The AMST Language: Formal Verification and Execution of Actor Systems

Carlos A. Varela(✉)

Rensselaer Polytechnic Institute, Troy, NY 12180, USA
cvarela@cs.rpi.edu

Abstract. We formalize Agha, Mason, Smith, and Talcott's actor language (AMST) [2] using the Athena proof assistant [3]. Since Athena is a dual deduction and computation language, we can both rigorously prove formal properties of actor systems, as well as synthesize executable actor code from the theory.

Keywords: Actor model · AMST language · Formal verification

1 Introduction

The actor model [1] is an elegant, simple, and expressive concurrent computation model. Agha, Mason, Smith, and Talcott formalized the model using a simple actor language extending the lambda calculus with three actor primitives [2]. We introduce an executable version of their language and call it *AMST*. Agha *et al.*'s paper presented the language's syntax, operational semantics, and theory, and we largely follow the original presentation. The lambda calculus [4] is used to model sequential computation within each actor. This sequential computing calculus is another elegant and simple model, yet Turing-complete, with only three syntactic constructs and one main semantic rule: β-reduction.

Athena is a many-sorted first-order logic proof assistant [3] that contains *methods* and *procedures* to abstract over deductions and computations respectively. When methods succeed, they produce *theorems*, which are guaranteed to be sound by the language, *i.e.*, they logically follow from the axioms assumed during the method's execution.

In this paper, we formalize AMST in Athena, providing a formal basis for reasoning with machine-checked proofs, as well as a method to *execute* AMST actor programs by viewing the theory not only as a set of first-order logic sentences encoding universally quantified equations, but also by viewing these equations as a set of left-to-right rewriting rules.

The paper's outline is as follows: Sect. 2 introduces the AMST actor language by way of examples illustrating how to execute AMST programs in Athena. Section 3 specifies AMST's syntax and operational semantics. The section includes a full machine-checkable proof of AMST actor expressions' unique decomposition into a reduction context and a redex. It also specifies computation sequences, computation trees, and fairness. Finally, Sect. 4 discusses the paper's contributions and potential future work.

J. Meseguer et al. (Eds.): Gul Agha Festschrift, LNCS 16120, pp. 36–59, 2026.
https://doi.org/10.1007/978-3-032-05291-9_2

2 AMST Actor Language

The AMST actor language is an extension of the untyped call-by-value lambda calculus with the following actor primitives:

- **send** for sending messages,
- **letactor** for actor creation, and
- **become** for changing behavior.

2.1 Trivial Examples

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

```
define B5 := (App Y
                  (Lambda 'y
                    (Lambda 'x
                      (Seq (Send (Var 'x) Five)
                           (Become (Var 'y))))))

define B5Test := (LetActor 'z B5 (Send (Var 'z) (Var 'a)))
```

where `Var`, `Lambda`, and `App` are constructs for lambda calculus expressions; `Seq` is syntactic sugar for sequential composition, `Y` is the call-by-value fixed-point recursion combinator, `Five` is syntactic sugar for Church numeral 5; `Send`, `LetActor`, and `Become` are actor primitives; and `'x`, `'y`, `'z`, and `'a` are Athena identifiers, which we use for variable and actor names.

Using Athena's `eval` procedure, we can use the AMST actor theory to execute this example, as follows:

```
> (eval (execute B5Test))

Term:
(Actors.kfg
 # Actor map redacted
 (MSet.insert
  (Actors.msg 'a
              (Lambda 'f
                      (Lambda 'x
                              (App (Var 'f)
                                   (App (Var 'f)
                                        (App (Var 'f)
                                             (App (Var 'f)
                                                  (App (Var 'f)
                                                       (Var 'x)))))))))
  MSet.null:(MSet.MSet Actors.Message)))
```

where we can observe that the final actor configuration contains a message to actor `'a` with value 5 in Church numeral form $\lambda f.\lambda x.(f\ (f\ (f\ (f\ (f\ x)))))$.

An equivalent expression of this behavior is:

```
define B5' := (App Y
                   (Lambda 'y
                     (Lambda 'x
                       (Seq (Become (Var 'y))
                            (Send (Var 'x) Five)))))
```

since the order of executing the `become` and the `send` cannot be observed.

