A Foundation for Actor Computation

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

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

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

Carolyn L. Talcott
Stanford University clt@sail.stanford.edu

Abstract

We present an actor language which is an extension of a simple functional language, and provide a precise operational semantics for this extension. Actor configurations represent open distributed systems, by which we mean that the specification of an actor system explicitly takes into account the interface with external components. We study the compositability of such systems. We define and study various notions of testing equivalence on actor expressions and configurations. The model we develop provides fairness. An important result is that the three forms of equivalence, namely, convex, must, and may equivalences, collapse to two in the presence of fairness. We further develop methods for proving laws of equivalence and provide example proofs to illustrate our methodology.

Contents

1 Introduction 2
   1.1 Overview 3
   1.2 Related Work 4

2 Actor Language Constructs 7
   2.1 Trivial Examples 8
   2.2 Actor Cells 9
   2.3 Join Continuations 9
   2.4 Notation 12

3 A Simple Lambda-Based Actor Language 12
   3.1 Syntax 13
   3.2 Reduction Semantics for Actor Configurations 13
   3.3 Composition of Actor Configurations 19

4 Equivalence of Expressions 20
   4.1 Events 20
   4.2 Three Equivalences 21
1 Introduction

The modern computing environment is becoming increasingly open and distributed. Research on semantics to support reasoning about components and configuration in open distributed systems is in its early stages. The main characteristics of an open distributed system are that such systems allow the addition of new components, the replacement of existing components, and changes in interconnections between components, largely without disturbing the functioning of the system. Open distributed systems require a discipline in which a component may not have any direct control over other components with which it is connected. Instead, the behavior of a component is locally determined by its initial state and the history of its interactions with the environment. Moreover, interactions between components may occur only through their interfaces. Thus, the internal state of a component must only be accessible through operations provided by the interface.

The actor model of computation has a built-in notion of local component and interface, and thus it is a natural model to use as a basis for a theory of open distributed computation. Specifically, we view actors as a model of coordination between autonomous interacting components. The local computation carried out by the components may be specified in any sequential language. However, we carry out the development of an actor semantics in a framework where local computation is specified using a functional language. We extend a functional programming language with actor coordination primitives to model open distributed systems. Our semantic theory uses techniques first developed in a functional setting. The transition system operational semantics extends the reduction system semantics of the functional language. The notion of observational equivalence studied generalizes the now standard notion of operational equivalence for functional languages. The resulting equational theory embodies that of the computational lambda calculus and preserves many of the advantages provided by functional programming for reasoning about programs and program transformations. The actor language we
study provides an alternative approach to concurrent extensions of functional languages that is object based rather than process based. Our components are reactive instead of active processes more common in other models of concurrency. Active processes correspond to a thread of control that disappears when execution of the thread is complete; unlike reactive processes, activity in them is not initiated by the receipt of a message. Note that receiving a message is analogous to a function being invoked.

1.1 Overview

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

In our model we make explicit the notion of open system component through the notion of an actor configuration. An actor configuration is a collection of individually named concurrently executing actors, a collection of messages in route, a set of receptionist names and a set of external actor names. Receptionists are the externally visible actors of the configuration, and names of external actors are references to actors outside the configuration. The receptionist and external actor names explicitly define the interface to the environment.

A common criticism of the actor model of computation is that actors do not compose. However actor configurations can be combined in a variety of ways to form new configurations. As a first step towards an algebra of operations on actor configurations, we define a composition operator on configurations. Composition is associative, commutative, and has a unit. This allows large complex configurations to be studied in parts and composed to form the full system. Unlike most notions of modularity and composability, which are static, the notions we define are fundamentally dynamic ones that allow for the interface between components to evolve over time.

Following the tradition of (Morris, 1968; Plotkin, 1975; Mason and Talcott, 1991; Felleisen and Hieb, 1992; Felleisen and Wright, 1991) we develop the semantics in two stages. The first stage consists in giving an operational semantics for actor configurations. In the second stage various notions of equivalence are investigated, both of expressions and of configurations. The operational semantics extends that of the embedded functional language in such a way that the equational theory of the functional language is preserved. In particular the equational laws of the computational lambda calculus (Moggi, 1988) as well as the usual laws for pairing and arithmetic hold. This provides a basis for a rich set of equational reasoning principles. There are also numerous equational laws that relate to actor computations, for instance allowing two adjacent message send operations to be permuted.

The operational semantics of actor configurations is defined by a transition re-
lation on configurations. An important aspect of the actor model is the fairness requirement: message delivery is guaranteed, and individual actor computations are guaranteed to progress. We make the fairness requirement explicit in our semantics by requiring infinite sequences of transitions on actor configurations to be fair. Although fairness makes some aspects of reasoning more complicated, it simplifies others. Certain classes of intuitively obvious equations fail to hold without the fairness requirement. The assumption of fairness allows equational specification of some liveness properties, and it is particularly important for modular reasoning about liveness properties.

Our equational theory is based on the notion of observational equivalence. Two expressions/configurations are said to be observationally equivalent if they give rise to the same observations, suitably defined, inside all observing contexts. This notion is closely related to testing equivalence (de Nicola and Hennessy, 1984). We prove that in the presence of fairness, the three standard notions of observational equivalence collapse to two. Observational equivalence provides a semantic basis for developing sound transformation rules for expressions.

Outline

The remainder of this paper is organized as follows. The rest of this section describes previous work on Actors and related models of concurrent computation. §2 gives the syntax and operational semantics of our actor language, and describes a composability result. In §3 we study the notion of observational equivalence for actor expressions. In §4 we state a variety of basic equational laws along with an intuitive explanation of how these laws are established. In §5 we develop methods for establishing expression equivalence, and use these methods to prove the laws of §4. §6 summarizes the highlights of this paper, and discusses future work.

1.2 Related Work

We discuss related work in actors, process algebras and concurrent functional languages.

1.2.1 Research on Actors

We may briefly summarize the principles underlying the actor model of computation that we use as follows. Actors are self-contained, concurrently interacting entities of a computing system. They communicate via message passing which is asynchronous and fair. Actors can be dynamically created and the topology of actor systems can change dynamically. The actor model is a primitive model of computation which nonetheless easily expresses a wide range of computation paradigms. It directly supports encapsulation and sharing, and provides a natural extension of both functional programming and object style data abstraction to concurrent open systems. (See (Agha, 1986a; Agha, 1990; Agha et al., 1993a) for more discussion of the actor model, and for many examples of programming with actors.)

G. Agha and others
Actor Computation

The actor model was originally proposed by Hewitt and the meaning of the term has evolved over time in the work of Hewitt and associates. We briefly describe the history of actor research, necessarily omitting some of the significant work.

In his early work on planner (Hewitt, 1971), Hewitt used the term actor to describe active entities which, unlike functions, went around looking for patterns to match in order to trigger activity. This concept was later developed into the scientific community metaphor where sprites examined facts and added to them in a monotonically growing knowledge base (Kornfeld and Hewitt, 1981). In (Hewitt et al., 1973), the notion of actors was close to that of an agent in Distributed Artificial Intelligence: actors have intentions, resources, contain message monitors and a scheduler. Irene Greif (Greif, 1975) developed an abstract model of actors in terms of event diagrams which recorded local events at each actor and the causal relations between events.

Baker and Hewitt (Baker and Hewitt, 1977) then formalized a set of axioms for concurrent computation which stated properties that events in actor systems must obey in order to prevent causality violations. The work in (Hewitt, 1977) contains the insight that the usual control structures could be represented as patterns of message passing between simple actors which had a conditional construct but no local state. It provided a basis for the continuation passing style of programming as found in Scheme (Abelson and Sussman, 1985).

In (Hewitt and Atkinson, 1979), the concept of serializer is described: a serializer localizes conditions for resumption of waiting processes and thus improves on monitors which require explicit signaling of dormant processes. A related notion, namely, that of guardians, was defined in (Attardi and Hewitt, 1978). A guardian regulates the use of shared resources, scheduling their access and providing protection and “recovery” boundaries. Guardians thus explicitly incorporated the notion of state. Lieberman implemented an actor language, Actl, incorporating guardians, serializers, and ‘rock bottom’ actors which is best described in (Lieberman, 1987).

Will Clinger (Clinger, 1981) developed a semantics of actor systems, showing the consistency of axioms proposed in (Baker and Hewitt, 1977). A key accomplishment of Clinger’s work was to show that a powerdomain semantics could be developed despite the fact that the underlying domain was incomplete due to fairness. The work did not develop a theory of actor systems — specifically, no notion of equivalence of actor systems was defined.

The model of actors we use is most closely resembles (Agha, 1986a) which defined a simple transition system for actors, and developed a notion of configurations, receptionists and external actors. The model was implemented by Carl Manning at MIT in the Acme programming language (Manning, 1987) and by Tomlinson and others at the Microelectronics and Computer Technology Consortium in the Rosette programming language (Tomlinson et al., 1989; Tomlinson et al., 1993). It has also provided a basis for dozens of other projects (Agha et al., 1989; Agha et al., 1991; Agha et al., 1993b). Some of the more recent research on actors has focused on coordination structures and meta-architectures (Agha et al., 1993a).
1.2.2 Other Related Research

We consider two areas of related work: process algebras and concurrent functional languages. In all of the work considered here the underlying model uses synchronous communication as primitive instead of asynchronous communication. Synchronous communication can be simulated with asynchronous primitives (Amadio, 1994) and vice-versa, and forms of both are required in a realistic programming language. Also in models considered, although processes may be dynamically created, they are not first class entities that can be directly manipulated. Channels are first class entities that can be dynamically created, communicated as values, and used for communication. One process can communicate with another only if they happen to share a communication channel. In general any number of different processes can send or receive on a given channel, thus processes can interfere with one another. In actor and other object based models, actors/objects are first class, history sensitive, entities whose identity can be communicated and used for communication with the identified object. The communication medium is not explicitly represented. Again, to some degree each choice of primitives can simulate the other. Further work is needed to clarify precisely the relative expressive powers of the two approaches. A more important general distinction is that of these models only (Reppy, 1992) incorporates fairness. Without fairness, specifications fail to compose — a process may behave correctly in isolation, but may fail to do so in the presence of other processes (even if it is not interacting with them).

Much of the work on process algebras is based on Milner's Calculus of Concurrent Systems (CCS) (Milner, 1983), Hoare's CSP (Hoare, 1985), and Milner's $\pi$-calculus (Milner et al., 1989). The work on process algebras and the $\pi$-calculus has focused on understanding elementary communications between processes, abstracting away other programming language issues. The CSP model assumes a fixed interconnection topology of processes, supports only static storage allocation, and disallows recursive procedures. The $\pi$-calculus is to concurrent programming as the pure $\lambda$-calculus is to functional programming in the sense that it provides a bare minimum of primitive notions: functions and function application in the case of the $\lambda$-calculus; and channels and communication in the case of the $\pi$-calculus. A variety of equivalence relations have been defined and studied. These include bisimulation (Milner, 1983; Milner, 1989) which defines a back-and-forth simulation relation between two processes, and trace-based equivalence (Brookes et al., 1984). Various logics have been developed for process algebras, including (Hoare, 1985; Milner, 1989; Bergstra and Klop, 1986).

Facile (Giacalone et al., 1989; Prasad et al., 1990; Thomsen et al., 1992) is perhaps the first language design based on the principle of combining a functional language with concurrency primitives. It differs from our approach in using the typed lambda calculus as the functional component and using concurrency primitives based on CSP and CCS. The underlying model of concurrency is one of processes communicating by synchronized message passing over typed channels. Any value of appropriate type, included higher-order values can be communicated over a channel. In (Giacalone et al., 1989) an operational semantics for Facile is
Actors Computation

given using a labelled transition system, and an abstract model of implementation is defined. In (Prasad et al., 1990) an algebraic semantics for Facile is given based on the operational semantics. The central notion here is bisimulation relations. Bismsulations are indexed by sets of channels called windows. This explicitly accounts for the interface of a system to its environment in much the same way as receptionists do for actor systems. Basic process algebra equations are established using bisimulations as well as the beta-value rule. The authors point out that establishing expression equivalence using bisimulations is much more complicated than establishing process algebra equations.

An early version of actor semantics presented in this paper appeared in (Agha et al., 1992). There we defined a notion of operational bisimulation that incorporated fairness. Operational bisimulations were intended to serve as tools for establishing observational equivalence, not as equivalence relations per se. We also found that operational bisimulations were not an effective tool for establishing expression equivalence, since it was difficult to find suitable bisimulations. This led us to develop the alternative methods presented in this paper.

Recently there have been two efforts (Reppy, 1991; Berry et al., 1992) to combine the work on ML with the work on process algebras to obtain a concurrent version of ML. The underlying concurrency model here is the same at that of Facile. In CML (Reppy, 1991) the channel interference problem can be avoided with the use of ML data abstraction mechanisms. A structured operational semantics and type inference system for a small kernel language contained in CML is described in (Reppy, 1991) and a type safety theorem is proved. In addition an implementation of CML is described. Basic properties of the operational semantics of similar kernel language are presented in (Berry et al., 1992). At this point, little has been done towards developing an equational theory concurrent extensions of ML.

2 Actor Language Constructs

Our actor language is an extension of a simple functional language which provides primitives for coordinating components which carry out local computation. An individual actor represents the smallest unit of coordination in the model. Our language provides a mechanism for specifying the creation and manipulation of actors. An actor's behavior is described by a lambda abstraction which embodies the code to be executed when a message is received. The actor primitives are: send; become; newadr; and initbeh.

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

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

Note that in open distributed systems, the order of arrival of messages from different external sources is nondeterministic. The become primitive is necessary to provide the history-sensitive behavior necessary to model asynchronous access to shared resources in such systems. A canonical example of the use of become is in modelling a shared bank account accessible by a two or more automatic teller machines. On the other hand, observe that the current behavior of an actor always remains a deterministic function of the sequence of messages that the actor has thus far received.

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

2.1 Trivial Examples

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

\[ b = \text{rec}(\lambda y. \lambda x. \text{seq}(\text{send}(x, 5), \text{become}(y))) \]

where seq is syntactic sugar for expressing sequential composition, and rec is a definable (in the pure \( \lambda \) fragment) call-by-value fixed-point combinator (cf. (Mason and Talcott, 1991)). An equivalent expression of this behavior is:

\[ b = \text{rec}(\lambda y. \lambda x. \text{seq}(\text{become}(y), \text{send}(x, 5))) \]

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

\[ e = \text{let}(z := \text{newadr()}) \text{seq}(\text{initbeh}(z, b), \text{send}(z, a)). \]

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

\[ \text{sink} = \text{rec}(\lambda b. \lambda m. \text{become}(b)). \]
Actor Computation

2.2 Actor Cells

It is easy to represent objects with local state in our language. As an example of this we describe an actor akin to an ML reference cell. The actor responds to two sorts of messages. The first sort of message is a get message, recognized by the get? operation. A get message contains the address (customer) to which the response (i.e. the current contents of the cell) should be sent. This address is accessed via the cust operation. A mkget(a) constructs a get message with customer a. The second message is a set message, recognized by the set? operation. A set message should contain the desired new contents of the cell-actor. This value is accessed via the contents operation. A mkset(c) constructs a set message with new contents c. Using these operations, the behavior of a cell is given by:

\[ B_{cell} = \text{rec}(\lambda b.\lambda c.\lambda m. \]
\[ \quad \text{if}(\text{get}?(m), \]
\[ \quad \quad \text{seq} \langle \text{become}(b(c)), \text{send}(\text{cust}(m), c) \rangle \]
\[ \quad \quad \text{if}(\text{set}?(m), \]
\[ \quad \quad \quad \text{become}(b(\text{contents}(m))), \]
\[ \quad \quad \quad \text{become}(b(c))) \rangle) \]

Evaluating

let \{ a \leftarrow \text{newadr}() \} \text{seq}(\text{initbox}(a, B_{cell}(0), e)) where

\[ e = \text{seq}(\text{send}(a, \text{mkset}(3)), \text{send}(a, \text{mkset}(4)), \text{send}(a, \text{mkget}(a))) \]

will result in the actor a receiving a message containing either 0, 3 or 4, depending on the arrival order. A cell is one of the simplest kinds of history sensitive object. The become primitive is the key to expressing history sensitivity by allowing new behaviors to be installed in response to messages. Accumulators, counters and gensym actors are easily constructed in a similar manner.

2.3 Join Continuations

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

\[ \text{treetprod} = \text{rec}(\lambda f.\lambda \text{tree}.) \]
\[ \quad \quad \text{if}(\text{isnat}(\text{tree}), \]
\[ \quad \quad \text{tree,} \]
\[ \quad \quad f(\text{left}(\text{tree})) * f(\text{right}(\text{tree}))) \]

In the above code, a tree is passed to treetprod which tests to see if the tree is a number (i.e. a leaf). If so it returns the tree, otherwise it subdivides the problem into two recursive calls. left and right are functions which pick off the left and
right branches of the tree. It is clear that the arguments to * may be evaluated concurrently. It is also clear that if a zero is encountered then the computation can terminate. In this example we only deal with the former optimization. The latter optimization, made using continuations, is treated in detail in (Talcott, 1993b; Talcott, 1985; Agha, 1990)

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

\[
B_{\text{treeprod}} = \\
\text{rec}(\lambda b. \lambda \text{self}. \lambda m. \\
\text{seq(become(b(self))),} \\
\text{if(isnat(tree(m)),} \\
\text{send(cust(m), tree(m)),} \\
\text{let\{newcust := newadr()\}} \\
\text{seq(initbeh(newcust, B_{\text{joincont}}(cust(m), 0, nil)),} \\
\text{send(self, <left(tree(m)), newcust>),} \\
\text{send(self, <right(tree(m)), newcust>))})
\]

\[
B_{\text{joincont}} = \text{rec}(\lambda b. \lambda \text{cust. A n args. A firstnum. A num} \\
\text{if(eq(nargs, 0),} \\
\text{become(b(cust, 1, num)),} \\
\text{seq(become(sink),} \\
\text{send(cust, firstnum * num))))
\]

When an tree product actor (with behavior \(B_{\text{treeprod}}\)) receives a tree of numbers that is not a leaf it creates a customer, called a join continuation, and sends two messages to itself to evaluate the two halves of the tree. These messages have the join continuation as customer. The join continuation (with behavior \(B_{\text{joincont}}\)) expects to receive two numbers representing the computation of the products of each of the two subtrees. When both numbers have arrived, the join continuation multiplies them and sends the result to its customer. Figure 1 shows some stages in the computation of of a tree production. The join continuations customer can be the original requester (root of the tree), or the join continuation of a higher branch point. Because multiplication is commutative, we need not be concerned about matching the responses to the order of the parameters. If we were dealing with an operator which was not commutative, we would need to tag the message corresponding to each argument and this tag would be returned with the response from the corresponding subcomputation. The replacement behavior of the join continuation would then depend on the order in which the evaluation of arguments was completed.

