y}\}) - \{x\} \subset \chi_1$.

2. IN: We have $C : [\rho_1, \chi_1]$ and $x \in \rho_1$. By Theorem 1, $\rho_1 \cap \chi_1 = \emptyset$ and $x \notin \chi_1$. By rules MSG and $COMP$ of Table 3.2, we deduce $C \mid \overline{x}\langle\tilde{y}\rangle : [\rho_1, \chi_1 \cup (\{\tilde{y}\} - \rho_1)]$.

3. OUT: We have $C : [\rho_2, \chi_2]$. By rule $COMP$ of Table 3.2, $C \mid \overline{x}\langle\tilde{y}\rangle : [\rho_2, (\chi_2 \cup \{x, \tilde{y}\}) - \rho_2]$. Then by rule RES of Table 3.2, $\rho_1 = \rho_2 - \{\tilde{y'}\}$ and $\{\tilde{y'}\} \subset \rho_2$. Therefore, $\rho_2 = \rho_1 \cup \{\tilde{y'}\}$. Also, by rule RES, $\chi_1 = (\chi_2 \cup \{x, \tilde{y}\}) - \rho_2$. By Theorem 1, $\chi_1 = \chi_2 \cup (\{x, \tilde{y}\} - \rho_2)$. Therefore, $\chi_2 \subset \chi_1$.

Induction step: If the last step of derivation uses the $EQUIV$ rule then a straight forward application of Theorem 4, shows that the hypothesis holds. Similarly, for the cases of $COMP$ and $HIDE$ rules, a simple application of $COMP$ and RES rules of Table 3.2 shows that the hypothesis holds. The details are simple and left to the reader.

4.3 Admissible Computations

4.3.1 Computations

A computation in the Actor Model corresponds to a sequence of transitions starting from a term. To facilitate the following discussions, for $C : [\rho, \chi]$, we define $recep(C) = \rho$ and $extern(C) = \chi$.

Definition 10 *A computation path, or just path for brevity, is a finite or infinite sequence of transitions such that the target of a transition is the source of the next. Thus, a path has the form*

$$C_0 \xrightarrow{\alpha_0} C_1 \xrightarrow{\alpha_1} \cdots C_i \xrightarrow{\alpha_i} \cdots, \quad \textit{where } 0 < i < \bowtie, \ \ \bowtie \in \mathbf{Nat} \cup \{\omega\}$$

for some term C_0. For the path p above, for $0 \leq i < \bowtie$, we define

1. $len(p) = \bowtie$
2. $p(i) = C_i \xrightarrow{\alpha_i} C_{i+1}$
3. $src(p, i) = C_i$
4. $tgt(p, i) = C_{i+1}$
5. $lbl(p, i) = \alpha_i$
6. $src(p) = src(p, 0) = C_0$
7. $recep(p, i) = recep(src(p, i))$

8. $recep(p) = recep(src(p)) \ \cup \ \cup_{0 \leq j < \bowtie} \ recep(tgt(p, j))$

9. $extern(p, i) = extern(src(p, i))$

10. $extern(p) = extern(src(p)) \ \cup \ \cup_{0 \leq j < \bowtie} \ extern(tgt(p, j))$

11. $prefix(p, i) = p' \ where \quad len(p') = i + 1 \ \wedge \ tgt(p', i) \equiv tgt(p, i) \ \wedge$
$\forall j \leq i.(src(p', j) \equiv src(p, j) \ \wedge \ lbl(p', j) \equiv lbl(p, j))$

12. $isprefix(p', p) \Leftrightarrow \exists i.prefix(p, i) = p'$

To avoid some pathological technical difficulties, we assume that for any path there are infinitely many names not used in the path, i.e. for every path p

$$o(\mathcal{N} - (recep(p) \cup extern(p))) = \aleph_0$$

We define $\mathcal{P}$ to be the set of all computation paths, and $\mathcal{P}_{fin}$ to be the set of all paths of finite length.

Note that a transition is a path of unit length. Further, Theorem 6 implies that for any $p \in \mathcal{P}$, $tgt(p, i)$ is a term for all $i < len(p)$.

Not all paths represent computations in the Actor Model. The paths that do are called admissible paths. A path may not be admissible because of two reasons. First, it need not satisfy the fairness requirement on message deliveries (Section 2.1.4). Second, the source of the path may evolve into a term which contains an internal actor with a forgotten external name, thus violating uniqueness of actor names. In Sections 4.3.2 and 4.3.3 we define predicates on $\mathcal{P}$ which check for these violations. The set of paths defined by these predicates is precisely the set of admissible paths. The approach we adopt closely resembles the one in [49].

4.3.2 Global Address Constraint

Since a term may lose external names as it evolves, it may eventually have an internal actor with one of these forgotten names. For instance, consider the following path.

$$(\nu x)(x().C|\overline{u}\langle\rangle|\overline{v}\langle x\rangle) \stackrel{\mathtt{out}(\overline{u}\langle\rangle)}{\longrightarrow} (\nu x)(x().C|\overline{v}\langle x\rangle) \stackrel{\mathtt{out}((\nu u)\overline{v}\langle u\rangle)}{\longrightarrow} u().C\{u/x\}$$

In the first transition the external name u is forgotten. In the second transition u is chosen as the name of the internal actor and is exported to the environment. This is possible because the *OUT* rule does not have access to the computational history of a term; it only ensures that the names chosen for non-receptionists whose names are being exported do not already occur free in the source of the transition. We impose a Global Address Constraint (GAC) on $\mathcal{P}$ which deems paths such as the above illegal.

$$GAC(p) \Leftrightarrow recep(p) \cap extern(p) = \emptyset$$

4.3.3 Fairness

Before we formally define the set of fair paths let us investigate our notion of fairness through a few examples. The following two paths are evidently unfair.

$$Diverge\langle x\rangle|\overline{x}\langle y\rangle|\overline{u}\langle v\rangle \stackrel{\tau}{\longrightarrow} Diverge\langle x\rangle|\overline{x}\langle y\rangle|\overline{u}\langle v\rangle \stackrel{\tau}{\longrightarrow} \cdots$$

$$Diverge\langle x\rangle \stackrel{\mathtt{in}(\overline{x}\langle y\rangle)}{\longrightarrow} Diverge\langle x\rangle|\overline{x}\langle y\rangle \stackrel{\mathtt{in}(\overline{x}\langle y\rangle)}{\longrightarrow} Diverge\langle x\rangle|\overline{x}\langle y\rangle|\overline{x}\langle y\rangle \stackrel{\mathtt{in}(\overline{x}\langle y\rangle)}{\longrightarrow} \cdots$$

$$\text{where } \; Diverge \stackrel{def}{=} (x)x(y).(\overline{x}\langle y\rangle \mid Diverge\langle x\rangle)$$

In every transition of the first path the message $\overline{x}\langle y\rangle$ is delivered to its target. But the message $\overline{u}\langle v\rangle$, which is targeted an external actor, is never delivered to the environment. In the second path, every transition involves the receipt of a message from the environment. Messages already in the configuration are never delivered to x. Note that both

these paths are infinitely long. In fact, every finite path is fair. This is because fairness only requires that the delivery of a given message is not delayed for infinite steps; but it can be delayed for any finite number of steps.

Our notion of fairness does *not* require that every message is eventually delivered to its target. Our fairness criteria is an instance of the general notion of *strong fairness* discussed in [26]. A message need not always be enabled for delivery at a given time. Depending on its behavior an actor is ready to accept only those messages that carry tuples of certain length (see rule IN of Table 4.2.2). On the other hand, messages that are targeted to external actors can always be delivered to the environment. Our fairness criteria only requires that there is no message that is infinitely often enabled and not delivered. For example, the following path is fair

$$Diverge\langle x\rangle|\overline{x}\langle y\rangle|\overline{x}\langle u,v\rangle \xrightarrow{\tau} Diverge\langle x\rangle|\overline{x}\langle y\rangle|\overline{x}\langle u,v\rangle \xrightarrow{\tau} \cdots$$

In every transition above, only $\overline{x}\langle y\rangle$ is enabled for delivery. Thus, all the silent actions correspond to its delivery. The path is still fair because $\overline{x}\langle u,v\rangle$ is never enabled.

