J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press

A Foundation for Actor Computation

Gul A. Agha
University of Illinois at Urbana-Champaign agha@cs.uiuc.edu

Ian A. Mason
University of Tasmania ian.mason@appcomp.utas.edu.au

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 an 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 composability 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 5
2 Our Actor Language 9
2.1 Trivial Examples 10
2.2 Actor Cells 10
2.3 Join Continuations 11
2.4 Notation. 14
3 A Simple Lambda-Based Actor Language 15
3.1 Syntax 15
3.2 Reduction Semantics for Actor Configurations 16
3.3 Composition of Actor Configurations 23
4 Equivalence of Expressions 24
4.1 Events 24
4.2 Three Equivalences 25

4.3 Partial Collapse 25
4.4 Equivalence of Configurations 27
5 Laws of Expression Equivalence 28
5.1 Functional Laws 29
5.2 Basic Laws for Actor Primitives 31
5.3 Introductory Examples Revisited 36
6 Proving Expression Equivalence 37
6.1 The General Method 38
6.2 Common Reduct Case 41
6.3 Equivalence by Two Stage Reduction 56
6.4 Equivalence of Reduction Contexts 59
7 Discussion 62
References 63
8 Index of Notations 67

1 Introduction

The modern computing environment is becoming increasingly open and distributed. Research on semantics to support reasoning about components and configurations 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 channel/process based. Our components are reactive in contrast to the 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 computational objects, activity in processes 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 and branching primitives and structure constructors, recognizers, and destructors) by including 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 for formalizing 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 *en 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 do compose. As a first step towards an algebra of operations on actor configurations, we define a composition operator. Composition on configurations is associative, commutative, and has a unit. This allows large complex configurations to be studied in parts and composed to form larger systems. 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 sending 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. We include fairness in our semantic model because we are developing a semantic theory of actors, and fairness is a feature of actor computation. Although fairness makes some aspects of reasoning more complicated, it simplifies others, and is essential in some cases. Certain classes of intuitively obvious equations fail to hold without the fairness requirement. For example two expressions should be considered equal if they differ only in that one of them creates an actor which has an infinite computation but never sends any messages: the additional actor created will have no observable effect. This equivalence fails to hold in the absence of a fairness requirement.

More generally, note that collection of active garbage does not preserve semantics without a fairness assumption. Although we only consider equational reasoning in this paper, this work is intended as starting point for a semantic theory that supports both equational reasoning and reasoning about safety and liveness properties of system components. The assumption of fairness allows equational specification of some liveness properties, and it is particularly important for modular reasoning about liveness properties. Without fairness, specifications fail to compose — a process may behave correctly in isolation, but fail to do so in the presence of other processes. Consider for instance a collection of independently operating server actors with identical functionality (we ignore the details of what service is provided). The specification of a single server may require that the server always service requests in a finite amount of time. Composing two servers with this property should compose the specifications, meaning both servers will make progress. But without fairness, one server could starve out the other and composition of the specifications will fail.

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. In this paper we study the semantics of our actor language, focusing on the equational theory and methods for establishing equivalence. Results of this study will be useful in the development of a sound proof calculus, but the development of a proof calculus is outside the scope of this paper. The language we define is not a full blown programming language, however our intent was to include enough features to bring out the technical problems that might arise, so that something like this langauge could serve as a kernel for a real implementatation.

We emphasize that this paper is not about the actor model per se, but about a concurrent extension of a simple functional language with concurrency primitives based on the actor model. The point is not to study an actor calculus analogous to

the π-calculus, but to define the semantics of a higher level language and study its theory of program equivalence. Thus we have sacrificed some of the elegance and generality of a π-calculus-like approach for something more specific and closer to programming practice in order to study these laws more directly.

There are many possible approaches for such a task. One possibility would be to give the higher level language a translational semantics based on a primitive actor calculus. The lack of a well-developed primitive actor calculus makes this approach less appealing, as we would first have to develop the primitive calculus and its equational theory. An alternative might be translation to the π-calculus, but here the mismatch in the choice of primitives makes this problematic for our objective of studying program equivalence. Also, with a transalational semantics, much work would still remain to develop the directly induced equational theory and it is likely that many desirable equations would be lost in the translation. Thus we have chosen to begin by directly defining an operational semantics for our language and studying forms of observational equivalence.

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 an informal introduction to our actor language. §3 gives the syntax and operational semantics of our actor language, and describes a composability result. In §4 we study the notion of observational equivalence for actor expressions and prove that in the presence of fairness two of the standard notions collapse to one. In §5 we state a variety of basic equational laws along with an intuitive explanation of how these laws are established. The use of the laws is illustrated by establishing several properties of the programs given in §2. In §6 we develop methods for establishing expression equivalence, and use these methods to prove the laws of §5. This section contains many technical details and can be read at various levels of detail (including zero) without greatly effecting the understanding of the remainder of the paper. The section begins with a local reading guide. §7 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 compu-

tation 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, 1986; Agha, 1990; Agha et al., 1993a) for more discussion of the actor model, and for many examples of programming with actors.

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 closer 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 demonstrated the use of continuation passing style in actor programs, which was carried over into Scheme (Steele and Sussman, 1975; 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, Act1, 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 is 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 semantic model of our actor language builds on that of (Agha, 1986) which defined a simple transition system for actors, and developed a notion of configurations, receptionists and external actors. This model was implemented by Carl Manning at MIT in the Acore 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). Yonezawa has developed the ABCL family (Yonezawa, 1990) of actor languages. The actor model has also been used as a foundation in designing a number of other concurrent object-oriented languages (c.f. (Yonezawa, 1990; Agha et al., 1989)).

1.2.2 Process Algebras

Much existing research giving rigorous semantics to concurrent languages falls into what could loosely be called the process algebra school. The most well-known process algebras are Milner's Calculus for Communicating Systems (CCS) (Milner, 1983), Hoare's CSP (Hoare, 1985), and Milner's π-calculus (Milner et al., 1989). Process algebra research focuses on understanding elementary communications between processes, abstracting away other programming language issues. Three points of contrast between the basic actor model and process calculi are: the choice of communication model, the choice of communicable values, and the issue of fairness. Process algebras typcially take synchronous communication as primitive instead of asynchronous communication ((Honda and Tokoro, 1991) defines a variant of the π-calculus with asynchronous communication). Synchronous communication can be simulated with asynchronous primitives (Amadio, 1994) and vice-versa, and it is probably the case that both will be required in any realistic concurrent programming language. In standard process algebra theory, processes can be dynamically created, but they are not first class entities that can be directly manipulated. 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 have the potential to inadvertantly interfere with one another. The π-calculus extends CCS in that it allows channels to be treated as first class entities that can be dynamically created and communicated as values. In contrast, in actor and other object based models, the communication medium is not explicitly represented. Actors/objects are first class, history sensitive entities whose identity can be communicated and used for communication with the identified object. 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 realistic programming language could very well need both as primitive notions.

An important distinction between actor and process algebra semantics is that actor semantics presupposes a fairness assumption while process algebra semantics does not.

The main contribution made by the process algebra school is in the realm of semantics. A wide variety of equivalence relations for process algrebras 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). A detailed comparison of the various equivalence

relations is given in (van Glabbeek, 1990). Numerous logics have been developed for process algebras, including (Hoare, 1985; Milner, 1989; Bergstra and Klop, 1986; Abramsky, 1991).

1.2.3 Concurrent Functional Languages

Both process algebras and primitive actor systems are too simple to be considered programming languages, however. There have been a number of languages developed using the approach we follow in this paper— combining concurrency primitives with a functional language. These languages include Amber (Cardelli, 1986), Facile (Giacalone et al., 1989; Prasad et al., 1990; Thomsen et al., 1992), CML (Reppy, 1991), Erlang (Armstrong et al., 1993), Obliq (Cardelli, 1994), and Pict (Pierce and Turner, 1994). Erlang and Obliq are object based languages (Erlang is essentially an actor language) while Facile, CML, and Pict have process algebra concurrency primitives. Of these only CML and Erlang require fairness. Except for Facile, and to a small extent Obliq, these efforts have focused on language design, and type systems, with less attention given to semantics and equivalences. 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. An equational calculus is given for a sequential core of Obliq (Abadi and Cardelli, 1994). The semantics of Pict is given by expansion into the *pi*-calculus core, but no effort has been made so far to develop the induced equational theory. In this paper we focus on developing equational laws for our basic actor language and leave to future work the study of richer actor language features; see (Agha et al., 1993a) for an extensive repertoire of communication and protocol primitives for actors.

The Facile project is perhaps the closest in spirit to our overall effort. Facile differs from our approach in that it uses the *typed* lambda calculus as the functional component and uses concurrency primitives inspired by process algebras. In (Prasad et al., 1990) an algebraic semantics for Facile is given based on an operational semantics. The central notion here is bisimulation relations. Bisimulations 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, along with a few simple functional laws, are established using bisimulations. The authors point out that establishing expression equivalence using bisimulations is much more complicated than establishing process algebra equations. An early version of the 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.

2 Our Actor Language

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`; and `letactor`.

`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.

`letactor` is for actor creation. `letactor`$\{x := b\}e$ creates an actor with initial behavior b, making the new address the value of the variable x. The expression e is evaluated in the extended environment. The variable x is also bound in the expressions b, thus allowing an actor to refer to itself if so desired. Like the Scheme `letrec` construct, multiple actors can be created, each possibly knowing the other. For example, if f and g are ternary lambda expressions, then `letactor`$\{x := \lambda m.f(x, y, m), y := \lambda m.g(x, y, m)\}$`send`$(x, z)$ creates two actors, referred to locally as x, and y, and sends x a message containing the address of an already created actor referred to as z. (When our intent is clear from the context, we simply say x to mean the actor referred to locally as x). x can send messages to itself and to y, and actor y can send messages to itself and to x. Moreover, x will be able to send messages to z as well, once the message is received.

`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. Here `become` is used to model the effects of asynchronous deposits and withdrawals. The `letactor` primitive cannot be used to model these effects as it would destroy the possibility of indeterminate shared access. 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.

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.

$$b5 = \texttt{rec}(\lambda y.\lambda x.\texttt{seq}(\texttt{send}(x, 5), \texttt{become}(y)))$$

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

$$b5' = \texttt{rec}(\lambda y.\lambda x.\texttt{seq}(\texttt{become}(y), \texttt{send}(x, 5)))$$

since the order of executing the **become** and the **send** cannot be observed. An expression that would create an actor with behavior $b5$ and send it another actor address a is

$$e = \texttt{letactor}\{z := b5\}\texttt{send}(z, a).$$

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

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

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:

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

Evaluating

$$\texttt{letactor}\{a := B_{\text{cell}}(0)\}e \quad \text{where}$$

$$e = \texttt{seq}(\texttt{send}(a, \texttt{mkset}(3)), \texttt{send}(a, \texttt{mkset}(4)), \texttt{send}(a, \texttt{mkget}(a)))$$

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 new-symbol generators 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.

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

In the above code, a *tree* is passed to `treeprod` which tests to see if the *tree* is a number (i.e., a leaf). If so it returns the tree, otherwise it subdivides the problem into two recursive calls. The functions `left` and `right` 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, 1993a; 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:

$$
\begin{aligned}
&B_{\texttt{treeprod}} = \\
&\quad \texttt{rec}(\lambda b.\lambda self.\lambda m. \\
&\qquad \texttt{seq}(\texttt{become}(b(self)), \\
&\qquad\quad \texttt{if}(\texttt{isnat}(\texttt{tree}(m)), \\
&\qquad\qquad \texttt{send}(\texttt{cust}(m), \texttt{tree}(m)), \\
&\qquad\qquad \texttt{letactor}\{newcust := B_{\texttt{joincont}}(\texttt{cust}(m), 0, \texttt{nil})\} \\
&\qquad\qquad\qquad \texttt{seq}(\texttt{send}(self, \texttt{pr}(\texttt{left}(\texttt{tree}(m)), newcust)), \\
&\qquad\qquad\qquad\quad \texttt{send}(self, \texttt{pr}(\texttt{right}(\texttt{tree}(m)), newcust))))))
\end{aligned}
$$

$$
\begin{aligned}
B_{\texttt{joincont}} = \texttt{rec}(\lambda b.\lambda cust.\lambda nargs.\lambda firstnum.\lambda num \\
&\texttt{if}(\texttt{eq}(nargs, 0), \\
&\quad \texttt{become}(b(cust, 1, num)), \\
&\quad \texttt{seq}(\texttt{become}(\texttt{sink}), \\
&\qquad \texttt{send}(cust, firstnum * num))))
\end{aligned}
$$

When a *tree product* actor (with behavior $B_{\texttt{treeprod}}$) 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 $B_{\texttt{joincont}}$) 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 a tree production. The join continuation's customer can be the original requester (root of the tree), or the join continuation of a higher branch point. Note that after receiving the first number, the join continuation must modify its behavior to remember that result while waiting for the second number. This is an example of where it is essential to use `become` to modify the existing actors behavior rather than simply creating a new actor with the desired behavior. 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 handle special cases or errors. For example, if the number 0 is encountered, the join continuation can immediately return a 0—in some cases without waiting for the results of evaluating the other subtree.