An advantage of explicit join continuations is that they provide considerable flexibility - they can be used to control the evaluation order, to do partial computations, and to do handle special cases or errors. For example, if the number 0
Fig. 1. The leaves of tree contain numbers to be multiplied and Cust is the actor to which the product is to be sent. (b) the tree is subdivided and two asynchronous messages are sent to compute the product on the two halves (concurrently). JC represents a newly created actor to which the two results should be sent. Each subtree will be recursively subdivided and new join-continuations will be successively created. (c) When the product of one of the subtrees has been computed, the value is sent to JC. (d) JC does a become to store the first number it receives. (e) The second product sends a value to JC. (f) JC multiplies the second number it receives with the number it had stored earlier and sends the result to Cust.
is encountered, the join continuation can immediately return a 0—in some cases without waiting for the results of evaluating the other subtree.

The above program may not be optimal in other respects. For example $send$ may be quite expensive. Consequently it may be prudent to check that the sub-computation is worth dispatching. This observation together with the fact, proved in §4, that $send$ of values commute leads to the possibly faster version:

$$B_{tree \text{product}} =
\text{rec} \lambda b. \lambda \text{self}. \text{Am}.
\quad \text{seq}(\text{become}(b),
\quad \quad \text{if}(\text{isnat}(\text{tree}(m)),
\quad \quad \quad \text{send}(\text{cust}(m), \text{tree}(m)),
\quad \quad \quad \text{let}\{\text{newcust} := \text{newadr}()\})
\quad \quad \text{seq}(\text{initbeh}(\text{newcust}, B_{joincont}(\text{cust}(m), 0, \text{nil}))
\quad \quad \quad \text{let}\{l := \text{left}(\text{tree}(m)), r := \text{right}(\text{tree}(m))\}
\quad \quad \quad \text{if}(\text{or}(\text{isnat}(l), \text{isnat}(r)),
\quad \quad \quad \quad \text{send}(\text{cust}(m), \text{treeprod}(l) \ast \text{treeprod}(r)),
\quad \quad \quad \quad \text{seq}(\text{send}(\text{self}, <l, \text{newcust}>)
\quad \quad \quad \quad \quad \quad \text{send}(\text{self}, <r, \text{newcust}>)))))$$

2.4 Notation.

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

3 A Simple Lambda-Based Actor Language

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

3.1 Syntax

We take as given countable sets \( X \) (variables), \( At \) (atoms), and for each \( n \in \mathbb{N} \) a (possibly empty) set of \( n \)-ary operations, \( C_n \) on \( At \). \( E_n \) is the set of primitive operations of arity \( n \), which includes \( C_n \), and \( E = \bigcup_{n \in \mathbb{N}} E_n \). We assume \( At \) contains \( \top, \bot \) for booleans, as well as constants for natural numbers, \( \mathbb{N} \). \( E \) contains arithmetic operations, recognizers \( isAtom \) for atoms, and \( isNat \) for numbers (arities 1, 1), branching \( br \) (arity 3), pairing \( isPr, pr \), \( 1^{st}, 2^{nd} \) (arities 1, 2, 1, 1), and actor primitives \( newadr, initbsh, send, \) and \( become \) (arities 0, 2, 2, 1). The sets of expressions, \( E \), value expressions (or just values), \( V \), and contexts (expressions with holes), \( C \), are defined inductively as follows.

**Definition** \(( E \ \forall \ \mathbb{C} )\):

\[
V = At \cup X \cup \lambda X. E \cup pr(V, V)
\]

\[
E = At \cup X \cup \lambda X. E \cup app(E, E) \cup E_n(E^n)
\]

\[
C = At \cup X \cup \lambda X. C \cup app(C, C) \cup E_n(C^n) \cup \{ \}
\]

We let \( x, y, z \) range over \( X \), \( v \) range over \( V \), \( e \) range over \( E \), and \( C \) range over \( C \). Since we are working with a syntactic reduction semantics, there is no distinction between a value expression and the value it denotes. Hence we use the terms value and value expression interchangeably. \( \lambda x.e \) binds the variable \( x \) in the expression \( e \). Two expressions are considered equal if they are the same up to renaming of bound variables. We say that a variable is fresh with respect to a context of use if it does not occur free or bound in any syntactic entity. We write \( \text{FV}(e) \) for the set of free variables of \( e \). We write \( e[x := e'] \) to denote the expression obtained from \( e \) by replacing all free occurrences of \( x \) by \( e' \), avoiding the capture of free variables in \( e' \). Contexts are expressions with holes. We use \( \bullet \) to denote a hole. \( C[e] \) denotes the result of replacing any holes in \( C \) by \( e \). Free variables of \( e \) may become bound in this process. \( \text{let, if, and seq} \) are the usual syntactic sugar, \( \text{seq} \) being a sequencing primitive.

\[
\text{let}\{x := e_0\}e_1 \text{ abbreviates } \text{app}(\lambda x.e_1, e_0)
\]

\[
\text{seq}(e_0, e_1) \text{ abbreviates } \text{app}(\text{app}(\lambda z.\lambda x.e_0), e_1)
\]

\[
\text{if}(e_0, e_1, e_2) \text{ abbreviates } \text{app}(\text{br}(e_0, \lambda z.e_1, \lambda z.e_2), \text{nil}) \text{ for } x \text{ fresh}
\]

3.2 Reduction Semantics for Actor Configurations

The operational semantics for actor systems is given by a transition relation on configurations. A configuration can be thought of as representing a global snapshot of an actor system with respect to some idealized observer (Agha, 1986b). It contains a collection of actors, messages, external actor names, and receptionist names. The sets of receptionists and external actors are the interface of an actor configuration to its environment. They specify what actors are visible and what actor connections must be provided for the configuration to function. Both the set of receptionists and the set of external actors may grow as the configuration evolves.
The state of the actors in a configuration is given by an actor map. An actor map is a finite map from actor addresses to actor states. Each actor state is one of
(a) uninitialized, having been newly created by an actor named $a$;
(b) ready to accept a message, where $b$ is its behavior, a lambda abstraction; or
(c) busy executing $e$, here $e$ represents the actor's current (local) processing state.

A message $m$ contains the address of the actor to whom it is sent and the message contents, $\langle a \leftarrow v \rangle$. The contents $v$ can be any value constructed from atoms and actor addresses using the pairing constructor. Lambda abstractions and structures containing lambda abstractions are not allowed to be communicated in messages. There are two reasons for this restriction. Firstly, allowing lambda abstractions to be communicated in values violates the actor principle that only an actor can change its own behavior, because a become in a lambda message may change the receiving actor behavior. Secondly, if lambda abstractions are communicated to external actors, there is no reasonable way to control what actor addresses are actually exported. This has unpleasant consequences in reasoning about equivalence, amongst other things. This restriction is not a serious limitation since the address of an actor whose behavior is the desired lambda abstraction can be passed in a message.

The transition relation determines the set of possible future configurations. We classify actor configuration transitions as internal or external to the configuration. The internal transitions are:
- rcv: receipt of a message by an actor not currently busy computing; and
- exec: an actor executing a step of its current computation.

The internal transitions involve a single active actor, which we call the focus actor for the transition. exec transitions may be purely local (a $\lambda$-transition), or messages may be sent, or a new actor may be created, or a newly created actor may be initialized. rcv transitions consume a message, putting the focus actor in a busy state.

In addition to the internal transitions of a configuration, there are $1/o$ transitions that correspond to interactions with external agents:
- in: arrival of a message to a receptionist from the outside; and
- out: passing a message out to an external actor.

To capture actor semantics, we require that computations be fair.

### 3.2.1 Actor Configurations

We assume that we are given a countable set $Ad$ of actor addresses. To simplify notation, we identify $Ad$ with $X$, and call variables used in this way actor names. This pun is useful for two reasons: it allows us to use expressions to describe actor states and message contents; and it allows us to avoid problems of choice of names for newly created actors by appealing to an extended form of alpha conversion. (See (Mason and Talcott, 1991; Felleisen and Hieb, 1992) for use of this pun to represent reference cells.)

**Definition** ($cV$ $As$ $M$): The set of communicable values, $cV$, the set of actor
states, \( \mathcal{A} \), and the set of messages, \( \mathcal{M} \), are defined as follows.

\[
\mathcal{V} \mathcal{V} = \mathcal{A} \mathcal{U} \mathcal{X} \cup \text{pr}(\mathcal{V} \mathcal{V}, \mathcal{V} \mathcal{V}) \quad \mathcal{A}s = (?_x) \cup (\mathcal{V}) \cup [\mathcal{E}] \quad \mathcal{M} = \langle \mathcal{V} \mathcal{V} \rangle
\]

We let \( \mathcal{V} \mathcal{V} \) range over \( \mathcal{V} \mathcal{V} \) and \( m \) range over \( \mathcal{M} \). Note that actor behaviors are not syntactically restricted to be lambda abstractions, nor are messages syntactically restricted to be of the form \( \langle \mathcal{A} \mathcal{D} \rangle \leftarrow \mathcal{V} \mathcal{V} \rangle \). The reduction system will prevent use of any ill-formed behavior or message. This is in keeping with the untyped nature of our language.

**Definition (Actor Configurations \( \mathcal{K} \))**: An actor configuration with actor map, \( \alpha \), multi-set of messages, \( \mu \), receptionists, \( \rho \), and external actors, \( \chi \), is written

\[
\left\{ \alpha \mid \mu \right\}^\rho_{\chi}
\]

where \( \rho, \chi \in \mathbf{P}_w[\mathcal{X}], \alpha \in \mathcal{X} \to \mathcal{A}s, \) and \( \mu \in \mathbf{M}_w[\mathcal{M}] \). Further, it is required that, letting \( A = \text{Dom}(\alpha) \), the following constraints are satisfied:

1. \( \rho \subseteq A \) and \( A \cap \chi = \emptyset \),
2. if \( a(a) = (?_a) \), then \( a' \in A \),
3. if \( a \in A \), then \( \text{FV}(\alpha(a)) \subseteq A \cup \chi \), and if \( \langle v_0 \leftarrow v_1 \rangle \in \mu \), then \( \text{FV}(v_i) \subseteq A \cup \chi \) for \( i < 2 \).

We let \( \mathcal{K} \) denote the set of actor configurations and let \( \kappa \) range over \( \mathcal{K} \). The receptionists \( \rho \) are names of actors within the configuration that are externally visible; all other actors in the (actor) configuration are local and thus inaccessible from the outside. The external actors \( \chi \) are names of actors that are outside this configuration but to which messages may be directed. A configuration in which both the receptionist and external actor sets are empty is said to be closed. For closed configurations we may omit explicit mention of the empty \( \rho \) and \( \chi \) sets. The actor map portion of a configuration is presented as a list of actor states each subscripted by the actor address which is mapped to this state. If \( \alpha'(a) = (b) \), and \( \alpha \) is \( \alpha' \) with \( a \) omitted from its domain, we write \( \alpha' \) as \( (a, (b)_{a}) \) to focus attention on \( a \). We follow a similar convention for other states subscripted with addresses.

The set of possible computations of an actor configuration is defined in terms of the labelled interleaving transition relation \( \rightarrow \) on configurations. Although this is, on the surface an interleaving semantics, it is easy to modify our transition system to obtain a truly concurrent semantics either by forming a labelled transition system with independence (Sassone et al., 1993), or by using concurrent rewriting (Meseguer, 1992).

### 3.2.2 Decomposition and Reduction

To describe the internal transitions other than message receipt, a non-value expression is decomposed uniquely into a reduction context filled with a redex. Reduction contexts identify the subexpression of an expression that is to be evaluated next, they correspond to the standard reduction strategy (left-first, call-by-value) of (Plotkin, 1975) and were first introduced in (Felleisen and Friedman, 1986). For further discussion of this method of defining reduction relations see (Honsell et al.,
In order to distinguish holes used for different purposes, we use the sign □ for the hole occurring in a reduction context, and call such holes *redex holes* (although they may in fact be filled with non-redex expressions).

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

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

$$\mathbb{R} = \{\emptyset\} \cup \text{app}(\mathbb{R}, \mathbb{R}) \cup \text{app}(\mathbb{V}, \mathbb{R}) \cup \mathbb{F}_n+m+1(\mathbb{V}^n, \mathbb{R}, \mathbb{E}^m)$$

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

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

**Lemma** (Unique decomposition):

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

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

**Proof**: An easy induction on the structure of $e$ □

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

**Definition** ($\lambda \rightarrow$):

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

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

where $\delta \in \mathbb{C}_n$, $v_1, \ldots, v_n \in \mathbb{A}^n$, and $\delta(v_1, \ldots, v_n) = v'$.

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

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

(fst) $R[\text{fst}(\text{pr}(v_0, v_1))] \xrightarrow{\lambda} R[v_0]$

 snd) $R[\text{snd}(\text{pr}(v_0, v_1))] \xrightarrow{\lambda} R[v_1]$

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

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

**Definition** ($\rightarrow$):

<fun : a>
Actor Computation

\[ e \xrightarrow{\lambda} e' \Rightarrow \langle \alpha, [e]_a \mid \mu \rangle_\chi \xrightarrow{\rho} \langle \alpha, [e']_a \mid \mu \rangle_\chi \]

\(<\text{new} : a, a'>\>

\[ \langle \alpha, [R\text{newadr}()]_a \mid \mu \rangle_\chi \xrightarrow{\rho} \langle \alpha, [R[a']]_a, (\_)_a' \mid \mu \rangle_\chi \quad a' \text{ fresh} \]

\(<\text{init} : a, a'>\>

\[ \langle \alpha, [R\text{initbeh}(a', v)]_a, (\_)_a' \mid \mu \rangle_\chi \xrightarrow{\rho} \langle \alpha, [R[\text{nil}]]_a, (v)_a \mid \mu \rangle_\chi \]

\(<\text{bec} : a, a'>\>

\[ \langle \alpha, [R\text{become}(v)]_a \mid \mu \rangle_\chi \xrightarrow{\rho} \langle \alpha, [R[\text{nil}]]_a, (v)_a \mid \mu \rangle_\chi \quad a' \text{ fresh} \]

\(<\text{send} : a, m\>

\[ \langle \alpha, (v)_a \mid \mu \rangle_\chi \xrightarrow{\rho} \langle \alpha, [\text{app}(v, cv)]_a \mid \mu \rangle_\chi \]

\(<\text{out} : m\>

\[ \langle \alpha \mid \mu, m \rangle_\chi \xrightarrow{\rho} \langle \alpha \mid \mu \rangle_\chi \]

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

\(<\text{in} : m\>

\[ \langle \alpha \mid \mu \rangle_\chi \xrightarrow{\rho} \langle \alpha \mid \mu, m \rangle_\chi^{(\text{FV}(cv) \setminus \text{Dom}(\alpha))} \]

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

In the rules for newadr and become, \( a' \) fresh abbreviates \( a' \notin \text{Dom}(\alpha) \cup \{ a \} \cup \chi \). We write \( \kappa_0 \xrightarrow{l} \kappa_1 \) if \( \kappa_0 \xrightarrow{l} \kappa_1 \) according to the rule labelled by \( l \). We call this triple a labelled transition. We say \( l \) is enabled in configuration \( \kappa \) if there is some \( \kappa' \) such that \( \kappa \xrightarrow{l} \kappa' \). The transitions are allowed to reason about sequences of transitions in terms of the rules applied, and to allow for alternative representation of computations, including: sequences of configurations; sequences of labelled transitions; and sequences of labels. Note that we have chosen the labels to include sufficient information that \( \kappa' \) is uniquely determined by \( \kappa \) and \( l \).

I/O transitions are transitions with tags in or out; rcv transitions are transitions with tag rcv. The remaining transitions are called exec transitions. The exec transitions correspond to the execution of functional or actor redexes.

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

A clone produced to carry on after a become is not allowed to initialize an actor created by its cloner. This is a technical simplification. With some additional bookkeeping we could keep track of cloners and allow clones to initialize. Alternatively, this technical detail would disappear if we used the \texttt{1stactor} construct for actor creation.

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

### 3.2.3 Computation Sequences and Paths

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

**Lemma (Anonymity):** If $\kappa$ be a computation sequence in the computation tree, $T(\kappa)$, then $\nu$ contains no transitions with label of the form $<\text{send} : a', \nu>$.  

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

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

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

1. $\text{Cfig}(\kappa, L, 0) = \kappa$
2. $\text{Cfig}(\kappa, L, i + 1) = \kappa'$ where $\text{Cfig}(\kappa, L, i) \xrightarrow{l_i} \kappa'$.

Thus, the path $\pi$ determined by $\kappa, L$ is the sequence

$[\text{Cfig}(\kappa, L, i) \xrightarrow{l_i} \text{Cfig}(\kappa, L, i + 1) \mid i < \infty]$.

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

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

### 3.2.4 Fairness

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

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

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

### 3.3 Composition of Actor Configurations

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

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

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

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

\((\kappa_0, \kappa_1)\) is a decomposition of \( \kappa \) if \( \kappa_i, i < 2 \) are composable configurations, and \( \kappa = \kappa_0 \parallel \kappa_1 \).

**Lemma (AC):** Let \( \kappa_i = \left\langle \alpha_i \mid \mu_i \right\rangle_{\chi_i}^{\rho_i}, i < 3 \) be pairwise composable configurations. And let \( \kappa_0 = \left\langle \emptyset \mid \emptyset \right\rangle \) be the empty configuration. Then

\[
\kappa_0 \parallel \kappa_1 = \kappa_1 \parallel \kappa_0
\]

\[
\kappa_0 \parallel \kappa_0 = \kappa_0
\]

\((\kappa_0 \parallel \kappa_1) \parallel \kappa_2 = \kappa_0 \parallel (\kappa_1 \parallel \kappa_2)\)

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

\[
((\chi_0 \cup \chi_1) - (\rho_0 \cup \rho_1)) \cup \chi_2) - (\rho_0 \cup \rho_1 \cup \rho_2)
\]

\[
= (\chi_0 \cup \chi_1 \cup \chi_2) - (\rho_0 \cup \rho_1 \cup \rho_2)
\]

\[
= (\chi_0 \cup ((\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2))) - (\rho_0 \cup \rho_1 \cup \rho_2)
\]

\]

Furthermore, it is possible to define composition operations on trees

\[
T(\kappa_0) \parallel T(\kappa_1) \quad \text{and} \quad F(\kappa_0) \parallel F(\kappa_1)
\]

such that

(1) The computation tree of the composition of actor configurations is the composition of the computation trees of the components.

\[
T(\kappa_0 \parallel \kappa_1) = T(\kappa_0) \parallel T(\kappa_1)
\]

(2) The set of fair computation paths of the composition of actor configurations is the composition of the fair computation paths of the components.

\[
F(\kappa_0 \parallel \kappa_1) = F(\kappa_0) \parallel F(\kappa_1)
\]

4 Equivalence of Expressions

In this section we study equivalence of expressions of our actor language. Our notion of equivalence is a combination of the now classic operational equivalence of (Plotkin, 1975) and testing equivalence of (de Nicola and Hennessy, 1984). For the deterministic functional languages of the sort Plotkin studied, this equivalence is defined as follows. Two program expressions are said to be equivalent if they behave the same when placed in any observing context. An observing context is some complete program with a hole, such that all of the free variables in the expressions being observed are captured when the expressions are placed in the hole. The notion of "behave the same" is (for deterministic functional languages) typically equi-termination, i.e. either both converge or both diverge.