We now proceed to the definition of fair paths. We immediately encounter two technical difficulties. First, given a path, to check if a message is infinitely often enabled we need to be able to track the message throughout the path. But this is not possible in arbitrary paths because terms in the path need not be in canonical form, and the $EQUIV$ rule allows alpha conversions across transitions. Second, given a transition with silent action, it is not possible to tell which message delivery caused the transition. For example, the transition $Divergex|\overline{x}\langle y\rangle|\overline{x}\langle z\rangle \xrightarrow{\tau} Divergex|\overline{x}\langle y\rangle|\overline{x}\langle z\rangle$ can be caused by delivery of either of the two messages.

To get around this problem we define canonical paths and annotations of canonical paths. A canonical path has all terms in it in canonical form, and does not allow alpha conversions across transitions. An annotation of a canonical path specifies for each silent transition in the path, a message in the configuration whose delivery could have caused that transition. It is straightforward to define the set of fair annotated canonical paths.

To define fairness on arbitrary paths we first define an equivalence relation on $\mathcal{P}$ which identifies paths which are evidently the same, and show that every path is equivalent to a path in canonical form. We then say a path is *observably* fair if it is equivalent to a canonical path which has a fair annotation. The fairness is only observational because a canonical path may have both fair and unfair annotations.

We now formalize the ideas presented above.

Definition 11 *The relation $\simeq_{\mathcal{P}}$ on $\mathcal{P}$ is defined as*

$$\begin{aligned} p_1 \simeq_{\mathcal{P}} p_2 \quad \Leftrightarrow \quad & len(p_1) = len(p_2) \ \wedge \ src(p_1) \simeq_{\mathcal{C}} src(p_2) \\ & \forall i < len(p_1).\ tgt(p_1, i) \simeq_{\mathcal{C}} tgt(p_2, i) \ \wedge \ lbl(p_1, i) \equiv lbl(p_2, i) \end{aligned}$$

Since $\simeq_{\mathcal{C}}$ is an equivalence relation, so is $\simeq_{\mathcal{P}}$.

Lemma 8 *The relation $\simeq_{\mathcal{P}}$ is an equivalence relation.*

Paths related by $\simeq_{\mathcal{P}}$ are the same because they exhibit the same (observable) sequence of actions on the same configuration. Moreover, the intermediate configurations are also identical. This, of course, is a very strong criteria for equating paths. But it suffices for our purposes; it is lax enough to allow every path to be equivalent to one in canonical form.

We would expect two paths identified by $\simeq_{\mathcal{P}}$ to either both satisfy, or both not satisfy the GAC constraint.

Lemma 9 *Let $p_1 \simeq_{\mathcal{P}} p_2$. Then $GAC(p_1) \Leftrightarrow GAC(p_2)$.*

Proof: By Definition 11, we have $len(p_1) = len(p_2) = l$ (say), $src(p_1) \simeq_{\mathcal{C}} src(p_2)$, and $tgt(p_1, i) \simeq_{\mathcal{C}} tgt(p_2, i)$ for all $i < l$. Then using Theorem 4, we deduce $recep(p_1) = recep(p_2)$ and $extern(p_1) = extern(p_2)$. Hence, $GAC(p_1) \Leftrightarrow GAC(p_2)$. $\square$

Now, we define the notion of canonical form for computation paths.

Definition 12 *A path p is said to be in canonical form if for all $i < len(p)$, $src(p)$ and $tgt(p,i)$ are of form $(\nu\tilde{x})C$ for some open C, and for some $u, \tilde{v}, \tilde{w}, C_1, C_2$*

1. *If $lbl(p,i) = \tau$ then $C \simeq_{\mathcal{C}} C_1 \mid u(\tilde{v}).C_2 \mid \overline{u}\langle\tilde{w}\rangle$, and $tgt(p,i)$ is of form $(\nu\tilde{x})C'$, where $C' \simeq_{\mathcal{C}} C_1 \mid C_2\{\tilde{w}/\tilde{v}\}$.*
2. *If $lbl(p,i) = \mathtt{in}(\overline{u}\langle\tilde{v}\rangle)$ then $tgt(p,i)$ is of form $(\nu\tilde{x})C'$, where $C' \simeq_{\mathcal{C}} C \mid \overline{u}\langle\tilde{v}\rangle$.*
3. *If $lbl(p,i) = \mathtt{out}((\nu\tilde{w})\overline{u}\langle\tilde{v}\rangle)$ then $C \simeq_{\mathcal{C}} C_1 \mid \overline{u}\langle\tilde{v}\rangle$, and $tgt(p,i)$ is of form $(\nu\tilde{y})C'$, where $\tilde{y} = \tilde{x} - \tilde{w}$, and $C' \simeq_{\mathcal{C}} C_1$.*

Note how we relate the source and target of each transition in the path to prevent renaming of non-receptionists across transitions.

We now sketch a proof of the fact that every path is equivalent to a canonical path.

Theorem 7 *For every path p there is a canonical path p' such that $p \simeq_{\mathcal{P}} p'$.*

Proof: By induction on the length of a derivation of a transition, we can show that the statement above is true for all paths of unit length. Given a path $C_1 \xrightarrow{\alpha} C_2$, in the induction step we can find canonical terms C_1', C_2', such that $C_1' \simeq_{\mathcal{P}} C_1$ and $C_2' \simeq_{\mathcal{P}} C_2$, and show that $C_1' \xrightarrow{\alpha} C_2'$ is canonical.

Given an arbitrary path p, using the above result, we find for each $i < len(p)$ a canonical transition t_i such that $p(i) \simeq_{\mathcal{P}} t_i$. Let $tgt(t_i) = (\nu\tilde{x})C_i$ and $src(t_{i+1}) = (\nu\tilde{y})C_i'$. Since $tgt(t_i) \simeq_{\mathcal{P}} src(t_{i+1})$, by Lemma 7, there is a permutation $\tilde{y'}$ of names in $\tilde{y}$ such that $C_i' \simeq_{\mathcal{C}} C_i\{\tilde{y'}/\tilde{x}\}$. Using these substitutions we can track an actor from its creation and check if it ever becomes a receptionist. The idea is to fix a name for the actor once and for all, avoiding renamings across transitions. This can be done as follows. If the actor becomes a receptionist then we directly choose its exported name. Else, we can pick a name for it, that has not already been chosen, from the set $\mathcal{N} - (recep(p) \cup extern(p))$. Note that by our assumption in Definition 10 there are always enough names to pick from.

With this trick we can show that there is a sequence of canonical paths $p_0, \cdots, p_i, \cdots$, where $i < len(p)$, such that $len(p_i) = i + 1$, $isprefix(p_j, p_k)$ for $j \leq k$ and $p_i \simeq_{\mathcal{P}} prefix(p, i)$. (But note that the trick does not give an effective construction of the sequence). Now, the path p' such that $len(p') = len(p)$ and $isprefix(p_i, p')$ for $i < len(p)$ is in canonical form and $p' \simeq_{\mathcal{P}} p$. $\square$

Now, we define annotation of canonical paths. We extend $\mathcal{A}$ to $\mathcal{A}'$ by adding receive actions which denote internal deliveries. A receive action is of form $\texttt{rcv}((\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle)$ and represents delivery of $\overline{x}\langle\tilde{y}\rangle$. The actor x is an internal actor, and $\tilde{y'}$ is a tuple of distinct names. The tuple $\tilde{y'}$ contains exactly the names in $x, \tilde{y}$ which denote non-receptionists. So, it is always the case that $\{\tilde{y'}\} \subset \{\tilde{y}\}$, and unlike output actions it may be the case that $x \in \{\tilde{y'}\}$.

Definition 13 *A function f : **Nat** $\rightarrow \mathcal{A}'$ is an annotation of a canonical path p if for all $i < len(p)$ and for some $u, \tilde{v}, \tilde{w}, \tilde{y}$*

1. *If $lbl(p, i) \neq \tau$ then $f(i) = lbl(p, i)$.*
2. *If $lbl(p, i) = \tau$ then $f(i) = \texttt{rcv}((\nu\tilde{w})\overline{u}\langle\tilde{v}\rangle)$, $src(p, i)$ is of form $(\nu\tilde{x})C$ where C is open, and $C \simeq_{\mathcal{C}} C_1 \mid u(\tilde{y}).C_2 \mid \overline{u}\langle\tilde{v}\rangle$, $\{\tilde{w}\} = \{\tilde{x}\} \cap \{u, \tilde{v}\}$, and $tgt(p, i)$ is of form $(\nu\tilde{x})C'$, where $C' \simeq_{\mathcal{C}} C_1 \mid C_2\{\tilde{v}/\tilde{y}\}$.*

A direct consequence of Definition 12 is that every canonical path has an annotation.