The above program may not be optimal in other respects. For example `send`s may be quite expensive. Consequently it may be prudent to check that the subcomputation is worth dispatching. This observation together with the fact, proved in §5, that `send`s of values commute leads to the possibly faster version:

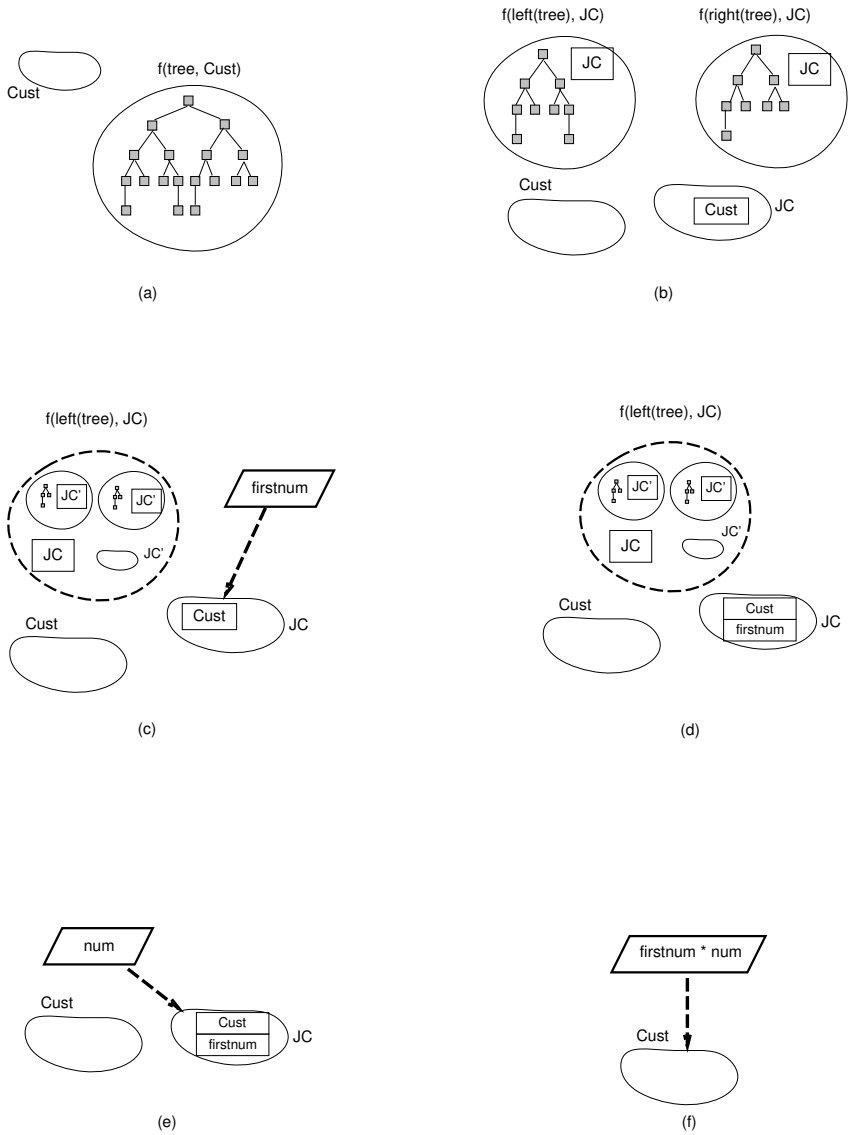

Fig. 1. The leaves of tree, contain numbers to be multiplied and Cust is the actor to which the product is to be sent. (b) the tree is subdivided and two asynchronous messages are sent to compute the product on the two halves (concurrently). JC represents a newly created actor to which the two results 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.

$$
\begin{array}{l}
B^1_{\texttt{treeprod}} = \\
\quad \texttt{rec}(\lambda b.\lambda self.\lambda m. \\
\qquad \texttt{seq}(\texttt{become}(b), \\
\qquad\quad \texttt{if}(\texttt{isnat}(\texttt{tree}(m)), \\
\qquad\qquad \texttt{send}(\texttt{cust}(m), \texttt{tree}(m)), \\
\qquad\qquad \texttt{let}\{l := \texttt{left}(\texttt{tree}(m)), r := \texttt{right}(\texttt{tree}(m))\} \\
\qquad\qquad\quad \texttt{letactor}\{newcust := B_{\texttt{joincont}}(\texttt{cust}(m), 0, \texttt{nil})\} \\
\qquad\qquad\qquad\qquad \texttt{if}(\texttt{or}(\texttt{isnat}(l), \texttt{isnat}(r)), \\
\qquad\qquad\qquad\qquad\quad \texttt{send}(\texttt{cust}(m), \texttt{treeprod}(l) * \texttt{treeprod}(r)), \\
\qquad\qquad\qquad\qquad\quad \texttt{seq}(\texttt{send}(self, \texttt{pr}(l, newcust)), \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{send}(self, \texttt{pr}(r, newcust)))))))
\end{array}
$$

By observing that join continuation is only used in one of the **if** branches we can transform $B^1_{\texttt{treeprod}}$ to a slightly more frugal version $B^2_{\texttt{treeprod}}$.

$$
\begin{array}{l}
B^2_{\texttt{treeprod}} = \\
\quad \texttt{rec}(\lambda b.\lambda self.\lambda m. \\
\qquad \texttt{seq}(\texttt{become}(b), \\
\qquad\quad \texttt{if}(\texttt{isnat}(\texttt{tree}(m)), \\
\qquad\qquad \texttt{send}(\texttt{cust}(m), \texttt{tree}(m)), \\
\qquad\qquad \texttt{let}\{l := \texttt{left}(\texttt{tree}(m)), r := \texttt{right}(\texttt{tree}(m))\} \\
\qquad\qquad\quad \texttt{if}(\texttt{or}(\texttt{isnat}(l), \texttt{isnat}(r)), \\
\qquad\qquad\qquad\quad \texttt{send}(\texttt{cust}(m), \texttt{treeprod}(l) * \texttt{treeprod}(r)), \\
\qquad\qquad\qquad\quad \texttt{letactor}\{newcust := B_{\texttt{joincont}}(\texttt{cust}(m), 0, \texttt{nil})\} \\
\qquad\qquad\qquad\qquad\qquad \texttt{seq}(\texttt{send}(self, \texttt{pr}(l, newcust)), \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{send}(self, \texttt{pr}(r, newcust)))))))
\end{array}
$$

Simple transformations such as this one are justified by the properties of the underlying actor primitives. We return to this example in §5.

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, ..., range over the set Y. Y^n is the set of sequences of elements of Y of length n. Y^* is the set of finite sequences of elements of Y. $\bar{y} = [y_1, \ldots, y_n]$ is the sequence of length $\mathrm{Len}(\bar{y}) = n$ with ith element y_i. (Thus $[\,]$ is the empty sequence.) $u * v$ denotes the concatenation of the sequences u and v. If u is a non-empty sequence, then $\mathrm{Last}(u)$

is the last element of u. $\mathbf{P}_\omega[Y]$ is the set of finite subsets of Y. $\mathbf{M}_\omega[Y]$ is the set of (finite) multi-sets with elements in Y. $Y_0 \xrightarrow{f} Y_1$ is the set of finite maps from Y_0 to Y_1. $[Y_0 \to Y_1]$ is the set of total functions, f, with domain Y_0 and range contained in Y_1. We write $\mathrm{Dom}(f)$ for the domain of a function and $\mathrm{Rng}(f)$ for its range. For any function f: $f\{y := y'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{y\}$, $f'(y) = y'$, and $f'(z) = f(z)$ for $z \neq y, z \in \mathrm{Dom}(f)$; and $f\rceil Y$ 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. In the core language, we replace the `letactor` construct for actor creation by two primitives: `newadr` and `initbeh`. This obeys the principle that λ is the only binding construct, and also permits simpler basic reduction rules. We treat `letactor` as an abbreviation along with `let` and others. `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. The allocation of a new address and initialization of the actor's behavior are separated in order to allow an actor to learn its own address upon initialization. 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.

3.1 Syntax

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

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

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

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

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

We let x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, e range over $\mathbb{E}$, and C range over $\mathbb{C}$. Since we are working with a syntactic reduction semantics, there is no distinction between a value expression and the value it denotes. Hence we use the terms *value* and *value expression* interchangeably. $\lambda x.e$ binds the variable x in the expression e. Two expressions are considered equal if they are the same up to α renaming (that

is, 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 $\mathrm{FV}(e)$ for the set of free variables of e. We write $e[x := e']$ to denote the expression obtained from e by replacing all free occurrences of x by e', avoiding the capture of free variables in e'. Contexts are expressions with holes. We use $\bullet$ to denote a hole. $C[e]$ denotes the result of replacing any holes in C by e. Free variables of e may become bound in this process.

`br` is a strict conditional, and the usual conditional construct `if` can be considered an abbreviation following Landin (Landin, 1964). `let` and `seq` are the usual syntactic sugar, `seq` being a sequencing primitive.

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

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

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

`letactor` is defined it terms of `newadr` and `initbeh` as follows:

$\texttt{letactor}\{x_1 := e_1, \ldots, x_n := e_n\}e$ abbreviates

$$\texttt{let}\{x_1 := \texttt{newadr}()\} \ldots \texttt{let}\{x_n := \texttt{newadr}()\}$$
$$\texttt{seq}(\texttt{initbeh}(x_1, e_1), \ldots, \texttt{initbeh}(x_n, e_n), e)$$

Note that free occurrences of the x_i in e_i and in e are bound in the `letactor` construct (to newly created actors). We will sometimes use the convention that

$\texttt{letactor}\{\bar{x} := \bar{e}\}e$ abbreviates $\texttt{letactor}\{x_1 := e_1, \ldots, x_n := e_n\}e$

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, 1986). 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

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

A message m contains the address of the actor to whom it is sent and the message contents, $<a \Leftarrow v>$. We restrict the contents v to be any value constructed from atoms and actor addresses using the pairing constructor. We call these values *communicable values*. 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 precise way to establish 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 π-calculus has a similar restriction on values that can be communicated, in that it does not allow processes to be communicated.

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 λ-transition), or a message 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 `i/o` transitions that correspond to interactions with external agents:

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

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

3.2.1 Actor Configurations

We assume that we are given a countable set $\mathbb{A}\mathrm{d}$ of actor addresses. To simplify notation, we identify $\mathbb{A}\mathrm{d}$ with $\mathbb{X}$ 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 ($\mathrm{c}\mathbb{V}$ $\mathbb{A}\mathrm{s}$ $\mathbb{M}$): The set of *communicable values*, $\mathrm{c}\mathbb{V}$, the set of *actor states*, $\mathbb{A}\mathrm{s}$, and the set of *messages*, $\mathbb{M}$, are defined as follows.

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

We let cv range over $\mathrm{c}\mathbb{V}$ and m range over $\mathbb{M}$. Note that actor behaviors are not syntactically restricted to be lambda abstractions, nor are messages syntactically restricted to be of the form $\mathtt{<}\mathbb{A}\mathrm{d} \Leftarrow \mathrm{c}\mathbb{V}\mathtt{>}$. 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 ($\mathbb{K}$)): An *actor configuration* with actor

map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written

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

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

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$,

(1) if $\alpha(a) = (?_{a'})$, then $a' \in A$,

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

We let $\mathbb{K}$ denote the set of actor configurations and let κ range over $\mathbb{K}$. The *receptionists* ρ are names of actors within the configuration that are externally visible; all other actors in the (actor) configuration are local and thus inaccessible from the outside. The *external* actors χ are names of actors that are outside this configuration but to which messages may be directed. A configuration in which both the receptionist and external actor sets are empty is said to be *closed*. For closed configurations we may omit explicit mention of the empty ρ and χ sets. The actor map portion of a configuration is presented as a list of actor states each subscripted by the actor address which is mapped to this state. If $\alpha'(a) = (b)$, and α is α' with a omitted from its domain, we write α' as $(\alpha, (b)_a)$ to focus attention on a. 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 transition relation $\mapsto$ on configurations. Although this is, on the surface an interleaving semantics, it is easy to modify our transition system to obtain a truly concurrent semantics either by forming a labelled transition system with independence (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., 1995). In order to distinguish holes used for different purposes, we use the sign $\square$ 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 ($\mathbb{E}_{\mathrm{rdx}}$ $\mathbb{R}$): The set of *redexes*, $\mathbb{E}_{\mathrm{rdx}}$, and the set of *reduction contexts*, $\mathbb{R}$, are defined by

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

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

We let R range over $\mathbb{R}$ and r range over $\mathbb{E}_{\mathrm{rdx}}$.

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.$\square$

Redexes can be split into two classes, purely functional and actor redexes. The actor redexes are $\texttt{newadr}()$, $\texttt{initbeh}(v_0, v_1)$, $\texttt{become}(v)$, and $\texttt{send}(v_0, v_1)$. Reduction rules for the purely functional case is given by a relation $\overset{\lambda}{\mapsto}_X$ on expressions. X is a finite collection of variables indicating the currently defined actor addresses, which hence are not atoms, pairs, or functions.

Definition ($\overset{\lambda}{\mapsto}_X$): Assume that $X \subset \mathbb{X}$ is a finite set of variables.

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

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

where $\delta \in \mathbb{G}_n$, $v_1, \ldots, v_n \in \mathbb{A}\mathrm{t}^n$, and $\delta(v_1, \ldots, v_n) = v'$.

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

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

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

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

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

The rules for `isatom` and `isnat` are analogous to that for `ispr`, in particular elements of X are not atoms and not numbers. If the set X is empty we write $\overset{\lambda}{\mapsto}$ rather than $\overset{\lambda}{\mapsto}_\emptyset$. The single-step transition relation $\mapsto$ on actor configurations is generated by the following rules.

Definition ($\mapsto$):

`<fun` : a`>`

$$e \overset{\lambda}{\mapsto}_X e' \Rightarrow \left\langle\!\!\left\langle \alpha, [e]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [e']_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

where $X = \mathrm{Dom}(\alpha) \cup \{a\} \cup \chi$

`<new` : a, a'`>`

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{newadr}()]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[a']]_a, (?_a)_{a'} \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

`<init` : a, a'`>`

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

$\texttt{<bec}: a, a'\texttt{>}$

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{become}(v)]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_{a'}, (v)_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

$\texttt{<send}: a, m\texttt{>}$

$$\left\langle\!\!\left\langle \alpha, [R[\texttt{send}(v_0, v_1)]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \mapsto \left\langle\!\!\left\langle \alpha, [R[\texttt{nil}]]_a \mid \mu, m \right\rangle\!\!\right\rangle^{\rho}_{\chi}$$

$$m = \texttt{<}v_0 \Leftarrow v_1\texttt{>}$$

$\texttt{<rcv}: a, cv\texttt{>}$

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

$\texttt{<out}: m\texttt{>}$

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

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

$\texttt{<in}: m\texttt{>}$

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

$$\text{if } m = \texttt{<}a \Leftarrow cv\texttt{>},\ a \in \rho \text{ and } \mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

In the lambda reduction rule, expressions are reduced under the assumption that $\mathrm{Dom}(\alpha) \cup \{a\} \cup \chi$ are in fact actor addresses. In the rules for `newadr` and `become`, a' fresh abbreviates $a' \notin \mathrm{Dom}(\alpha) \cup \{a\} \cup \chi$. Each rule is given a label l consisting of a tag indicating the primitive instruction, and additional parameters. We write $\kappa_0 \xrightarrow{l} \kappa_1$ if $\kappa_0 \mapsto \kappa_1$ according to the rule labelled by l. We call this triple a labelled transition.

`i/o` transitions are transitions with tags `in` or `out`. In all cases other than `i/o` transitions the first parameter names the *focus* actor of the transition. `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. The transitions are labelled to allow us to reason about sequences of transitions in terms of the rules applied, and to allow for alternative representation of computations, including: sequences of configurations; sequences of labelled transitions; and sequences of labels. Note that we have chosen the labels to include sufficient information that κ' is uniquely determined by κ and l.