4.1 Events

The first step is to find proper notions of "observing context" and "behave the same" in an actor setting. The analogue of an observing context is an observing actor configuration: a configuration that contains an actor state with a hole. Since termination is not relevant for actor configurations, we instead introduce an observer primitive, \texttt{event} and \texttt{observe} whether or not in a given computation, \texttt{event} is executed. Our approach is identical in spirit to that used in defining testing equivalence for CCS (de Nicola and Hennessy, 1984).

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

\[
\langle e : a \rangle \quad \mathcal{R}[\text{event}] \quad \mu \quad \mathcal{R}[\text{nil}] \quad \mu
\]

**Definition (\( \text{obs} \))**: For an expression \( e \), the observing configurations are configurations over the extended language of the form \( \langle \alpha, [C]_\delta \mid \mu \rangle \), such that filling the hole in \( C \) with \( e \) results in a closed configuration. We use \( \text{obs} \) to denote the set of observing configurations, and let \( O \) range over \( \text{obs} \).

In our definition of observing configuration, the holes appear in the current state of an single executing actor. It is not hard to see that allowing holes in any actor state does not change the resulting notion of equivalence. A generalization of this fact, (oex) is proved in §5.

Since the language is nondeterministic, three different outcomes are possible in place of the two in the deterministic case: either \text{event} occurs for all possible executions, it occurs in some executions but not others, or it never occurs. We observe \text{event} transitions in the fair paths. We say that a path succeeds, \( s \), if an \text{event} transition occurs in it. This is the basic unit of observation; on top of this derived notions can be defined. We say a computation is observed to fail, \( f \), if it is not observed to succeed. \( \text{obs}(\pi) \) is \( s \) if \( \pi \) succeeds, and \( f \) otherwise, and \( \text{Obs}(\kappa) \) is the set of observations possible for all paths of a closed actor configuration.

**Definition (observations)**: Let \( \kappa \) be a configuration of the extended language, and let \( \pi = |\kappa_i \xrightarrow{i} \kappa_{i+1} | i < \infty \) be a fair path, i.e. \( \pi \in \mathcal{F}(\kappa) \). Define

\[
\text{obs}(\pi) = \begin{cases} s & \text{if } (\exists i < \infty)(i = \langle e : a \rangle) \\ f & \text{otherwise} \end{cases}
\]

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

### 4.2 Three Equivalences

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

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

1. (testing or convex or Plotkin or Egli-Milner)
   \( e_0 \cong_1 e_1 \) if \( \text{Obs}(O[e_0]) = \text{Obs}(O[e_1]) \) for all observing contexts \( O \in \text{obs} \)
2. (must or upper or Smyth)
   \( e_0 \cong_2 e_1 \) if \( \text{Obs}(O[e_0]) = s \iff \text{Obs}(O[e_1]) = s \) for all observing contexts \( O \in \text{obs} \)
3. (may or lower or Hoare)


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

By construction each of these equivalence relations is a congruence.

Theorem (congruence):

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

4.3 Partial Collapse

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

Theorem (partial collapse):

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

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

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

To demonstrate (1 = 2) we consider a fixed pair of expressions $e_0, e_1$ and categorize their closing configuration contexts $O$ according to what observations are made by $O[e_0]$ and $O[e_1]$. We say $O$ is labelled $o : o'$ for $o, o' \in \{s, sf, f\}$ just if $\text{Obs}(O[e_0]) = o$ and $\text{Obs}(O[e_1]) = o'$. This partitions the observing configuration contexts of $e_0$ and $e_1$ into nine sets labeled $o : o'$ for $o, o' \in \{s, sf, f\}$.

Lemma (sets) characterizes the various possibilities for equivalence in terms of which sets must be empty.

Lemma (sets):

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

This characterization is summarized in the picture below. Here $\times$ indicates that the set must be empty, and $\bigvee$ indicates that the set might be non-empty. The two $s'$d cases in $\cong_2$ are cases in which sets are allowed in theory to be non-empty, but lemma $(f, sf)$ below shows they in fact never arise.

<table>
<thead>
<tr>
<th></th>
<th>$\cong_1$</th>
<th>$\cong_2$</th>
<th>$\cong_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$e_0$</td>
<td>$e_1$</td>
<td>$e_1$</td>
</tr>
<tr>
<td>$s$</td>
<td>$\sqrt{\times \times}$</td>
<td>$s$</td>
<td>$s$</td>
</tr>
<tr>
<td>$sf$</td>
<td>$\times \sqrt{\times \times}$</td>
<td>$sf$</td>
<td>$sf$</td>
</tr>
<tr>
<td>$f$</td>
<td>$\times \times \sqrt{\bigvee}$</td>
<td>$f$</td>
<td>$f$</td>
</tr>
</tbody>
</table>

The key to collapsing $\cong_2$ into $\cong_1$ is the observation that if $\text{Obs}(O[e_0]) = f$ and
\textbf{Actor Computation} 23

\[ \text{Obs}(O[e_1]) = \text{sf} \text{ it is always possible to construct a } O^* \text{ such that } \text{Obs}(O^*[e_0]) = s, \text{ and } \text{Obs}(O^*[e_1]) = \text{sf}. \]

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

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

**Proof (partial collapse):**

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

3 \( \not\leftrightarrow \) 1 We construct expressions \( e_0, e_1 \) such that \( e_0 \cong_3 e_1 \), but \( \lnot(e_0 \cong_2 e_1) \).

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

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

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

Now we extend the notion of observational equivalence to configurations.

Definition (Observing Configurations): For an actor configuration \( \kappa = \left\langle \alpha \mid \mu \right\rangle^\rho_x \) the observing configurations are configurations over the extended language of the form \( \kappa' = \left\langle \alpha' \mid \mu' \right\rangle^\rho_x \) such that \( \kappa' \) is composable (in the sense of §2.3) with \( \kappa \).

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

Definition (Hide(\kappa)): \( \text{Hide}(\left\langle \alpha \mid \mu \right\rangle^\rho_x) = \left\langle \alpha \mid \mu \right\rangle^\emptyset_x \)

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

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

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

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

(i) \( \kappa_0 = \left\langle \alpha, [C[e_0]]_a \mid \mu \right\rangle^\rho_x \cong \left\langle \alpha, [C[e_1]]_a \mid \mu \right\rangle^\rho_x = \kappa_1 \)

(ii) \( \kappa'_0 = \left\langle \alpha, (\lambda x. C[e_0])_a \mid \mu \right\rangle^\rho_x \cong \left\langle \alpha, (\lambda x. C[e_1])_a \mid \mu \right\rangle^\rho_x = \kappa'_1 \)

Proof:

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

(ii) We need to show that \( \text{Obs}(\text{Hide}(\kappa'_0 \parallel \kappa')) = \text{Obs}(\text{Hide}(\kappa'_1 \parallel \kappa')) \) for any observing \( \kappa' \). Pick any \( \pi_0 \in \mathcal{F}(\text{Hide}(\kappa'_0 \parallel \kappa')) \), then we must find \( \pi_1 \in \mathcal{F}(\text{Hide}(\kappa'_1 \parallel \kappa')) \) such that \( \text{obs}(\pi_0) = \text{obs}(\pi_1) \). There are two cases to consider. Either actor \( a \) never becomes active, or it becomes active first after \( k \) steps of computation. In the first case, the \( e_i \) are never touched, so both computations proceed uniformly, thus their observation and fairness behavior both correspond. In the second case, consider the step where \( a \) receives its first message:

\[ \left\langle \alpha', (\lambda x. C[e_0])_a \mid \mu, a := cv \right\rangle \]

\[ \xrightarrow{cv \mapsto a, cv} \left\langle \alpha', [\text{app}(\lambda x. C[e_0], cv)]_a \mid \mu \right\rangle = O[e_0] \]

Factor \( \pi_0 = \nu[e_0] \ast \pi'_0 \), where \( \pi'_0 \in \mathcal{F}(O[e_0]) \) and \( \nu[] \) denotes a sequence where each configuration in the sequence contains a hole that computes uniformly in the hole. Thus, \( \text{Obs}(O[e_0]) = \text{Obs}(O[e_1]) \) because \( e_0 \cong e_1 \) and \( O \) is a configuration.
context. This means by the definition of $\text{Obs}$ there is a path $\pi'_1 \in \mathcal{F}(\nu[e_1])$, such that $\text{obs}(\pi'_0) = \text{obs}(\pi'_1)$. Let $\pi_1 = \nu[e] \ast \pi'_1$. Then, $\pi_1 \in \mathcal{F}($Hide$(\kappa_1 \parallel \kappa'))$, since by construction it is a computation for (Hide$(\kappa_1 \parallel \kappa')$), and because $\pi$ fair implies $\nu \ast \pi$ is fair for any $\nu$ such that $\nu \ast \pi$ is a computation path. Moreover, $\text{obs}(\pi_0) = \text{obs}(\pi_1)$ because any event transitions in $\nu[e_0]$ also occur in $\nu[e_1]$, and $\text{obs}(\pi'_0) = \text{obs}(\pi'_1)$ by hypothesis.

□

5 Laws of Expression Equivalence

With a notion of equivalence on actor expressions defined, a library of useful equivalences can be established. The first part of this section contains a collection of purely functional laws that continue to hold in the actor setting. The second part contains a collection of laws for manipulating expressions that involve actor primitives. These laws are established in §5.

5.1 Functional Laws

Since our reduction rules preserve the evaluation semantics of the embedded functional language, many of the equational laws for this language (cf. (Talcott, 1993b)) continue to hold in the full actor language. A first simple observation is that communicable values are observationally equivalent iff they are the same value expression.

**Lemma (eq):**

\[
\text{cv}_0 \equiv \text{cv}_1 \iff \text{cv}_0 = \text{cv}_1
\]

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

\[
O = \left\{ [\text{if}(\text{eq}(\text{cv}_0, \bullet), \text{event}, \text{nil})]_a \right\}
\]

distinguishes the atom $\text{cv}_0$ from any non-atom (and any other atom). Similarly,

\[
O = \left\{ [\text{let}\{x := 0\} \text{let}\{y := 1\} \text{if}(\text{eq}(x, \bullet), \text{event}, \text{nil})]_a \right\}
\]

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

The laws of the computational lambda calculus (Moggi, 1988), and the laws for conditional and pairing continue to hold in the actor setting.

**Theorem (functional laws):**

- **(beta-v)** $\text{app} (\lambda x. e, v) \equiv e[x := v]$
- **(if)** $\text{if}(v, e_1, e_2) \equiv e_1$ if $v \in (\texttt{At} - \{\texttt{nil}\}) \cup \bot \cup \text{pr}(\top, \bot)$
- **(ifn)** $\text{if}(\texttt{nil}, e_1, e_2) \equiv e_2$
- **(ifelim)** $\text{if}(v, e, e) \equiv e$
(iflam) \[ \lambda x.\text{if}(v, e_1, e_2) \equiv \text{if}(v, \lambda x.e_1, \lambda x.e_2) \quad x \not\in \text{FV}(v) \]
(ispr) \[ \text{ispr}(\text{pr}(w_0, v_1)) \equiv t, \]
(isprn) \[ \text{ispr}(v) \equiv \text{nil} \quad v \in \mathcal{A} \cup \mathcal{L} \]
(fst) \[ 1^\text{st}(\text{pr}(w_0, v_1)) \equiv w_0 \]
(snd) \[ 2^\text{nd}(\text{pr}(w_0, v_1)) \equiv v_1 \]

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

**Theorem (red-exp):**
\[ e_0 \xrightarrow{\lambda} e_1 \Rightarrow e_0 \equiv e_1 \]

The theorem (rcx) is a special case of a theorem proved in (Talcott, 1989).

**Theorem (rcx):** If \( R \) is a reduction context, then
\[ \text{let}(x := e) R[x] \cong R[e] \]
\[ \text{let}(x := e) e_0 \cong \text{let}(x := e) R[e_0] \]
\[ \text{if}(e, e_1, e_2) \cong \text{if}(e, R[e_1], R[e_2]) \]

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

\[ (\text{app}) \quad e_0(e_1) \cong (\lambda f. f(e_1))(e_0) \]
\[ (\text{cmps}) \quad f(g(e)) \cong (\lambda x. f(g(x)))(e) \]
\[ (\text{id}) \quad \text{app}(\lambda x. e, e) \cong e \]

Some useful corollaries of (rcx) are the following.

**Corollary (uni-rcx):**
\[ \text{let}(\text{arg}) \quad v(\text{let}(x := e_0)e_1) \cong \text{let}(x := e_0)v(e_1) \]
\[ (\text{if}) \quad \text{if}(\text{if}(e_0, e_1, e_2), e_3, e_4) \cong \text{if}(e_0, \text{if}(e_1, e_4, e_5), \text{if}(e_2, e_3, e_6)) \]

The above laws are really about equivalence of reduction contexts. They are instances of the operational law (red-rcx). Two reduction contexts are considered equivalent if placing an arbitrary expression in the redex hole results in equivalent expressions. The law (red-rcx) says that if two reduction contexts have a common \( \lambda \)-reduct when the redex hole is filled with a fresh variable (standing for an arbitrary value expression), then they are equivalent.

**Theorem (red-rcx):** If there is some \( e' \) such that \( R_0[x] \xrightarrow{\lambda} e' \) and \( R_1[x] \xrightarrow{\lambda} e' \) where \( x \) is a fresh variable, then \( R_0[e] \cong R_1[e] \) for any \( e \).

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

To make these ideas more precise we define \( \text{Hang} \) and \( \text{Infin} \) as follows.

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

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

**Theorem (hang-infin):** If \( e_0, e_1 \in \text{Hang} \cup \text{Infin} \), then \( e_0 \simeq e_1 \).

### 5.2 Basic Laws for Actor Primitives

Now we consider the equational properties of the actor primitives, \( \text{send}, \text{become}, \text{newadr}, \) and \( \text{initbeh} \). As is the case for a language with state operations, \( \text{seq}(e, e) \simeq e \) fails to hold because the computation \( e \) could have effects such as message sends.

A stronger analogy exists between the actor primitives and the reference primitives \( \{ \text{mk}, \text{get}, \text{set} \} : \text{newadr} \) is an allocation primitive analogous to \( \text{mk} \); \( \text{initbeh} \) and \( \text{become} \) update or alter state; while the effect of \( \text{send} \) depends on the state in a way analogous to get. There are limits to this analogy, for example \( \text{send} \) does not return anything of interest. In fact \( \text{send}, \text{become}, \) and \( \text{initbeh} \) all return \( \text{nil} \) as values. Thus

\[
(\text{triv}) \quad \vartheta(\hat{x}) \simeq \text{seq}(\vartheta(\hat{x}), \text{nil}) \quad \text{for } \vartheta \in \{\text{send}, \text{become}, \text{initbeh}\}
\]

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

\[
(\text{non-delay}) \quad \text{let}\{y := e_0\}\text{let}\{x := \text{newadr}()\}e_1 \not\simeq \text{let}\{x := \text{newadr}()\}\text{let}\{y := e_0\}e_1
\]

where \( x \) is not free in \( e_0 \). Since, if evaluation of \( e_0 \) executes a \( \text{become} \) and \( e_1 \) is of the form \( \text{initbeh}(x, v) \), then the left-hand side evaluation of \( e_1 \) will succeed, while the right-hand side evaluation of \( e_1 \) will suspend.

Once allocated, an actor behavior is updated by either \( \text{become} \) or \( \text{initbeh} \). In analogy with \( \text{set} \), both \( \text{become} \) and \( \text{initbeh} \) satisfy certain, slightly different, cancellation laws:

\[
(\text{can-b}) \quad \text{seq}(\text{become}(v_0), \text{become}(v_1)) \simeq \text{seq}(\text{become}(v_0), \text{nil}) \simeq \text{become}(v_0)
\]

\[
(\text{can-i}) \quad \text{seq}(\text{initbeh}(v, v_0), \text{initbeh}(v, v_1)) \simeq \text{seq}(\text{initbeh}(v, v_0), \text{stuck}) \not\simeq \text{seq}(\text{initbeh}(v, v_0), \text{bot})
\]

Note the difference between these two principles. In the case of \( \text{become} \) the second call is equivalent to \( \text{nil} \), while in the case of \( \text{initbeh} \) it is stuck (which is equivalent to diverging).
5.2.1 Commuting Operations

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

**Definition (commutes):** We say two operations \( \vartheta_0 \) and \( \vartheta_1 \) commute if

\[
\text{let}\{x_0 := \vartheta_0(y)\}\text{let}\{x_1 := \vartheta_1(z)\}e \cong \text{let}\{x_1 := \vartheta_1(z)\}\text{let}\{x_0 := \vartheta_0(y)\}e
\]

for all \( e \in \mathbb{E} \), \( x_0 \notin e \), \( x_1 \notin e \) and \( y \neq z \). Similarly we say two expressions \( e_0 \) and \( e_1 \) commute iff

\[
\text{let}\{x_0 := e_0\}\text{let}\{x_1 := e_1\}e \cong \text{let}\{x_1 := e_1\}\text{let}\{x_0 := e_0\}e
\]

if the obvious hygiene conditions hold.

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

\[
e_0 = \text{let}\{y := \text{newadr}()\}\text{let}\{z := \text{become}(b)\}\text{initbeh}(y, b')
e_1 = \text{let}\{z := \text{become}(b)\}\text{let}\{y := \text{newadr}()\}\text{initbeh}(y, b')
\]

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

\[
\left\llceil (\lambda x.\text{event}())_{a_1}, [\text{seq}\{\text{send}(a_0, 0)\}]_a \mid \emptyset \right\rrceil
\]

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

\[
\begin{align*}
\text{com-ss} & \quad \text{seq}(\text{send}(v_1, v_1), \text{send}(v_2, v_2)) \cong \text{seq}(\text{send}(v_2, v_3), \text{send}(v_0, v_1)) \\
\text{send also commutes with become} \\
\text{com-sb} & \quad \text{seq}(\text{send}(a_0, v_0), \text{become}(v_1)) \cong \text{seq}(\text{become}(v_1), \text{send}(a_0, v_0))
\end{align*}
\]

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

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

**Proof (partial):** If \( \vartheta(y) \) diverges, then

\[
\text{let}\{x := \vartheta(y)\}\text{let}\{x_1 := \text{send}(a, v)\}e
\]

will not execute the send, whereas

\[
\text{let}\{x_1 := \text{send}(a, v)\}\text{let}\{x := \vartheta(y)\}e
\]
will execute the send. Consequently the two expressions are easily distinguished. Similarly with the two operations become and initbeh. 