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

```
define Sink:= (App Y
                   (Lambda 'b
                           (Lambda 'm
                                   (Become (Var 'b)))))

define SinkTest := (LetActor 'z Sink (Send (Var 'z) (Var 'a)))
```

In this example, execution shows the following final configuration:

```
> (eval (execute SinkTest))

Term:
(Actors.kfg
 (FMap.update
  (pair 'f0
        (Ready
         (Lambda 'y
                 (App (App (Lambda 'x
                                   (App (Lambda 'b
                                                (Lambda 'm
                                                        (Become (Var 'b))))
                                        (Lambda 'y
                                                (App (App (Var 'x)
                                                          (Var 'x))
                                                     (Var 'y)))))
                           (Lambda 'x
                                   (App (Lambda 'b
                                                (Lambda 'm
                                                        (Become (Var 'b))))
                                        (Lambda 'y
                                                (App (App (Var 'x)
                                                          (Var 'x))
                                                     (Var 'y))))))
                      (Var 'y)))))
  (FMap.update
   (pair 'f1
         (Lambda 'x
                 (Var 'x)))
   (FMap.update
    (pair 'main
          (Lambda 'x
                  (Var 'x)))
    FMap.empty-map:(FMap.Map Ide AExp))))
 MSet.null:(MSet.MSet Actors.Message))
```

where we can observe the final actor configuration including an actor map with actor `'f0` encoding the `Sink` behavior—after expansion of the `Y` combinator, the actors `'f1` and `main` with `Nil` behavior—represented as the identity combinator $\lambda x.x$, and an empty multiset (`null`) representing the fact that the message to the sink actor was effectively consumed. We can also trace the actor program to see the transition labels in the actor computation as a list:

```
> (eval (trace SinkTest))

Term: (:: (Actors.fun 'main)
          (:: (Actors.fun 'main)
              (:: (Actors.fun 'main)
                  (:: (Actors.new 'main 'f0)
                      (:: (Actors.snd 'main)
                          (:: (Actors.rcv 'f0
```

```
                                          (Var 'a))
                              (:: (Actors.fun 'f0)
                                  (:: (Actors.bec 'f0 'f1)
                                      nil:(List Actors.Label))))))))
```

where we can observe in the initial `'main` actor—produced by `execute` and `trace`—three functional transitions converting the `Sink` expression into a value (expanding the `Y` combinator), the creation of the `Sink` actor named `'f0`, the sending of message containing `'a` from `'main` to `'f0`, the reception of the message by `'f0`, and function application becoming ready to receive new messages with the same `Sink` behavior, and letting the (empty) continuation be processed by anonymous actor `'f1`.

`(execute e)` is used to start the computation in a configuration with a single `'main` actor mapped to the initial program `e`, and an empty network (multiset of messages.) It is defined in Athena as follows:

```
declare execute : [AExp] -> AConfig

assert* execute-def :=
[((execute e) = (reduce* (kfg (['main e] ++ empty-map)
                              empty-mset)
                         (gen-fresh 'f 30)))]
```

where we pass an initial list of 30 fresh names to choose from, which explains why the actor executing the `Sink` behavior is named `'f0` and the continuation of the `Sink` actor after `Become` is named `'f1`[1].

2.2 Actor Cells

A *reference cell*—one of the simplest stateful concurrent programs—can be represented in the AMST actor language, as follows:

```
define Cell :=
(App Y
  (Lambda 'b
    (Lambda 'c
      (Lambda 'm
        (If (App Get? (Var 'm))
            (Seq (Become (App (Var 'b) (Var 'c)))
                 (Send (App Cust (Var 'm)) (Var 'c)))
            (If (App Set? (Var 'm))
                (Become (App (Var 'b) (App Contents (Var 'm))))
                (Become (App (Var 'b) (Var 'c)))))))))

define CellTest :=
(LetActor 'a
  (App Cell Zero)
  (Seq (Send (Var 'a) (App MkSet Three))
       (Seq (Send (Var 'a) (App MkSet Four))
            (Send (Var 'a) (App MkGet (Var 'c))))))
```

with auxiliary functions to create *get* and *set* message types, and access their contents as follows:

```
define MkGet    := (Lambda 'c (Pr True (Var 'c)))
define MkSet    := (Lambda 'c (Pr False (Var 'c)))
```