Definition (Enabled(κ, l))**:** A transition l is said to be *enabled* in the configuration κ, written Enabled(κ, l), iff there is a configuration κ' such that $\kappa \xrightarrow{l} \kappa'$. The transition is said to be *disabled* iff it is not enabled. Formally

$$\mathrm{Enabled}(\kappa, l) \;\Leftrightarrow\; (\exists \kappa' \in \mathbb{K})(\kappa \xrightarrow{l} \kappa')$$

Note that of the eight particular forms of configuration transitions, only the $\texttt{<in}: m\texttt{>}$ transition is always enabled (provided the message is of the correct form).

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. We chose not to, in order to have a consistently lazy dynamic error checking policy.

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

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

3.2.3 Computation Sequences and Paths

Definition (Computation trees): If κ is a configuration, then we define the *computation tree* for κ, $\mathcal{T}(\kappa)$, to be the set of all finite sequences of labelled transitions of the form $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < n]$ for some $n \in \mathbb{N}$, with $\kappa = \kappa_0$. We call such sequences *computation sequences* and let ν range over them.

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

Definition (Computation paths $\pi \in \mathcal{T}^{\infty}(\kappa)$): The sequences of a computation tree are partially ordered by the initial segment relation. An *infinite computation path* from κ is a maximal linearly ordered set of computation sequences in the computation tree, $\mathcal{T}(\kappa)$. A *finite computation path* from κ is a linearly ordered set of computation sequences in the computation tree, $\mathcal{T}(\kappa)$, which is maximal with respect to transitions other than `in`. The reason for this minor distinction is that, as noted before, `in` transitions are always enabled and unconditional maximality would eliminate the possibility of finite computation paths. Note that a path can also be regarded as a (possibly infinite) sequence of labelled transitions. We use $\mathcal{T}^{\infty}(\kappa)$ to denote the set of all paths from κ, and let π range over computation paths. When thinking of a path as a possibly infinite sequence we write $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ where $\bowtie \in \mathbb{N} \cup \{\omega\}$ is the length of the sequence.

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

Definition (Cfig)**:** Let κ be a configuration and let $L = [l_i \mid i < \bowtie]$ be a sequence of labels corresponding to a computation from κ. The ith configuration of

the computation from κ determined by L, $\mathrm{Cfig}(\kappa, L, i)$, is defined by induction on i as follows.

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

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

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

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

This notation has the advantage that when an initial starting configuration is fixed, either implicitly or explicitly, computation sequences in the computation tree can be identified with sequences of labels. When the sequence L is finite we let $\mathrm{Cfig}(\kappa, L)$ denote the final configuration: $\mathrm{Cfig}(\kappa, L) = \mathrm{Cfig}(\kappa, L, \mathrm{Len}(L))$. Note that label sequences are related to CSP-like traces, but differ in that they are possibly infinite, and that our labels include more information than simply communication.

Definition (multi-step transition): Let $L = [l_j \mid j < n]$ be a finite sequence of transition labels (possibly empty). L is a *multi-step transition* $\kappa \xrightarrow{L} \kappa'$ just if $\mathrm{Cfig}(\kappa, L) = \kappa'$. 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 but does not. We begin by making the notion of *fairness* more formal.

Definition ($\mathrm{Fair}(\pi)$): A path $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ in the computation tree $\mathcal{T}^{\infty}(\kappa)$ is *fair*, written $\mathrm{Fair}(\pi)$ if each enabled transition (other than `in` transitions) eventually happens or becomes permanently disabled.

$$\begin{aligned}\mathrm{Fair}(\pi) \;\Leftrightarrow\; & (\forall i < \bowtie)(\exists l)((\mathrm{Enabled}(\kappa_i, l) \wedge \neg(\exists m \in \mathbb{M})(l = \texttt{<in}: m\texttt{>})) \Rightarrow \\ & ((\exists j \geq i)(\kappa_j \xrightarrow{l} \kappa_{j+1}) \vee (\exists j > i)(\forall k > j)(\neg\mathrm{Enabled}(\kappa_k, l))))\end{aligned}$$

The transition system has the property that if l is enabled in κ_i then either it remains enabled in every subsequent configuration until it is executed, or else l has the form `<rcv` $: a, cv$`>` and for some $j \geq i$ a is busy and never again becomes ready to accept a message.

Definition ($\mathcal{F}(\kappa)$): For a configuration κ we define $\mathcal{F}(\kappa)$ to be the subset of $\mathcal{T}^{\infty}(\kappa)$ that are fair.

$$\mathcal{F}(\kappa) = \{\pi \in \mathcal{T}^{\infty}(\kappa) \mid \mathrm{Fair}(\pi)\}$$

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

Lemma (Fairness): A finite path is fair.

Proof (fairness): If $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < M-1]$, then by maximality we must have that $(\forall l \neq \texttt{<in}: m\texttt{>})\neg\mathrm{Enabled}(\kappa_M, l)$. Consequently the path is fair, since all enabled transitions have either occurred or become disabled. □

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

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

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

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

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

$$\kappa_0 \parallel \kappa_1 = \kappa_1 \parallel \kappa_0$$

$$\kappa_0 \parallel \kappa_\emptyset = \kappa_0$$

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

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

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

□

Furthermore, it is possible to independently define composition operations on sets P of computation sequences or paths

$$P_0 \parallel P_1$$

such that

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

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

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

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

Details of this construction are omitted for space considerations.

4 Equivalence of Expressions

In this section we study the 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 that 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, `event` and observe whether or not in a given computation, `event` is executed. Our approach is similar in spirit to that used in defining testing equivalence for CCS (de Nicola and Hennessy, 1984).

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

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

Definition ($\mathbb{O}$): The *observing configurations* are configurations over the extended language of the form $\left\langle\!\!\left\langle \alpha, [C]_a \mid \mu \right\rangle\!\!\right\rangle$. We use $\mathbb{O}$ to denote the set of observing configurations, and let O range over $\mathbb{O}$. Placing an expression in an observing configuration is just filling the holes of the context with that expression. Thus, if O is an observing configuration as above, then $O[e] = \left\langle\!\!\left\langle \alpha, [C[e]]_a \mid \mu \right\rangle\!\!\right\rangle$. For an given expression e, the observing configurations for e are those $O \in \mathbb{O}$ such that $O[e]$ a closed configuration.

In our definition of observing configuration, the holes appear in the current state of an single executing actor. It is not hard to see that allowing holes in any actor state does not change the resulting notion of equivalence. A generalization of this fact, (**ocx**) is proved in §6.

Since the language is nondeterministic, three different outcomes are possible in place of the two in the deterministic case: either `event` occurs for all possible computation paths, it occurs in some computation paths but not others, or it never occurs. We observe `event` transitions in the fair paths. We say that a computation path succeeds, **s**, if an `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 path is observed to fail, **f**, if it is not observed to succeed. $obs(\pi)$ is **s** if π succeeds, and

f otherwise, and $Obs(\kappa)$ encodes the set of observations possible for all paths of a closed actor configuration.

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

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

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

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 **sf** observation may be considered as good as an **s** observation, and a new equivalence arises if these observations are equated. Similarly, an **sf** observation may be as bad as an **f** observation. We define the following three equivalences.

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

(1) (testing or convex or Plotkin or Egli-Milner)
$e_0 \cong_1 e_1$ iff $Obs(O[e_0]) = Obs(O[e_1])$ for all observing contexts $O \in \mathbb{O}$

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

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

By construction each of these equivalence relations is a congruence.

Theorem (congruence):

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

4.3 Partial Collapse

Note that may-equivalence ($\cong_3$) 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 $o, o' \in \{\mathbf{s}, \mathbf{sf}, \mathbf{f}\}$ just if $Obs(O[e_0]) = o$ and $Obs(O[e_1]) = o'$. This partitions the observing configuration contexts of e_0 and e_1 into nine sets labeled $o : o'$ for $o, o' \in \{\mathbf{s}, \mathbf{sf}, \mathbf{f}\}$.

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

This characterization is summarized in the picture below. Here $\times$ indicates that the set must be empty, and $\surd$ indicates that the set might be non-empty. The two $*$'d cases in $\cong_2$ are cases in which sets are allowed to be non-empty by the definition, but lemma (**f.sf**) below shows they are in fact always empty.

$\cong_1$

		e_1		
		s	**sf**	**f**
e_0	**s**	$\surd$	$\times$	$\times$
	sf	$\times$	$\surd$	$\times$
	f	$\times$	$\times$	$\surd$

$\cong_2$

		e_1		
		s	**sf**	**f**
e_0	**s**	$\surd$	$\times$	$\times$
	sf	$\times$	$\surd$	$*$
	f	$\times$	$*$	$\surd$

$\cong_3$

		e_1		
		s	**sf**	**f**
e_0	**s**	$\surd$	$\surd$	$\times$
	sf	$\surd$	$\surd$	$\times$
	f	$\times$	$\times$	$\surd$

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

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

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

Proof (partial collapse):

$1 = 2$ Assume $e_0 \cong_2 e_1$. Then by (**sets**) the sets labeled $\mathbf{s} : \mathbf{sf}$, $\mathbf{sf} : \mathbf{s}$, $\mathbf{s} : \mathbf{f}$, $\mathbf{f} : \mathbf{s}$

are all empty. By (**f.sf**) **f** : **sf** and **sf** : **f** must also be empty. Hence by (**sets**), $e_0 \cong_1 e_1$. $\square 1 = 2$

$1 \Rightarrow 3$ By (**sets**) $\square 1 \Rightarrow 3$

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

$\square$

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): The observing configurations for an actor configuration, $\kappa = \left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle_{\chi}^{\rho}$, are configurations over the extended language of the form $\kappa' = \left\langle\!\!\left\langle \alpha' \mid \mu' \right\rangle\!\!\right\rangle_{\rho}^{\chi}$. Note that if κ' is an observing configuration for κ, then κ' is composable (in the sense of §3.3) with κ.

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

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

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

We observe that, no two closed configurations can be distinguished by an external observer.

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

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

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

(ii) $\kappa'_0 = \left\langle\!\!\left\langle \alpha, (\lambda x.C[e_0])_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} \cong \left\langle\!\!\left\langle \alpha, (\lambda x.C[e_1])_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi} = \kappa'_1$

Proof : (i) We need to show that $Obs(\mathrm{Hide}(\kappa_0 \parallel \kappa')) = Obs(\mathrm{Hide}(\kappa_1 \parallel \kappa'))$ for any observing κ'. Note however that $\mathrm{Hide}(\kappa_0 \parallel \kappa') = O[e_0]$ and $\mathrm{Hide}(\kappa_1 \parallel \kappa') = O[e_1])$ for some $O \in \mathbb{O}$, so the result follows directly from the definition of $\cong$.
(ii) We need to show that $Obs(\mathrm{Hide}(\kappa'_0 \parallel \kappa')) = Obs(\mathrm{Hide}(\kappa'_1 \parallel \kappa'))$ for any observing κ'. Pick any $\pi_0 \in \mathcal{F}(\mathrm{Hide}(\kappa'_0 \parallel \kappa'))$, then we must find $\pi_1 \in \mathcal{F}(\mathrm{Hide}(\kappa'_1 \parallel \kappa'))$ such that $obs(\pi_0) = obs(\pi_1)$. There are two cases to consider. Either actor a never becomes active, or it becomes active first after k steps of computation. In the first case, the e_i are never touched, so both computations proceed uniformly, thus their observation and fairness behavior both correspond. In the second case, consider the step where a receives its first message:

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

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

□

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 §6. These laws are not intended as a proof system for reasoning about actor systems, but as illustrations of the laws that can

be established using the methods of §6. Much work remains to develop a usable proof system. We conclude the section by establishing properties of some of the examples given in §2.

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, 1993a)) continue to hold in the full actor language. A first simple observation is that two communicable values are observationally equivalent iff they are the same value expression.

Lemma (cv):

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

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

$$O = \left\langle\!\!\left\langle\; [\mathtt{if}(\mathtt{eq}(cv_0, \bullet), \mathtt{event}(), \mathtt{nil})]_a \;\middle|\; \right\rangle\!\!\right\rangle$$

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

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

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

The laws of the untyped computational lambda calculus (Moggi, 1988), and the laws for conditional and pairing continue to hold in the actor setting. The following theorem is a sample of such laws.

Theorem (functional laws):

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

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

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

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

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

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

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

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

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

Each of these laws (except for (**iflam**)) is a consequence of the following operational law, established in §6.2.6.

Theorem (red-exp):

$$e_0 \overset{\lambda}{\mapsto} e_1 \;\Rightarrow\; e_0 \cong e_1$$

The law (**iflam**) is established in §6.3.3. The theorem (**rcx**) is a special case of a theorem proved in (Talcott, 1989).

Theorem (rcx): If R is a reduction context and $x \notin \mathrm{FV}(R)$, then

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

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

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

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

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

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

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

Corollary (uni-rcx):

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

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

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

The above laws are really about equivalence of reduction contexts. They are instances of the operational law (**red-rcx**), established in §6.4.4. 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 λ-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 $R_0[x] \overset{\lambda}{\mapsto} e'$ and $R_1[x] \overset{\lambda}{\mapsto} e'$ 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. Note that if the reductions involve non-lambda steps the result clearly fails, since they could have different effects such as the sending of messages that other actors in the configuration may detect. We let $\mathtt{stuck} \in \mathit{Hang}$ be a prototypical stuck expression, for example $\mathtt{app}(0, 0)$, and let $\mathtt{bot} \in \mathit{Infin}$ be a prototypical expression with infinite computation, for example $\mathtt{app}(\lambda x.\mathtt{app}(x, x), \lambda x.\mathtt{app}(x, x))$.

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

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

Definition (*Infin*): Let *Infin* the set of (non-value) expressions e such that every

closed instance has an infinite lambda reduction sequence. Thus $e \in \mathit{Infin}$ just if we can find e_j for $j \in \mathbb{N}$ such that $e_0 = e$ and $e_j \overset{\lambda}{\mapsto} e_{j+1}$.

The following theorem is established in §6.2.7.

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

5.2 Basic Laws for Actor Primitives

Now we consider the equational properties of the actor primitives, `send`, `letactor`, `become`, `newadr`, and `initbeh`. These laws are established in §6.2.8 and §6.2.9. As is the case for a language with operations that modify state, $\mathtt{seq}(e, e) \cong e$ fails to hold because the evaluation of e can have effects such as message sends. A stronger analogy exists between the actor primitives and the reference primitives $\{\mathtt{mk}, \mathtt{get}, \mathtt{set}\}$ (see (Mason and Talcott, 1991; Honsell et al., 1995)). The construct `letactor` (that is, `newadr` combined with `initbeh`) is an allocation primitive analogous to `mk`. The primitive `become` updates state analogously to `set`. The effect of `send` depends on the state in a way analogous to `get`. There are limits to this analogy, for example `send` does not return anything of interest. Since `send`, `become`, and `initbeh` all return `nil` as values we have the following law.