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

\[ e_0 = \text{seq(initbeh}(a_0, b_0), \text{send}(a_1, 0)) \]
\[ e_1 = \text{seq(send}(a_1, 0), \text{initbeh}(a_0, b_0)) \]

are distinguished by

\[ O_0 = \left\langle (\lambda z. \text{event()} \ a_1, (? \ a_1) \ a_0, [\ast]_a \mid \mu \right\rangle \]

for \( a \neq a_1 \), or by

\[ O_1 = \left\langle (\lambda z. \text{event()} \ a_1, (b) \ a_0, [\ast]_a \mid \mu \right\rangle \]

Also,

\[ e_0 = \text{seq(initbeh}(a_0, b_0), \text{become}(\lambda z. \text{send}(a_1, 0))) \]
\[ e_1 = \text{seq(become}(\lambda z. \text{send}(a_1, 0)), \text{initbeh}(a_0, b_0)) \]

are distinguished by \( O_0, O_1 \) if \( \mu \) contains \( \langle a \leftarrow 0 \rangle \).

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

We summarize these results in the following:

**Lemma (commutes):**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>s</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>i</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>b</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

(n) newadr commutes with send, newadr, and initbeh, but not with become.
(s) send commutes with send, become, and newadr, but not with initbeh.
(i) initbeh commutes with newadr, but not with send, become, and initbeh.
(b) become commutes with send, but not with initbeh, newadr, or become.

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

**Lemma (commutes-if):** If \( \vartheta \) commutes with \( e_0 \) and \( e_1 \), then it also commutes with \( \text{if}(z,e_0,e_1) \)

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

Using these basic principles we can prove more complex properties, the following theorem being the most obvious.
Theorem (commutes): Suppose that $e_0$ and $e_1$ are built up from $\forall$ using only the operations, if and let. Furthermore suppose every operation occurring in $e_0$ commutes with every operation occurring in $e_1$. Then

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

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

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

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

by hypothesis

$$\cong \text{let}\{x_0 := \text{let}\{z := \theta(y)\}R[z]\}\text{let}\{x_1 := e_1\}e$$

by (cong) and (letx)

$$\cong \text{let}\{z := \theta(y)\}\text{let}\{x_0 := R[z]\}\text{let}\{x_1 := e_1\}e$$

by (let.dis)

$$\cong \text{let}\{z := \theta(y)\}\text{let}\{x_1 := e_1\}\text{let}\{x_0 := R[z]\}e$$

by the induction hypothesis and (cong)

$$\cong \text{let}\{x_1 := e_1\}\text{let}\{z := \theta(y)\}\text{let}\{x_0 := R[z]\}e$$

by the induction hypothesis and (cong)

$$\cong \text{let}\{x_1 := e_1\}\text{let}\{x_0 := \text{let}\{z := \theta(y)\}R[z]\}e$$

by (let.dis)

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

by (cong) and (letx)

\[\square\]

Note that the theorem fails in the case when the expressions contain app and $\lambda$ due to the possibility of divergence.

6 Proving Expression Equivalence

In this section we develop methods for proving expressions observationally equivalent. The methods are a variation on the techniques used to prove the (cit) theorem (a form of context lemma) in (Mason and Takcott, 1991).

Before outlining the methods we motivate the difficulty of establishing equivalences. A relatively simple case is $\text{succ}(0) \cong 1$. It is simple for two reasons: there are no free variables occurring, and only one step of computation separates $\text{succ}(0)$
and 1. By the definition of $\simeq$, we need to establish

$$\text{Obs}(O[\text{succ}(0)]) = \text{Obs}(O[1])$$

for all observing contexts $O$. To establish this, we construct, for each computation path $\pi_0 \in \mathcal{F}(O[\text{succ}(0)])$, a $\pi_1 \in \mathcal{F}(O[1])$ such that $\text{obs}(\pi_0) = \text{obs}(\pi_1)$ and conversely. We call such a construction a path correspondence. Informally, the path correspondence is constructed as follows. First consider how from a path $\pi_0$ in $\mathcal{F}(O[\text{succ}(0)])$ we obtain a path in $\mathcal{F}(O[1])$. At each point in $\pi_0$ where the $\text{succ}(0)$ is reduced to 1, we remove this step, giving path $\pi_1$. Describing this operation in detail requires care, for there could be other independent occurrences of the computation of subterm $\text{succ}(0)$ in $O$ which are not to be removed. We then can show that $\pi_1$ is a computation for $O[1]$, with the same observable outcome and same fairness property as $\pi_0$. The converse construction is similar, except steps computing $\text{succ}(0)$ are inserted into $\pi_0$ each time the 1 first appears in a reduction context.

These two expressions differ by only one step of computation; in general they could differ by more than one step, and could both reduce to a common reduct rather than one reducing to the other, e.g. $\text{pred}(\text{succ}(1)) \simeq \text{succ}(0)$. The complication arising from this case is the two-step execution of $\text{pred}(\text{succ}(1))$ may be interleaved with computations of other actors and a local replacement thus not possible. To solve this problem the computation path is put in an equivalent canonical form with both steps adjacent. In general we place as many steps of an individual actor adjacent as possible.

A complication also arises in proving equations that may contain free variables, for instance $\text{1st}(\text{pr}(x, 0)) \simeq x$. Such expressions may be self-substituted: if $\text{1st}(\text{pr}(x, 0))$ occurs in the local context $\text{app}(\lambda y.\text{app}(y, y), \lambda x.\cdot)$, upon computing the free $x$ in $\text{1st}(\text{pr}(x, 0))$ will be replaced with $\lambda x.\text{1st}(\text{pr}(x, 0)))$. This means the necessary replacements are not flat but may be nested. A notion of generalized hole is introduced to account for this nesting.

In general we give methods for establishing three different varieties of expression equivalence; the above informal description describes only the first variant. The three variants are as follows.

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

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

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

We provide examples of the use of these techniques by using them to establish the equational laws stated in §4.
6.1 The General Method

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

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

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

1. For the common reduct case we define templates by adding a single hole, $\circ$, for expressions. We call this hole an expression hole.

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

3. For the equivalence of reduction contexts we need an entirely new kind of hole, $\phantomcirc$, for reduction contexts. We call it a reduction context hole. Note that occurrences of holes will be filled by reduction contexts and are not to be confused with redex holes. As far as we are aware the introduction of holes that are filled by contexts is completely novel.

For each variant, syntactic classes $X$ are annotated with the signs of the sorts of holes they contain: $X$ for expression holes; $\bigcirc X$ for expression and lambda abstraction holes; and $\bigcirc X$ for reduction context holes. We prefix the names of these classes by $E$, $\text{LE}$, or $\text{R}$, respectively. Thus $E$-expressions are expression templates with holes for expressions, $\bigcirc \mathbb{E}$ is the set of $E$-expressions, and we let $\circ e$ range over $\bigcirc \mathbb{E}$. A similar convention holds for the other syntactic classes and hole types.

The idea underlying the construction of a path correspondence to establish equivalence is the same for each of the three cases. It relies on the ability to localize differences in computations as multi-steps (§2.2.3), and to use holes to formalize the aspects of computation that are independent of the local differences. Consider the case of proving expressions equivalent using templates with expression holes. We consider fair computation paths starting from an $E$-configuration with holes filled by one of the expressions, say $e_0$. For each such path, $\pi_0$, we show how to obtain a sequence of $E$-configurations satisfying two conditions. The first is that filling the
Actors Computation

holes in the sequence of E-configurations with \( \sigma_0 \) (and filling in transition labels) yields \( \tau_0 \). The second is that filling the holes in the sequence of E-configurations with \( \sigma_1 \) (and filling in transition labels) yields a fair computation path with the same observation. The other two cases are simple variations on this idea.

One of the keys ideas in uniform computation is to ensure that transitions commute with hole filling; except of course when the hole is touched, i.e. information about the contents of the hole is required to carry out the step. Consider the schematic redex \( \text{app}(\lambda x. e, v) \). We need a notation that allows us to carry out this reduction in such a way that filling the hole and then reducing gives the same result as reducing and then filling the hole. For this purpose we associate with each hole a substitution to be applied when the hole is filled. The domain of the substitution also determines the variables of an expression that are trapped at the hole. This localizes trapping and allows renaming of lambda-variables even in the presence of holes (which is not the case for traditional notions of expression context). A detailed development of this notation can be found in (Talcott, 1991; Talcott, 1993a). We use \( \sigma[\varphi] \) to denote an expression hole with associated substitution \( \varphi \) (which may in turn have expressions holes in its range), a similar notation holds for the other classes of holes: \( \tau[\varphi] \) for abstraction holes, and \( \circ[\varphi] \) for reduction context holes.

To simplify definitions of syntactic classes we treat \( \text{app} \) on a par with elements of \( \mathbb{F}_2 \). We use \( \Theta_n \) for syntactic operations of arity \( n \), and \( \Theta^n_n \) to indicate the operations of the extended language (i.e. \( \Theta_0 \) extended to include event). Thus:

**Definition (\( \Theta_n \Theta^n_n \))**:

\[
\Theta_2 = \mathbb{F}_2 \cup \{ \text{app} \} \quad \Theta_n = \mathbb{F}_n \quad \text{for } n \neq 2
\]

\[
\Theta^n_0 = \Theta_0 \cup \{ \text{event} \} \quad \Theta^n_n = \Theta_n \quad \text{for } n \neq 0
\]

As the last technical detail, we make precise the sense in which we are able to localize differences in computations as multi-steps. We first define the notion of **thread fragment**, and then show that any family of disjoint thread fragments in a computation path can be regroup as multi-step without effecting the observation made of the path.

A thread fragment, \( I \), at \( a \) in \( \pi \) is a finite subsequence of exec transitions of \( \pi \) with focus actor \( a \) or a clone of \( a \) created by a become such that any gaps in the sequence are transitions with some other focus, or at \( a \) after a new message receipt.

**Definition (thread fragment)**: Let

\[
\mathcal{F}(\kappa) = \left\{ \kappa_i \xrightarrow{\delta_i} \kappa_{i+1} \mid i < \infty \right\}, \text{ and } \pi \in \mathcal{F}(\kappa),
\]

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

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

Then \( I \) is a **thread fragment** at \( a \) in \( \pi \) if

1. \( k_a = \text{exec}(a, cv) \) for some \( cv \),
2. \( L(I, \pi) \) contains no \( \text{exec} \), \( \text{in} \) or \( \text{out} \) and
3. \( L(I, \pi) \) is a computation for \( \left\langle \alpha_{\text{exec}}(a) \mid \emptyset \right\rangle \).

Condition (3) makes explicit that a thread fragment is essentially running the focus actor in a configuration with only itself. With no receives only that actor or its become clones can execute.
Theorem (infinite-macro-steps): Let $\pi = \{\kappa_i \xrightarrow{\kappa_{i+1}} \} i < \mathcal{I} \in \mathcal{F}(K)$. Let $I_j = [i_{j,0}, \ldots, i_{j,n_j}]$ for $j < J \leq \infty$ be a family (possibly infinite) of thread fragments in $\pi$ such that
(a) if $j < j'$ then $i_{j,0} < i_{j',0}$, and
(b) if $j \neq j'$ the $I_j$ and $I_j'$ have empty intersection.
Then there is a bijection, $\xi$, on $\mathcal{I}$ such that letting $\pi' = \text{Cfig}(\kappa_0, [\xi(i) \mid i < \mathcal{I}])$
(recall the definition of Cfig from §2.2.3)
(1) $\pi' \in \mathcal{F}(\kappa_0)$
(2) $\xi(i_{j,k+1}) = \xi(i_{j,k}) + 1$ for $j \in J$, and $0 \leq k < n_j$.
Part (2) says that in $\pi'$ the thread fragments of $\pi$ marked by $I_j$ for $j \in J$ occur as multi-steps, that is, with no interleaved computation steps. Note that $\text{obs}(\pi') = \text{obs}(\pi)$.
Proof: $\xi$ is computed by induction on the index set, permuting the exec steps of each successive fragment across interleaved steps the obvious way. By the definition of thread fragment and the disjointness requirement, we see that permutations only involve moving exec steps before steps with distinct focus. Hence the resulting sequence of labels defines a computation. Also the enabledness is not affected by such permutations (except possibly enabling a transition earlier). All transitions that occur in $\pi$ also occur in $\pi'$ this means that fairness is also preserved. \[ \Box \]

6.2 Common Reduct Case

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

6.2.1 E-Syntax

As mentioned above, syntactic classes, $X$, with expression holes are indicated by the mark $^0X$. Metavariables ranging over these classes are indicated by the same mark, and we prefix the names of these classes by $E$. Thus we have E-expressions where $^0\alpha$ ranges over $^0\alpha$, E-configurations where $^0\alpha$ ranges over $^0\alpha$, etc. We first define the E- analogs of expression, value expression, and value substitution.

Definition ($^0\alpha$ $^0\nu$ $^0\beta$):

\[
^0\nu = \lambda \alpha \cup \mathcal{X} \cup \alpha \cdot ^0\alpha \cup \text{pr}(^0\nu, ^0\nu) \\
^0\alpha = ^0\nu \cup \Theta_n(\Theta_n) \cup [^0\beta] \\
^0\beta = \mathcal{X} \rightarrow (^0\nu)
\]

As before, $\lambda$ is the only binding operator, and free variables of E-expressions are defined as follows:

Definition ($\text{FV}^0(\alpha)$ $\text{FV}^0(\beta)$):

\[
\text{FV}^0(\alpha) = \begin{cases} 
\text{FV}^0(\beta) & \text{if } \alpha = [\beta] \\
\{\alpha\} & \text{if } \alpha \in \mathcal{X} \\
\text{FV}^0(\alpha) \setminus \{x\} & \text{if } \alpha = x.\alpha_0 \\
\text{FV}^0(\alpha_1) \cup \ldots \cup \text{FV}^0(\alpha_n) & \text{if } \alpha = \theta(\alpha_1, \ldots, \alpha_n) \text{ and } \theta \in \Theta_n
\end{cases}
\]

G. Agha and others
\[ FV(\sigma) = \bigcup_{x \in \text{Dom}(\sigma)} FV(\sigma(x)) \]

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

**Definition** (\( \sigma_1 \odot \sigma_2 \)): Substitution is extended to E-expressions as follows:

\[
\sigma_1 \odot \sigma_2 = \begin{cases} 
\sigma_1 & \text{if } \sigma_1 \in \text{Dom}(\sigma) \\
\sigma_2 & \text{if } \sigma_2 \in \text{Dom}(\sigma) \\
\lambda z.\sigma_0[\sigma_1(Dom(\sigma) - \{z\})] & \text{if } \sigma_1 = \lambda z.\sigma_0 \text{ and } z \notin FV(\sigma)\\
\theta(\sigma_0[\sigma_1], \ldots, \sigma_n[\sigma_1]) & \text{if } \sigma_1 = \theta(\sigma_0[\sigma_1], \ldots, \sigma_n[\sigma_1]) \text{ and } \theta \in \Theta_n^e
\end{cases}
\]

\( \sigma_1 \odot \sigma_2 = \lambda x \in \text{Dom}(\sigma_2).\sigma_2(x)[\sigma_1] \)

As defined here substitution is a partial operation. Using renaming substitutions we can define a renaming in the usual way. We consider E-expressions (and entities containing them) to be equivalent if they differ only by a renaming. Thus, for any substitution we can always choose an \( \alpha \) variant so that substitution is defined. Note that such renaming is not possible in the case of traditional contexts where holes have no associated substitution (c.f. (Talcott, 1993a)).

Expression hole filling is defined by induction on the structure of \( \sigma \). We let \( \sigma[0 := e] \) be the result of filling expression holes in \( \sigma \) with \( e \). Like substitution, we avoid capture of free variables in \( e \) by lambda binding. All capture is done at hole occurrences by the associated substitution.

**Definition** (\( \sigma[0 := e] \)):

\[
\sigma[0 := e] = \begin{cases}
\sigma & \text{if } e = \sigma[0] \\
\theta(\sigma_1[0 := e], \ldots, \sigma_n[0 := e]) & \text{if } e = \theta(\sigma_1, \ldots, \sigma_n) \text{ and } \theta \in \Theta_n^e \\
\lambda x.\sigma[0 := e] & \text{if } e = \lambda x.\sigma \text{ and } x \text{ not free in } \sigma
\end{cases}
\]

\( \sigma[0 := e] = \lambda x \in \text{Dom}(\sigma).\sigma(x)[0 := e] \)

The following example illustrates hole filling and variable scoping. Let

\[
\sigma = \{y := \sigma\} \\
\sigma = \lambda z.\sigma \[\sigma\] \\
e = \theta(x, y)
\]

then

\[
\sigma[0 := e] = \lambda w.\sigma[0 := e][0 := e] \quad \text{note the change in bound variable}
\]

\[
= \lambda w.\sigma[0 := e][0 := e] \\
= \lambda w.\sigma[0 := e][0 := e] \quad \text{by the change in bound variable}
\]

\[
\sigma[0 := e] = \{y := \sigma[0 := e]\}
\]

\[
= \{y := \lambda w.\sigma[0 := e][0 := e]\}
\]
$\circ[e := e] = \lambda z. (e[\sigma][o := e])$

$= \lambda z. (e[\sigma][o := e])$

$= \lambda z. \theta(x, \lambda w. \text{if } w(\theta(x), 0), z))$

The following lemma is the key to developing a notion of uniform computation.

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

$\circ[e][o := e'] = \circ[e][o := e'][\sigma[o := e']]$

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

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

$\circ[e][\sigma][o := e']$

$= (\lambda z. \circ[e_0][\sigma][o := e'])[o := e']$

$= \lambda z. (\circ[e_0][\sigma][o := e'])$ by hygiene assumptions

$= \lambda z. (\circ[e_0][o := e'][\sigma[o := e']])$ by the Induction Hypothesis

$= (\lambda z. \circ[e_0][o := e'])[\sigma[o := e']][o := e']$ by hygiene assumptions

$= \circ[e][o := e'][\sigma[o := e']]$

If $\circ[e] = o[\sigma']$ then

$\circ[e][\sigma][o := e']$

$= o[\sigma'][\sigma][o := e']$

$= (o[\lambda z \in \text{Dom}(\sigma') \circ[e'][z]][\sigma][o := e'])[o := e']$

$= e'[\lambda z \in \text{Dom}(\sigma') \circ[e'][z][\sigma][o := e']]$ by the Induction Hypothesis

$= e'[\circ[e'][o := e'] \circ \circ[\sigma][o := e']]$ by hygiene assumptions

$= \circ[e][o := e'][\sigma[o := e']]$

Next we define analogs of redex and reduction context.