We now define fairness on annotated canonical paths. For this, we define two predicates, *Enabled* and *Fires*. The predicate *Enabled* tells if a message in a canonical term is enabled for delivery, and *Fires* tells if a message is delivered in a given action. In the following, C is open.

$Enabled((\nu\tilde{x})C, \overline{u}\langle\tilde{v}\rangle) \Leftrightarrow$
$(C \simeq_{\mathcal{C}} C_1 \mid \overline{u}\langle\tilde{v}\rangle \ \wedge \ u \in extern((\nu\tilde{x})C)) \ \vee \ (C \simeq_{\mathcal{C}} C_1 \mid u(\tilde{w}).C_2 \mid \overline{u}\langle\tilde{v}\rangle \ \wedge \ len(\tilde{v}) = len(\tilde{w}))$

$Fires(f, i, \overline{x}\langle\tilde{y}\rangle) \Leftrightarrow \exists\tilde{z}.(f(i) = \mathtt{rcv}((\nu\tilde{z})\overline{x}\langle\tilde{y}\rangle) \ \vee \ f(i) = \mathtt{out}((\nu\tilde{z})\overline{x}\langle\tilde{y}\rangle))$

Definition 14 *Every annotation of a finite canonical path is fair. An annotation f of an infinite canonical path p is fair if*

$$\forall x, \tilde{y}, i < len(p).\ Enabled(src(p,i), \overline{x}\langle\tilde{y}\rangle) \Rightarrow$$
$$\exists j \geq i.\ Fires(f, j, \overline{x}\langle\tilde{y}\rangle) \vee \neg\exists k \geq j.\ Enabled(tgt(p,k), \overline{x}\langle\tilde{y}\rangle)$$

We are now ready to define fair paths in general.

Definition 15 *A path p is observably fair if there is a canonical path p' such that $p \simeq_{\mathcal{P}} p'$ and p' has a fair annotation.*

An infinite canonical path may have both fair and unfair annotations. A path is just the observable projection of an actual computation in a configuration. Therefore, several computations may project to the same path. For example, consider the path

$$Diverge\langle x\rangle \mid \overline{x}\langle y\rangle \mid \overline{x}\langle z\rangle \xrightarrow{\tau} Diverge\langle x\rangle \mid \overline{x}\langle y\rangle \mid \overline{x}\langle z\rangle \xrightarrow{\tau} \cdots$$

For this path, the annotation f where

$$f(i) = \begin{cases} \mathtt{rcv}(\overline{x}\langle y\rangle) & i \text{ is even} \\ \mathtt{rcv}(\overline{x}\langle z\rangle) & \text{otherwise} \end{cases}$$

is fair. But the annotation g where $g(i) = \mathtt{rcv}(\overline{x}\langle y\rangle)$ for all i, is unfair. Thus, our fairness notion is observational.

Chapter 5

Examples

In this chapter, we illustrate System A through some examples. In Section 5.1 we present an encoding of call-return communication. We use this communication pattern extensively in the other examples. In Chapter 3 we claimed that our calculus is powerful enough to encode data types such as booleans and naturals. In Section 5.2 we present an encoding of boolean values and operations. In Section 5.3 we present an encoding of natural numbers and some basic arithmetic operations. In Section 5.4 we show how abstract data types can be represented in our calculus. Specifically, we present an encoding of stack with push and pop operations.

We hope that the reader has had enough experience with the type system by now. We therefore leave it to the reader to verify that our encodings are well-typed.

Definition 16 *The relations* $\Longrightarrow$ *and* $\stackrel{s}{\Longrightarrow}$ *for any* $s \in \mathcal{A}^*$ *are defined as follows*

1. $C \Longrightarrow C'$ *means that there is a sequence of zero or more silent transitions* $C \stackrel{\tau}{\longrightarrow} \cdots \stackrel{\tau}{\longrightarrow} C'$.

2. *Let* s *be a sequence of actions* $\alpha_1 \cdots \alpha_n$. *Then* $C \stackrel{s}{\Longrightarrow} C'$ *means* $C \Longrightarrow C_1 \stackrel{\alpha_1}{\longrightarrow} C'_1 \cdots \Longrightarrow C_n \stackrel{\alpha_n}{\longrightarrow} C'_n \Longrightarrow C'$.

The following lemma is easy to prove.

Lemma 10 *If* $C_1 \Longrightarrow C_2$ *then*

1. $C \mid C_1 \Longrightarrow C \mid C_2$
2. $(\nu x)C_1 \Longrightarrow (\nu x)C_2$

Proof: By induction on the length of an expansion of $C_1 \Longrightarrow C_2$. $\Box$

Notation 2 *In order to define configurations parametric upon names, we use macro definitions such as* $B(\tilde{x}) \stackrel{\triangle}{=} C$*, where* $\tilde{x}$ *is a tuple of distinct names and contains all the free names in* C*. For every occurrence of* $B(\tilde{y})$ *we assume that* $\tilde{x}$ *and* $\tilde{y}$ *are of the same length. All occurrences of* $B(\tilde{y})$ *in preterms are implicitly replaced with* $C\{\tilde{y}/\tilde{x}\}$*. Macro definitions cannot be recursive.*

Remark: Of course, behavior definitions can be seen as macro definitions with additional constraints. In fact, we can do away with behavior definitions by allowing recursive macro definitions. But this would significantly complicate the type system; deriving the interface of a term would involve solving recursive equations over $P(\mathcal{N})$.

5.1 Call/Return Communication

A call/return communication involves a caller which sends a message to a callee and waits for the reply before initiating a dependent computation. This communication pattern can be encoded in actor systems by using the technique of passing continuations which is familiar in the context of functional programming. The caller creates a continuation (an actor) which embodies the dependent computation and passes the continuation's name to the callee. The continuation waits for the reply and then initiates the appropriate computation. Meanwhile, the caller is free to receive other messages (see Figure 5.1). For example, following is an actor x which receives a message with name y, and calls y.

$$x(y).(\nu z)(z(\tilde{u}).C \mid B\langle x, \tilde{v}\rangle \mid \overline{y}\langle \tilde{w}, z\rangle)$$

Here the actor z is the continuation. Unlike what is shown in Figure 5.1, the reply need not necessarily be from the callee. The callee may pass the continuation's name to other actors which may then reply. However, since the continuation is created with a fresh name it will not receive arbitrary messages; only the callee and other actors which receive the continuation's name may reply to it.

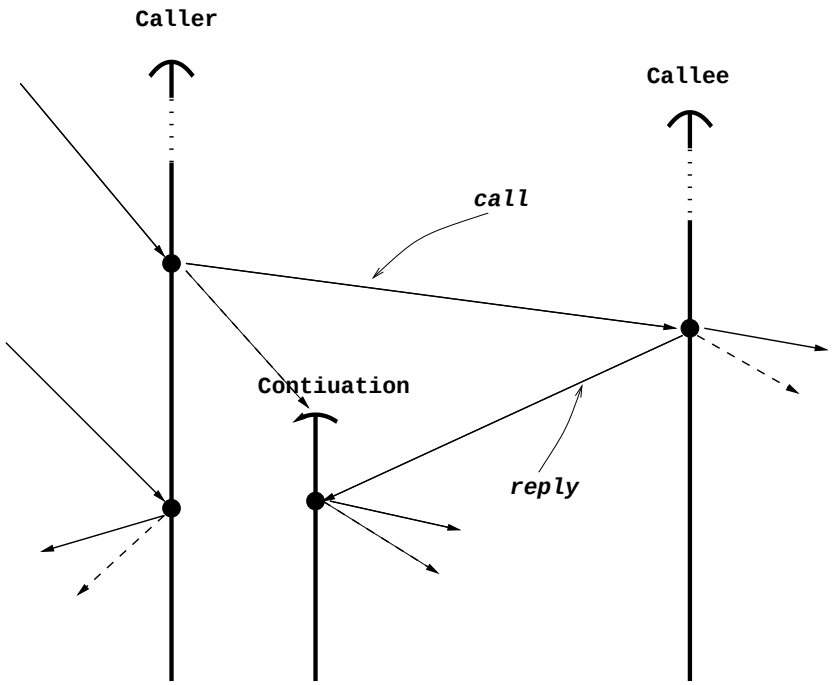

Figure 5.1: An event diagram illustrating call/return communication using continuations.