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

That `letactor` is an allocation primitive analogous to `mk` manifests itself in the following (**delay**) and (**gc**) laws.

(delay) $\mathtt{let}\{y := e_0\}\mathtt{letactor}\{\bar{x} := \bar{v}\}e \cong \mathtt{letactor}\{\bar{x} := \bar{v}\}\mathtt{let}\{y := e_0\}e$

(gc) $\mathtt{letactor}\{\bar{x} := \bar{v}\}\, e \cong e$

where in (delay) no x_i is free in e_0, and y is not free in $\bar{x}, \bar{v}$, and in (gc) no x_i is free in e. Note that, because we have not allowed clones to initialize newly spawned actors, the analogous property for `newadr` alone fails to hold. Namely,

(non-delay)

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

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

A `letactor` law analogous to (**if.dist**) is the following

(if.letact)
$$\begin{aligned}\mathtt{letactor}\{\bar{x} := \bar{v}\}\mathtt{if}(e_0, e_1, e_2) \cong \mathtt{if}(e_0,\ & \\ & \mathtt{letactor}\{\bar{x} := \bar{v}\}\, e_1, \\ & \mathtt{letactor}\{\bar{x} := \bar{v}\}\, e_2)\end{aligned}$$

if no x_i is free in e_0. As a simple application of the laws already presented we show how (**if.letact**) follows from a slightly simpler version (**if.letact.z**) where the test

expression of the `if` is assumed to be a variable:

(if.letact.z) $\texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{if}(z, e_1, e_2) \cong \texttt{if}(z,$
$\texttt{letactor}\{\bar{x} := \bar{v}\}e_1,$
$\texttt{letactor}\{\bar{x} := \bar{v}\}e_2)$

Proof (if.letact):

$\texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{if}(e_0, e_1, e_2)$

$\cong \texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{let}\{z := e_0\}\texttt{if}(z, e_1, e_2)$

by (**rcx.letx**) and (**congruence**)

$\cong \texttt{let}\{z := e_0\}\texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{if}(z, e_1, e_2)$ by (**delay**)

$\cong \texttt{let}\{z := e_0\}\texttt{if}(z, \texttt{letactor}\{\bar{x} := \bar{v}\}e_1, \texttt{letactor}\{\bar{x} := \bar{v}\}e_2)$

by (**if.letact.z**) and (**congruence**)

$\cong \texttt{if}(e_0, \texttt{letactor}\{\bar{x} := \bar{v}\}e_1, \texttt{letactor}\{\bar{x} := \bar{v}\}e_2)$ by (**rcx.letx**)

□**if.letact**

Some other simple properties of `letactor` are (**perm**) and (**split**):

(perm) $\texttt{letactor}\{x_1 := v_1, \ldots, x_n := v_n\}e$
$\cong$
$\texttt{letactor}\{x_{p(1)} := v_{p(1)}, \ldots, x_{p(n)} := v_{p(n)}\}e$
if p is a permutation of $\{1, \ldots, n\}$

(split) $\texttt{letactor}\{x_1 := v_1, \ldots, x_{n+k} := v_{n+k}\}e$
$\cong$
$\texttt{letactor}\{x_1 := v_1, \ldots, x_k := v_k\}$
$\texttt{letactor}\{x_{k+1} := v_{k+1}, \ldots, x_{n+k} := v_{n+k}\}e$
if $\{x_{k+1}, \ldots, x_{n+k}\} \cap \mathrm{FV}(v_1, \ldots, v_k) = \emptyset$

And immediate consequence of (**perm**) and (**split**) is

(perm-split) $\texttt{letactor}\{x_{k+1} := v_{k+1}, \ldots, x_{n+k} := v_{n+k}\}$
$\texttt{letactor}\{x_1 := v_1, \ldots, x_k := v_k\}e$
$\cong$
$\texttt{letactor}\{x_1 := v_1, \ldots, x_k := v_k\}$
$\texttt{letactor}\{x_{k+1} := v_{k+1}, \ldots, x_{n+k} := v_{n+k}\}e$
if $\{x_{k+1}, \ldots, x_{n+k}\} \cap \mathrm{FV}(v_1, \ldots, v_k) = \emptyset$, and
$\{x_1, \ldots, x_k\} \cap \mathrm{FV}(v_{k+1}, \ldots, v_{n+k}) = \emptyset$.

Once allocated, an actor behavior is initialized by `initbeh` and updated by `become`. In analogy with `set` both `become` and `initbeh` satisfy certain, slightly

different, cancellation laws:

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

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

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

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 ϑ_0 and ϑ_1 *commute* if

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

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

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

provided that $x_0 \notin \mathrm{FV}(e_1)$ and $x_1 \notin \mathrm{FV}(e_0)$.

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

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

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

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

$$\left\langle\!\!\left\langle\, (\lambda x.\texttt{event}())_{a_0}, [\texttt{seq}(\bullet, \texttt{send}(a_0, 0))]_a \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle$$

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

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

`send` also commutes with `become`

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

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

case, where an effect of a subcomputation is only observable if the computation completes. We say that a primitive ϑ is *total* if for any configuration of the form $\left\langle\!\!\left\langle \alpha, [R[\vartheta(\bar{v})]]_a \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$ there is a reduction step with a as the focus actor.

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

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

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

will not execute the `send`, whereas

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

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

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

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

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

are distinguished by

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

for $a \neq a_1$, or by

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

Also,

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

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

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

(**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 (**com-ss,com-sb**) would no longer hold.

We summarize these results in the following:

Lemma (commutes):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by hypothesis

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

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

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

by (**let.dis**)

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

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

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

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

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

by (**let.dis**)

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

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

□

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

5.3 Introductory Examples Revisited

To illustrate the application of the actor expression laws we establish some properties of the actor behaviors introduced in §2. First we show that the behaviors $b5$ and $b5'$ from §2.1 are equivalent.

Lemma (b5):

$$b5 \cong b5'$$

where

$$b5 = \texttt{rec}(\lambda y.\lambda x.\texttt{seq}(\texttt{send}(x,5),\texttt{become}(y)))$$

$$b5' = \texttt{rec}(\lambda y.\lambda x.\texttt{seq}(\texttt{become}(y),\texttt{send}(x,5)))$$

Proof: By (**commutes**) $\texttt{seq}(\texttt{send}(x,5),\texttt{become}(y))) \cong \texttt{seq}(\texttt{become}(y),\texttt{send}(x,5)))$. The result follows using the congruence property of $\cong$. □

A generalization of the (**gc**) property of `letactor` is that allocation of an actor with suitably restricted behavior followed by sending it a message and then forgetting that actor is equivalent to the external effects of applying that behavior. A simple example of this is the following property of cells. (Recall that the behavior of a cell, B_{cell}, was defined in §2.2)

(cellb) $\texttt{letactor}\{a := B_{\text{cell}}(0)\}\texttt{send}(a,\texttt{mkget}(c)) \cong \texttt{send}(c,0)$

This property is proved in §6.2.9.

Three actor behaviors, $B_{\texttt{treeprod}}$, $B^1_{\texttt{treeprod}}$, and $B^2_{\texttt{treeprod}}$ were defined in §2.3 for computing the `treeprod` function (also defined in §2.3). Note that $B_{\texttt{treeprod}}$ and $B^1_{\texttt{treeprod}}$ are not equivalent as lambda expressions. They are only equivalent under the assumption that the *self* parameters are bound to the actor in which the code is executing, or at least one of equivalent behavior. The statement of equivalence has the form:

$$\texttt{letactor}\{a := B_{\texttt{treeprod}}(a)\}a \cong \texttt{letactor}\{a := B^1_{\texttt{treeprod}}(a)\}a$$

Proving this is beyond the scope of the methods developed this paper, as it requires reasoning about interactions with the enviornment, not simply isolated local computations. However, we can use the actor laws developed here to show that the third variation, $B^2_{\texttt{treeprod}}$, is equivalent to $B^1_{\texttt{treeprod}}$.

Lemma (tp.1.2):

$$B^1_{\texttt{treeprod}} \cong B^2_{\texttt{treeprod}}$$

Proof: This follows from the equivalence of

$$\texttt{if}(e_0, e_1, \texttt{letactor}\{newcust := b\}e_2) \cong \texttt{letactor}\{newcust := b\}\texttt{if}(e_0, e_1, e_2)$$

where

$e_0 = \texttt{or}(\texttt{isnat}(l), \texttt{isnat}(r))$

$e_1 = \texttt{send}(\texttt{cust}(m), \texttt{treeprod}(l) * \texttt{treeprod}(r))$

$b = B_{\texttt{joincont}}(\texttt{cust}(m), 0, \texttt{nil})$

$e_2 = \texttt{seq}(\texttt{send}(\mathit{self}, \texttt{pr}(l, \mathit{newcust})), \texttt{send}(\mathit{self}, \texttt{pr}(r, \mathit{newcust})))$

which in turn follows from (**gc**) and (**if.letact**). □

6 Proving Expression Equivalence

In this section we develop methods for proving expressions observationally equivalent. In the remainder of this initial part we discuss informally some of the complications that arise to motivate our proof technique. In §6.1 we given an informal outline of the general method and briefly discuss the three special cases developed here. This subsection concludes with some techical matters that can be skipped by the reader who wants only to understand the intuitions and not the technical details. In §6.2 we treat the first special case – equivalence by common reduct. The initial subsections develop the necessary technical details. §6.2.6 contains the main theorem for the case of finite reduction involving no actor primitives. The statement and initial informal part of the proof can be understood without digging into the technical details by just thinking of the meta variables decorated with little circle superscripts as denoting syntactic entites with holes in which expressions to be compared can be placed and observing the presence of holes does not effect computation except when a hole is exposed (touched). In the remainder of the subsection we give some technical details for extensions of the basic method for the common reduct case. §6.2.7 treats stuck and infinite (functional) computations. §6.2.8 extends the proof of §6.2.6 to prove the delay theorem for `letactor`. §6.2.9 generalizes the basic method to treat reductions involving actor primitives. The remaining two subsections give the technical details for applying the general method to two stage reduction and equivalence of reduction contexts.

To illustrate the complications that can arise in attempting to establish equivalences we consider a simple case: $\texttt{succ}(0) \cong 1$. It is simple for two reasons: there are no free variables occurring, and only one step of computation separates $\texttt{succ}(0)$ and 1. By the definition of $\cong$, we need to establish

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

for all observing contexts O. To establish this, we construct, for each computation path $\pi_0 \in \mathcal{F}(O[\texttt{succ}(0)])$, a $\pi_1 \in \mathcal{F}(O[1])$ such that $obs(\pi_0) = obs(\pi_1)$. Similarly for each computation path $\pi_1 \in \mathcal{F}(O[1])$, we construct a $\pi_0 \in \mathcal{F}(O[\texttt{succ}(0)])$ such that $obs(\pi_0) = obs(\pi_1)$. We call such a construction a *path correspondence*. Informally, the path correspondence is constructed as follows. First consider how from a path π_0 in $\mathcal{F}(O[\texttt{succ}(0)])$ we obtain a path in $\mathcal{F}(O[1])$. At each point in π_0 where the $\texttt{succ}(0)$ is reduced to 1, we remove this step, giving path π_1. Describing this operation in detail requires care, for there could be other independent occurrences of the reductions of $\texttt{succ}(0)$ in π_0 which are not to be removed. We then can

show that π_1 is a computation for $O[1]$, with the same observable outcome and same fairness property as π_0. The converse construction is similar, except steps computing $\mathtt{succ}(0)$ are inserted into π_0 each time the 1 first appears in a reduction context. Again this must be done only for occurrences of 1 arising from placing 1 in the holes of O.

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. $\mathtt{pred}(\mathtt{succ}(1)) \cong \mathtt{succ}(0)$. The complication arising from this case is the two-step execution of $\mathtt{pred}(\mathtt{succ}(1))$ can be interleaved with computations of other actors and thus a local replacement is not possible. To solve this problem the computation path is put in an equivalent canonical form with both steps adjacent. In general we may cluster together as many steps of an individual actor as necessary.

A complication also arises in proving equations that may contain free variables, for instance $\mathtt{1}^{\mathtt{st}}(\mathtt{pr}(x,0)) \cong x$. Such expressions may be self-substituted: if $\mathtt{1}^{\mathtt{st}}(\mathtt{pr}(x,0))$ occurs in the local context $\mathtt{app}(\lambda y.\mathtt{app}(y,y), \lambda x.\bullet)$, upon computing the free x in $\mathtt{1}^{\mathtt{st}}(\mathtt{pr}(x,0)))$ will be replaced with $\lambda x.\mathtt{1}^{\mathtt{st}}(\mathtt{pr}(x,0)))$. 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 reduce 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 §5.

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 $e_0 \cong e_1$ if $Obs(ct[e_0]) = Obs(ct[e_1])$ for all templates $ct \in CT$. To establish the equality of observations, it is sufficient to construct a path correspondence. That

is, to provide for each $\pi_0 \in \mathcal{F}(ct[e_0])$, a $\pi_1 \in \mathcal{F}(ct[e_1])$ such that $obs(\pi_0) = obs(\pi_1)$ and conversely. 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 *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 $\triangleright_j$ for $j \in \mathbb{N}$. Note that these holes are filled by values, specifically by lambda abstractions, not simply by expressions. Since the lambda abstractions may contain free variables, we need a family of holes corresponding to the different enviroments in which they are closed.

(3) For the equivalence of reduction contexts we need an entirely new kind of hole, $\diamond$, 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: ${}^{\circ}X$ for expression holes; ${}^{\circ\triangleright}X$ for expression and lambda abstraction holes; and ${}^{\diamond}X$ for reduction context holes. We prefix the names of these classes by E-, LE-, or R- respectively. Thus E-expressions are expression templates with holes for expressions, ${}^{\circ}\mathbb{E}$ is the set of E-expressions, and we let ${}^{\circ}e$ range over ${}^{\circ}\mathbb{E}$. 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-step transitions (§3.2.3), 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, π_0, we show how to obtain a sequence of E-configurations satisfying two conditions. The first is that filling the holes in the sequence of E-configurations with e_0 (and filling in transition labels) yields π_0. The second is that filling the holes in the sequence of E-configurations with e_1 (and expanding multi-step transitions) yields a fair computation path with the same observation. The other two cases are simple variations on this idea.

6.1.1 Some Preliminary Technical Details

One of the keys requirements for 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 $\mathtt{app}(\lambda x.\bullet, v)$. We need a notation that allows us to carry out this reduction in such a way that filling the hole and 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 and discussion of related ideas can be found in (Talcott, 1991; Talcott, 1993b). We use $\circ[^{\circ}\sigma]$ to denote an expression hole with associated substitution $^{\circ}\sigma$ (which may in turn have expressions holes in its range), a similar notation holds for the other classes of holes: $\triangleright_j[^{\circ\triangleright}\sigma]$ for abstraction holes, and $\diamond[^{\diamond}\sigma]$ for reduction context holes.