**Definition (\circ[R] \circ[R_{\text{rdx}}]):**

$\circ[R] = (\Theta R) \cup \Theta_{m+n+1} (\circ[R], \circ[R], \circ[R])$

$\circ[R_{\text{rdx}}] = \Theta^n (\circ[R])$

Note that E-reduction contexts possess two types of holes, consequently we must disambiguate the process of hole filling. Note that the unique occurrence of a redex hole is not adorned with a substitution, consequently the process of filling the
redex hole, \( \square \), with the E-expression, \( ^0e \), remains unchanged, and we denote it by \( ^0R[\square] := ^0e \).

In the case of multiple hole filling we write \( ^0R[\circ := e_0][\square := e] \) for the result of filling the expression holes with \( e_0 \), and the redex hole with \( e \).

**Lemma (E-properties):**

1. \( ^0R[\circ := e_0][\square := e] = ^0R[\circ := e][\square := e_0] \)
2. Filling an E-expression, E-reduction context, or E-redex with an expression yields an expression, reduction context, or redex, respectively.

### 6.2.2 E-Expression Decomposition

We now give a decomposition lemma for E-expressions: An E-expression \(^0e\) is either an E-value (element of \( ^0V \)) or it can be decomposed uniquely into an E-reduction context filled with either an E-redex or with an expression hole.

**Lemma (E-expression decomposition):**

1. \( ^0e \in ^0V \), or
2. \( (\exists ! ^0R, ^0r)(^0e = ^0R[\square := ^0r]) \), or
3. \( (\exists ! ^0R, ^0r)(^0e = ^0R[\circ := ^0r]) \)

**Proof:** An easy induction on the structure of \(^0e\). We consider two example cases.

First, suppose \( ^0e = \circ[\mathcal{R}] \). Then we have case (2) with \( ^0R = \square \). Second, suppose \( ^0e = \theta(\circ \mathcal{E}_2, \ldots, \circ \mathcal{E}_9) \). If \( \circ \mathcal{E}_i \in ^0V \) for \( 1 \leq i \leq 9 \), then we have case (1) with \( ^0R = \square \) (and \( ^0r = ^0e \)). If \( \circ \mathcal{E}_i \not\in ^0V \) for some \( 1 \leq i \leq 9 \), assume \( k \) to be the least such \( i \). Then by the induction hypothesis, \( \circ \mathcal{E}_k \) decomposes either as (i) \( ^0R[\square := ^0r] \), or as (ii) \( ^0R[\circ := \circ[\mathcal{R}]] \). Taking \( ^0R = \theta(\circ \mathcal{E}_1, \ldots, \circ \mathcal{E}_{k-1}, ^0R', \circ \mathcal{E}_{k+1}, \ldots, \circ \mathcal{E}_9) \) we obtain the desired decomposition of \( ^0e \).

### 6.2.3 E-Configurations

An E-configuration, \(^0\alpha\), is formed in the same manner as a simple configuration, using E-expressions and E-values instead of simple expressions and values.

**Definition \((^0\alpha)\):**

\[
^0\alpha = \left\langle \begin{array}{c}
\overline{^0\beta} \\
\square
\end{array} \right| \overline{^0M} \right\rangle^0
\]

where

\[
^0\beta = \overline{^0\delta} \Rightarrow ^0\beta \Rightarrow
\]

\[
^0\alpha = (\overline{^0V}) \cup \{ \overline{^0\alpha} \} \cup \{ ^0\beta \}
\]

\[
^0\text{M} = ^0\alpha \leftarrow \overline{^0V}
\]

and the constraints specified in the definition of actor configurations in §2 are satisfied.† We let \( ^0\alpha \) range over \(^0\text{W} \), and \( ^0\beta \) range over \(^0\beta \). Filling expression holes

† The only condition whose meaning is altered in this general setting is (2), where the
of an E-configuration, E-actor map, E-actor state, E-multiset of messages, and E-messages is defined in the obvious manner. Let $^0X$ stand generically for an element of one of these E-syntactic classes, then we define $^0X[\alpha := e]$ as follows:

**Definition ($^0X[\alpha := e]$):**


eq^0X[\alpha := e] = 
\begin{cases} 
\langle \alpha[\alpha := e] \mid \gamma[\alpha := e] \rangle & \text{if } ^0X = \langle \alpha \mid \gamma \rangle \\
\lambda x \in \text{Dom}(\alpha).\alpha(x)[\alpha := e] & \text{if } ^0X = \alpha \\
(\lambda x.\gamma)[\alpha := e] & \text{if } ^0X = (\lambda x.\gamma) \\
\gamma[e][\alpha := e] & \text{if } ^0X = \gamma \\
(\gamma_\lambda) & \text{if } ^0X = (\gamma_\lambda) \\
\gamma[m][\alpha := e] & \text{if } ^0X = \gamma_m \\
\gamma_\nu[\alpha := e] = \nu_1[\alpha := e] & \text{if } ^0X = \gamma_\nu[\nu_1] 
\end{cases}

An E-configuration, $\alpha$, is **closing for** $e$ if $^0\alpha[\alpha := e]$ is a closed configuration. Dually an expression $e$ is a **valid filling** on an E-configuration, $\alpha$, if $^0\alpha[\alpha := e]$ is a configuration. As for atoms and variables, the notion of communicable value remains unchanged and we do not introduce new notation for these. In particular, although messages may have holes, a message with a hole can effectively be ignored. This is because holes in E-values must occur inside $\lambda$'s and hence filling these holes cannot yield communicable values or actor addresses. Thus a message with a hole can never be processed. The next lemma expresses the fact that closing E-configurations make just the same observations as simple observing contexts.

**Lemma (ocx):** $e_0 \models e_1$ iff $\text{Obs}(\alpha[\alpha := e_0]) = \text{Obs}(\alpha[\alpha := e_1])$ for all $\alpha$ that close $e_0, e_1$.

**Proof**: The backward implication is easy to see, since $\emptyset$ is (with suitable translation to account for trapping at holes rather than at lambdas) a subset of $^0\alpha$.

The idea for the proof of the forward implication is to define for each configuration context $^0\alpha$, an observing context $O$ whose computations give rise to the same set of observations. In fact $O$ evolves to $\alpha$ in a finite number of steps. For an E-expression $e$ we define $e^*$ to be the result of recursively replacing decorated holes $e[\sigma]$ by applications $\text{app}(\ldots \text{app}(\lambda x_1 \ldots \lambda x_n.\alpha, \sigma(x_1)^*), \ldots, \sigma(x_n)^*)$ where $\{x_1, \ldots, x_n\} = \text{Dom}(\sigma)$. Let $\alpha = \langle \alpha \mid \mu \rangle$, let $A = \{a_i \mid i < n\} = \text{Dom}(\alpha)$, and define $O = \langle \{\nu_i\}_{a} \mid \emptyset \rangle$ where $\hat{a} \notin A$, and $\nu_{\alpha}$ is constructed as follows. Let $E = \{i < n \mid (\exists e_i.\alpha(a_i) = [\nu_i])\}$, and let $n_E$ be the cardinality of $E$.

$I = \{i < n \mid (\exists e_i.\alpha(a_i) = [\nu_i])\}$.

$B_i(a_0, \ldots, a_{n-1}) = \nu_i^*$, if $\alpha(a_i) = [\nu_i]$.

$B_i(a_0, \ldots, a_{n-1}) = \lambda a.\text{seq}(\text{send}(a, 0), \nu_i^*)$, if $\alpha(a_i) = [\nu_i]$.

$\mu = \langle x_j \leftarrow \nu_j^* \mid j < m \rangle$

Define

$$W_{\alpha} = \text{rec}(\lambda b.\lambda k.\lambda m.\text{if}(\text{seq}(k, 0), \text{seq}(\text{send}(z_j, (\nu_j^*)_{j<k}), \text{become}(b(k-1))))$$

free variables of any hole occurrences (namely the free variables in the range of the associated substitution) must be taken into consideration.
Actors Computation

\( e_\alpha = \text{let} \{ a_i := \text{newadr()} \}_{i \leq n} \)
\( \text{seq}(\text{initbeh}(a_i, B_i(a_0, \ldots, a_{n-1}))_{i \in I \cup E}, \)
\( \text{send}(a_i, \hat{a})_{i \in E}, \)
\( \text{become}(W_\alpha(n_E)) \)

Now, we claim that any computation of \( c_\alpha[o := e] \) has a corresponding computation (with same observations) of \( O[e] \) obtained by accepting all the startup messages, sending and accepting the acts, and completing the computation of the initializing actor (which can then be ignored). Conversely any computation of \( O[e] \) has a corresponding computation of \( c_\alpha[o := e] \) obtained by ignoring the finite amount of initializing activity. A more detailed proof can be given along the lines of the proof of the theorem (fun-red-eq) below. \( \square \)

6.2.4 \( E \)-Reduction

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

**Definition** (\( \rightarrow \)):

\( \beta \text{-v} \quad R[\square := \text{app}(\lambda x. \circ e, \circ v)] \rightarrow R[\square := \circ e[x := \circ v]] \)

\( \text{br} \quad R[\square := \text{br}(\circ v_1, \circ v_2, \circ v)] \rightarrow \begin{cases} R[\square := \circ v_1] & \text{if } \circ v \in \circ \forall \setminus \{ \text{nil} \} \\ R[\square := \circ v_2] & \text{if } \circ v = \text{nil} \end{cases} \)

\( \text{eq} \quad R[\square := \text{eq}(\circ v_0, \circ v_1)] \rightarrow \begin{cases} R[\square := \text{t}] & \text{if } \circ v_0 = \circ v_1 \in \text{At} \\ R[\square := \text{nil}] & \text{if } \circ v_0, \circ v_1 \in \text{At} \text{ and } \circ v_0 \neq \circ v_1 \end{cases} \)

**Definition** (\( \rightarrow \)):

\( \text{fun} : a \quad \circ e \rightarrow \circ e' \Rightarrow \begin{cases} \circ e, [\circ e]_a \mid \circ \gamma & \rightarrow \begin{cases} \circ e, [\circ e']_a \mid \circ \gamma' \end{cases} & \text{if } a \text{ fresh} \\ \circ e, [\circ e]_a \mid \circ \gamma & \rightarrow \begin{cases} \circ e, [\circ e']_a \mid \circ \gamma' \end{cases} & \text{if } a \text{ fresh} \end{cases} \)

\( \text{new} : a, a' \quad \begin{cases} \circ e, [\circ R[\square := \text{newadr()}]]_a \mid \circ \gamma & \rightarrow \begin{cases} \circ e, [\circ R[\square := a']]_a, (\circ ? a')_a' \mid \circ \gamma' \end{cases} & a \text{ fresh} \\ \circ e, [\circ R[\square := a']]_a, (\circ ? a')_a' \mid \circ \gamma' & \rightarrow \begin{cases} \circ e, [\circ R[\square := \text{nil}]]_a, (\circ ? a)_a' \mid \circ \gamma' \end{cases} & a' \text{ fresh} \end{cases} \)

\( \text{bec} : a, a' \quad \begin{cases} \circ e, [\circ R[\square := \text{become}(\circ v)]_a \mid \circ \gamma & \rightarrow \begin{cases} \circ e, [\circ R[\square := \text{nil}]]_a', (\circ v)_a' \mid \circ \gamma' \end{cases} & a' \text{ fresh} \\ \circ e, [\circ R[\square := \text{nil}]]_a', (\circ v)_a' \mid \circ \gamma' & \rightarrow \begin{cases} \circ e, [\circ R[\square := \text{send}(\circ v_0, \circ v_1)]_a \mid \circ \gamma' \end{cases} & a' \text{ fresh} \end{cases} \)

\( \text{send} : a, m \quad \begin{cases} \circ e, [\circ R[\square := \text{send}(\circ v_0, \circ v_1)]_a \mid \circ \gamma' \end{cases} & \rightarrow \begin{cases} \circ e, [\circ R[\square := \text{send}(\circ v_0, \circ v_1)]_a \mid \circ \gamma' \end{cases} & a' \text{ fresh} \end{cases} \)
6.2.5 E-Uniform Computation

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

**Definition (E-hole touching):** Let \( \kappa = \left\langle \alpha, [\mu] \right\rangle \). We say that \( \kappa \) touches a hole at \( a \) if \( \kappa(a) = [\mu] \) for some \( \mu \).

We say that a transition \( \kappa \xrightarrow{t} \kappa' \) touches a hole relative to a decomposition \( \kappa = \kappa[l := e] \) if \( l \) has focus \( a \) and \( \kappa \) touches a hole at \( a \).

**Lemma (E-Uniform Computation):**

1. If \( \kappa \xrightarrow{t} \kappa' \) then \( \kappa[l := e] \xrightarrow{t} \kappa'[l := e] \) for any valid filling expression \( e \).
2. If \( \kappa \) has no transition with focus \( a \) (and \( a \) is an actor of \( \kappa \)), then either \( \kappa \) touches a hole at \( a \) or \( \kappa[l := e] \) has no transition with focus \( a \) for any valid filling expression \( e \).
3. If \( \kappa \xrightarrow{l} \kappa' \) and \( \kappa = \kappa[l := e] \), then either the transition touches a hole or we can find \( \kappa' \) such that \( \kappa' = \kappa'[l := e] \) and \( \kappa \xrightarrow{l} \kappa' \).

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

**Proof (2):** Assume \( \kappa = \left\langle \alpha, [\mu] \right\rangle \) has no transition with focus \( a \), and \( \kappa \) does not touch a hole at \( a \). Then one of the following holds:

1. \( \kappa(a) = (\?a) \)
2. \( \kappa(a) = (\?v) \) and \( \mu \) contains no messages deliverable to \( a \)
3. \( \kappa(a) = [\gamma] \)
4. \( \kappa(a) = [\beta] \) where \( \beta \) is the address of an uninitialized actor created by \( a \)
5. \( \kappa(a) = [\gamma] \) where \( \gamma \) is a non-actor redex that is stuck.

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

**Proof (3):** Assume \( \kappa = \left\langle \alpha, [\mu] \right\rangle \xrightarrow{l} \kappa' = \left\langle \alpha', [\mu'] \right\rangle \), \( \kappa = \kappa[l := e] \), the transition does not touch the hole. Thus \( \kappa = \left\langle \alpha, [\mu] \right\rangle \) where \( \alpha = \kappa[l := e] \) and \( \mu = \gamma[l := e] \). We want to find \( \kappa' \), \( \mu' \) such that \( \kappa \xrightarrow{l} \kappa' = \left\langle \alpha', [\mu'] \right\rangle \), \( \alpha' = \kappa'[l := e] \), and \( \mu' = \gamma'[l := e] \). Since we are considering closed configurations there are no 1/o transitions. Thus, we need to consider only two cases: \( \text{recv} \) transitions and \( \text{exec} \) transitions. We split the \( \text{exec} \) transitions into functional and actor primitives.

**Receive:** \( l = \left\langle \text{recv} : a, cv \right\rangle,  \left\langle a \iff cv \right\rangle \in \mu \), and \( \alpha(a) = (v) \). Thus \( \alpha(a) = (v) \)
with \( v = v_0[0 := e] \). Thus we let \( \alpha' = \alpha\{ a := [\text{app}(v, cv)] \} \), and \( \gamma = \gamma' \cup \{ a := cv \} \).

**Execution-lambda:** \( l = \langle \text{fun} : a \rangle \), \( \alpha(a) = [R[\varnothing := r]] \) and \( r \xrightarrow{\lambda} e' \). Thus \( \alpha(a) = [\gamma[\varnothing := \gamma']] \) with \( R = \gamma[\varnothing := e] \), and \( r = \gamma'[\varnothing := e] \). Thus we want to find \( \gamma' \) such that \( r \xrightarrow{\lambda} e' \). Then \( \alpha' = \alpha\{ a := [\gamma[\varnothing := \gamma']] \} \) and \( \gamma' = \gamma \). If \( r = \text{app}(\lambda x. e_0, v) \) (\( z \) chosen fresh), then \( e' = e_0[z := v] \), and \( r = \text{app}(\lambda x. e_0, v) \) where \( e_0 = e_0[\varnothing := e] \) and \( v = v[\varnothing := e] \). Take \( \gamma' = e_0[\varnothing := v] \) and use (fil-subst). If \( r = \text{eq}(v_0, v_1) \), then \( \gamma = \text{eq}(v_0[\varnothing := v_1]) \) where \( v_j = v_j[\varnothing := e] \) for \( j < 2 \). \( e' \) is \( t \) or \( \text{nil} \) and we may take \( \gamma' = e' \). The remaining cases are similar.

**Execution-actor:** If \( l = \langle \text{send} : a \rangle \), then \( \alpha(a) = [R[\varnothing := \text{send}(v_0, v_1)]] \alpha'(a) = [R[\varnothing := \text{send}(v_0, v_1)]] \), \( \mu' = \mu \cup \{ v_0 \leftarrow v_1 \} \). Also \( \alpha(a) = [R[\varnothing := \text{send}(v_0, v_1)]] \) where \( R = \gamma[\varnothing := e] \), and \( v_j = v_j[\varnothing := e] \) for \( j < 2 \). Take \( \alpha' = \alpha\{ a := [\gamma[\varnothing := \text{nil}]] \} \) and \( \gamma' = \gamma \cup \{ v_0 \leftarrow v_1 \} \).

If \( l = \langle \text{become} : a, a' \rangle \), then \( a' \) is fresh, \( \alpha(a) = [R[\varnothing := \text{become}(v)]] \alpha'(a) = (v), \alpha'(a') = ([R[\varnothing := \text{become}(v)]] \), and \( \mu' = \mu \). Also \( \alpha(a) = [R[\varnothing := \text{become}(v)]] \) where \( R = \gamma[\varnothing := e] \), and \( v = v[\varnothing := e] \). Take \( \alpha' = \alpha\{ a := (v), a' := [R[\varnothing := \text{nil}]] \} \) and \( \gamma' = \gamma \).

If \( l = \langle \text{new} : a, a' \rangle \), then \( a' \) is fresh, \( \alpha(a) = [R[\varnothing := \text{newadr}(v)]] \alpha'(a) = [R[\varnothing := \text{newadr}(v)]] \), \( \alpha'(a') = (\gamma_0) \), and \( \mu' = \mu \). Also \( \alpha(a) = [R[\varnothing := \text{newadr}(v)]] \) where \( R = \gamma[\varnothing := e] \). Take \( \alpha' = \alpha\{ a := [R[\varnothing := \text{nil}]], a' := (\gamma_0) \} \) and \( \gamma' = \gamma \).