[1] In general, we want to pass a lazily generated infinite list of fresh names.

```
define Get?      := 1st
define Set?      := (App (App Compose Not) 1st)
define Cust      := 2nd
define Contents := 2nd
```

Executing this code can result in customer actor `'c` receiving a message containing 0, 3, or 4, depending on the cell's message processing order. Following is an execution where the customer actor gets 0, and the cell contains value 3 at the end of the execution:

```
> (eval (execute CellTest))

Term:
(Actors.kfg
 (FMap.update
  (pair 'f0
        (Ready (Lambda 'm
               # context redacted here for readability
                       (Send (App (Lambda 'p
                                          (App (Var 'p)
                                               (Lambda 'x
                                                       (Lambda 'y
                                                               (Var 'y)))))
                                  (Var 'm))
                             (Lambda 'f
                                     (Lambda 'x
                                             (App (Var 'f)
                                                  (App (Var 'f)
                                                       (App (Var 'f)
                                                            (Var 'x))))))))))
               # context redacted here for readability
  # additional "garbage" actors redacted
  FMap.empty-map:(FMap.Map Ide AExp))
 (MSet.insert (Actors.msg 'c
                          (Lambda 'f
                                  (Lambda 'x
                                          (Var 'x))))
              MSet.null:(MSet.MSet Actors.Message)))
```

where 0 is represented as the Church numeral $\lambda f.\lambda x.x$, and the cell on a subsequent `get` request will send to the customer (`(Cust m)`) the Church numeral 3: $\lambda f.\lambda x.(f\ (f\ (f\ x)))$. Notice that booleans are represented as $\lambda x.\lambda y.x$ for `True`, and $\lambda x.\lambda y.y$ for `False`, and pairs are represented as abstractions that take `True` to obtain the first element, and `False` to obtain the second, which is why `(Cust m)`—which uses `(2nd m)`—takes `m` as a pair and applies `False` to it, to obtain the customer.

2.3 Join Continuations

A prototypical recursive computation such as a binary tree product can be made concurrent using the AMST actor language as follows:

```
define TProd :=
  (App Y
    (Lambda 'b
      (Lambda 'self
        (Lambda 'm
          (Seq
           (Become (App (Var 'b) (Var 'self)))
           (If (App IsNat (App Tree (Var 'm)))
```

```
9                (Send (App Cust (Var 'm)) (App Tree (Var 'm)))
10               (LetActor
11                'nc
12                (App JoinC (App Cust (Var 'm)))
13                (Seq (Send (Var 'self)
14                           (Pr (App Left (App Tree (Var 'm)))
15                               (Var 'nc)))
16                     (Send (Var 'self)
17                           (Pr (App Right (App Tree (Var 'm)))
18                               (Var 'nc))))))))))
19
20 define JoinC :=
21   (Lambda 'c
22     (Lambda 'n1
23       (Become (Lambda 'n2
24                 (Send (Var 'c) (App (App Mult (Var 'n1)) (Var 'n2)))))))
```

where `TProd` denotes the behavior of the actor computing the tree product, and `JoinC` denotes the behavior of a *join continuation* actor that receives the products of the left and right sub-trees (computed concurrently) and sends their product to its customer.

A sample interaction follows:

```
define Tree1 := (Pr (Pr One Two) False)
define Tree2 := (Pr (Pr Tree1 Three) False)

define TProdTest2 :=
  (LetActor 't (App TProd (Var 't))
               (Send (Var 't) (Pr Tree2 (Var 'c))))
```

where leaf trees are encoded as Church numerals and non-leaf trees are encoded as `(Pr (Pr Left Right) False)`[2], resulting in the following output (redacted to include only the network component of the final configuration):

```
> (eval (execute TProdTest2))
#...
(MSet.insert
 (Actors.msg 'c
             (Lambda 'x
               (App (Lambda 'f
                      (Lambda 'x
                        (App (Var 'f)
                             (App (Var 'f)
                                  (App (Var 'f)
                                       (Var 'x))))))
                    (App (Lambda 'x
                           (App (Lambda 'f
                                  (Lambda 'x
                                          (App (Var 'f)
                                               (App (Var 'f)
                                                    (Var 'x)))))
                                (App (Lambda 'f
                                       (Lambda 'x
                                         (App (Var 'f)
                                              (Var 'x))))
                                     (Var 'x))))
                         (Var 'x)))))
 MSet.null:(MSet.MSet Actors.Message))
#...
```