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

Definition (Θ_n Θ_n^e):

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

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 segment*, and then show that any family of disjoint thread segments in a computation path can be regrouped as multi-steps without effecting the fairness or observation made of the path.

A thread segment, I, at a in π is a finite subsequence of $\mathtt{exec}$ transitions of π with focus actor a or a clone of a created by a $\mathtt{become}$ such that any gaps in the sequence are transitions with some other focus, or at a after a new message receipt.

Definition (thread segment): Let

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

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

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

Then I is a *thread segment* at a in π if

(1) $L(I, \pi)$ contains no $\mathtt{rcv}$, $\mathtt{in}$ or $\mathtt{out}$, and

(2) $L(I, \pi)$ is a computation for $\left\langle\!\!\left\langle \alpha_{i_0} \rfloor \{a\} \mid \emptyset \right\rangle\!\!\right\rangle^{\{a\}}_{\mathrm{FV}(\alpha_{i_0}(a))}$.

As a consequence l_{i_0} has focus a. Note that condition (**2**) makes explicit that a thread segment is essentially running the focus actor in a configuration with only itself. With no $\mathtt{rcv}$s, $\mathtt{in}$s, or $\mathtt{out}$s only that actor or its $\mathtt{become}$ clones can execute.

Theorem (infinite-macro-steps): Let $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie] \in \mathcal{F}(\kappa)$. Let $I_j = [i_{j,0}, \ldots i_{j,n_j}]$ for $j < J \leq \bowtie$ be a (possibly infinite) family of thread segments in π 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, ξ, on $\bowtie$ such that letting $\pi' = \mathrm{Cfig}(\kappa_0, [l_{\xi(i)} \mid i < \bowtie])$ (recall the definition of Cfig from §3.2.3)

(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 π' the thread segments of π marked by I_j for $j \in J$ occur as multi-steps, that is, with no interleaved computation steps. Note that $obs(\pi') = obs(\pi)$.

Proof : ξ is constructed by induction on the index set, one permutes the `exec` steps of each successive segment across interleaved steps in the obvious way. By the definition of thread segment 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 effected by such permutations (except possibly enabling a transition earlier). All transitions that occur in π also occur in π' this means that fairness is also preserved. □

The notion of thread segment I at an actor in a path π can be generalized to allow transitions of a subconfiguration – a group of actors and messages. The key requirements are as before that $L(I, \pi)$ is a computation for the subconfiguration, and that none of the transitions involve interaction with exterior configuration – i.e. no `in` or `out` transitions (receives of internal messages are allowed, but messages from other parts of the configuration are not allowed to come in).

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

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

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

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

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

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

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

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

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

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

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

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

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

As defined here substitution is a partial operation. Using renaming substitutions we can define α renaming in the usual way. We consider E-expressions (and entities containing them) to be equivalent if they differ only by α renaming. Thus, for any substitution we can always choose an α 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, 1993b)).

Expression hole filling is defined by induction on the structure of ${}^\circ e$. We let ${}^\circ e[\circ := e]$ be the result of filling expression holes in ${}^\circ e$ 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 (${}^\circ e[\circ := e]$)**:**

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

$${}^\circ\sigma[\circ := e] = \lambda x \in \mathrm{Dom}({}^\circ\sigma).{}^\circ\sigma(x)[\circ := e]$$

The following example illustrates hole filling and variable scoping. Let

$${}^\circ v = \lambda x.\mathtt{if}(x, \circ[\{y := 0\}], z)$$

$${}^\circ\sigma = \{y := {}^\circ v\}$$

$${}^\circ e = \lambda z.\circ[{}^\circ\sigma]$$

$$e = \theta(x, y).$$

Then

$$
\begin{aligned}
{}^\circ v[\circ := e] &= \lambda w.\mathtt{if}(w, \circ[\{y := 0\}][\circ := e], z) \qquad \text{note the change in bound variable} \\
&= \lambda w.\mathtt{if}(w, e[\{y := 0\}], z) \\
&= \lambda w.\mathtt{if}(w, \theta(x, 0), z) = v \\
{}^\circ\sigma[\circ := e] &= \{y := {}^\circ v[\circ := e]\} \\
&= \{y := \lambda w.\mathtt{if}(w, \theta(x, 0), z)\} \\
{}^\circ e[\circ := e] &= \lambda z.(\circ[{}^\circ\sigma][\circ := e]) \\
&= \lambda z.(e[{}^\circ\sigma[\circ := e]]) \\
&= \lambda z.\theta(x, \lambda w.\mathtt{if}(w, \theta(x, 0), z))
\end{aligned}
$$

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

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

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

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

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

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

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

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

□

Next we define analogs of redex and reduction context.

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

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

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

By our conventions $^\circ r$ range over $^\circ\mathbb{E}_{\mathrm{rdx}}$ and $^\circ R$ range over $^\circ\mathbb{R}$. 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, $^\circ e$, remains unchanged, and we denote it by $^\circ R[\square := {}^\circ e]$.

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

Lemma (E-properties):

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

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

6.2.2 E-Expression Decomposition

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

Lemma (E-expression decomposition):

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

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

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

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

6.2.3 E-Configurations

An E-configuration, $^\circ\kappa$, is formed in the same manner as a simple configuration, using E-expressions and E-values instead of simple expressions and values.

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

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

where

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

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

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

and the constraints specified in the definition of actor configurations in §3 are satisfied.† We let $^\circ\kappa$ range over $^\circ\mathbb{K}$, and $^\circ\alpha$ range over $^\circ\mathbb{A}\mathrm{c}$. Filling expression holes of an E-configuration, E-actor map, E-actor state, E-multiset of messages, and E-messages is defined in the obvious manner. Let $^\circ X$ stand generically for an element of one of these E-syntactic classes, then we define $^\circ X[\circ := e]$ as follows:

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

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

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

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

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

$E = \{i < n \mid (\exists {}^\circ e_i)({}^\circ\alpha(a_i) = [^\circ e_i])\}$, and let n_E be the cardinality of E.
$I = \{i < n \mid (\exists {}^\circ v_i)({}^\circ\alpha(a_i) = (^\circ v_i))\}$.
$B_i(a_0, \ldots, a_{n-1}) = {}^\circ v_i^\star$, if $^\circ\alpha(a_i) = (^\circ v_i)$.

† The only condition whose meaning is altered in this general setting is (**2**), where the free variables of any hole occurrences (namely the free variables in the range of the associated substitution) must be taken into consideration.

$B_i(a_0, \ldots, a_{n-1}) = \lambda a.\mathtt{seq}(\mathtt{send}(a, 0), {}^\circ e_i^\star), \quad \text{if } {}^\circ\alpha(a_i) = [{}^\circ e_i].$
$\mu = \{\langle z_j \Leftarrow {}^\circ v_j' \rangle \mid j < n_M\}$

Define

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

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

Now, we claim that for any computation of ${}^\circ\kappa[\circ := e]$ there is a corresponding computation (with same observations) of $O[e]$ obtained by accepting all the startup messages, sending and accepting the acknowledgments, and completing the computation of the initializing actor (which can then be ignored). Conversely any computation of $O[e]$ has a corresponding computation of ${}^\circ\kappa[\circ := e]$ obtained by ignoring the finite amount of initializing activity. A more detailed proof can be given along the lines of the proof of the theorem (**fun-red-eq**) below. □

6.2.4 E-Reduction

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

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

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

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

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

Definition ($\mapsto$):

$\langle \mathtt{fun} : a \rangle \qquad {}^\circ e \overset{\lambda}{\mapsto}_{\mathrm{Dom}({}^\circ\alpha) \cup \{a\}} {}^\circ e' \Rightarrow \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ e]_a \mid {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ e']_a \mid {}^\circ\mu \right\rangle\!\!\right\rangle$

$\langle \mathtt{new} : a, a' \rangle \qquad \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{newadr}()]]_a \mid {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto$

$\qquad \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := a']]_a, (?_a)_{a'} \mid {}^\circ\mu \right\rangle\!\!\right\rangle \qquad a' \text{ fresh}$

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

$\qquad \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{nil}]]_a, ({}^\circ v)_{a'} \mid {}^\circ\mu \right\rangle\!\!\right\rangle$

$\langle \mathtt{bec} : a, a' \rangle \qquad \left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \mathtt{become}({}^\circ v)]]_a \mid {}^\circ\mu \right\rangle\!\!\right\rangle \mapsto$

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

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

$$\left\langle\!\!\left\langle {}^\circ\alpha, [{}^\circ R[\square := \texttt{nil}]]_a \;\middle|\; {}^\circ\mu, m \right\rangle\!\!\right\rangle \qquad m = \texttt{<}{}^\circ v_0 \Leftarrow {}^\circ v_1\texttt{>}$$

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

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, any transition step leading from that configuration is either independent of what appears in the holes, or it explicitly uses information about the contents of some hole occurrence.

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

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

Lemma (E-Uniform Computation):

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

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

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

Proof (1): This is proved by considering cases on the transition rule applied. The only interesting case is (**beta-v**). This follows from (**fil-subst**) $\square_1$

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

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

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

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

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

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

In each of these cases, it easy to see that there will be no transition with focus a enabled when the expressions holes are filled. $\square_2$

Proof (3): Assume $\kappa = \left\langle\!\!\left\langle \alpha \;\middle|\; \mu \right\rangle\!\!\right\rangle \xrightarrow{l} \kappa' = \left\langle\!\!\left\langle \alpha' \;\middle|\; \mu' \right\rangle\!\!\right\rangle$, $\kappa = {}^\circ\kappa[\circ := e]$, and the transition does not touch the hole. Thus ${}^\circ\kappa = \left\langle\!\!\left\langle {}^\circ\alpha \;\middle|\; {}^\circ\mu \right\rangle\!\!\right\rangle$ where $\alpha = {}^\circ\alpha[\circ := e]$ and $\mu = {}^\circ\mu[\circ := e]$. We want to find ${}^\circ\alpha'$, ${}^\circ\mu'$ such that ${}^\circ\kappa \xrightarrow{l} {}^\circ\kappa' = \left\langle\!\!\left\langle {}^\circ\alpha' \;\middle|\; {}^\circ\mu' \right\rangle\!\!\right\rangle$, $\alpha' = {}^\circ\alpha'[\circ := e]$, and $\mu' = {}^\circ\mu'[\circ := e]$. Since we are considering closed configurations there

are no `i/o` transitions. Thus, we need to consider only two cases `rcv` transitions and `exec` transitions. We split the `exec` transitions into functional and actor primitives.

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

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

Execution-actor:

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

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

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

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

$\square_3$ $\square_{\mathbf{E-uniform}}$

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 ${}^\circ\sigma$ whose domain contains the free variables of e_0, e_1, either $e_j[{}^\circ\sigma]$ hangs for $j < 2$, or there is some ${}^\circ e_c$ such that $e_j[{}^\circ\sigma]$ reduces in 0 or more $\overset{\lambda}{\mapsto}_{\mathrm{FV}(\mathrm{Rng}({}^\circ\sigma))}$ steps to ${}^\circ e_c$ uniformly, then $e_0 \cong e_1$.

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

Proof : Assume that for each closing ${}^\circ\sigma$ there is ${}^\circ e_{c,j}$ such that, letting $X = \mathrm{FV}(\mathrm{Rng}({}^\circ\sigma))$, $e_j[{}^\circ\sigma] \overset{\lambda}{\mapsto}_X \ldots \overset{\lambda}{\mapsto}_X {}^\circ e_{c,j}$ $j < 2$, uniformly, and either ${}^\circ e_{c,j}$ is (uniformly) stuck for $j < 2$, or ${}^\circ e_{c,0} = {}^\circ e_{c,1}$. In either case we call ${}^\circ e_{c,j}$ the common

reduct. We want to show that $e_0 \cong e_1$. By (**ocx**) it is sufficient to show that $Obs({}^\circ\kappa[\circ := e_0]) = Obs({}^\circ\kappa[\circ := e_1])$ for any ${}^\circ\kappa$ that is a closing E-configuration for e_0 and e_1. To do this, we show that for any $\pi_0 = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i \in \bowtie] \in \mathcal{F}({}^\circ\kappa[\circ := e_0])$ we can find $\pi_1 \in \mathcal{F}({}^\circ\kappa[\circ := e_1])$ such that $obs(\pi_0) = obs(\pi_1)$. (The case with 0 and 1 interchanged is symmetric.)

Informally, by the uniformity property of computations, we see that replacing occurrences of e_0 by e_1 has no effect on a computation except where a hole is touched. Using (**infinite macro-steps**) we can localize non-uniform steps so that when a hole is touched, reduction to a common reduct occurs in a single multi-step (which involves no event transitions). Thus we may obtain a computation for ${}^\circ\kappa[\circ := e_1]$ by replacing occurrences of e_0 by e_1 and replacing multi-step transitions reducing $e_0[{}^\circ\sigma]$ to its common reduct by multi-step transitions reducing $e_1[{}^\circ\sigma]$ to its common reduct. To ensure completeness/fairness of the result, we need to take account of the case where a hole $\circ[{}^\circ\sigma]$ is exposed, but the multi-step for $e_0[{}^\circ\sigma]$ is trivial and hence does not appear as a transition. We do this by inserting the corresponding multi-step for $e_1[{}^\circ\sigma]$ at the point where the hole is first exposed. Such holes then effectively disappear, since they are either filled with a stuck expression or with the same expression. Now we make this informal argument more rigorous, by the following steps (details to be filled in below):

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

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

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

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

Step (1) We analyze and decompose π_0 to obtain

(i) for each $i < \bowtie$, an integer n_i and a sequence of decompositions ${}^\circ\kappa_{i,j}$ for $j \leq$

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

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

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

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

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

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

(1) l_i is execution in a hole;

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

(3) l_i touches a hole and

(3.1) enters the hole

(3.2) does not enter the hole

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

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

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

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

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

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

where each L_i is either a single (uniform) transition, or a multi-step reduction of an occurrence of e_0 to its common reduct, and the E-configurations obtained by the above decomposition method. For each i, if ${}^\circ\alpha_{i,0}(a) = [{}^\circ R[\square := \circ[{}^\circ\sigma]]]$, 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 ${}^\circ\alpha_{i,0}$ an an empty transition ${}^\circ\kappa_{i,0}[\circ := e_0] \xrightarrow{[\,]} {}^\circ\kappa_{i,0}[\circ := e_0]$.

Step (3) We let π_1 be the path

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

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

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

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

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

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

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

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

We now prove (**hang-infin**) (see §5.1), which says that any two expressions that hang or have infinite computations are observationally equivalent.