If \( l = \langle \text{init} : a, a' \rangle \), then \( \alpha(a) = [R[\varnothing := \text{initbeh}(a', v)]] \alpha'(a) = (\gamma_0) \), \( \alpha'(a') = (\gamma_0) \), \( \gamma_0 = (v) \), and \( \mu' = \mu \). Also \( \alpha(a) = [R[\varnothing := \text{initbeh}(a', v)]] \) where \( R = \gamma[\varnothing := e] \), and \( v = v[\varnothing := e] \), and \( \alpha'(a') = (\gamma_0) \). Take \( \alpha' = \alpha\{ a := [R[\varnothing := \text{nil}]], a' := (v) \} \) and \( \gamma' = \gamma \).

**O3 O2 uniform**

### 6.2.6 The Common Expression Reduct Theorem

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

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

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

**Proof:** Assume that for each closing \( \sigma \) there is \( e_{c,j} \) such that \( e_j[\sigma] \xrightarrow{\lambda} \ldots \xrightarrow{\lambda} e_{c,j} \) \( j < 2 \), uniformly, and either \( e_{c,j} \) and (uniformly) stuck for \( j < 2 \), or \( e_{c,0} = e_{c,1} \).

In either case we call \( e_{c,j} \) the common reduct. We want to show that \( e_0 \cong e_1 \). By \( \text{(oes)} \) it is sufficient to show that \( \text{Obs}(\kappa[\varnothing := e_0]) = \text{Obs}(\kappa[\varnothing := e_1]) \) for any \( \kappa \) that is a closing E-configuration for \( e_0 \) and \( e_1 \). To do this, we show that for any \( \pi_0 \in \mathcal{F}(\kappa[\varnothing := e_0]) \) \( \xrightarrow{\kappa_i} \kappa_{i+1} \) \( \epsilon \in \mathcal{E} \) we can find \( \pi_1 \in \mathcal{F}(\kappa[\varnothing := e_1]) \) such that \( \text{obs}(\pi_0) = \text{obs}(\pi_1) \). (The case with 0 and 1 interchanged is symmetric.)
Informally, by the uniformity property of computations, we see that replacing occurrences of \( \varepsilon_0 \) by \( \varepsilon_1 \) has no effect on a computation except where a hole is touched. Using (infinite macro-steps) we can localize non-uniform steps so that when a hole is touched, reduction to a common reduct occurs in a single multi-step (which involves no event transitions). Thus we may obtain a computation for \( \kappa_{[0]} := \varepsilon_1 \) by replacing occurrences of \( \varepsilon_0 \) by \( \varepsilon_1 \) and replacing multi-step transitions reducing \( \varepsilon_0[\sigma] \) to its common reduct by multi-step transitions reducing \( \varepsilon_1[\sigma] \) to its common reduct. To insure completeness/fairness of the result, we need to take account of the case where a hole \( \circ[\sigma] \) is exposed, but the multi-step for \( \varepsilon_0[\sigma] \) is trivial and hence does not appear as a transition. We do this by inserting the corresponding multi-step for \( \varepsilon_1[\sigma] \) at the point where the hole is first exposed. Such holes then effectively disappear, since they are either filled with a stuck expression or with the same expression. Now we make this informal argument more rigorous, by the following steps:

(1) We analyze the configurations occurring in \( \pi_0 \) and record occurrences of \( \varepsilon_0 \) in holes descending from \( \kappa \). This gives us decompositions \( \kappa_i[0] := \varepsilon_0 \) of \( \kappa_i \). In the cases where a hole is touched such that \( \varepsilon_0 \) is its common reduct, we fill that hole with \( \varepsilon_0 \) giving a new E-configuration \( \kappa'_i \) with one less hole, such that \( \kappa'_i[0] := \varepsilon_0 \) is \( \kappa_i \). This process of filling holes with common reducts continues until the transition \( \varepsilon \) is either uniform or touches a hole in which \( \varepsilon_0 \) is not its common reduct. We also record subsequences of transitions corresponding to uniform reduction of such occurrences of \( \varepsilon_0 \) to its common reduct.

(2) Using (infinite-macro-steps) we may assume that the path is expressed in terms of multi-step transitions such that the recorded subsequences of transitions corresponding to non-trivial reduction to a common reduct are single multi-steps. We also insert copies of \( \kappa_i \) for each hole that is filled with a common reduct, remembering the corresponding decomposition, and insert empty multi-steps between these copies. We also insert a copy of \( \kappa_i \) and a connecting empty multi-step for each hole that occurs in a reduction context that is not touched — because the occurrence of \( \varepsilon_0 \) is stuck, or because it is a value and placing it in the redex hole produces either a value or a stuck state.

(3) Form \( \pi_1 \) by filling the holes of \( \kappa_i \) with \( \varepsilon_1 \) and replacing multi-steps for \( \varepsilon_0 \) by corresponding multi-steps for \( \varepsilon_1 \). Note that empty multi-steps may expand to non-trivial reductions of occurrences of \( \varepsilon_1 \) to its common reduct. It is easy to see that \( \pi_1 \) is a computation path. The argument that it is complete and fair relies on the insertion of multi-steps, and uses the same case analysis that was used in the uniform computation lemma.

**Step (1)** We analyze and decompose \( \pi_0 \) to obtain

(i) for each \( i \leq n_i \), an integer \( n_i \) and a sequence of decompositions \( \kappa_{i,j} \) for \( j \leq n_i \) such that \( \kappa_i = \kappa_{i,j}[0] := \varepsilon_0 \) for \( j \leq n_i \) and such that \( \kappa_{i,n_i} \xrightarrow{j} \kappa_{i+1,0} \) uniformly, or the transition touches a hole in which \( \varepsilon_0 \) has non-trivial reduction to its common reduct. (We call this entering the hole.) \( n_i \) will be 0 except in the case of a hole touched in which \( \varepsilon_0 \) is its own common reduct. Then we fill that hole with the common reduct and redecompose.
(ii) The set $I$ of indices of transitions that enter holes

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

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

**Stage $(0,0)$** $J_0 = \emptyset$, and $J_0$ is the empty map. $\alpha_{0,0} = \alpha$.

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

1. $k$ is execution in a hole;
2. $k$ is uniform wrt $\alpha_{i,j}$
3. $k$ touches a hole and
   3.1 enters the hole
   3.2 does not enter the hole

**Case 1**: This case occurs if $i$ is an element of $M = J_i(m)$ for some $m \in I_i$. Thus
$n_i = j$ and $l_i = \langle \text{fun} : a \rangle$ for some $a$. We move to stage $(i+1,0)$ with $I_{i+1} = I_i$
$J_{i+1} = J_i$, $\mu_{i+1,0} = \mu_{i,j}$, and $\alpha_{i+1,0} = \alpha_{i,j}(a := [J'R[\mathcal{R} := \alpha_{e_m,k+1}]])$ where $\alpha_R$
and $\alpha_{e_m,k+1}$ are obtained as follows. Let $k$ be the index of $i$ in $M$. The hole is entered
at stage $(m,m)$ with $\alpha_{e_m,m}(a) = [J'R[\mathcal{R} := \alpha_{e_m,k+1}]]$. Let $\alpha_{e_m,0} = \alpha_0[\vartheta]$, let $n$ be
the length of $M$, and let $[\alpha_{e_m,k} \xrightarrow{\vartheta} \alpha_{e_m,k+1} \mid k < n]$ be the thread of computation
reducing $\alpha_0[\vartheta]$ to its common reduct $\alpha_{e_m,k}$. Note that $\alpha_{i,j}(a) = [J'R[\mathcal{R} := \alpha_{e_m,k}]]$.

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

**Case 3.1**: In this case $k = \langle \text{fun} : a \rangle$ for some $a$, $n_i = j$ and we move to stage
$(i+1,0)$ with $I_{i+1} = I_i \cup \{i\}$, $J_{i+1} = J_i \{i := M\}$, $\mu_{i+1,0} = \mu_i$, and $\alpha_{i+1} = 
\alpha_i(a := [J'R[\mathcal{R} := \alpha_{e_{i+1}}]])$ where $M$, $\alpha_R$, and $\alpha_{e_{i+1}}$ are obtained as follows. Let
$\alpha_{i,j}(a) = [J'R[\mathcal{R} := \alpha_{e_i,k}]]$, let $\alpha_{e_{i+1}} = \alpha_{e_i,k}$, and let $[\alpha_{e_k,k}[\vartheta] \xrightarrow{\vartheta} \alpha_{e_k,k+1} \mid k < n+1]$
be the thread of computation reducing $\alpha_0[\vartheta]$ to its common reduct $\alpha_{e_{i+1}}$. By
fairness, there is a sequence of indices $M = [i_k \mid k < n+1]$ with $i_0 = i$, $i_k < i_{k+1}$
for $k < n + 1$ such that $M$ is the macro step corresponding to the above lambda
reduction.

**Case 3.2**: We move to stage $(i,j+1)$ with $I_{i+1} = I_i$, $J_{i+1} = J_i$, $\mu_{i,j+1} = \mu_{i,j}$, and
$\alpha_{i,j+1} = \alpha_{i,j}(a := [J'R[\mathcal{R} := \alpha_{e_i,k}]])$ where $a$, $\alpha_R$, $\vartheta$ are obtained as follows.
$a$ is the focus of $l_i$, and $\alpha_{i,j}(a) = [J'R[\mathcal{R} := \alpha_{e_i,k}]]$, with $\alpha_0[\vartheta]$ equal to its common
reduct.

**Step (2)** The family $J(i)$ for $i \in I$ satisfies the conditions of (infinite-macro-
step). Hence we may assume that $\pi_0$ has the form

$$
[\alpha_{i,0}[\varnothing := e_0] \xrightarrow{\vartheta} \ldots \xrightarrow{\vartheta} \alpha_{i,n_i}[L_i \xrightarrow{\vartheta} \alpha_{i+1,0}[\varnothing := e_0]] \mid i < \infty
$$

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

**Actor Computation**

43
Step (3) We let $\pi_1$ be the path

$$[[\alpha_{i,0}[o := e_1] \overset{L_i}{\rightarrow} \ldots \overset{L_{i,n-1}}{\rightarrow} \alpha_{i,n}[o := e_1] \overset{L'_i}{\rightarrow} \alpha_{i+1,0}[o := e_1]] \mid i < \infty]$$

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

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

It remains to check that fairness has been preserved. Suppose some transition $l$ is enabled at stage $i$ in $\pi_1$. We have three cases:

**Receive:** Suppose $l$ is receipt of $<a \leftarrow c,v>$. Then $\alpha_{i}(a) = (\lambda x.\gamma)$ and hence $l$ is enabled in $\pi_0$ at stage $i$. If $l$ fires in $\pi_0$ at stage $i' \geq i$, then it also fires at this stage in $\pi_1$. Suppose $l$ never fires in $\pi_0$. Then by fairness, there is some $i' > i$ such that $\alpha_{i'}(a)$ is an executing state for $j \geq i'$. By construction $l$ is permanently disabled at $i'$ in $\pi_1$ as well.

**Uniform Execution:** Suppose $l$ is an execution step with focus $a$ where $\alpha_{i}(a) = [\alpha R[\alpha := \gamma]]$. Then $l$ is enabled in $\pi_0$ at $i$, it can not be disabled, and must occur in $\pi_0$ at some stage $i' \geq i$ and hence will occur in $\pi_1$ at that stage.

**Hole Touching:** Suppose $l$ is an execution step by $a$ with $\alpha_{i}(a) = [\gamma]$ where $\alpha e = \alpha R[\gamma := o[\gamma]]$. First assume $e_1[\gamma]$ reduces. If $\alpha e[\gamma] = e_0$ does not reduce, then by construction, the transition is taken in $\pi_1$ as soon as it is enabled. If $\alpha e[\gamma] = e_0$ reduces, then a transition will eventually be taken at $a$ in $\pi_0$, and the $l$ will be taken at the corresponding point in $\pi_1$. Suppose $e_1[\gamma]$ does not reduce. Then it must be a value, hence the common reduct. Hence the reduction of $e_0$ is enabled in $\pi_0$ and will eventually be taken. $\alpha R$ has the form $\alpha R[\gamma := o[\gamma]]$. If all the E-expressions $\alpha e$ are E-values, then $l$ must be reduction of the redex in $\alpha R$ and this is also enabled now in $\pi_0$. Otherwise consider decomposition of the first non-value element of $\alpha e$ and repeat this argument. Since we are now looking at a smaller E-expression, we eventually reach the point where the step enabled in $\pi_1$ corresponds to one in $\pi_0$ and hence will occur eventually.

6.2.7 The Proof of the Equivalence of Hanging and Lambda-Divergence

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

**Proof (hang-infin):** Assume $e_0, e_1 \in Hang \cup Infin$. We want to show that $e_0 \cong e_1$. Let $\gamma = \left< \alpha \mid \beta \right>$ be a closing E-configuration for $e_0$ and $e_1$. Assume $\pi_0 \in F(\gamma[0 := e_0]) = [\kappa_i \overset{L_i}{\rightarrow} \kappa_{i+1} \mid i \in \infty]$. We want to find $\alpha_{i}$ and $L_i$, such that $\kappa_i = \alpha_{i}[0 := e_0]$ where $\kappa_i = \left< \alpha_{i} \mid \gamma_{i} \right>$ and, letting $\pi_1 = [\kappa_i[0 := e_1] \overset{L'_i}{\rightarrow} \alpha_{i+1}[0 := e_1] \mid i \in \infty]$, we have $\pi_1 \in F(\gamma[0 := e_1])$ and $\pi_1 = \alpha_{i}$. (Actually, we let holes in $\pi_j$ be filled by any expression of the same class as $e_j$.) For the base case we have $\alpha_0 = \alpha$. Assume we have $\alpha_i$. Suppose $e_0 \in Hang$. Let $a$ be
the focus of \( i \). We first consider each \( a' \) other than \( a \) such that \( \alpha_i(a') = \circ R[\mathbf{0} := \circ[\sigma]] \). If \( e_1 \in \text{Hang} \) then we just insert any steps needed to reach the stuck state (we assume that they are already macroized for \( \pi_0 \)). If \( e_1 \in \text{Infin} \), then insert the step to reach the next element of its infinite sequence. Now we consider the transition label \( i \). If it does not touch a hole, then \( \alpha_{i+1} \) is given by the uniform transition lemma. Suppose \( \alpha_i(a) = \circ R[\mathbf{0} := \circ[\sigma]] \). Then \( \alpha_{i+1} \) has the same decomposition, just possibly different expressions (of the same class) filling the holes. \( \Box \)

6.2.8 The Proofs of the Actor Primitive Laws

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

**Proof (triv):** Suppose \( e_0, e_1 \) are taken from the pair \( \theta(\bar{x}), \text{seq}(\theta(\bar{x}), \text{nil}) \) where \( \theta \in \{\text{send, become, initbeh}\} \). Suppose \( i \) is an execution by \( a \) with \( \alpha_i(a) = [\circ R[\mathbf{0} := \circ[\sigma]]] \). We define \( \alpha_{i+1} \) for each form \( \theta(\bar{x}) \).

(\texttt{send}(x_0, x_1)) \( \alpha_{i+1} = \alpha_i\{a := [\circ R[\mathbf{0} := \text{nil}]]\} \). Also, \( \eta_{i+1} = \eta_i \cup \{<v_0 \leftarrow \circ v_1>\} \), where \( v_j \) is \( x_j[\sigma] \). (Recall that any holes appearing in messages must be inside lambdas and hence the message is ill-formed and can be ignored.)

(\texttt{become}(x_0)) \( \alpha_{i+1} = \alpha_i\{a := \circ v_0[\sigma], a' := [\circ R[\mathbf{0} := \text{nil}]]\} \) where \( a' \) is a new (anonymous) actor identifier.

(\texttt{initbeh}(x_0, x_1)) \( \alpha_{i+1} = \alpha_i\{a := \circ R[\mathbf{0} := \text{nil}]], \sigma(x_0) := \sigma(x_1)\} \). Since the step occurs in \( \pi_0 \), we may assume \( a' = \sigma(x_0) \in \text{Dom}(\alpha_i) \), and \( \alpha_i(a') = (?a) \).

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

\[
\text{seq(become}(v_0), \text{become}(v_1)), \quad \text{seq(become}(v_0), \text{nil}).
\]

Assume \( i \) is an execution by \( a \) with \( \alpha_i(a) = [\circ R[\mathbf{0} := \circ[\sigma]]] \). Define

\[
\alpha_{i+1} = \alpha_i\{a := (v_0[\sigma]), a' := [\circ R[\mathbf{0} := \text{nil}]]\}
\]

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

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

\[
\text{seq(initbeh}(v, v_0), \text{initbeh}(v, v_1)), \quad \text{seq(initbeh}(v, v_0), \text{stuck}).
\]

(The equivalence with \texttt{stuck} replaced by \texttt{bot} follows from (hang-inf).) Assume \( i \) is an execution by \( a \) with \( \alpha_i(a) = [\circ R[\mathbf{0} := \circ[\sigma]]] \). Define

\[
\alpha_{i+1} = \alpha_i\{a := [\circ R[\mathbf{0} := \text{bot}]], \sigma(v) := \circ[\sigma(v_0)]\}
\]
Since the step occurs in \( \pi_0 \), we may assume \( a' = \varphi(v) \in \text{Dom}(\alpha_i) \), and \( \alpha_i(a') = (?_a) \). Officially, in \( \pi_1 \), \( \alpha_{i+1}(a) \) should be \( [\alpha_{i+1}\{\varphi(v), v_1]\}] \). But, as in the proof of (hang-infin) we treat stuck expressions as indistinguishable. □

**Proof (commutes):** Suppose \( e_0, e_1 \) are taken from one of the pairs in the commutes lemma, except (n-i). Assume \( l_i \) is an execution by \( a \) with \( \alpha_i(a) = [\alpha_{i}\{\varphi(v), v_1]\}] \).