[2] `isNat`–as used in the `TProd` behavior's line 8–is a lambda calculus function that returns `True` for Church numerals, and `2nd` for pairs: $\lambda e.((e\ \lambda x.\mathrm{True})\ \mathrm{True})$.

where we notice that the customer actor `'c` receives a message encoding $3 \times 2 \times 1$ (Church numeral multiplication is function composition.)

3 A Simple Lambda-Based Actor Language

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

3.1 Syntax

The syntax for AMST actor language expressions is given in Fig. 1, where the first three syntactic forms encode the lambda calculus, the `Br` form permits branching in a call-by-value language, and the last four syntactic forms encode the three actor primitives, plus a `Ready` primitive that allows an actor to change state without creating an anonymous actor[3].

```
structure AExp := (Var Ide)
                | (Lambda Ide AExp)
                | (App AExp AExp)
                | (Br AExp AExp AExp)
                | (LetActor Ide AExp AExp)
                | (Send AExp AExp)
                | (Become AExp)
                | (Ready AExp)
```

Fig. 1. AMST actor language syntax.

We use the lambda calculus to represent nil, booleans, numbers, and pairs as follows:

```
# Combinators:
define I := (Lambda 'x (Var 'x))
define Compose := (Lambda 'f
                    (Lambda 'g
                      (Lambda 'x
                         (App (Var 'f) (App (Var 'g) (Var 'x))))))
define Nil := I

# Booleans
define True  := (Lambda 'x (Lambda 'y (Var 'x)))
define False := (Lambda 'x (Lambda 'y (Var 'y)))
define Not := (Lambda 'b (App (App (Var 'b) False) True))
```

[3] The `(Ready b)` expression is used to signal that an actor can receive new messages with behavior `b`, a lambda abstraction, and it is written `(b)` in [2] which is distinguished from `[e]` which denotes sequential computing within the actor. In [5], we use `Ready` instead of `Become` to reduce the number of actor creations, which is useful when we implement actors as active objects. In AMST, we keep both forms, and we could show that `(Ready b)` is equivalent to `(Become b)` with an empty continuation.

```
#  Church Numerals
define Zero := (Lambda 'f (Lambda 'x (Var 'x)))
define One :=  (Lambda 'f (Lambda 'x (App (Var 'f) (Var 'x))))
define Two :=  (Lambda 'f (Lambda 'x (App (Var 'f)
                                           (App (Var 'f) (Var 'x)))))
define Three := # ...
define Four := # ...
define Five := # ...
define Succ := (Lambda 'n
                 (Lambda 'f (Lambda 'x
                   (App (Var 'f)
                        (App (App (Var 'n) (Var 'f)) (Var 'x))))))
define Mult := Compose

# Pairs
define PrC  := (Lambda 'x
                (Lambda 'y
                  (Lambda 'b
                    (App (App (Var 'b) (Var 'x)) (Var 'y)))))
define 1st := (Lambda 'p (App (Var 'p) True))
define 2nd := (Lambda 'p (App (Var 'p) False))

define Pr := lambda (x y)
               (App (App PrC x) y)
```

Notice that `Pr` is an Athena procedure, which uses its `lambda` syntactic form, that takes two AMST expressions `x` and `y` and produces the AMST expression `(App (App PrC x) y)`.