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

Proof (hang-infin): Assume $e_0, e_1 \in \mathit{Hang} \cup \mathit{Infin}$. We want to show that

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

6.2.8 Proof of the Delay Law

(delay) $\texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{let}\{y := e_0\}e \cong \texttt{let}\{y := e_0\}\texttt{letactor}\{\bar{x} := \bar{v}\}e$

where no x_i in $\bar{x}$ is free in e_0, and y is not free in $\bar{x}, \bar{v}$.

Proof (delay): The argument is similar to that used in the (**fun-red-eq**). We outline the key steps in constructing the path correspondence. Let

$e_l = \texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{let}\{y := e_0\}e$, and

$e_r = \texttt{let}\{y := e_0\}\texttt{letactor}\{\bar{x} := \bar{v}\}e$.

Let ${}^\circ\kappa$ be a closing E-configuration for e_l and e_r. We want to show that for any $\pi_l = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i \in \bowtie] \in \mathcal{F}({}^\circ\kappa[\circ := e_l])$ we can find $\pi_r \in \mathcal{F}({}^\circ\kappa[\circ := e_r])$ such that $obs(\pi_l) = obs(\pi_r)$ and conversely. Let π_l be as above. Using (**infinite macro-steps**) we can assume that steps in the evaluation of `letactor` constructs of the form $\texttt{letactor}\{\bar{x} := \bar{v}\}$ occur a single multi-step. We let $\texttt{<leta}: a, \bar{a}\texttt{>}$ label a letactor multi-step with focus a and new actors $\bar{a}$ ($\bar{a}$ and $\bar{x}$ have the same length). Using this assumption, we obtain a computation π_r for ${}^\circ\kappa[\circ := e_r]$ roughly by replacing occurrences of e_l by e_r and shifting the actor creation multi-step to the return from evaluation of e_0.

In more detail, we analyze the configurations occurring in π_l and record occurrences of e_l in holes descending from ${}^\circ\kappa$. This gives us decompositions ${}^\circ\kappa_i^l[\circ := e_l]$ of κ_i. Simultaneously, for each i, we define a corresponding E-configuration ${}^\circ\kappa_i^r$, a (multi-step) label L_i, and a set of pairs X_i of indices such that, if $(q, p) \in X_i$ this means that $q < i$, l_q enters a hole, and l_p is the step that returns the value of the occurrence of e_0 in the hole. If the computation of e_0 does not return a value, then p is ∞.

Stage (0) ${}^\circ\kappa_0^l = {}^\circ\kappa_0^r = {}^\circ\kappa$, and $X_0 = \emptyset$.

Stage $(i+1)$ Assume that we have ${}^\circ\kappa_j^l$, ${}^\circ\kappa_j^r$, and X_j, for $j \leq i$, and L_j for $j < i$. ${}^\circ\kappa_j^l$ and ${}^\circ\kappa_j^r$ differ in two ways. There may be actors in ${}^\circ\kappa_j^l$ that are not in ${}^\circ\kappa_j^r$. These will be actors created at hole entering steps whose exit has not yet be

reached. There will be no messages mentioning these actors in ${}^\circ\kappa^l_j$. In addition, for executing actors a whose computation has entered one or more holes (it is possible that evaluation of $e_0[{}^\circ\sigma]$ will touch a hole) and not yet exited, the actors state will have the form

(l) ${}^\circ e^l = {}^\circ R^l_1[\square := \texttt{let}\{x := {}^\circ e^l_1\}\, e[{}^\circ\sigma_1]]$

(r) ${}^\circ e^r = {}^\circ R^r_1[\square := \texttt{let}\{x := {}^\circ e^r_1\}\, e'[{}^\circ\sigma_1]]$

where $e' = \texttt{letactor}\{\bar{x} := \bar{v}\}\, e$, and ${}^\circ e^l_1$, ${}^\circ e^r_1$ are the same, or decompose similarly if a hole has been entered inside the `let` argument. There are three cases to consider:

(1) l_i enters a hole; and

(2) l_i is uniform with respect to ${}^\circ\kappa^l_i$

(2.1) l_i is a return step – (q, i) is in X_i for some q

(2.2) l_i is not a return step

Case 1: ${}^\circ\kappa^j_i$ has the form ${}^\circ\kappa^j_{i,0} \parallel \left\langle\!\!\left\langle\, [\![{}^\circ R^j[\square := \circ[{}^\circ\sigma]]]\!]_a \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle^a_\chi$ for $j \in \{l, r\}$, and l_i has the form $\texttt{<leta}: a, \bar{a}\texttt{>}$. Let p be the index of the step in π_l that return the value of this occurrence of e_0, or ∞ if there is no such step. Then $L_i = [\,]$, $X_{i+1} = X_i \cup \{(i, p)\}$, ${}^\circ\kappa^r_{i+1} = {}^\circ\kappa^r_i$, and ${}^\circ\kappa^l_{i+1}$ is such that

$$ {}^\circ\kappa^j_{i,0} \parallel \left\langle\!\!\left\langle\, [\![{}^\circ R^j[\square := e_l[{}^\circ\sigma]]]\!]_a \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle^a_\chi \xrightarrow{\texttt{<leta}: a, \bar{a}\texttt{>}} {}^\circ\kappa^l_{i+1}. $$

Case 2.1: In this case, there is some q such that $(q, i) \in X_i$, l_i has the form $\texttt{<fun}: a\texttt{>}$, and l_q has the form $\texttt{<leta}: a', \bar{a}\texttt{>}$, ($a$ is either a' or a clone resulting from execution of a `become`). ${}^\circ\kappa^l_i$ has the form

$$ {}^\circ\kappa^l_{i,0} \parallel \left\langle\!\!\left\langle\, [\![{}^\circ R^l[\square := \texttt{let}\{x := {}^\circ v\}\, e[{}^\circ\sigma]]]\!]_a \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle^a_\chi $$

and ${}^\circ\kappa^r_i$ has the form

$$ {}^\circ\kappa^r_{i,0} \parallel \left\langle\!\!\left\langle\, [\![{}^\circ R^r[\square := \texttt{let}\{x := {}^\circ v\}\, e'[{}^\circ\sigma]]]\!]_a \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle^a_\chi $$

where $e' = \texttt{letactor}\{\bar{x} := \bar{v}\}\, e$, and ${}^\circ\alpha^j_q(a) = [\![{}^\circ R^j[\square := \circ[{}^\circ\sigma]]]\!]$. Let

$$ {}^\circ\kappa^j_{i+1} = {}^\circ\kappa^j_{i,0} \parallel \left\langle\!\!\left\langle\, [\![{}^\circ R^r[\square := e[{}^\circ\sigma']]]\!]_a, (v_k[\square := e[{}^\circ\sigma']])_{a_k}\, 1 \le k \le \mathrm{Len}(\bar{v}) \;\middle|\; \emptyset \,\right\rangle\!\!\right\rangle^a_\chi $$

where ${}^\circ\sigma' = {}^\circ\sigma\{x := {}^\circ v, \bar{x} := \bar{a}\}$, and let $L_i = [l_i, \texttt{<leta}: a, \bar{a}\texttt{>}]$, and $X_{i+1} = X_i$.

Case 2.2: In this case ${}^\circ\kappa^j_i \xrightarrow{l_i} {}^\circ\kappa^j_{i+1}$ uniformly for $j \in \{l, r\}$, and we take $L_i = l_i$, and $X_{i+1} = X_i$.

It is now straightforward to show that

$$ \pi_r = [{}^\circ\kappa^r_i[\circ := e_r] \xrightarrow{L_i} {}^\circ\kappa^r_{i+1}[\circ := e_r] \;\middle|\; i < \bowtie] \in \mathcal{F}({}^\circ\kappa[\circ := e_r]) $$

and that $obs(\pi_l) = obs(\pi_r)$.

The converse direction is similar. Here, to ensure completeness/fairness of the result, we need to take account of the case where a hole $\circ[{}^\circ\sigma]$ is exposed, but there is no transition for $e_0[{}^\circ\sigma]$. As before we do this by inserting the letactor multi-step for e_l at the point where the hole is first exposed.

$\square$**delay**

6.2.9 The Proofs of the Remaining 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 idea is to generalize the notion of common reduct to allow for reduction of actor primitives, thus producing not just an actor state, but a fragment of an E-configuration. To express the generalized common reduct theorem, we first must say when we consider two E-configurations to be essentially the same. The idea is that two E-configurations are essentially the same if we ignore inactive actors not known to any other actors, and we allow replacement of hanging expressions or expressions with infinite computations by expressions of the same class.

Definition (essential sameness): E-expressions, ${}^{\circ}e_j$ for $j < 2$ are said to be essentially the same if they are the same expression, or if both are in $Hang \cup Infin$.

E-configurations, ${}^{\circ}\kappa_j$ for $j < 2$ are said to be essentially the same, for focus actor a_f, if there are ${}^{\circ}\alpha_j$, ${}^{\circ}\alpha_j^g$, ${}^{\circ}\mu$, and χ for $j < 2$ such that

(1) ${}^{\circ}\kappa_j = \left\langle\!\!\left\langle {}^{\circ}\alpha_j, {}^{\circ}\alpha_j^g \mid {}^{\circ}\mu \right\rangle\!\!\right\rangle_{\chi}^{\emptyset}$,

(2) $a_f \in \mathrm{Dom}({}^{\circ}\alpha_0)$, $\mathrm{Dom}({}^{\circ}\alpha_0) = \mathrm{Dom}({}^{\circ}\alpha_1)$ and ${}^{\circ}\alpha_0(a)$ is essentially the same as ${}^{\circ}\alpha_1(a)$ for $a \in \mathrm{Dom}({}^{\circ}\alpha_1)$, and

(3) $\mathrm{Dom}({}^{\circ}\alpha_j^g) \cap \mathrm{FV}({}^{\circ}\alpha_j, {}^{\circ}\mu, \chi) = \emptyset$ and ${}^{\circ}\kappa_j$ does not touch a hole at a and a is not enabled in ${}^{\circ}\kappa_j$ for any transitions, for $a \in \mathrm{Dom}({}^{\circ}\alpha_j^g)$ and $j < 2$.

To state the general result we begin with a few convenient definitions. First, we define an operation constructing an actor map whose range is unitialized actors created by a given actor.

Definition ($New(N, a)$): Let N be a finite set of actor addresses, with $a \notin N$. Define $New(N, a)$ to be the actor map with domain N such that $New(N, a)(a') = (?_a)$ for $a' \in N$.

Next we define the notion of a multi-step being initial for a configuration.

Definition (Initial multi-step): A multi-step L is said to be initial for a configuration κ if for any path $\pi \in \mathcal{F}(\kappa)$, there is a permutation equivalent path $\pi' \in \mathcal{F}(\kappa)$ such that L is the initial sequence of steps of π'. By permutation equivalent we mean that π' is obtained from π by permuting steps of π that are elements of L such that permuted pairs of steps have different focus actors.

Definition (Common generalized reduct): We call $(N, a, \chi, {}^{\circ}\sigma)$ an instance for the pair (e_0, e_1) if N, $\{a\}$, χ are pairwise disjoint finite sets of actor addresses, the domain of ${}^{\circ}\sigma$ contains the free variables of e_0, e_1, and the free variables of the range of ${}^{\circ}\sigma$ are among $N \cup \{a\} \cup \chi$. For such an instance we call

$$ {}^{\circ}\kappa_0^j = \left\langle\!\!\left\langle New(N, a), [\![e_j[{}^{\circ}\sigma]]\!]_a \mid \emptyset \right\rangle\!\!\right\rangle_{\chi}^{a} $$

the initial E-configuration. Two expressions e_0 and e_1 have common generalized reducts if for each instance $(N, a, \chi, {}^{\circ}\sigma)$ there are E-configurations, ${}^{\circ}\kappa^j$, and multi-steps L_j containing no input/output transitions, such that

(1) ${}^{\circ}\kappa_0^j \xrightarrow{L_j} {}^{\circ}\kappa^j$ for $j < 2$,

(2) ${}^{\circ}\kappa^j$ is essentially the same as ${}^{\circ}\kappa^1$, and

(3) L_j is initial for any configuration ${}^{\circ}\kappa_0^j \parallel \kappa$ such that κ and ${}^{\circ}\kappa_0^j$ are composable.

We note that the notion of common generalized reduct can easily be further generalized to allow for the case where the common reduct can be reached by more than one sequence of steps. We omit the details.

Theorem (gen-red-eq): If e_0 and e_1 have common generalized reducts, then $e_0 \cong e_1$.

Proof : The proof is almost the same as the proof of (**fun-red-eq**) (in §6.2.6)). We adopt the strategy used in the proof of (**hang-infin**) to insert (useless) steps of lambda-diverging computations to maintain fairness. The main changes are in the construction stages (i, j) for the cases 1 – execution in a hole at a, and (3.1) touching and entering a hole at a. To establish notation we consider case 3.1 first. Let N is the set of addresses of unitialized actors in ${}^\circ\kappa_{i,j}$ with creator a, and let ${}^\circ\alpha_{i,j}(a) = [\![{}^\circ R[\square := \circ[{}^\circ\sigma]]]\!]$. Pick χ so that $(N, a, \chi, {}^\circ\sigma)$ is an instance for (e_0, e_1) and let ${}^\circ\kappa_0^0$ be the initial configuration for this instance, modified by placing the state of a in the context ${}^\circ R$.

$$ {}^\circ\kappa_0^0 = \left\langle\!\!\left\langle New(N, a), [\![{}^\circ R[\square := e_0[{}^\circ\sigma]]]\!]_a \;\middle|\; \emptyset \right\rangle\!\!\right\rangle_\chi^a $$

Thus ${}^\circ\kappa_{i,j}$ has the form ${}^\circ\kappa'_{i,j} \parallel {}^\circ\kappa_0^0$. We let $L_0 = [l_q \mid q < p]$ be the multi-step selected from the collection given by the hypothesis to match the path under consideration, and let ${}^\circ\kappa_p^0$ be the given configuration. Thus there is a computation sequence $[{}^\circ\kappa_q^0 \xrightarrow{l_q} {}^\circ\kappa_{q+1}^0 \mid q < p]$. Let i_q be the index in π_0 of l_q. Then we move to stage $(i+1, 0)$ as follows: $n_j = i$, $I_{i+1} = I_i \cup \{i\}$, $J_{i+1} = J_i\{i := [i_q \mid q < p]$, ${}^\circ\kappa_{i+1,0} = {}^\circ\kappa'_{i,j} \parallel {}^\circ\kappa_1^0$.

For the case 1, i is an element of $J_i(m)$ for some $m \in I_i$. We move to stage $(i+1, 0)$ using the data of 3.1 with i replaced by m. $\square$

Note that the theorems (**fun-red-eq**) and (**hang-infin**) are special cases of (**gen-red-eq**).