\( (s\cdot s) \)

\[ \text{let}\{x_0 := \text{send}(v_0, v_1)\}\text{let}\{x_1 := \text{send}(v_2, v_3)\}\]

\[ \text{let}\{x_1 := \text{send}(v_2, v_3)\}\text{let}\{x_0 := \text{send}(v_0, v_1)\} \]

Define

\[ \alpha_{i+1} = \alpha_i\{a := [\alpha_{i}\{\varphi(v), v_1]\}]\} \]

\[ v_j = v_j[\varphi] \text{ for } j < 4 \]

\[ \mu_{i+1} = \mu_i \cup \{v_0 \leftarrow v_1, v_1 \leftarrow v_3\} \]

\( (s\cdot n) \)

\[ \text{let}\{x_0 := \text{send}(v_0, v_1)\}\text{let}\{x_1 := \text{newadr}\}\]

Let \( a_1 \) be fresh and define

\[ \alpha_{i+1} = \alpha_i\{a := [\alpha_{i}\{\varphi(v), v_1]\}, a_1 := (?_a)\} \]

\[ v_j = v_j[\varphi] \text{ for } j < 2 \]

\[ \mu_{i+1} = \mu_i \cup \{v_0 \leftarrow v_1\} \]

\( (n\cdot n) \)

\[ \text{let}\{x_0 := \text{newadr}\}\text{let}\{x_1 := \text{newadr}\}\]

Let \( a_0, a_1 \) be fresh and define

\[ \alpha_{i+1} = \alpha_i\{a := [\alpha_{i}\{\varphi(v), v_1]\}, a_0 := (?_a) \} \]

\[ a_1 := (?_a) \]

\( (n\cdot n) \)

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

\[ \text{let}\{x_0 := \text{newadr}\}\text{let}\{x_1 := \text{initbeh}(v, v_1)\}\]

\[ \text{let}\{x_1 := \text{initbeh}(v, v_1)\}\text{let}\{x_0 := \text{newadr}\}\]
attempts the initialization first. Suppose \( \alpha_{i,j}(a) = [^2R]_i o[\sigma]] \), such that when \( \text{init} \text{beh}(v, v_1) \) is placed in the expression hole no transition is enabled. We insert a conditional multi-step which is the empty sequence of steps when filling holes with \( e_0 \) and the new add transition when filling holes with \( e_1 \). \( \alpha_{i,j+1}(a) \) is obtained by filling the hole with the stuck \text{init} \text{beh}. As before, we do not distinguish between different stuck expressions placed in the hole. Also, in \( \pi_1 \) there will be an extra uninitialized actor carried along untouched for each such insertion. These extra uninitialized actors can also be ignored. The remainder of the construction is as before. \( \square_{n-i} \)

6.3 Equivalence by Two Stage Reduction

There is one remaining equivalence to establish using common reducts:

\[(\texttt{if}\texttt{lam}) \; \lambda x.\texttt{if}(v, e_1, e_2) \cong \texttt{if}(v, \lambda x. e_1, \lambda x. e_2) \; \quad x \notin \text{FV}(v)\]

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

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

6.3.1 LE-Syntax

Syntactic classes \( X \) with both expression and lambda holes are indicated by the mark \( ^{0^b}X \), and we prefix the names of these classes by \( \text{LE-} \), thus we have \( \text{LE-} \) expressions, \( \text{LE-} \) configurations, etc. The defining clauses are as before with two exceptions: lambda holes are added to the clause generating values; and \( \text{app}[\nu_j][^{0^b}\sigma], \; ^{0^b}\nu \) is omitted from the class of \( \text{LE-} \) redexes. The latter exception is made in order to preserve the property that redexes reduce uniformly.

**Definition** \( ^{0^b}V, \; ^{0^b}E, \; ^{0^b}S, \; ^{0^b}R, \; ^{0^b}E_{\text{reda}} \):

\[
^{0^b}V = A + X \cup \lambda X. \; ^{0^b}E \cup \text{pr}(^{0^b}V, ^{0^b}V) \cup \nu \cap \{^{0^b}\nu\}
\]

\[
^{0^b}E = ^{0^b}V \cup \Theta^e_n(^{0^b}E_n) \cup \{\nu \cap ^{0^b}\nu\}
\]

\[
^{0^b}S = X \rightarrow ^{0^b}V
\]

\[
^{0^b}R = \{\square\} \cup \Theta^m_{m+n}(^{0^b}V^m, \; ^{0^b}E, \; ^{0^b}E_n)
\]

\[
^{0^b}E_{\text{reda}} = \Theta^e_n(^{0^b}E_n) - \text{app}[^{0^b}\nu_j][^{0^b}\nu], \; ^{0^b}V
\]

Note that lambda holes can occur in the range of a value substitution, and as arguments in redexes, except in the function position of an application. Using the double index convention, we write \( ^{0^b}e[^{0^b}\nu_j := \varphi_j] \) to indicate the simultaneous filling of the holes \( \nu_j \) with the corresponding lambdas \( \varphi_j \) from some previously specified family.
\{\varphi_j\}_{j \in J} of lambda abstractions. The definitions of substitution, free variables, and hole filling are entirely analogous to the expression hole case and we omit them.

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

**Lemma (LE-expression decomposition):**

\[ (0) \quad \mathcal{L}^e \in \mathcal{L}^v, \quad \text{or} \]
\[ (1) \quad \exists \mathcal{L}^R, \mathcal{L}^e [\mathcal{L}^e = \mathcal{L}^R[\mathcal{L}^R := \mathcal{L}^R]], \quad \text{or} \]
\[ (2) \quad \exists \mathcal{L}^R, \mathcal{L}^e [\mathcal{L}^e = \mathcal{L}^R[\mathcal{L}^R := \mathcal{L}^R]], \quad \text{or} \]
\[ (3) \quad \exists \mathcal{L}^R, \mathcal{L}^e, \mathcal{L}^e [\mathcal{L}^e = \mathcal{L}^R[\mathcal{L}^R := \text{app}(\varphi_j, \mathcal{L}^e)]] \]

6.3.2 LE-computation

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

**Definition (\(\mathcal{L}_\mathcal{K}\)):**

\[
\mathcal{L}_\mathcal{K} = \left\{ \mathcal{L}^\alpha \mid \mathcal{L}^\alpha \mathcal{M} \right\}_{\mathcal{X}}
\]

where

\[
\mathcal{L}^\alpha = \mathcal{L}^\alpha \xrightarrow{\mathcal{L}^\alpha} \mathcal{L}^\alpha
\]

\[
\mathcal{L}^\alpha \mathcal{M} = \mathcal{L}^\alpha \mathcal{V} \cup \mathcal{L}^\alpha \mathcal{X} \cup \{(?_\mathcal{X})\}
\]

and the constraints specified in the definition of actor configurations in §2. are satisfied. We let \(\mathcal{L}_\mathcal{K}\) range over \(\mathcal{L}_\mathcal{K}\), and \(\mathcal{L}_\mathcal{K}\) range over \(\mathcal{L}_\mathcal{K}\). Filling expression and abstraction holes of an LE-configuration, LE-actor map, LE-actor state, LE-multiset of messages, and LE-messages is defined in the obvious manner.

An LE-configuration, \(\mathcal{L}_\mathcal{K}\), is closing for \(\mathcal{L}\) and a family \(\{\varphi_j\}_{j \in J}\) of lambda abstractions if \((\mathcal{L}_\mathcal{K})[\mathcal{L} := \mathcal{L}][\varphi_j := \varphi_j] = \lambda\) is a closed configuration.

**Definition (LE-Reduction):** The reduction relations \(\xrightarrow{\lambda}\) and \(\xrightarrow{\mathcal{L}}\) are extended to the generalized domains in the obvious fashion, simply by liberally annotating metavariables with \(\mathcal{L}\)’s. We omit the details.

**Definition (LE-hole touching):** If \(\mathcal{L}_\mathcal{K} = \left\{ \mathcal{L}^\alpha \mid \mathcal{L}^\alpha \mathcal{M} \right\}\), then \(\mathcal{L}_\mathcal{K}\) touches a hole at \(a\) if \(\mathcal{L}^\alpha(a) = \mathcal{L}^\alpha\mathcal{E}\) and either \(\mathcal{L}^\alpha = \mathcal{L}^\alpha\mathcal{R}[\mathcal{L}^\alpha := \mathcal{L}^\alpha\mathcal{R}]\) or \(\mathcal{L}^\alpha = \mathcal{L}^\alpha\mathcal{R}[\mathcal{L}^\alpha := \text{app}(\varphi_j, \mathcal{L}^\alpha\mathcal{M})]\). A transition from \(\mathcal{L}_\mathcal{K}[\mathcal{L} := \mathcal{L}][\varphi_j := \varphi_j]\) touches a hole at \(a\) if the focus actor of the transition is \(a\) and \(\mathcal{L}_\mathcal{K}\) touches a hole at \(a\).

Note that since an abstraction hole must be filled with a value, they are not touched in the same ways as arbitrary expression holes, in particular if the transition
is a \(<\text{fun} : a>\) execution step where \(\varphi_a(a) = [\varphi_e]\) and \(\varphi_e = \varphi^e_R[\boxempty := \text{app}(\lambda z. \varphi^e_z, v_j[\varphi^e])])\),
then this is not considered touching the hole, \(v_j\).

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

Lemma (E-Uniform Computation):
(1) If \(\varphi^e \xrightarrow{\lambda} \varphi^{e'}\), then \(\varphi^e[0 := e][v_j := \varphi_f] \xrightarrow{\lambda} \varphi^{e'}[0 := e][v_j := \varphi_f]\) for any
valid filling expression \(e\) and family of lambda abstractions \(\varphi_f\).
(2) If \(\varphi^e\) has no transition with focus \(a\) (and \(a\) is an actor of \(\varphi^e\)), then either \(\varphi^e\)
touches a hole at \(a\) or \(\varphi^e[0 := e][v_j := \varphi_f]\) has no transition with focus \(a\) for any
valid filling expression \(e\) and family of lambda abstractions \(\varphi_f\).
(3) If \(\kappa \xrightarrow{\lambda} \kappa'\) and \(\kappa = \varphi^e[0 := e][v_j := \varphi_f]\), then either the transition touches a
hole or we can find \(\varphi^{e'}\) such that \(\kappa' = \varphi^{e'}[0 := e][v_j := \varphi_f]\) and \(\varphi^e \xrightarrow{\lambda} \varphi^{e'}\).

Proof: Similar to the proof of (E-uniform computation) Now there are two
cases in which a hole is touched in the decomposition of \(\varphi^e\), namely cases (2) and
(3) of the decomposition lemma. \(\square\)

6.3.3 LE-Main Theorem

Now we have developed sufficient notation and machinery to state and prove a
general result giving equivalence via two-stage reduction.

Theorem (eq-reduct): Let \(e_0, e_1, \varphi_0, \varphi_1, j, j\) for \(j < J\) be such that for each \(\varphi^e\)
we can find \(j \in J\) such that \(e_i[\varphi^e]\) reduces uniformly via \(\lambda\) steps to \(\varphi_i, j[\varphi^e]\) for \(i < 2\),
and that for each \(\varphi^e, \varphi^v\) and \(j \in J\) we can find \(\varphi^e_\lambda\) such that \(\text{app}(\varphi_i, j[\varphi^e], \varphi^v)\)
reduces uniformly via \(\lambda\) steps to \(\varphi_i, j[\varphi^e]\) for \(i < 2\). Then \(e_0 \equiv e_1\).

Corollary (eq-reduct): (if.\lam) is an example. Here we take
\[
e_0 = \lambda x. \text{if}(v, e_a, e_b)
\]
\[
e_1 = \text{if}(v, \lambda x. e_a, \lambda x. e_b) \text{ where } x \notin \text{FV}(v)
\]
\(J = \{a, b\}\)
\(\varphi_i, j = \lambda x. \text{if}(v, e_a, e_b)\) for \(j \in J\)

Proof: Let \(\varphi^e = \{\varphi^e, \varphi^e\} \) be a closing LE-configuration for \(e_0, e_1, \varphi_0, \varphi_1, j\) \(j \in J\).
Assume \(\pi_0 \in \mathcal{F}(\varphi^e[0 := e_0][v_j := \varphi_0]) = [\kappa_i \xrightarrow{\lambda} \kappa_{i+1} \mid i \in \Box].\) We want to
find \(\varphi^e, \varphi_i, \) and \(L_i\), such that \(\kappa_i = \varphi^e[0 := e_0]\) and, letting \(\pi_1 = \varphi^e[0 := e_1][v_j := \varphi_1]\) \(L_i\)
\(\varphi^e[0 := e_1][v_j := \varphi_1]\) \(i \in \Box\), we have \(\pi_1 \in \mathcal{F}(\varphi^e[0 := e_1])\) and \(\text{obs}(\pi_0) = \text{obs}(\pi_1)\). At each stage \(i\) we first consider dangling steps. Suppose
\(\varphi^e[a] = \varphi^e[0 := o]\) and the reduction of \(e_0\) is trivial, i.e. \(e_0[\varphi^e] = \varphi_0, j[\varphi^e]\)
for some \(j\). Then we prefix \(L_i\) with the transitions for \(e_1[\varphi^e] \xrightarrow{\lambda} \varphi_1, j[\varphi^e]\) and convert the
E-hole to \(v_j\). This is done for each \(a \in \text{Dom}(\varphi^e)\) meeting the condition.

Now we consider the decomposition of the configuration at stage \(i + 1\) in the case \(k\)
touches a hole. Suppose \(k\) is an execution by \(a\) with \(\varphi_1, i(a) = \varphi^e[0 := o[\varphi^e]]\). By
the elimination of ‘dangling steps’ we may assume that \( e_0 \) is not a value expression and hence the execution occurs at the hole. Suppose also that \( e_i[\rho^*] \xrightarrow{\lambda} \psi_{i_j}[\rho^*] \).

Define \( \rho^* \alpha_{i+1} = \rho^* \alpha_i \{ a := [\rho^* R[\mathbf{b} := \psi_{i_j}[\rho^*]]] \} \).

Suppose \( R \) is an execution by \( a \) with \( \rho^* \alpha_i(a) = [\rho^* R[\mathbf{b} := \psi_{i_j}[\rho^*], \rho^* \theta]] \).

Suppose also that \( \psi_{i_j}[\rho^*] \) reduces to \( e_i[\rho^*] \) by \( \xrightarrow{\lambda} \) steps. Define \( \rho^* \alpha_{i+1} = \rho^* \alpha_i \{ a := [\rho^* R[\mathbf{b} := e_i[\rho^*]]] \} \).

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

### 6.4 Equivalence of Reduction Contexts

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

#### 6.4.1 R-Syntax

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

**Definition** (\( \circ \mathcal{E} \circ \mathcal{V} \)):

\[
\begin{align*}
\circ \mathcal{V} & = \mathcal{A} \cup \mathcal{X} \cup \lambda \mathcal{E} \cup \text{pr}(\circ \mathcal{V}, \circ \mathcal{V}) \\
\circ \mathcal{E} & = \circ \mathcal{V} \cup \Theta_n^\circ(\circ \mathcal{E}^m) \cup \circ \mathcal{S}[\mathbf{b} := \circ \mathcal{E}] \\
\circ \mathcal{S} & = \mathcal{X} \xrightarrow{\lambda} \circ \mathcal{V}
\end{align*}
\]

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

**Definition** (\( \circ e[\circ := R] \)):

\[
(\circ e[\mathbf{b} := \circ e])[\circ := R] = R[\sigma][\mathbf{b} := \circ e[\circ := R]]
\]

where \( \sigma = \circ \sigma[\circ := R] = \lambda x \in \text{Dom}(\circ \sigma). \circ \sigma(x)[\circ := R] \)

**Definition** (\( \circ \mathcal{R} \circ \mathcal{R}_\text{rdx} \)):

\[
\begin{align*}
\circ \mathcal{R} & = \{ \circ \mathbf{b} \cup \Theta_{m+n+1}(\circ \mathcal{V}^m, \circ \mathcal{E}^m) \cup \circ \mathcal{S}[\mathbf{b} := \circ \mathcal{E}] \\
\circ \mathcal{R}_\text{rdx} & = \Theta_n^\circ(\circ \mathcal{V}^m)
\end{align*}
\]

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

\[
\begin{align*}
\circ \mathcal{B} & := R[\mathbf{b}] = \mathcal{B} \\
\circ \mathcal{B} & := \circ e \end{align*}
\]

\[
(\circ e[\mathbf{b} := \circ R])[\circ := R] = R[\mathbf{b} := \circ R[\circ := R]]
\]

where \( \sigma = \circ \sigma[\circ := R] = \lambda x \in \text{Dom}(\circ \sigma). \circ \sigma(x)[\circ := R] \)
(o ε [ [] := oR])[[[] := o e]] = o ε [ [] := oR][[] := o e]]

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

An R-expression ε e is either an R-value (element of o \emptyset) or it can be decomposed uniquely into an R-reduction context filled with either an R-redex or an R-hole.

**Lemma (R-expression decomposition):**

1. ε e ∈ o \emptyset, or
2. (∃! R, ε r)(ε e = o R[[] := o r]), or
3. (∀! R, ε s, ε t)(ε e = o R[[] := o ε s][[] := o ε t])

**Proof:** An easy induction on the structure of ε e.

### 6.4.2 R-Configurations

**Definition (ε \mathbb{K}):** An R-configuration for reduction contexts ε \kappa is formed like a configuration, but using R-expressions instead of simple expressions.

\[ ε \mathbb{K} = \left\{ \begin{array}{c} \varepsilon \alpha \mathbb{C} \mid \varepsilon \mathbb{M} \end{array} \right\}_{X}^{o} \]

\[ \varepsilon \alpha \mathbb{C} = \varepsilon \Pi \rightarrow o \mathbb{A}s \]

\[ \varepsilon \mathbb{A}s = (o \emptyset) \cup (ε \mathbb{K}) \cup \{ (? \alpha \Pi) \} \]

\[ \varepsilon \mathbb{M} = \ll o \emptyset \ll \rightarrow o \emptyset \]

We let \kappa range over ε \mathbb{K}, and \alpha range over ε \alpha \mathbb{C}. Filling holes of an R-configuration is analogous to filling holes of an E-configuration. An R-configuration ε \kappa is closing for R if \kappa[\varnothing := R] is a closed configuration.

### 6.4.3 R-Reduction

**Definition (touching R-holes):** A transition \ell from \kappa[\varnothing := R] touches an R-hole at a if \ell is an execution transition with focus a and execution state of \kappa at a decomposes according to case (2) of the decomposition lemma.

### 6.4.4 R-Uniform Computation

**Lemma (Uniform Computation):** An R-redex reduces or hangs uniformly (for a given enabling occurrence in a configuration). Hence transitions not touching an R-hole are uniform. More precisely, if \kappa \alpha[\varnothing := R] \rightarrow_{\ell_{k}} \kappa_{i+1}, with focus a, that does not touch an R-hole at a, then \ell_{k} is either a receive or an execution transition in which the execution state of \kappa_{i} at a decomposes according to case (1) of the decomposition lemma. Let \kappa_{i} = \left\langle \varepsilon \alpha_{i} \mid \varepsilon \mu_{i} \right\rangle. Then the decomposition of \kappa_{i+1} = \left\langle \varepsilon \alpha_{i+1} \mid \varepsilon \mu_{i+1} \right\rangle is defined as follows.
Receive: In the receive case we must have that $^x\alpha_i(a) = (\lambda x.\tilde{e})$. Thus $^x\alpha_{i+1} = ^x\alpha_i(a := [\text{app}(\lambda x.\tilde{e}, cv)])$, and $^x\mu_i = ^x\mu_{i+1} + a \leftarrow cv$.

**Uniform execution:** In the uniform execution case $^x\alpha_i(a) = [\tilde{e}]$, where $^e$ has the form $^eR[\cdot ::= \tilde{v}]$. In this case the step is independent of what fills the holes. Thus we can find $^x\alpha_{i+1}$ such that $^x\alpha_i \xrightarrow{L_i} ^x\alpha_{i+1}$ uniformly.