This technique of using continuations does not work if the caller itself has to receive the reply before it can process any other messages. One such scenario is where the reply determines the behavior with which the caller has to process future messages. However, because of persistence of actors the caller has to assume a replacement behavior immediately after it makes the call. Therefore, it could receive a message which is not necessarily the reply.

This situation can be handled by the concept of insensitive actors, also discussed in [1]. An insensitive actor buffers all communications until it receives a message with contents that satisfy a given condition. On receiving such a message it assumes an appropriate replacement behavior and resumes processing messages, which may be the buffered ones or freshly received.

Consider the following actor

$$x(\tilde{u}, y).(\overline{y}\langle \tilde{u}, x\rangle \mid x(\tilde{v}).C)$$

Suppose the intended behavior of x is to call y and wait for the reply before receiving any other messages. The above encoding does not represent this behavior because after sending the message to y, x is free to receive messages which need not be the reply. Using the concept of insensitive actors, the caller can be encoded as follows

$$\begin{aligned}
Caller(x, \tilde{z}) &\stackrel{\triangle}{=} x(\tilde{u}, y).(\nu p)(Proxy\langle p, x\rangle \mid \overline{y}\langle \tilde{u}, p\rangle \mid InsenseCaller\langle x, p, \tilde{w}\rangle) \\
&\qquad \texttt{where } \{\tilde{w}\} = fn(C) - \{x, \tilde{v}\},\ \ \{\tilde{z}\} = \{\tilde{w}\} - \{\tilde{u}, y\} \\
&\qquad\qquad p \notin \{x, y, \tilde{u}, \tilde{v}, \tilde{w}\} \\
Proxy &\stackrel{def}{=} (x, y)x(\tilde{v}).(\overline{y}\langle \tilde{v}, x\rangle \mid Proxy\langle x, y\rangle) \\
InsenseCaller &\stackrel{def}{=} (x, p, \tilde{w})x(\tilde{v}, u).[u = p](C,\ \overline{x}\langle \tilde{v}, u\rangle | InsenseCaller\langle x, p, \tilde{w}\rangle) \\
&\qquad \texttt{where } \ u \notin \{x, p, \tilde{w}, \tilde{v}\}
\end{aligned}$$

It is straight forward to verify that the definitions above are well typed. To check the definition for $InseneCaller$, using $C : [\{x\}, \chi]$ for some χ, the reader may verify that

$$x(\tilde{v}, u).[u = p](C,\ \overline{x}\langle u, \tilde{v}\rangle | InsenseCaller\langle x, p, \tilde{w}\rangle) : [\{x\}, \{p, \tilde{w}\}].$$

The caller creates a proxy and passes its name in the call. The proxy just waits for a reply, tags the reply with its name and forwards it to the caller. The caller meanwhile assumes an insensitive behavior, buffering all communications until it receives a message from the proxy (see Figure5.2). In fact, it is straightforward to show that if

$$InsenseCaller\langle x, p, \tilde{w}\rangle \mid \overline{x}\langle \tilde{v'}, u'\rangle \xrightarrow{\tau} C'$$

then

$$C' \simeq_{\mathcal{C}} \begin{cases} C\{\tilde{v'}/\tilde{v}\} & \text{if } u' = p \\ InsenseCaller\langle x, p, \tilde{w}\rangle \mid \overline{x}\langle \tilde{v'}, u'\rangle & \text{otherwise} \end{cases}$$

Note that, since the proxy is created with the fresh name the caller will not accidentally receive a message which is not a reply but contains the proxy's name.

Following is an example which illustrates these ideas. Let

$$Callee \stackrel{def}{=} (x)x(\tilde{u}, p).(\overline{p}\langle \tilde{u}\rangle \mid Callee\langle x\rangle)$$

Assuming $len(\tilde{u}) = len(\tilde{v}) = len(\tilde{u'}) = len(\tilde{v'})$, and $\sigma = \{\tilde{u'}/\tilde{u}\}$, following is an annotated canonical path for a possible computation in $Caller(x, \tilde{z}) \mid Callee\langle y\rangle \mid \overline{x}\langle \tilde{u'}, y\rangle$.

$Caller(x, \tilde{z}) \mid Callee\langle y\rangle \mid \overline{x}\langle \tilde{u'}, y\rangle$

$\xrightarrow{\texttt{rcv}(\overline{x}\langle \tilde{u'}, y\rangle)}$ $(\nu p)(InsenseCaller\langle x, p, \sigma(\tilde{w})\rangle \mid \overline{y}\langle \tilde{u'}, p\rangle \mid Callee\langle y\rangle \mid Proxy\langle p, x\rangle)$

$\xrightarrow{\texttt{in}(\overline{x}\langle \tilde{v'}, p'\rangle)}$ $(\nu p)(InsenseCaller\langle x, p, \sigma(\tilde{w})\rangle \mid \overline{y}\langle \tilde{u'}, p\rangle \mid Callee\langle y\rangle \mid Proxy\langle p, x\rangle \mid \overline{x}\langle \tilde{v'}, p'\rangle)$

$\xrightarrow{\texttt{rcv}(\overline{x}\langle \tilde{v'}, p'\rangle)}$ $(\nu p)(InsenseCaller\langle x, p, \sigma(\tilde{w})\rangle \mid \overline{y}\langle \tilde{u'}, p\rangle \mid Callee\langle y\rangle \mid Proxy\langle p, x\rangle \mid \overline{x}\langle \tilde{v'}, p'\rangle)$

$\xrightarrow{\texttt{rcv}(\overline{y}\langle \tilde{u'}, p\rangle)}$ $(\nu p)(InsenseCaller\langle x, p, \sigma(\tilde{w})\rangle \mid \overline{p}\langle \tilde{u'}\rangle \mid Callee\langle y\rangle \mid Proxy\langle p, x\rangle \mid \overline{x}\langle \tilde{v'}, p'\rangle)$

$\xrightarrow{\texttt{rcv}(\overline{p}\langle \tilde{u'}\rangle)}$ $(\nu p)(InsenseCaller\langle x, p, \sigma(\tilde{w})\rangle \mid \overline{x}\langle \tilde{u'}, p\rangle \mid Callee\langle y\rangle \mid Proxy\langle p, x\rangle \mid \overline{x}\langle \tilde{v'}, p'\rangle)$

$\xrightarrow{\texttt{rcv}(\overline{x}\langle \tilde{u'}\rangle)}$ $(\nu p)(C\{\sigma(\tilde{w})/\tilde{w}\}\{\tilde{u'}/\tilde{v}\} \mid Callee\langle y\rangle \mid Proxy\langle p, x\rangle \mid \overline{x}\langle \tilde{v'}, p'\rangle)$

The event diagram for this computation is shown in Figure 5.2.

We will use the above encoding of call/return communication in some of the examples we present in this chapter. To simplify the presentation we introduce the following shorthands.