We also use the lambda calculus to represent let, sequencing, conditionals, and recursion, as follows:

```
# sequencing combinator
define Let := lambda (x e1 e2)
                (App (Lambda x e2) e1)
define SeqC := (Lambda 'x
                (Lambda 'y
                  (App (Lambda 'z (Var 'y)) (Var 'x))))

define Seq := lambda (e1 e2)
                (App (App SeqC e1) e2)

# conditional form using Br
define If := lambda (b t e)
               let {fresh := (gen-fresh-e 'if (App t e))}
               (App (Br b (Lambda fresh t) (Lambda fresh e)) (Var 'ifoo))

# applicative order Y combinator
define Y :=
(Lambda 'f
  (App (Lambda 'x (App (Var 'f)
                       (Lambda 'y (App (App (Var 'x) (Var 'x)) (Var 'y)))))
       (Lambda 'x (App (Var 'f)
                       (Lambda 'y (App (App (Var 'x) (Var 'x)) (Var 'y)))))))
```

3.2 Reduction Semantics for Actor Configurations

The operational semantics for the AMST actor language is given by a labeled transition relation on configurations. Actor configurations contain an actor map and a multi-set of messages representing messages in-transit.

```
datatype Message := (msg Ide AExp)

datatype AConfig := (kfg (FMap.Map Ide AExp) (MSet.MSet Message))
```

where `Ide` denotes Athena identifiers, such as `'a` which we use for variable and actor names; and `AExp` denotes an actor expression, as specified in Sect. 3.1.

Notation. `(FMap.Map Ide AExp)` denotes a finite map from identifiers to actor states, represented as actor expressions. `(MSet.MSet Message)` denotes a multiset of messages. We define and overload the following operators:

```
define ++ := FMap.++
overload ++ MSet.++

define at  := FMap.at
define dom := FMap.dom

define -  := MSet.-
overload - FMap.-

define in := MSet.in
overload in Set.in
overload in List.in
```

so that we may use `[a e] ++ amap` to denote the extension of actor map `amap` to include a mapping from `a` to `e`; but also so that we may use `(msg a v) ++ n` to denote the addition of a message to actor `a` with content `v` to the network, represented as multiset `n`. Furthermore, we may use `amap at a` to obtain the actor state corresponding to `a` in the actor map `amap`; `dom amap` to obtain the domain of actor map `amap`; `n - (msg a v)` to remove one occurrence of message `(msg a v)` from the network `n`; `amap - a` to remove the actor `a` from the actor map `amap`; `(msg a v) in n` to check whether a message is in the network; and `a in dom amap` to check for membership of actor named `a` in actor map `amap`.

Actor Configurations. An actor configuration with actor map `amap` and network `n` is denoted as `(kfg amap n)`.

The set of possible computations of an actor configuration is defined in terms of the labeled transition relation `reduce`:

```
# operational semantics transition labels
datatype Label := (fun Ide)
               |  (new Ide Ide)
               |  (snd Ide)
               |  (rcv Ide AExp)
               |  (bec Ide Ide)

# given a config and a label produces the next config
declare reduce : [AConfig Label] -> AConfig
```

where the five label types denote how the actor `a`—*in focus*—makes progress as follows:

- `(fun a)`, to denote functional (lambda calculus β-reduction) progress,
- `(new a a')`, to denote creation of actor `a'` by `a`,
- `(snd a)`, to denote message sending by actor `a`,
- `(rcv a v)`, to denote message reception by actor `a` with message content `v`, and
- `(bec a a')` to denote becoming a new actor behavior, and letting an anonymous actor `a'` carry out the remaining computation.

Decomposition and Reduction. To describe actor transitions, we classify expressions into *value* and *non-value* expressions:

```
structure AVal  := (VarV Ide)
                |  (LambdaV Ide AExp)

structure AE    := (Val AVal)
                |  (AppE AE AE)
                |  (BrE AE AE AE)
                |  (LetActorE Ide AE AE)
                |  (SendE AE AE)
                |  (BecomeE AE)
                |  (ReadyE AE)
```

with the idea to decompose non-value actor expressions (`AE`) uniquely into a reduction context filled with a redex. Reduction contexts identify the subexpression of an expression that is to be evaluated next, to correspond to a left-first, call-by-value evaluation strategy.