Now we show how to establish equivalence of expressions of the form $R_j[\cdot ::= e]$ for $j < 2$ where the $R_j[\cdot ::= v]$ have a common reduct for any value expression.

**Theorem (eq-r):** If for $z$ fresh, there is some $e$ such that $R_j[\cdot ::= z]$ reduces uniformly via 0 or more $\lambda$ steps to $e$ for $j < 2$, then $R_0[\cdot ::= e] \cong R_1[\cdot ::= e]$ for any $e$.

**Corollary (eq-r):** $(\text{app}), (\text{cmps}), (\text{id}), (\text{letx}), (\text{let.dist}), (\text{if.dist})$ are instances of (eq-r).

In fact we prove a slightly more general result, since we use a weaker assumption on the reduction contexts: for each $^e$, we can find $^e_\tilde{e}$ such that $R_j^x[\cdot ::= \tilde{v}]$ reduces uniformly via 0 or more $\lambda$ steps to $^e_\tilde{e}$ for $j < 2$.

**Proof (eq-r):** Suppose $R_0, R_1$ are reduction contexts that we wish to establish the observational equivalence of. That is, we want to show $R_0[\cdot ::= e] \cong R_1[\cdot ::= e]$ for all expressions $e$. Let $^e\psi = \left\langle ^e\alpha \mid ^e\mu \right\rangle$ be a closing $\R$-configuration for $R_0, R_1$.

Assume $\pi_0 \in \mathcal{F}(^e\psi[\cdot ::= R_0]) = [\kappa_0 \xrightarrow{L_0} \kappa_{i+1} \mid i \in \iota]$. We want to find $^e\kappa_{i+1}$, and $L_i$, such that $\kappa_i = ^e\kappa_{i}[\cdot ::= R_0]$ and, letting $\pi_1 = [^e\kappa_i[\cdot ::= R_1] \xrightarrow{L_1} ^e\kappa_{i+1}[\cdot ::= R_1] \mid i \in \iota]$, we have $\pi_1 \in \mathcal{F}(^e\psi[\cdot ::= R_1])$ and $\text{obs}(\pi_0) = \text{obs}(\pi_1)$. As in the expression context case, we can focus our attention on the construction of the actor configuration part, since here also deliverable messages cannot have holes. Assume we have $^e\alpha_i$ and consider cases on the transition label $L_i$; we have three cases to consider. For the base case we have $^e\alpha_0 = ^e\alpha$. At stage $i$ suppose $^e\alpha_i(a') = ^eR[\cdot ::= \tilde{v}]$. If $R_0^x[\cdot ::= \tilde{v}]$ is the common form (i.e. the reduction is trivial), then we prefix $L_i$ with steps for reduction of $R_0^x[\cdot ::= \tilde{v}]$ to common form and remove this hole. This is carried out for each $a'$ in the domain of $^e\alpha_i$.

Now, suppose $L_i$ is an execution with focus $a$ that touches a hole. Thus $^e\alpha_i(a) = [\tilde{e}]$ where $\tilde{e}$ has the form $^eR[\cdot ::= ^e\tilde{v}]$. Suppose also that $R_j^x[\cdot ::= \tilde{v}] \xrightarrow{\lambda} ^e\tilde{e}$ for $j < 2$. Then we define $^e\alpha_{i+1} = ^e\alpha_i[a ::= [\tilde{e}]$.

Let $\pi_1 = [^e\kappa_i[\cdot ::= R_1] \xrightarrow{L_1} ^e\kappa_{i+1}[\cdot ::= R_1] \mid i \in \iota]$ be the constructed computation path. Note that the transitions are the same except for the points where holes are touched, but these differences are not observable. Clearly, under the assumptions on $R_j$, $\pi_1$ is a computation path. It remains to show that the construction preserves fairness.

Suppose some transition $L$ is enabled at stage $i$ in $\pi_1$. If $L$ is a receive or uniform execution, then $L$ is also enabled in $\pi_0$ at stage $i$ and will eventually occur, uniformly. Suppose $L$ is an execution step by $a$ with $^e\alpha_i(a) = [^eR[\cdot ::= ^e\tilde{v}]$]. Either $R_j^x[\cdot ::= \tilde{v}]$ is in common form and the transition occurs as soon as it is enabled in $\pi_1$, or a transition is enabled at this hole in $\pi_0$. This transition will eventually occur. If $R_j^x[\cdot ::= \tilde{v}]$ is not in common form, $L$ will happen in $\pi_1$ at the same stage. If $R_j^x[\cdot ::= \tilde{v}]$ is in common form, then we must consider whether $L$ occurs at the hole
7 Discussion

In this paper we presented an operational semantics of actor computation. The actor language is an extension of a call-by-value functional language by primitives for creating and manipulating actors. Central to the theory is the concept of an actor configuration that makes explicit the notion of open system component. A composition operation on actor configurations is defined. It is associative, commutative, has a unit, and is thus a first step towards an algebra of configurations. The operational semantics is defined by a labelled transition relation on configurations. This notion of computation incorporates fairness is an essential way. This operational semantics is used to define a notion of observational equivalence of expressions based on traditional operational and testing equivalence. An interesting consequence of fairness is that the classic three testing equivalences collapse to two. Methods for establishing equivalence of expressions are developed and a plethora of laws of expression equivalence that incorporates the equational theory of the embedded functional language are presented. We expect that these methods will be useful in developing equational theories for other concurrent extensions of functional languages such as CML or FACILE.

7.1 Future Directions

The theory presented is perhaps best viewed as a starting point for further research rather than a final product. In particular there at least four directions for further research. First, work is needed to develop an algebra of operations on configurations. Treating configurations as objects would allow us to abstract over specific linguistic constructs. Second, a systematic study of notions of equivalence within the actor theory, including notions of configuration equivalence. Third, developing a logic for specifying components as actor configurations which would provide methods for verifying that programs implementing components meet their specifications as well as methods for refining specifications into implementations. Finally, it would be useful to determine whether or not there is a context lemma (similar to ([cin])) for observational equivalence of expressions? 

In hindsight, the theory as presented here might benefit from replacing the newadr, initbeh choice of language primitives by a letactor primitive, and by making the operational semantics truly concurrent. Work is in progress to formalize the semantics presented in this paper in the PVS proof development system (Owre et al., 1992) and to develop mechanically checked proofs of the main theorems. This is a first step towards building of a set of tools for computer aided specifical
tion and design of open distributed systems. The formalization work incorporates these modifications.

Acknowledgements
The authors would like to thank Carl Hewitt and Richard Weyhrauch for many discussions about actor computation that served as a foundation for this work. This research was partially supported by ARPA contract NAG2-703, NSF grants CCR-8917606, CCR-8915663, CCR-9109070 and INT-89-20626, and ARPA and NSF joint contract CCR 90-07195, ONR contract N00014-90-J-1899, and by the Digital Equipment Corporation.

References
Intelligence Laboratory.
Computer Science, 34:83–133.
Felleisen, M. and Friedman, D. (1986). Control operators, the SECD-machine, and the
\( \lambda \)-calculus. In Wirsing, M., editor, Formal Description of Programming Concepts III,
Report Rice COMP TR91-160, Rice University Computer Science Department. To
appear, Information and Computation.
tent and functional programming. International Journal of Parallel Programming,
MIT, Project MAC.
Hewitt, C. (1971). Description and Theoretical Analysis (Using Schemata) of PLANNER:
A Language for Proving Theorems and Manipulating Models in a Robot. PhD thesis,
MIT.
Artificial Intelligence, 8(3):323–364.
artificial intelligence. In Proceedings of 1973 International Joint Conference on Artificial
Intelligence, pages 235–245.
Transactions on Systems, Man, and Cybernetics, SMC-11(1).
and Tokoro, M., editors, Object-Oriented Concurrent Programming, pages 9–36. MIT
Press.
Master's thesis, MIT, Artificial Intelligence Laboratory.
Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. The-
LFCS-88-86, University of Edinburgh.
Massachusetts Institute of Technology.


8 Index of Notations (Not including §5)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>§</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{N} )</td>
<td>The natural numbers, ( i, j, \ldots, n \in \mathbb{N} )</td>
<td>1.3</td>
</tr>
<tr>
<td>( Y^n )</td>
<td>Sequences from ( Y ) of length ( n ), ( \bar{y} = [y_1, \ldots, y_n] \in Y^n )</td>
<td>1.3</td>
</tr>
<tr>
<td>( Y^* )</td>
<td>Finite sequences from ( Y )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>The empty sequence</td>
<td>1.3</td>
</tr>
<tr>
<td>( \text{Len}(\bar{y}) )</td>
<td>The length of the sequence ( \bar{y} )</td>
<td>1.3</td>
</tr>
<tr>
<td>( u \ast v )</td>
<td>The concatenation of the sequences ( u ) and ( v )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \text{Last}(u) )</td>
<td>The last element of the sequence ( u )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \mathcal{P}_\omega[Y] )</td>
<td>Finite subsets of ( Y )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \mathcal{M}_\omega[Y] )</td>
<td>Finite multi-sets with elements in ( Y )</td>
<td>1.3</td>
</tr>
<tr>
<td>( Y_0 \xrightarrow{\delta} Y_1 )</td>
<td>Finite maps from ( Y_0 ) to ( Y_1 )</td>
<td>1.3</td>
</tr>
<tr>
<td>( Y_0 \to Y_1 )</td>
<td>Total functions from ( Y_0 ) to ( Y_1 )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \text{Dom}(f) )</td>
<td>The domain of the function ( f )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \text{Rng}(f) )</td>
<td>The range of the function ( f )</td>
<td>1.3</td>
</tr>
<tr>
<td>( f{y := y'} )</td>
<td>An extension to, or alteration of, the function ( f )</td>
<td>1.3</td>
</tr>
<tr>
<td>( f \mid Y )</td>
<td>The restriction of ( f ) to the set ( Y )</td>
<td>1.3</td>
</tr>
<tr>
<td>( \mathcal{X} )</td>
<td>A countably infinite set of variables, ( x, y, z \in \mathcal{X} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( \text{At} )</td>
<td>Atoms</td>
<td>2.1</td>
</tr>
<tr>
<td>( t, \text{nil} )</td>
<td>Atoms playing the role of booleans</td>
<td>2.1</td>
</tr>
<tr>
<td>( \mathcal{C}_n )</td>
<td>( n )-ary algebraic operations</td>
<td>2.1</td>
</tr>
<tr>
<td>( F )</td>
<td>Operations, ( \delta \in F )</td>
<td>2.1</td>
</tr>
<tr>
<td>( F_n )</td>
<td>( n )-ary operation symbols</td>
<td>2.1</td>
</tr>
<tr>
<td>( F_0 )</td>
<td>Zero-ary operation symbols ( \supseteq {\text{newadr}} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( F_1 )</td>
<td>Unary operation symbols ( \supseteq {\text{isatom, isnat, ispr, isst, isnd, become}} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( F_2 )</td>
<td>Binary operation symbols ( \supseteq {\text{pr, initbeh, send}} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( F_3 )</td>
<td>Ternary operation symbols ( \supseteq {\text{br}} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( \mathcal{L} )</td>
<td>( \lambda )-abstractions, ( \lambda x.e \in \mathcal{L} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( \mathcal{V} )</td>
<td>Value expressions, ( v \in \mathcal{V} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( \mathcal{E} )</td>
<td>Expressions, ( e \in \mathcal{E} )</td>
<td>2.1</td>
</tr>
<tr>
<td>( \lambda x.e )</td>
<td>Abstractions</td>
<td>2.1</td>
</tr>
<tr>
<td>( \text{app}(e_0, e_1) )</td>
<td>Application</td>
<td>2.1</td>
</tr>
<tr>
<td>( \delta(\bar{e}) )</td>
<td>Application of operations</td>
<td>2.1</td>
</tr>
<tr>
<td>( \text{if}(e_0, e_1, e_2) )</td>
<td>Conditional branching</td>
<td>2.1</td>
</tr>
<tr>
<td>( \text{let}{x := e_0}e_1 )</td>
<td>Lexical variable binding</td>
<td>2.1</td>
</tr>
<tr>
<td>( \text{seq}(e_1, \ldots, e_n) )</td>
<td>Sequencing construct</td>
<td>2.1</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>§</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>---</td>
</tr>
<tr>
<td>$\text{FV}(e)$</td>
<td>The free variables of the expression $e$</td>
<td>2.1</td>
</tr>
<tr>
<td>$e[x := e']$</td>
<td>The result of substituting $e'$ for $x$ in $e$</td>
<td>2.1</td>
</tr>
<tr>
<td>$C$</td>
<td>Contexts, $C \in C$</td>
<td>2.1</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>The hole in contexts</td>
<td>2.1</td>
</tr>
<tr>
<td>$C[e]$</td>
<td>The result of filling the context with $e$</td>
<td>2.1</td>
</tr>
<tr>
<td>$\mathcal{A}_d$</td>
<td>Actor addresses, identified with $X$</td>
<td>2.2</td>
</tr>
<tr>
<td>$\mathcal{A}_s$</td>
<td>Actor states, $(\mathcal{A}_d), (b), [e] \in \mathcal{A}_s$</td>
<td>2.2</td>
</tr>
<tr>
<td>$(\mathcal{A}_d)$</td>
<td>An uninitialized actor created by $a$</td>
<td>2.2</td>
</tr>
<tr>
<td>$(b)$</td>
<td>An actor with behavior $b$ ready to accept a message</td>
<td>2.2.1</td>
</tr>
<tr>
<td>$[e]$</td>
<td>A processing actor with current computation $e$</td>
<td>2.2.1</td>
</tr>
<tr>
<td>$\mathcal{M}$</td>
<td>Messages, $\langle \mathcal{M} \rangle \in \mathcal{M}$</td>
<td>2.2.1</td>
</tr>
<tr>
<td>$c^\mathcal{V}$</td>
<td>Communicable values, $cv \in c^\mathcal{V}$</td>
<td>2.2.1</td>
</tr>
<tr>
<td>$\llangle \alpha \mid \mu \rrangle^\chi$</td>
<td>An actor configuration with:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\alpha$ - an actor map</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\mu$ - a multi-set of messages</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\chi$ - the external actors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\rho$ - the receptionists</td>
<td></td>
</tr>
<tr>
<td>$X$</td>
<td>Actor configurations, $\kappa \in X$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$\mathcal{E}_{\mathcal{D}X}$</td>
<td>The set of redexes, $r \in \mathcal{E}_{\mathcal{D}X}$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>The set of reduction contexts, $R \in \mathcal{R}$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$\square$</td>
<td>The reduction context hole</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>The reduction relation for functional redexes, $\kappa_0 \lambda \kappa_1$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>The reduction relation for configurations, $\kappa_0 \rightarrow \kappa_1$ via the rule labelled by $l$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>Labels:</td>
<td>Transition labels, $l \in \text{Labels}$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{fun} : a&gt;$</td>
<td>A functional transition</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{new} : a, a'&gt;$</td>
<td>newadex redex transition with focus $a$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{init} : a, a'&gt;$</td>
<td>initbeh redex transition with focus $a$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{bec} : a, a'&gt;$</td>
<td>become redex transition with focus $a$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{send} : a, m&gt;$</td>
<td>send redex transition with focus $a$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{rcv} : a, cv&gt;$</td>
<td>The receipt of a message with focus $a$</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{out} : m&gt;$</td>
<td>A message exiting the configuration</td>
<td>2.2.2</td>
</tr>
<tr>
<td>$&lt;\text{in} : m&gt;$</td>
<td>A message entering the configuration</td>
<td>2.2.2</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>§</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$\mathcal{T}(\kappa)$</td>
<td>All finite sequences of labeled transitions from $\kappa$, $\nu \in \mathcal{T}(\kappa)$</td>
<td>2.2.3</td>
</tr>
<tr>
<td>$\mathcal{T}^\infty(\kappa)$</td>
<td>the set of all computation paths in $\mathcal{T}(\kappa)$, $\pi \in \mathcal{T}^\infty(\kappa)$</td>
<td>2.2.3</td>
</tr>
<tr>
<td>$\in \in \mathbb{N} \cup {\omega}$</td>
<td>The length of a finite or infinite sequence</td>
<td>2.2.3</td>
</tr>
<tr>
<td>$\text{Cfig}(\kappa, L, i)$</td>
<td>The $i$th configuration of the computation from $\kappa$ via $L$</td>
<td>2.2.3</td>
</tr>
<tr>
<td>$\kappa \xrightarrow{L} \kappa'$</td>
<td>A multi-step transition</td>
<td>2.2.3</td>
</tr>
<tr>
<td>$\mathcal{F}(\kappa)$</td>
<td>the fair subset of $\mathcal{T}^\infty(\kappa)$</td>
<td>2.2.4</td>
</tr>
<tr>
<td>$\text{event}$</td>
<td>A zero-ary primitive/observation</td>
<td>3.1</td>
</tr>
<tr>
<td>$\langle e : a \rangle$</td>
<td>An observation transition</td>
<td>3.1</td>
</tr>
<tr>
<td>$\left&lt; \alpha, [C]_a \mid \mu \right&gt;$</td>
<td>An observing configuration</td>
<td>3.1</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>The set of observing configurations, $O \in \emptyset$</td>
<td>3.1</td>
</tr>
<tr>
<td>$s$</td>
<td>Signifies that an event transition occurs</td>
<td>3.1</td>
</tr>
<tr>
<td>$f$</td>
<td>Signifies that an event transition does not occur</td>
<td>3.1</td>
</tr>
<tr>
<td>$\text{obs}(\pi)$</td>
<td>The s/f classification of the path $\pi$</td>
<td>3.1</td>
</tr>
<tr>
<td>$\text{Obs}(\kappa)$</td>
<td>The s/f classification of the configuration, $\kappa \in {s, f, sf}$</td>
<td>3.1</td>
</tr>
<tr>
<td>$\equiv_1 e_1$</td>
<td>Testing or Convex or Plotkin or Egli-Milner equivalence</td>
<td>3.2</td>
</tr>
<tr>
<td>$\equiv_2 e_1$</td>
<td>Must or Upper or Smyth equivalence</td>
<td>3.2</td>
</tr>
<tr>
<td>$\equiv_3 e_1$</td>
<td>May or Lower or Hoare equivalence</td>
<td>3.2</td>
</tr>
<tr>
<td>$\equiv e_1$</td>
<td>Operational equivalence (either $\equiv_1$ or equivalently $\equiv_2$)</td>
<td>3.2</td>
</tr>
<tr>
<td>$\text{Hang}$</td>
<td>The set of all stuck expressions, $\text{stuck} \in \text{Hang}$</td>
<td>4.1</td>
</tr>
<tr>
<td>$\text{Infin}$</td>
<td>The set of all diverging expressions, $\text{bot} \in \text{Infin}$</td>
<td>4.1</td>
</tr>
</tbody>
</table>