1. We write $x(\tilde{v}) : p.C$, where $p \notin \{x, \tilde{v}\} \cup fn(C)$, to mean $B\langle x, p, \tilde{w}\rangle$, where B is a fresh behavior identifier, $\{\tilde{w}\} = fn(C) - \{x, \tilde{v}\}$, and

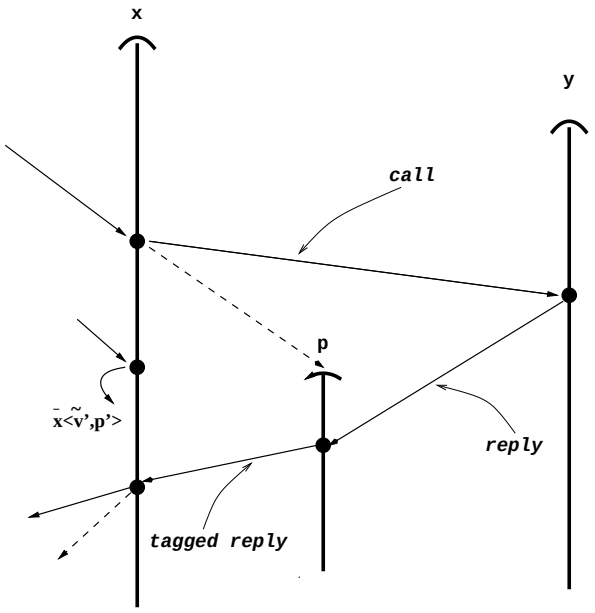

Figure 5.2: An event diagram illustrating call/return communication using insensitive behaviors. Note that the caller x buffers the message $\overline{x}\langle\tilde{v'}, p'\rangle$ until it receives the reply to its call.

$$B \stackrel{def}{=} (x, p, \tilde{w})x(\tilde{v}, u).[u = p](C,\ \overline{x}\langle\tilde{v}, u\rangle | B\langle x, p, \tilde{w}\rangle) \quad \text{where} \ \ u \notin \{x, p, \tilde{v}, \tilde{w}\}$$

2. For the proxies used in call return communication, we assume a family of behavior definitions indexed by naturals.

$$Proxy_i \stackrel{def}{=} (x, y)x(\tilde{u}).(\overline{y}\langle\tilde{u}, x\rangle \mid Proxy_i\langle x, y\rangle) \quad \text{where } len(\tilde{u}) = i$$

5.2 Booleans

We encode booleans as configurations with a single actor which is also a receptionist. In the following, $\underline{T}$ defines the behavior of the receptionist in the encoding of *true*, and $\underline{F}$ for *false*.

$$\underline{T} \stackrel{def}{=} (x)x(c, y_1, y_2).(\overline{c}\langle y_1\rangle \mid \underline{T}\langle x\rangle)$$
$$\underline{F} \stackrel{def}{=} (x)x(c, y_1, y_2).(\overline{c}\langle y_2\rangle \mid \underline{F}\langle x\rangle)$$

Both the behaviors accept messages containing three names. The first name is assumed to be the customer name. The behavior $\underline{T}$ replies back to the customer with the second name, while $\underline{F}$ replies back with the third name. The actor $\underline{T}\langle x\rangle$ can be thought of as the value *true* available at name x. Note that an actor with any of these behaviors never changes its behavior. As a consequence, the values we have defined are persistent; they do not disappear after being read once.

The negation function can be encoded as a configuration with multiple actors and a single receptionist as follows.

$$Not(x) \triangleq (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_1\langle p, x\rangle)$$

where

$$\begin{aligned}
B_1 &\stackrel{def}{=} (x)x(c).(\nu u)(\underline{F}\langle u\rangle \mid \overline{c}\langle u\rangle \mid B_1\langle x\rangle)\\
B_2 &\stackrel{def}{=} (x)x(c).(\nu u)(\underline{T}\langle u\rangle \mid \overline{c}\langle u\rangle \mid B_2\langle x\rangle)\\
B_3 &\stackrel{def}{=} (x, y, z, p)x(v, c).(\overline{v}\langle p, y, z\rangle \mid B_3'\langle x, y, z, p, c\rangle)\\
B_3' &\stackrel{def}{=} (x, y, z, p, c)x(v) : p.(\overline{v}\langle c\rangle \mid B_3\langle x, y, z, p\rangle)
\end{aligned}$$

$Not(x)$ can be thought of as the function *not* available at name x. Evaluation of the function is initiated by sending a message containing a value and a customer to x. The customer eventually receives the negation of the value sent.

It is illuminating to see the following correspondence between this encoding and the psuedocode below

```
def  Not(v, c)
     if (v = T)          :   B_3⟨x, y, z, p⟩ | Proxy_1⟨p, x⟩
        send(c, F)       :   B_1⟨y⟩
     else
        send(c, T)       :   B_2⟨z⟩
```

The testing of condition is encoded as a call return communication. The commands are encoded as actors; execution of these commands corresponds to delivery of messages to these actors. The environment in which these commands are executed is specified in the message delivered.

As an illustration we show the following

$$Not(x) \mid \underline{F}\langle u\rangle \mid \overline{x}\langle u, c\rangle \stackrel{\mathtt{out}((\nu v)\overline{c}\langle v\rangle)}{\Longrightarrow} Not(x) \mid \underline{F}\langle u\rangle \mid \underline{T}\langle v\rangle$$

The following computation path is one possible expansion of the above

$$\begin{aligned}
&Not(x) \mid \underline{F}\langle u\rangle \mid \overline{x}\langle u, c\rangle \\
&\quad\equiv\quad (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid B_3\langle x, y, z, p\rangle \mid \overline{x}\langle u, c\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{F}\langle u\rangle) \\
&\quad\stackrel{\tau}{\longrightarrow} (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid B_3'\langle x, y, z, p, c\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{F}\langle u\rangle \mid \overline{u}\langle p, y, z\rangle) \\
&\quad\stackrel{\tau}{\longrightarrow} (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid B_3'\langle x, y, z, p, c\rangle \mid Proxy_1\langle p, x\rangle \mid \overline{p}\langle z\rangle \mid \underline{F}\langle u\rangle) \\
&\quad\stackrel{\tau}{\longrightarrow} (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid B_3'\langle x, y, z, p, c\rangle \mid \overline{x}\langle z, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{F}\langle u\rangle) \\
&\quad\stackrel{\tau}{\longrightarrow} (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid \overline{z}\langle c\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{F}\langle u\rangle) \\
&\quad\stackrel{\tau}{\longrightarrow} (\nu y, z, p, v)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid \underline{T}\langle v\rangle \mid \overline{c}\langle v\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{F}\langle u\rangle) \\
&\quad\stackrel{\mathtt{out}((\nu v)\overline{c}\langle v\rangle)}{\longrightarrow} (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid \underline{T}\langle v\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{F}\langle u\rangle) \\
&\quad\equiv\quad Not(x) \mid \underline{F}\langle u\rangle \mid \underline{T}\langle v\rangle
\end{aligned}$$

Following is the encoding of boolean *and*

$$\begin{aligned}
And(x) \stackrel{\triangle}{=}\ & (y, z, u, p_1, p_2)(B_1\langle y\rangle \mid B_2\langle z\rangle \mid B_3\langle u, z, y, p_1\rangle \mid B_4\langle x, y, u, p_2\rangle \mid \\
& Proxy_1\langle p_1, u\rangle \mid Proxy_1\langle p_2, x\rangle)
\end{aligned}$$

where

$$\begin{aligned}
B_4 &\stackrel{def}{=} (x, y, z, p)x(v_1, v_2, p).(\overline{v_1}\langle z, y\rangle \mid B_4'\langle x, y, z, p, v_2, c\rangle) \\
B_4' &\stackrel{def}{=} (x, y, z, p, v, c)x(w) : p.([w = y](\overline{w}\langle c\rangle,\ \overline{w}\langle v, c\rangle) \mid B_4\langle x, y, z, p\rangle)
\end{aligned}$$

The ideas behind this encoding are similar to those used for negation. The reader may verify the following

$$And(x) \mid \underline{T}\langle u\rangle \mid \underline{F}\langle v\rangle \mid \overline{x}\langle u,v,c\rangle \stackrel{\mathtt{out}((\nu w)\overline{c}\langle w\rangle)}{\Longrightarrow} And(x) \mid \underline{T}\langle u\rangle \mid \underline{F}\langle v\rangle \mid \underline{F}\langle w\rangle$$

The computation above involves 9 transitions with silent actions followed by the output.