Following is the definition of actor redexes (`ARedex`) and actor reduction contexts (`ARC`):

```
structure ARedex := (AppR AVal AVal)
                 |  (BrR AVal AE AE)
                 |  (LetActorR Ide AVal AE)
                 |  (SendR AVal AVal)
                 |  (BecomeR AVal)
                 |  (ReadyR AVal)

structure ARC    :=  Hole
                 |   (AppCL AVal ARC)
                 |   (AppCR ARC AE)
                 |   (BrC ARC AE AE)
                 |   (LetActorC Ide ARC AE)
                 |   (SendCL AVal ARC)
                 |   (SendCR ARC AE)
                 |   (BecomeC ARC)
                 |   (ReadyC ARC)
```

An expression `e` (in `AExp`) 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 1 (Unique decomposition). $\forall e.(e \in AVal \vee (\exists! R, r)(e = R[r]))$.

To formally prove this lemma in Athena, we will first introduce some notation. To determine whether an expression is a value, we introduce the `isValue` predicate:

```
declare isValue : [AExp] -> Boolean

assert* isValue-def :=
[(isValue   (Var v))
 (isValue   (Lambda v e))
 (~ isValue (App e1 e2))
 (~ isValue (Br e1 e2 e3))
 (~ isValue (LetActor v e1 e2))
 (~ isValue (Send e1 e2))
 (~ isValue (Become e'))
 (~ isValue (Ready e'))]
```

where only the variable name and lambda abstraction forms are considered value expressions.

We also introduce a function `fill-in` to represent $R[r]$, *i.e.*, reduction context `R` filled in with redex `r`, as follows:

```
# R[r] is denoted as (fill-in R r)
declare fill-in : [ARC AExp] -> AExp

assert* fill-in-def :=
  [((fill-in Hole e)                = e)
   ((fill-in (AppCL val R) e)       = (App (val->aexp val) (fill-in R e)))
   ((fill-in (AppCR R ae) e)        = (App (fill-in R e) (ae->aexp ae)))
   ((fill-in (BrC R ae ae') e)      =
                            (Br (fill-in R e) (ae->aexp ae) (ae->aexp ae')))
   ((fill-in (LetActorC v R ae) e) = (LetActor v (fill-in R e) (ae->aexp ae)))
   ((fill-in (SendCL val R) e)      = (Send (val->aexp val) (fill-in R e)))
   ((fill-in (SendCR R ae) e)       = (Send (fill-in R e) (ae->aexp ae)))
   ((fill-in (BecomeC R) e)         = (Become (fill-in R e)))
   ((fill-in (ReadyC R) e)          = (Ready (fill-in R e)))]
```

where we use auxiliary functions to convert the operational semantics-internal data types `AVal` and `AE` back to the AMST actor expressions `AExp`.

Finally, we define a `next-redex` function to take an actor expression in `AExp`, and optionally decompose it into a reduction context `R` (in `ARC`) and a redex `r` (in `ARedex`). If the expression is a value, we produce `NONE`, as follows:

```
# next-redex takes an actor expression AExp and optionally produces a
# reduction context R and a redex r
declare next-redex : [AExp] -> (Option (Pr ARC ARedex))

assert* next-redex-def :=
  [((next-redex (Var v))            = NONE)
   ((next-redex (Lambda v e))       = NONE)
   ((next-redex (App e1 e2)         = (SOME (pr Hole (AppR (aexp->val e1)
                                                           (aexp->val e2)))))
                                    <== (  (next-redex e1 = NONE)
                                         & (next-redex e2 = NONE)))
   ((next-redex (App e1 e2)         = (SOME (pr (AppCL (aexp->val e1) R) r)))
                                    <== (  (next-redex e1 = NONE)
                                         & (next-redex e2 = (SOME (pr R r)))))
   ((next-redex (App e1 e2)         = (SOME (pr (AppCR R (aexp->ae e2)) r)))
                                    <== (next-redex e1 = (SOME (pr R r))))
   ((next-redex (Br e1 e2 e3)       = (SOME (pr Hole (BrR (aexp->val e1)
                                                          (aexp->ae e2)
                                                          (aexp->ae e3)))))
                                    <== (next-redex e1 = NONE))
   ((next-redex (Br e1 e2 e3)       = (SOME (pr (BrC R (aexp->ae e2)
                                                    (aexp->ae e3)) r)))
                                    <== (next-redex e1 = (SOME (pr R r))))
   ((next-redex (LetActor v e1 e2) = (SOME (pr Hole (LetActorR v (aexp->val e1)
                                                                 (aexp->ae e2)))))
                                    <== (next-redex e1 = NONE))
   ((next-redex (LetActor v e1 e2) = (SOME (pr (LetActorC v R (aexp->ae e2)) r)))
                                    <== (next-redex e1 = (SOME (pr R r))))
   ((next-redex (Send e1 e2)        = (SOME (pr Hole (SendR (aexp->val e1)
                                                            (aexp->val e2)))))
                                    <== (  (next-redex e1 = NONE)
                                         & (next-redex e2 = NONE)))
   ((next-redex (Send e1 e2)        = (SOME (pr (SendCL (aexp->val e1) R) r)))
                                    <== (  (next-redex e1 = NONE)
                                         & (next-redex e2 = (SOME (pr R r)))))
   ((next-redex (Send e1 e2)        = (SOME (pr (SendCR R (aexp->ae e2)) r)))
                                    <== (next-redex e1 = (SOME (pr R r))))
   ((next-redex (Become e)          = (SOME (pr Hole (BecomeR (aexp->val e)))))
                                    <== (next-redex e = NONE))
   ((next-redex (Become e)          = (SOME (pr (BecomeC R) r)))
```

```
                                 <== (next-redex e = (SOME (pr R r))))
((next-redex (Ready e)           = (SOME (pr Hole (ReadyR (aexp->val e)))))
                                 <== (next-redex e = NONE))
((next-redex (Ready e)           = (SOME (pr (ReadyC R) r)))
                                 <== (next-redex e = (SOME (pr R r))))
]
```