Corollary (gen-red-eq): The following laws are instances of (**gen-red-eq**): (**triv**), (**gc**), (**if.letact.v**), (**perm**), (**split**), (**can-b**), (**can-i**), (**commutes**), and (**cellb**).

We sketch the proofs for (**gc**), (**if.letact.v**), (**can-i**), and (**cellb**). The remaining cases use similar arguments and are left to the reader. In each case we need to establish that the pair of expressions to be shown equivalent have common generalized reducts. For this we have to find (for each instance $(N, a, \chi, {}^\circ\sigma)$) the common reduct configuration, and show that the set of multi-steps leading to this configuration is initial.

(gc) $\quad e_0 = \texttt{letactor}\{\bar{x} := \bar{v}\}e \cong e = e_1 \qquad$ where $\bar{x}$ not free in e.

Proof (gc): Let $(N, a, \chi, {}^\circ\sigma)$ be an instance for (gc) and let ${}^\circ\kappa_0^j$ be the corresponding initial configurations for $j < 2$. Let L_0 be the multi-step that creates and initializes the actors specified by $\texttt{letactor}\{\bar{x} := \bar{v}\}$. Let L_1 be the empty multi-step. It is easy to see that the L_j are initial. The end E-configurations are essentially the same since the differ only in the choice of ${}^\circ\alpha_j^g$. $\square_{\mathbf{gc}}$

(if.letact.v) $\texttt{letactor}\{\bar{x} := \bar{v}\}\texttt{if}(z, e_1, e_2) \cong \texttt{if}(z,$
$\texttt{letactor}\{\bar{x} := \bar{v}\}\, e_1,$
$\texttt{letactor}\{\bar{x} := \bar{v}\}\, e_2)$

if z is not an element of $\bar{x}$.

Proof (if.letact.v): Let $(N, a, \chi, {}^{\circ}\sigma)$ be an instance for (if.letact.v) and let ${}^{\circ}\kappa_0^j$ be the corresponding initial configurations for $j < 2$. Let L_0 be the multi-step that creates and initializes the actors specified by $\texttt{letactor}\{\bar{x} := \bar{v}\}$, and then branches according to ${}^{\circ}\sigma(z)$. Let L_1 be the multi-step that branches according to ${}^{\circ}\sigma(z)$, and then creates and initializes the actors specified by $\texttt{letactor}\{\bar{x} := \bar{v}\}$, It is easy to see that the L_j are initial. The end E-configurations are in fact the same. $\square_{\textbf{if.letact.v}}$

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

Proof (can-i): The second equation follows from (**hang-infin**). For the first equation, let $(N, a, \chi, {}^{\circ}\sigma)$ be an instance for (**can-i**) and let ${}^{\circ}\kappa_0^j$ be the corresponding initial configurations for $j < 2$. There are two cases to consider according to whether or not ${}^{\circ}\sigma(v)$ is an actor address in N or not. If ${}^{\circ}\sigma(v) \in N$, then let L_j be the multi-step that initializes ${}^{\circ}\sigma(v)$ for $j < 2$. If ${}^{\circ}\sigma(v) \notin N$, then let L_j consist of the empty multi-step for $j < 2$. It is easy to see that in either the L_j are initial. The end E-configurations are essentially the same in either case because they differ only in replacing a hung expression by a hung or lambda-infinite expression. $\square_{\textbf{can-i}}$

(cellb) $\texttt{letactor}\{b := B_{\text{cell}}(0)\}\texttt{send}(b, \texttt{mkget}(v)) \cong \texttt{send}(v, 0)$

(Recall that the behavior of a cell, B_{cell}, was defined in §2.2)

Proof (cellb): Let $(N, a, \chi, {}^{\circ}\sigma)$ be an instance for (**cellb**) and let ${}^{\circ}\kappa_0^j$ be the corresponding initial configurations for $j < 2$. Let L_0 be the multi-step that creates the cell actor, sends it the message $\texttt{mkget}(v)$, delivers the message, and executes the transitions with cell actor focus, until the actor has no more enable transitions, i.e. until the `become` is executed. Let L_1 be the multi-step that does the send. The end configurations differ only by the presence of an inaccessible, inactive actor – the cell actor, since after the send to the cell actor, it is not known to any other actors. Clearly L_1 is initial. L_0 is initial, because there can only be one message sent to the cell actor, and the delivery and processing of that message can always be permuted ahead of any other transitions. $\square_{\textbf{cellb}}$

6.3 Equivalence by Two Stage Reduction

There is one remaining equivalence to establish using common reducts:

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

The intuitive reasoning behind this equivalence is that for any closing substitution (allowing holes, and actor addresses in the range) the two expressions reduce to

equivalent lambda expressions. In fact these lambda expressions have the property that when applied to any argument they reduce to a common expression.

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

6.3.1 LE-Syntax

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

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

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

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

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

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

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

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

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

Lemma (LE-expression decomposition):

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

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

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

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

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

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

where

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

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

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

and the constraints specified in the definition of actor configurations in §3. are satisfied. We let $^{\circ\triangleright}\kappa$ range over $^{\circ\triangleright}\mathbb{K}$, and $^{\circ\triangleright}\alpha$ range over $^{\circ\triangleright}\mathbb{A}\mathrm{c}$. Filling expression and abstraction holes of an LE-configuration, LE-actor map, LE-actor state, LE-multiset of messages, and LE-messages is defined in the obvious manner.

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

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

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

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

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

Lemma (LE-Uniform Computation):

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

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

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

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

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 $e_0, e_1, \varphi_{0,j}, \varphi_{1,j}$ for $j < J$ be such that for each closing ${}^{\circ\triangleright}\sigma$ we can find $j \in J$ such that $e_i[{}^{\circ\triangleright}\sigma]$ reduces uniformly via $\overset{\lambda}{\mapsto}_{\mathrm{FV}(\mathrm{Rng}({}^{\circ\triangleright}\sigma))}$ steps to $\varphi_{i,j}[{}^{\circ\triangleright}\sigma]$ for $i < 2$, and that for each ${}^{\circ\triangleright}\sigma$, ${}^{\circ\triangleright}v$, and $j \in J$ we can find ${}^{\circ\triangleright}e_c$ such that $\mathtt{app}(\varphi_{i,j}[{}^{\circ\triangleright}\sigma], {}^{\circ\triangleright}v)$ reduces uniformly via $\overset{\lambda}{\mapsto}_{\mathrm{FV}(\mathrm{Rng}({}^{\circ\triangleright}\sigma))}$ steps to ${}^{\circ\triangleright}e_c$ for $i < 2$. Then $e_0 \cong e_1$.

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

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

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

$$J = \{a, b\}$$

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

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

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

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

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

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

6.4 Equivalence of Reduction Contexts

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Lemma (R-expression decomposition):

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

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

(2) $(\exists! {}^\diamond R, {}^\diamond\sigma, {}^\diamond v)({}^\diamond e = {}^\diamond R[\square := \diamond^{\diamond\sigma}[\square := {}^\diamond v]])$

Proof : An easy induction on the structure of $^\diamond e$. $\square$

6.4.2 R-Configurations

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

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

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

$$^\diamond\mathbb{A}\mathrm{s} = (^\diamond\mathbb{V}) \cup [^\diamond\mathbb{E}] \cup \{(?_{\mathbb{A}\mathrm{d}})\}$$

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

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

6.4.3 R-Uniform Computation

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

Lemma (Uniform Computation): An R-redex reduces or hangs uniformly (for a given enabling occurrence in a configuration). Hence transitions not touching an R-hole are uniform. More precisely, if $^\diamond\kappa_i[\diamond := R] \xrightarrow{l_i} \kappa_{i+1}$, with focus a, that does not touch an R-hole at a, then l_i is either a receive or an execution transition in which the execution state of $^\diamond\kappa_i$ at a decomposes according to case (1) of the decomposition lemma. Let $^\diamond\kappa_i = \left\langle\!\!\left\langle\, ^\diamond\alpha_i \;\middle|\; ^\diamond\mu_i \,\right\rangle\!\!\right\rangle$. Then the decomposition of $\kappa_{i+1} = \left\langle\!\!\left\langle\, ^\diamond\alpha_{i+1} \;\middle|\; ^\diamond\mu_{i+1} \,\right\rangle\!\!\right\rangle$ is defined as follows.

Receive: In the receive case we must have that $^\diamond\alpha_i(a) = (\lambda x.^\diamond e)$. Thus $^\diamond\alpha_{i+1} = {}^\diamond\alpha_i\{a := [\mathtt{app}(\lambda x.^\diamond e, cv)]$, and $^\diamond\mu_i = {}^\diamond\mu_{i+1} + \langle a \Leftarrow cv\rangle$.

Uniform execution: In the uniform execution case $^\diamond\alpha_i(a) = [^\diamond e]$. where $^\diamond e$ has the form $^\diamond R[\square := {}^\diamond r]$. In this case the step is independent of what fills the holes. Thus we can find $^\diamond\kappa_{i+1}$ such that $^\diamond\kappa_i \xrightarrow{l_i} {}^\diamond\kappa_{i+1}$ uniformly.

6.4.4 R-Main theorem

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

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

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

In fact we prove a slightly more general result, since we use a weaker assumption

on the reduction contexts: for each ${}^{\diamond}\sigma$, ${}^{\diamond}v$, we can find ${}^{\diamond}e_c$ such that $R_j^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v]$ reduces uniformly via 0 or more $\overset{\lambda}{\mapsto}_{\mathrm{FV}(\mathrm{Rng}({}^{\diamond}\sigma))}$ steps to ${}^{\diamond}e_c$ 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 $R_0[\square := e] \cong R_1[\square := e]$ for all expressions e. Let ${}^{\diamond}\kappa = \left\langle\!\!\left\langle {}^{\diamond}\alpha \;\middle|\; {}^{\diamond}\mu \right\rangle\!\!\right\rangle$ be a closing R-configuration for R_0, R_1. Assume $\pi_0 \in \mathcal{F}({}^{\diamond}\kappa[\diamond := R_0]) = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \;\middle|\; i \in \bowtie]$. We want to find ${}^{\diamond}\kappa_i$, and L_i, such that $\kappa_i = {}^{\diamond}\kappa_i[\diamond := R_0]$ and, letting $\pi_1 = [{}^{\diamond}\kappa_i[\diamond := R_1] \xrightarrow{L_i} {}^{\diamond}\kappa_{i+1}[\diamond := R_1] \;\middle|\; i \in \bowtie]$, we have $\pi_1 \in \mathcal{F}({}^{\diamond}\kappa[\diamond := R_1])$ and $obs(\pi_0) = obs(\pi_1)$. As in the expression context case, we can focus our attention on the construction of the actor configuration part, since here also deliverable messages cannot have holes. Assume we have ${}^{\diamond}\alpha_i$ and consider cases on the transition label l_i. we have three cases to consider. For the base case we have ${}^{\diamond}\alpha_0 = {}^{\diamond}\alpha$. At stage i suppose ${}^{\diamond}\alpha_i(a') = {}^{\diamond}R[\square := {}^{\diamond}{}^{\diamond\sigma}[\square := {}^{\diamond}v]]$. If $R_0^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v]$ is the common form (i.e. the reduction is trivial), then we prefix L_i with steps for reduction of $R_1^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v]$ to common form and remove this hole. This is carried out for each a' in the domain of ${}^{\diamond}\alpha_i$.

Now, suppose l_i is an execution with focus a that touches a hole. Thus ${}^{\diamond}\alpha_i(a) = [{}^{\diamond}e]$ where ${}^{\diamond}e$ has the form ${}^{\diamond}R[\square := {}^{\diamond}{}^{\diamond\sigma}[\square := {}^{\diamond}v]]$. Suppose also that $R_j^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v] \overset{\lambda}{\mapsto}_{\mathrm{FV}(\mathrm{Rng}({}^{\diamond}\sigma))} {}^{\diamond}e_c$ for $j < 2$. Then we define ${}^{\diamond}\alpha_{i+1} = {}^{\diamond}\alpha_i\{a := [{}^{\diamond}R[\square := {}^{\diamond}e_c]]\}$.

Let $\pi_1 = [{}^{\diamond}\kappa_i[\diamond := R_1] \xrightarrow{L_i} {}^{\diamond}\kappa_{i+1}[\diamond := R_1] \;\middle|\; i < \bowtie]$ be the constructed computation path. Note that the transitions are the same except for the points where holes are touched, but these differences are not observable. Clearly, under the assumptions on R_j, π_1 is a computation path. It remains to show that the construction preserves fairness.

Suppose some transition l is enabled at stage i in π_1. If l is a receive or uniform execution, then l is also enabled in π_0 at stage i and will eventually occur, uniformly. Suppose l is an execution step by a with ${}^{\diamond}\alpha_i(a) = [{}^{\diamond}R[\square := {}^{\diamond}{}^{\diamond\sigma}[\square := {}^{\diamond}v]]]$. Either $R_0^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v]$ is in common form and the transition occurs as soon as it is enabled in π_1, or a transition is enabled at this hole in π_0. This transition will eventually occur. If $R_1^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v]$ is not in common form, l will happen in π_1 at the same stage. If $R_1^{{}^{\diamond}\sigma}[\square := {}^{\diamond}v]$ is in common form, then we must consider whether l occurs at the hole – i.e. whether or not the common form is an R-value expression. If l occurs at the hole, then the same transition is now enabled in π_0 and will eventually occur in both paths. Otherwise consider the decomposition after the transition in π_0. As shown before, this reduces to considering a proper subexpression of ${}^{\diamond}R[\square := {}^{\diamond}{}^{\diamond\sigma}[\square := {}^{\diamond}v]]$ and the process will eventually terminate with a uniform execution.

$\square_{\mathbf{eq-r}}$

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, and we incorporate fairness into the semantics by restricting the set of admissible computation paths. 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.

The theory presented is perhaps best viewed as a starting point for further research rather than a final product. There are several directions for further research. Work is needed to develop an algebra of operations on configurations. Treating configurations as objects would allow us to abstract over specific linguistic constructs. A set of laws and proof principles adequate for reasoning about actor programs is needed. These would include structural induction as well as principles analogous to the simulation (co-) induction principles we developed for reasoning about streams, mutable data, and objects (Talcott, 1993a; Mason and Talcott, 1991; Mason and Talcott, 1994). 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, in (Mason and Talcott, 1991; Honsell et al., 1995) a variant of Milner's context lemma (Milner, 1977), called the (**ciu**) theorem, is proved. This reduces the number of contexts that must be considered to establish an observational equivalence law and greatly simplifies the proofs. It is an open question whether such a reduction can be made in the case of actor systems.