5.3 Natural Numbers

Natural numbers can be built from the constructors 0 and S. Accordingly, we define the following two behaviors.

$$\begin{aligned} Zero &\stackrel{def}{=} (x)x(c,u_1,u_2).(\overline{c}\langle u_1,x\rangle \mid Zero\langle x\rangle) \\ Succ &\stackrel{def}{=} (x,y)x(c,u_1,u_2).(\overline{c}\langle u_2,y\rangle \mid Succ\langle x,y\rangle) \end{aligned}$$

As in the case of booleans, we encode natural numbers as configurations with a single receptionist.

$$\begin{aligned} \underline{0}(x) &\triangleq Zero\langle x\rangle \\ \underline{S^{n+1}0}(x) &\triangleq (\nu y)(Succ\langle x,y\rangle \mid \underline{S^n 0}(y)) \end{aligned}$$

The number $S^n 0$ is encoded as a sequence of $n+1$ actors each pointing to the next (the last one points to itself). The first n actors have the behavior $Succ$ and the last one has behavior $Zero$. Only the first actor is the receptionist to the entire configuration. As in our encoding for booleans, both the behaviors accept messages with three names, the first of which is assumed to denote the customer. The behavior $Succ$ replies back to the customer with the third name and the name of next actor in the sequence, while $Zero$ replies back with the second name and its own name.

We now encode addition of natural numbers. Our aim is to define a configuration $Add(x)$ with a single receptionist x such that the following holds

$$Add(x) \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u,v,c\rangle \stackrel{\mathtt{out}((\nu w)\overline{c}\langle w\rangle)}{\Longrightarrow} Add(x) \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \underline{S^{n+m} 0}(w)$$

$Add(x)$ may be thought of as the addition function available at name x.

We first define $AddTo(x)$ such that

$$AddTo(x) \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^m 0}(v) \\ \Longrightarrow \quad AddTo(x) \mid (\nu u)(\underline{S^{n+m} 0}(u) \mid \overline{c}\langle u\rangle) \mid \underline{S^m 0}(v)$$

We will then use $AddTo(x)$ to define $Add(x)$.

$$AddTo(x) \stackrel{\triangle}{=} (\nu y, z, p)(B_1\langle y\rangle \mid B_2\langle z, x\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_2\langle p, x\rangle)$$

where

$$\begin{aligned} B_1 &\stackrel{def}{=} (x)x(v_1, v_2, c).(\overline{c}\langle v_1\rangle \mid B_1\langle x\rangle) \\ B_2 &\stackrel{def}{=} (x, y)x(v_1, v_2, c).(\nu u)(Succ\langle u, v_1\rangle \mid \overline{y}\langle u, v_2, c\rangle \mid B_2\langle x, y\rangle) \\ B_3 &\stackrel{def}{=} (x, y, z, p)x(v_1, v_2, c).(\overline{v_2}\langle p, y, z\rangle \mid B_3'\langle x, y, z, p, v_1, c\rangle) \\ B_3' &\stackrel{def}{=} (x, y, z, p, v, c)x(u, w) : p.(\overline{u}\langle v, w, c\rangle \mid B_3\langle x, y, z, p\rangle) \end{aligned}$$

The relation between this encoding and the pseudo code below is illuminating.

def AddTo(v_1, v_2, c)
  if $(v_2 = 0)$ **then** : $B_3\langle x, y, z, p\rangle \mid Proxy_2\langle p, x\rangle$
    send(v_1, c) : $B_1\langle y\rangle$
  else
    AddTo$(v_1 + 1, v_2 - 1, c)$: $B_2\langle z, x\rangle$

Lemma 11 $AddTo(x) \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^m 0}(v) \\ \stackrel{\tau}{\Longrightarrow} \quad AddTo(x) \mid (\nu u)(\underline{S^{n+m} 0}(u) \mid \overline{c}\langle u\rangle) \mid \underline{S^m 0}(v)$

Proof: We prove this by induction on m.

Base Case: The reader may verify that

$$AddTo(x) \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{0}(v) \stackrel{\tau}{\longrightarrow}{}^5 AddTo(x) \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{c}\langle u\rangle) \mid \underline{0}(v)$$

Induction Step: Assuming the given proposition we show

$$AddTo(x) \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^{m+1}0}(v) \Longrightarrow AddTo(x) \mid (\nu u)(\underline{S^{n+m+1}0}(u) \mid \overline{c}\langle u\rangle) \mid \underline{S^{m+1}0}(v)$$

$$\begin{aligned}
&AddTo(x) \mid (\nu u)(\underline{S^n 0}(u) \mid \overline{x}\langle u, v, c\rangle) \mid \underline{S^{m+1}0}(v) \\
&\quad \equiv \quad (\nu y, z, p, u)(B_1\langle y\rangle \mid B_2\langle z, x\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_2\langle p, x\rangle \mid \underline{S^n 0}(u) \mid \\
&\qquad\qquad \underline{S^{m+1}0}(v) \mid \overline{x}\langle u, v, c\rangle) \\
&\quad \xrightarrow{\tau}{}^{5} \quad (\nu y, z, p, u, w)(B_1\langle y\rangle \mid B_2\langle z, x\rangle \mid B_3\langle x, y, z, p\rangle \mid Proxy_2\langle p, x\rangle \mid \underline{S^{n+1}0}(u) \mid \\
&\qquad\qquad Succ\langle v, w\rangle \mid \underline{S^m 0}(w) \mid \overline{x}\langle u, w, c\rangle) \\
&\quad \equiv \quad (\nu w)(Succ\langle v, w\rangle \mid AddTo(x) \mid (\nu u)(\underline{S^{n+1}0}(u) \mid \overline{x}\langle u, w, c\rangle) \mid \underline{S^m 0}(w)) \\
&\quad \stackrel{\tau}{\Longrightarrow} \quad (\nu w)(Succ\langle v, w\rangle \mid AddTo(x) \mid (\nu u)(\underline{S^{n+m+1}0}(u) \mid \overline{c}\langle u\rangle) \mid \underline{S^m 0}(w)) \\
&\qquad\qquad \text{by induction hypothesis and Lemma 10} \\
&\quad \equiv \quad AddTo(x) \mid (\nu u)(\underline{S^{n+m+1}0}(u) \mid \overline{c}\langle u\rangle) \mid \underline{S^{m+1}0}(v) \qquad \square
\end{aligned}$$

We are now ready to define $Add(x)$.

$$Add(x) \stackrel{\triangle}{=} (\nu y, p)(B_4\langle x, y, p\rangle \mid AddTo(y) \mid Proxy_1\langle p, x\rangle)$$

where

$$\begin{aligned}
B_4 &\stackrel{def}{=} (x, y, p)x(v_1, v_2, c).(\nu u)(\underline{0}(u) \mid \overline{y}\langle u, v_1, p\rangle \mid B_4'\langle x, y, p, v_2, c\rangle) \\
B_4' &\stackrel{def}{=} (x, y, p, v, c)x(w) : p.(\overline{y}\langle w, v, c\rangle \mid B_4\langle x, y, p\rangle)
\end{aligned}$$

Lemma 12 $Add(x) \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u, v, c\rangle \stackrel{\mathtt{out}((\nu w)\overline{c}\langle w\rangle)}{\Longrightarrow} Add(x) \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \underline{S^{n+m} 0}(w)$

Proof:

$$\begin{aligned}
&Add(x) \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u, v, c\rangle \\
&\quad \equiv \quad (\nu y, p)(B_4\langle x, y, p\rangle \mid AddTo(y) \mid Proxy_1\langle p, x\rangle \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \overline{x}\langle u, v, c\rangle) \\
&\quad \xrightarrow{\tau} \quad (\nu y, p, w)(\underline{0}(w) \mid \overline{y}\langle w, u, p\rangle \mid B_4'\langle x, y, p, v, c\rangle \mid AddTo(y) \mid Proxy_1\langle p, x\rangle \mid
\end{aligned}$$