where we use auxiliary functions to convert AMST actor expressions `AExp` into the operational semantics-internal structures `AVal` and `AE`.

Now, we can express Lemma 1 on unique decomposition of expressions:

```
define unique-redex-lemma :=
  (forall e . ((isValue e) |
               (exists R r . ((next-redex e = SOME (pr R r)) &
                  (forall R' r' . ((next-redex e = SOME (pr R' r')) ==>
                                  ((R = R') & (r = r'))))))))
```

The proof is by structural induction on actor expressions:

```
let {premise := lambda (e)
                  ((isValue e) |
                   (exists R r . ((next-redex e = SOME (pr R r)) &
                      (forall R' r' . ((next-redex e = SOME (pr R' r')) ==>
                                      ((R = R') & (r = r')))))));
     eVal := method (e)
               (!chain<- [    (premise e)
                          <== (isValue e)                [prop-taut]
                          <== true                       [isValue-def]]);
     uniqueM := method (e R r redex-e)
                  pick-any R' r'
                    assume (next-redex e = SOME (pr R' r'))
                      (!chain<- [((R = R') & (r = r'))
                          <== ((pr R r) = (pr R' r'))          [pr-dt-axioms]
                          <== ((SOME (pr R r)) = (SOME (pr R' r')))
                                                               [opt-axioms]
                          <== (redex-e & (next-redex e = SOME (pr R' r')))
                                [method (p)
                                  (!combine-equations
                                    (!sym (!left-and p))
                                    (!sym (!right-and p)))]
                          <== true                             [prop-taut]]);
     e1-cases := method (e e1 R1 r1 R2 r2)
       let {ih-e1 := (premise e1);
            existsP := (exists R r . ((next-redex e = SOME (pr R r))) &
              (forall R' r' . ((next-redex e = SOME (pr R' r')) ==>
                               ((R = R') & (r = r')))));
            _ := conclude existsP
              (!cases ih-e1
                assume e1-val := (isValue e1)
                  let {no-redex-e1 :=
                         (!chain<- [    (next-redex e1 = NONE)
                                    <== e1-val [val-no-redex]]);
#   ((next-redex e = (SOME (pr R1 r1))) <== (next-redex e1 = NONE))
                       redex-e := (!chain<- [
                             ((next-redex e) = (SOME (pr R1 r1)))
                         <== no-redex-e1                  [next-redex-def]]);
                       unique := (!uniqueM e R1 r1 redex-e);
                       _ := (!both redex-e unique)
                      }
                  (!egen* existsP [R1 r1])
                assume e1-noval :=
                  (exists R r . ((next-redex e1 = SOME (pr R r)) &
                     (forall R' r' . ((next-redex e1 = SOME (pr R' r')) ==>
                                     ((R = R') & (r = r'))))))
#   ((next-redex e = (SOME (pr R2' r2')))
#         <== (next-redex e1 = (SOME (pr R-e1 r-e1))))
                  pick-witnesses R-e1 r-e1 for e1-noval e1-redex-unique
```

```
                    let {sub-redex := (make-sub [[R R-e1] [r r-e1]]);
                         R2' := (sub-redex R2);
                         r2' := (sub-redex r2);
                         redex-e1 := conclude
                           (next-redex e1 = SOME (pr R-e1 r-e1))
                             (!left-and e1-redex-unique);