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. They would also like to thank the referees for careful reading of earlier versions, and numerous helpful criticisms and editor Phil Wadler for his interest and help with the revision process.

This research was partially supported by ARPA contract NAG2-703, ARPA/ONR grant N00014-94-1-0775 NSF grants CCR-8917606, CCR-8915663, CCR-9109070 CCR-9221774 CCR-9312580 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

Abadi, M. and Cardelli, L. (1994). A theory of primitive objects: Untyped and first-order

systems. In *Theoretical Aspects of Computer Software*, number 789 in Lecture Notes in Computer Science, pages 296–320. Springer-Verlag.

Abelson, H. and Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company.

Abramsky, S. (1991). A domain equation for bisimulation. *Information and Computation*, 92(2):161–218.

Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass.

Agha, G. (1990). Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141.

Agha, G., Frølund, S., Kim, W., Panwar, R., Patterson, A., and Sturman, D. (1993a). Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14.

Agha, G., Hewitt, C., Wegner, P., and Yonezawa, A., editors (1991). *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*. OOPS Messenger, vol. 2, no. 2.

Agha, G., Mason, I. A., Smith, S. F., and Talcott, C. L. (1992). Towards a theory of actor computation. In *The Third International Conference on Concurrency Theory (CONCUR '92)*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer Verlag.

Agha, G., Wegner, P., and Yonezawa, A., editors (1989). *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*. Special Issue of SIGPLAN Notices.

Agha, G., Wegner, P., and Yonezawa, A., editors (1993b). *Research Directions in Concurrent Object Oriented Programming*. MIT Press.

Amadio, R. M. (1994). Translating core facile. Technical Report ECRC-1994-3, European Computer-Industry Research Centre.

Armstrong, J., Virding, R., and Williams, M. (1993). *Concurrent Programming in Erlang*. Prentice Hall.

Attardi, G. and Hewitt, C. (1978). Specifying and proving properites of guardians for distributed systems.

Baker, H. G. and Hewitt, C. (1977). Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP.

Bergstra, J. A. and Klop, J. W. (1986). *Process Algebra: Specification and Verification in Bisimulation Semantics*. Number CWI Monograph 4 in Mathematics and Computer Science, II. North-Holland.

Brookes, S., Hoare, C., and Roscoe, A. (1984). A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599.

Cardelli, L. (1986). Amber. In *Combinators and functional programming languages*, volume 242 of *Lecture notes in computer science*. Springer-Verlag.

Cardelli, L. (1994). Obliq: A language with distributed scope. Technical Report SRC Research Report 122, Digital Equipment Corporation, Systems Research Center.

Clinger, W. D. (1981). Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory.

de Nicola, R. and Hennessy, M. C. B. (1984). Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133.

Felleisen, M. and Friedman, D. (1986). Control operators, the SECD-machine, and the λ-calculus. In Wirsing, M., editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland.

Felleisen, M. and Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271.

Felleisen, M. and Wright, A. K. (1991). A syntactic approach to type soundness. Technical Report Rice COMP TR91-160, Rice University Computer Science Department. To appear, *Information and Computation*.

Giacalone, A., Mishra, P., and Prasad, S. (1989). Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160.

Greif, I. (1975). Semantics of communicating parallel processes. Technical Report 154, 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.

Hewitt, C. (1977). Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364.

Hewitt, C. and Atkinson, R. (1979). Specifications and proof techniques for serializers. *IEEE Transactions on Software Engineering*, SE-5(1):10–23.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modulal actor formalism for artificial intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*, pages 235–245.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Honda, K. and Tokoro, M. (1991). An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag.

Honsell, F., Mason, I. A., Smith, S. F., and Talcott, C. L. (1995). A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90.

Kornfeld, W. A. and Hewitt, C. (1981). The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1).

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6:308–320.

Lieberman, H. (1987). Concurrent object-oriented programming in act-1. In Yonezawa, A. and Tokoro, M., editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press.

Manning, C. (1987). ACORE: The design of a core actor language and its compiler. Master's thesis, MIT, Artificial Intelligence Laboratory.

Mason, I. A. and Talcott, C. L. (1991). Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327.

Mason, I. A. and Talcott, C. L. (1994). Reasoning about object systems in vtloe. submitted to International Journal of Foundations of Computer Science.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155.

Milner, R. (1977). Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22.

Milner, R. (1983). Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Milner, R., Parrow, J. G., and Walker, D. J. (1989). A calculus of mobile processes, Parts I and II. Technical Report ECS-LFCS-89-85, -86, Edinburgh University.

Moggi, E. (1988). Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh.

Morris, J. H. (1968). *Lambda calculus models of programming languages*. PhD thesis, Massachusetts Institute of Technology.

Pierce, B. C. and Turner, D. N. (1994). Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, Lecture Notes in Computer Science. Springer-Verlag. To appear, 1995.

Plotkin, G. (1975). Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159.

Prasad, S., Giacalone, A., and Mishra, P. (1990). Operational and algebraic semantics for facile: A symmetric integration of concurrent and functional programming. In Paterson, M. S., editor, *17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780.

Reppy, J. H. (1991). An operational semantics of first-class synchronous operations. Technical Report TR 91-1232, Cornell University.

Sassone, V., Nielsen, M., and Winskel, G. (1993). A hierarchy of models for concurrency. In *The Fourth International Conference on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science. Springer Verlag.

Steele, G. L. and Sussman, G. J. (1975). Scheme, an interpreter for extended lambda calculus. Technical Report Technical Report 349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory.

Talcott, C. L. (1985). *The essence of Rum: A theory of the intensional and extensional aspects of Lisp-type computation.* PhD thesis, Stanford University.

Talcott, C. L. (1989). Programming and proving with function and control abstractions. Technical Report STAN-CS-89-1288, Stanford University Computer Science Department.

Talcott, C. L. (1991). Binding structures. In Lifschitz, V., editor, *Artificial Intelligence and Mathematical Theory of Computation.* Academic Press.

Talcott, C. L. (1993a). A theory for program and data specification. *Theoretical Computer Science*, 104:129–159.

Talcott, C. L. (1993b). A theory of binding structures and its applications to rewriting. *Theoretical Computer Science*, 112:99–143.

Thomsen, B., Leth, L., and Giacalone, A. (1992). Some issues in the semantics of facile distributed programming. Technical Report ECRC-1992-32, European Computer-Industry Research Centre.

Tomlinson, C., Cannata, P., Meredith, G., and Woelk, D. (1993). The extensible services switch in carnot. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):16–20.

Tomlinson, C., Kim, W., Schevel, M., Singh, V., Will, B., and Agha, G. (1989). Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93.

van Glabbeek, R. J. (1990). *Comparative concurrency semantics and refinement of actions.* PhD thesis, Free University of Amsterdam, Centre for Mathematics and Computer Science.

Yonezawa, A. (1990). *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge Mass.

8 Index of Notations

Symbol	Description	§
$\mathbb{N}$	The natural numbers, $i, j, \ldots, n \in \mathbb{N}$	2.4
Y^n	Sequences from Y of length n, $\bar{y} = [y_1, \ldots, y_n] \in Y^n$	2.4
Y^*	Finite sequences from Y	2.4
$[]$	The empty sequence	2.4
$\mathrm{Len}(\bar{y})$	The length of the sequence $\bar{y}$	2.4
$u * v$	The concatenation of the sequences u and v	2.4
$\mathrm{Last}(u)$	The last element of the sequence u	2.4
$\mathbf{P}_\omega[Y]$	Finite subsets of Y	2.4
$\mathbf{M}_\omega[Y]$	Finite multi-sets with elements in Y	2.4
$Y_0 \xrightarrow{\mathrm{f}} Y_1$	Finite maps from Y_0 to Y_1	2.4
$Y_0 \to Y_1$	Total functions from Y_0 to Y_1	2.4
$\mathrm{Dom}(f)$	The domain of the function f	2.4
$\mathrm{Rng}(f)$	The range of the function f	2.4
$f\{y := y'\}$	An extension to, or alteration of, the function f	2.4
$f\rceil Y$	The restriction of f to the set Y	2.4
$\mathbb{X}$	A countably infinite set of variables, $x, y, z \in \mathbb{X}$	3.1
$\mathbb{A}\mathrm{t}$	Atoms	3.1
$\mathtt{t}, \mathtt{nil}$	Atoms playing the role of booleans	3.1
$\mathbb{G}_n$	n-ary algebraic operations	3.1
$\mathbb{F}$	Operations, $\delta \in \mathbb{F}$	3.1
$\mathbb{F}_n$	n-ary operation symbols	3.1
$\mathbb{F}_0$	Zero-ary operation symbols $\supseteq \{\mathtt{newadr}\}$	3.1
$\mathbb{F}_1$	Unary operation symbols $\supseteq \{\mathtt{isatom}, \mathtt{isnat}, \mathtt{ispr}, \mathtt{1}^{\mathtt{st}}, \mathtt{2}^{\mathtt{nd}}, \mathtt{become}\}$	3.1
$\mathbb{F}_2$	Binary operation symbols $\supseteq \{\mathtt{pr}, \mathtt{initbeh}, \mathtt{send}\}$	3.1
$\mathbb{F}_3$	Ternary operation symbols $\supseteq \{\mathtt{br}\}$	3.1
$\mathbb{L}$	λ-abstractions, $\lambda x.e \in \mathbb{L}$	3.1
$\mathbb{V}$	Value expressions, $v \in \mathbb{V}$	3.1
$\mathbb{E}$	Expressions, $e \in \mathbb{E}$	3.1
$\lambda x.e$	Abstractions	3.1
$\mathtt{app}(e_0, e_1)$	Application	3.1
$\delta(\bar{e})$	Application of operations	3.1
$\mathtt{if}(e_0, e_1, e_2)$	Conditional branching	3.1
$\mathtt{let}\{x := e_0\}e_1$	Lexical variable binding	3.1
$\mathtt{seq}(e_1, \ldots, e_n)$	Sequencing construct	3.1

Symbol	Description	§
$\mathrm{FV}(e)$	The free variables of the expression e	3.1
$e[x := e']$	The result of substituting e' for x in e	3.1
$\mathbb{C}$	Contexts, $C \in \mathbb{C}$	3.1
$\bullet$	The hole in contexts	3.1
$C[e]$	The result of filling the context with e	3.1
$\mathbb{A}\mathrm{d}$	Actor addresses, identified with $\mathbb{X}$	3.2
$\mathbb{A}\mathrm{s}$	Actor states, $(?_a), (b), [e] \in \mathbb{A}\mathrm{s}$	3.2
$(?_a)$	An uninitialized actor created by a	3.2
(b)	An actor with behavior b ready to accept a message	3.2.1
$[e]$	A processing actor with current computation e	3.2.1
$\mathbb{M}$	Messages, $<\mathbb{V} \Leftarrow \mathbb{V}> \in \mathbb{M}$	3.2.1
$\mathrm{c}\mathbb{V}$	Communicable values, $cv \in \mathrm{c}\mathbb{V}$	3.2.1
$\left\langle\!\!\left\langle \alpha \mid \mu \right\rangle\!\!\right\rangle^{\rho}_{\chi}$	An actor configuration with:	
	α – an actor map	
	μ – a multi-set of messages	
	ρ – the receptionists	
	χ – the external actors	3.2.1
$\mathbb{K}$	Actor configurations, $\kappa \in \mathbb{K}$	3.2.2
$\mathbb{E}_{\mathrm{rdx}}$	The set of redexes, $r \in \mathbb{E}_{\mathrm{rdx}}$	3.2.2
$\mathbb{R}$	The set of reduction contexts, $R \in \mathbb{R}$	3.2.2
$\square$	The reduction context hole	3.2.2
$\overset{\lambda}{\mapsto}_X$	The reduction relation for functional redexes, $e_0 \overset{\lambda}{\mapsto}_X e_1$	3.2.2
$\mapsto$	The reduction relation for configurations, $\kappa_0 \mapsto \kappa_1$	3.2.2
$\kappa_0 \xrightarrow{l} \kappa_1$	$\kappa_0 \mapsto \kappa_1$ via the rule labelled by l	3.2.2
Labels :	Transition labels, $l \in$ Labels	3.2.2
`<fun` : a`>`	A functional transition	3.2.2
`<new` : a, a'`>`	`newadr` redex transition with focus a	3.2.2
`<init` : a, a'`>`	`initbeh` redex transition with focus a	3.2.2
`<bec` : a, a'`>`	`become` redex transition with focus a	3.2.2
`<send` : a, m`>`	`send` redex transition with focus a	3.2.2
`<rcv` : a, cv`>`	The receipt of a message with focus a	3.2.2
`<out` : m`>`	A message exiting the configuration	3.2.2
`<in` : m`>`	A message entering the configuration	3.2.2

Symbol	Description	§
$\mathcal{T}(\kappa)$	All finite sequences of labeled transitions from κ, $\nu \in \mathcal{T}(\kappa)$	3.2.3
$\mathcal{T}^{\infty}(\kappa)$	the set of all computation paths in $\mathcal{T}(\kappa)$, $\pi \in \mathcal{T}^{\infty}(\mathbb{K})$	3.2.3
$\bowtie \in \mathbb{N} \cup \{\omega\}$	The length of a finite or infinite sequence	3.2.3
$\mathrm{Cfig}(\kappa, L, i)$	The ith configuration of the computation from κ via L	3.2.3
$\kappa \xrightarrow{L} \kappa'$	A multi-step transition	3.2.3
$\mathcal{F}(\kappa)$	the fair subset of $\mathcal{T}^{\infty}(\kappa)$	3.2.4
`event`	A zero-ary primitive/observation	4.1
`<e` : a`>`	An observation transition	4.1
$\langle\!\langle \alpha, [C]_a \mid \mu \rangle\!\rangle$	An observing configuration	4.1
$\mathbb{O}$	The set of observing configurations, $O \in \mathbb{O}$	4.1
s	Signifies that an `event` transition occurs	4.1
f	Signifies that an `event` transition does not occur	4.1
$obs(\pi)$	The **s**/**f** classification of the path π	4.1
$Obs(\kappa)$	The **s**/**f** classification of the configuration, κ, $\in \{\mathbf{s}, \mathbf{f}, \mathbf{sf}\}$	4.1
$e_0 \cong_1 e_1$	Testing or Convex or Plotkin or Egli-Milner equivalence	4.2
$e_0 \cong_2 e_1$	Must or Upper or Smyth equivalence	4.2
$e_0 \cong_3 e_1$	May or Lower or Hoare equivalence	4.2
$e_0 \cong e_1$	Operational equivalence (either $\cong_1$ or equivalently $\cong_2$)	4.2
Hang	The set of all stuck expressions, `stuck` $\in$ *Hang*	5.1
Infin	The set of all diverging expressions, `bot` $\in$ *Infin*	5.1