$$
\begin{aligned}
& && \underline{S^n 0}(u) \mid \underline{S^m 0}(v)) \\
\equiv & && (\nu y, p)(B_4'\langle x, y, p, v, c\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{S^m 0}(v) \mid AddTo(y) \mid \\
& && (\nu w)(\underline{0}(w) \mid \overline{y}\langle w, u, p\rangle) \mid \underline{S^n 0}(u)) \\
\stackrel{\tau}{\Longrightarrow} & && (\nu y, p)(B_4'\langle x, y, p, v, c\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{S^m 0}(v) \mid AddTo(y) \mid \\
& && (\nu w)(\underline{S^n 0}(w) \mid \overline{p}\langle w\rangle) \mid \underline{S^n 0}(u)) \\
& && \qquad \text{by Lemma 11 and Lemma 10} \\
\stackrel{\tau}{\longrightarrow}^2 & && (\nu y, p)(B_4\langle x, y, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{S^n 0}(u) \mid AddTo(y) \mid \\
& && (\nu w)(\underline{S^n 0}(w) \mid \overline{y}\langle w, v, c\rangle) \mid \underline{S^m 0}(v)) \\
\stackrel{\tau}{\Longrightarrow} & && (\nu y, p)(B_4\langle x, y, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{S^n 0}(u) \mid AddTo(y) \mid \\
& && (\nu w)(\underline{S^{n+m} 0}(w) \mid \overline{c}\langle w\rangle) \mid \underline{S^m 0}(v)) \\
& && \qquad \text{by Lemma 11 and Lemma 10} \\
\stackrel{\mathtt{out}((\nu w)\overline{c}\langle w\rangle)}{\longrightarrow} & && (\nu y, p)(B_4\langle x, y, p\rangle \mid Proxy_1\langle p, x\rangle \mid \underline{S^n 0}(u) \mid AddTo(y) \mid \\
& && \underline{S^{n+m} 0}(w) \mid \underline{S^m 0}(v)) \\
\equiv & && Add(x) \mid \underline{S^n 0}(u) \mid \underline{S^m 0}(v) \mid \underline{S^{n+m} 0}(w) \qquad \square
\end{aligned}
$$

We hope the reader is convinced that our calculus is powerful enough to encode other arithmetic operations.

5.4 Stack

In this section we show how complex data types can be encoded in System A. Specifically, we present an encoding of a stack with *push* and *pop* operations. This example has been adapted from [1].

We implement a stack as a linked list of actors. Each actor in the list, called a node, has two acquaintances: a content and the next node in the list. When a node receives a push request it creates a new node with its content and link, and changes its own acquaintances to the value pushed and the node created. When a node (other than the last one) receives a pop request it replies back to the specified customer with its content and forwarders all future requests to its link. Following definitions capture these behaviors of a node

$$Cell \stackrel{def}{=} (x, v, l, PUSH, BOT)\ x(arg, op).$$
$$[op = PUSH]((\nu u)(Cell\langle u, v, l, PUSH, BOT\rangle \mid Cell\langle x, arg, u, PUSH, BOT\rangle),$$
$$\overline{arg}\langle v\rangle \mid [v = BOT](Cell\langle x, v, l, PUSH, BOT\rangle, Forwarder\langle x, l\rangle))$$

$$Forwarder \stackrel{def}{=} (x, l)x(arg, op).(\overline{l}\langle arg, op\rangle \mid Forwarder\langle x, l\rangle)$$

A special name BOT is used as the bottom of stack marker. The last node in the list has this name as its content and link.

A stack is defined as follows

$$Stack(PUSH, POP, BOT) \stackrel{\triangle}{=} (\nu x)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid$$
$$Cell\langle x, BOT, BOT, PUSH, BOT\rangle)$$

A stack instance is a configuration with two receptionists, one for each supported operation. Both the receptionists are proxies which forward tagged requests to the first node of an internal list such as the one described above. Names of other internal actors (list nodes) are never exported to the environment. We thus have total containment of data and operations valid on it.

Note that the arrival orders (see Section 2.2) at the receptionists are in no way related to the order in which requests are "processed" at the internal list. In fact, the arrival order(s) at the first node in the list which is not a forwarder, is the order in which request are processed[1].

Following is an example interaction with a stack. Figure 5.3 illustrates this computation.

$$Stack(PUSH, POP, BOT)$$
$$\stackrel{\mathtt{in}(\overline{PUSH}\langle v_1\rangle)}{\longrightarrow} (\nu x)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid$$
$$Cell\langle x, BOT, BOT, PUSH, BOT\rangle \mid \overline{PUSH}\langle v_1\rangle)$$
$$\stackrel{\tau}{\longrightarrow}^2 \quad (\nu x, y)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid$$

[1] A formal specification of a stack and proof of correctness of our implementation are beyond the scope of this dissertation. Work in this direction is in progress

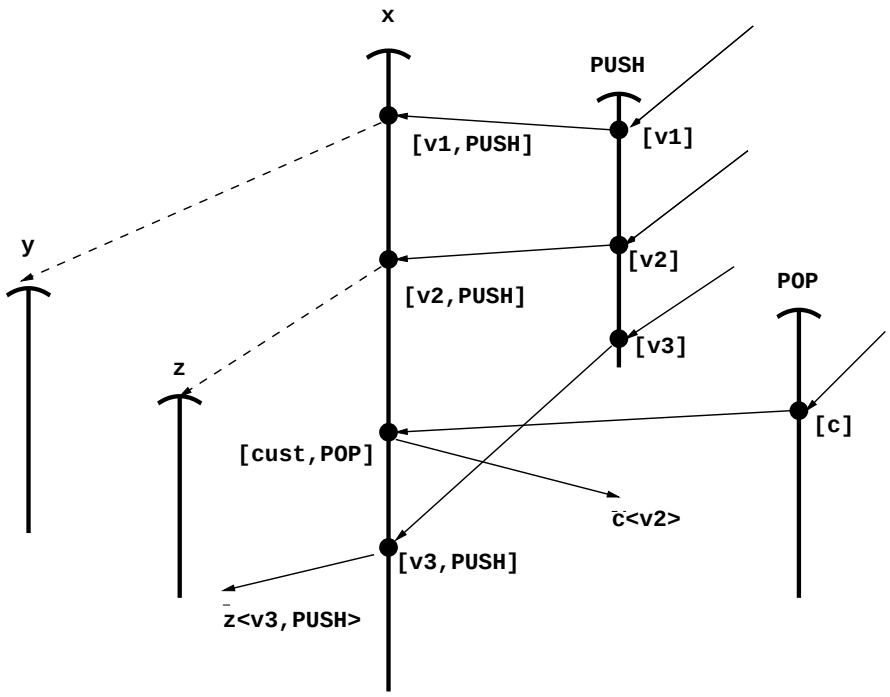

Figure 5.3: An event diagram depicting a computation in our stack implementation. For the purpose of illustration, events are annotated with corresponding message contents.

$$
\begin{aligned}
& Cell\langle x, v_1, y, PUSH, BOT\rangle \mid Cell\langle y, BOT, BOT, PUSH, BOT\rangle) \\
\xrightarrow{\mathtt{in}(\overline{PUSH}\langle v_2\rangle)} \; & (\nu x, y)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid \\
& Cell\langle x, v_1, y, PUSH, BOT\rangle \mid Cell\langle y, BOT, BOT, PUSH, BOT\rangle \mid \\
& \overline{PUSH}\langle v_2\rangle) \\
\xrightarrow{\tau}{}^{2} \; & (\nu x, y, z)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid \\
& Cell\langle x, v_2, z, PUSH, BOT\rangle \mid Cell\langle z, v_1, y, PUSH, BOT\rangle \mid \\
& Cell\langle y, BOT, BOT, PUSH, BOT\rangle) \\
\xrightarrow{\mathtt{in}(\overline{POP}\langle c\rangle)} \; & (\nu x, y, z)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid \\
& Cell\langle x, v_2, z, PUSH, BOT\rangle \mid Cell\langle z, v_1, y, PUSH, BOT\rangle \mid \\
& Cell\langle y, BOT, BOT, PUSH, BOT\rangle \mid \overline{POP}\langle c\rangle) \\
\xrightarrow{\tau}{}^{2} \; & (\nu x, y, z)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid
\end{aligned}
$$

$$
\begin{aligned}
& Forwarder\langle x, z\rangle \mid Cell\langle z, v_1, y, PUSH, BOT\rangle \mid \\
& Cell\langle y, BOT, BOT, PUSH, BOT\rangle \mid \overline{c}\langle v_2\rangle) \\
\xrightarrow{\texttt{out}(\overline{c}\langle v_2\rangle)} \quad & (\nu x, y, z)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid \\
& Forwarder\langle x, z\rangle \mid Cell\langle z, v_1, y, PUSH, BOT\rangle \mid \\
& Cell\langle y, BOT, BOT, PUSH, BOT\rangle) \\
\xrightarrow{\texttt{in}(\overline{PUSH}\langle v_3\rangle)} \quad & (\nu x, y, z)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid \\
& Forwarder\langle x, z\rangle \mid Cell\langle z, v_1, y, PUSH, BOT\rangle \mid \\
& Cell\langle y, BOT, BOT, PUSH, BOT\rangle \mid \overline{PUSH}\langle v_3\rangle) \\
\xrightarrow{\tau}{}^{2} \quad & (\nu x, y, z)(Proxy_1\langle PUSH, x\rangle \mid Proxy_1\langle POP, x\rangle \mid \\
& Forwarder\langle x, z\rangle \mid Cell\langle z, v_1, y, PUSH, BOT\rangle \mid \\
& Cell\langle y, BOT, BOT, PUSH, BOT\rangle \mid \overline{z}\langle v_3, PUSH\rangle)
\end{aligned}
$$

Chapter 6

Conclusion

We have so far just laid the foundation for further semantical investigations on the Actor Model. We indicate a few prominent research directions, some of which are already being pursued.

We are currently investigating testing equivalences [18] in System A. The type system we impose cuts down the number of legal experiments (observing contexts) that may be performed on (composed with) a given configuration. The fairness constraint cuts down the number of computations that an experiment may engage the configuration in. These together give us coarser equivalences which are relevant in practice. For example, the uniqueness and persistence properties together help hide details of internal implementation by ensuring that a context cannot observe internal communications in the configuration being tested. Similarly, fairness ensures that collecting active garbage such as that described in Section 2.1.4 does not alter semantics.

In [3], testing equivalences are investigated in an actor-based concurrent extension of a functional language. It is shown that in the presence of fairness the three forms of equivalence, namely, convex, must, and may equivalences, collapse to two. It remains to be seen if such a collapse also occurs in our system.

We are also studying denotational models for our calculus such as the interaction paths and event diagrams [51]. Specific questions of interest are if these models are composi-

tional in our setting, and how the equivalences based on these relate to testing equivalences. Answers to these questions could give direct characterizations of testing equivalences without quantification on all observing contexts, and facilitate the development of proof systems (induced algebras) to establish equivalences.

Numerous equivalences have been investigated on asynchronous variants of π-calculus. We are particularly interested in the notions of asynchronous bisimulation [4] and barbed congruence [29]. We would like to investigate how these notions and their algebraic theories map to System A.

System A could serve as a meta-language for interpretation of concurrent object oriented languages. Such interpretations will be useful in validating program transformations such as those suggested in [23]. Of immediate interest is a translational semantics of the actor-based language presented in [3], and validation of its equational laws.

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 1996.

[4] R. Amadio, I.Castellani, and D. Sangiorgi. On Bisimulations for Asynchronous π-Calculus. In *Proceedings of CONCUR '96*. Springer-Verlag, 1996. LNCS 1119.

[5] M. Astley and G. Agha. Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management. In *Sixth International Symposium on the Foundations of Software Engineering*, November 1998.

[6] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[7] H. G. Baker and C. Hewitt. Laws for Communicating Parallel Processes. *IFIP Congress*, pages 987–992, August 1977.

[8] G. Berry and G. Boudol. Chemical abstract machine. *Proc of* 17^{th} *Annual Symposium on Principles of Programming Languages*, 1990.

[9] Michele Boreale and D. Sangiorgi. Bisimulation in Name Passing Calculi without Matching. *Proceedings of LICS*, 1998.

[10] G. Boudol. Asynchrony and the pi-calculus. Technical Report 1702, Department of Computer Science, Inria Univeristy, May 1992.

[11] C. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[12] W.D. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusettes Institute of Technology, AI Laboratory, 1981.

[13] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join Calculus. *Proceedings of POPL*, 1996.

[14] S. Frolund. Inheritence of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In O.L. Madsen, editor, *European Conference on Object-Oriented Programming*, pages 185–196, 1992. LNCS 615.

[15] S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach for Synchronization*. MIT Press, November 1996.

[16] M. Gaspari and G. Zavattaro. An Algebra of Actors. Technical Report UBLCS-97-4, Department of Computer Science, Univeristy of Bologna (Italy), May 1997.

[17] I. Greif. Semantics of Communicating Parallel Processes. Technical Report 154, MIT, Project MAC, 1975.

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

[19] 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.

[20] C. Hewitt. Viewing Control Structures as Patterns of Message Passing. *Journal of Artificial Intelligence*, 8(3):323–364, September 1977.

[21] C.A.R. Hoare. *Communication Sequential Processes*. Prentice Hall, 1985.

[22] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Fifth European Conference on Object-Oriented Programming*, July 1991. LNCS 512, 1991.

[23] C.B. Jones. Constraining Interference in an Object-Based Design Menthod. In M.C. Gaudel and J.P. Jouannaud, editors, *Proc. TAPSOFT'93*, pages 136–150. Springer Verlag, 1993. LNCS 668.

[24] W. Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana Champaign, 1997.

[25] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *SuperComputing*, 1996.

[26] A.R. Krzysztof and E.R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer Verlag, 1991.

[27] H. Lieberman. Concurrent Object-Oriented Programming in Act1. In *Object Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[28] C. Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, 1987.

[29] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In *Proceeding of ICALP '98*. Springer-Verlag, 1998. LNCS 1443.

[30] J. Meseguer. Rewriting Logic as a Unified Model of Concurrency. Technical Report SRI-CSI-90-02, SRI International, Computer Science Laboratory, February 1990.

[31] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[32] R. Milner. The Polyadic π-calculus: a Tutorial. In F.L. Hamer, W.Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer-Verlag, 1993.

[33] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[34] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, Part I. Technical Report ECS-LFCS-89-85, Department of Computer Science, Edinburgh University, June 1989.

[35] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, Part II. Technical Report ECS-LFCS-89-86, Department of Computer Science, Edinburgh University, June 1989.

[36] R. Panwar, W. Kim, and G. Agha. Efficient Compilation of Call/Return Communication for Actor-Based Programming Languages. In *High Performance Parallel Computing*, 1996.

[37] R. Panwar, W. Kim, and G. Agha. Parallel Implementations of Irregular Problems Using High-Level Actor Language. In *International Parallel Processing Symposium*, August 1996.

[38] J. Parrow. Mismatching and Early Equivalence. In *Pi-Calculus Note JP13, Manuscript*, 1990.

[39] J. Parrow. An introduction to pi-calculus. In Bergstra, Ponce, and Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000.

[40] J. Parrow and D. Sangiorgi. Algebraic Theories of Name-Passing Calculi. *Information and Computation*, 120(2), 1995.

[41] B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[42] B. C. Pierce and D. N. Turner. Pict: A programming Language Based on the Pi-Calculus. Technical Report CSCI-476, Indiana University, March 1997.

[43] S. Ren, G. Agha, and M. Saito. A Modular Approach for Programming Distributed Real-Time Systems. *Journal of Parallel and Distributed Computing*, 1996.

[44] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Edinburgh University, 1992.

[45] D. Sangiorgi. Typed Pi-Calculus at Work: A Proof of Jone's Transformation on Concurrent Objects. *Theory and Practice of Object-Oriented Systems*, 1997.

[46] D. Sangiorgi. An Interpretation of Typed Objects into Typed Pi-Calculus. *Information and Computation*, 143(1), 1998.

[47] D. Sangiorgi. The Name Discipline of Uniform Receptiveness. *Theoretical Computer Science*, 221, 1999.

[48] D.C. Sturman. *Modular Specification of Interaction in Distributed Computing*. PhD thesis, University of Illinois at Urbana Champaign, 1996.

[49] C. Talcott. An Actor Rewriting Theory. In *Electronic Notes in Theoretical Computer Science 5*, 1996.

[50] C. Talcott. Interaction Semantics for Components of Distributed Systems. In E.Najm and J.B. Stefani, editors, *Formal Methods for Open Object Based Distributed Systems*. Chapman & Hall, 1996.

[51] C. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3), 1998.

[52] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An Object-Oriented Concurrent System Architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[53] C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In *Third International Conference on Coordination Models and Languages (COORDINATION '99)*, pages 166–182, 1999. LNCS 1594.

[54] N. Venkatasubramanian and C. Talcott. Meta-Architectures for Resource Management in Open Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, August 1995.

[55] D. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.

[56] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.