                         redex-e := (!chain<- [
                               (next-redex e = (SOME (pr R2' r2')))
                           <== redex-e1                  [next-redex-def]]);
                         unique := (!uniqueM e R2' r2' redex-e);
                         _ := (!both redex-e unique)
                        }
                    (!egen* existsP [R2' r2'])
              )}
       (!chain<- [(premise e) <== existsP  [prop-taut]]);
     e1-e2-cases := method (e e1 e2 R1 r1 R2 r2 R3 r3)
       let {ih-e1 := (premise e1);
            ih-e2 := (premise e2);
            existsP := (exists R r . ((next-redex e = SOME (pr R r))) &
              (forall R' r' . ((next-redex e = SOME (pr R' r')) ==>
                               ((R = R') & (r = r')))));
            _ := conclude existsP
              (!cases ih-e1
                assume e1-val := (isValue e1)
                let {no-redex-e1 := (!chain<- [
                         (next-redex e1 = NONE) <== e1-val [val-no-redex]])}
                  (!cases ih-e2
                    assume e2-val := (isValue e2)
#   ((next-redex e = (SOME (pr R1 r1)))
#    <== (  (next-redex e1 = NONE) & (next-redex e2 = NONE)))
                      let {no-redex-e2 := (!chain<- [
                           (next-redex e2 = NONE) <== e2-val [val-no-redex]]);
                           redex-e := (!chain<- [
                               ((next-redex e) = (SOME (pr R1 r1)))
                           <== (no-redex-e1 & no-redex-e2) [next-redex-def]]);
                           unique := (!uniqueM e R1 r1 redex-e);
                           _ := (!both redex-e unique)
                          }
                      (!egen* existsP [R1 r1])
                    assume e2-noval :=
                      (exists R r . ((next-redex e2 = SOME (pr R r)) &
                        (forall R' r' . ((next-redex e2 = SOME (pr R' r')) ==>
                                         ((R = R') & (r = r'))))))
#   ((next-redex (App e1 e2) = (SOME (pr R2' r2')))
#       <== ((next-redex e1 = NONE) & (next-redex e2 = (SOME (pr R r)))))
                      pick-witnesses R-e2 r-e2 for e2-noval e2-redex-unique
                        let {sub-redex := (make-sub [[R R-e2] [r r-e2]]);
                             R2' := (sub-redex R2);
                             r2' := (sub-redex r2);
                             redex-e2 := conclude
                               (next-redex e2 = SOME (pr R-e2 r-e2))
                                 (!left-and e2-redex-unique);
                             redex-e := (!chain<- [
                                   (next-redex e = (SOME (pr R2' r2')))
                               <== (no-redex-e1 & redex-e2)
                                                          [next-redex-def]]);
                             unique := (!uniqueM e R2' r2' redex-e);
                             _ := (!both redex-e unique)
                            }
                        (!egen* existsP [R2' r2'])
                  )
                assume e1-noval :=
                  (exists R r . ((next-redex e1 = SOME (pr R r)) &
                    (forall R' r' . ((next-redex e1 = SOME (pr R' r')) ==>
                                     ((R = R') & (r = r'))))))
#   ((next-redex e = (SOME (pr R3' r3')))
#    <== (next-redex e1 = (SOME (pr R r))))
                  pick-witnesses R-e1 r-e1 for e1-noval e1-redex-unique
```

```
117                     let {sub-redex := (make-sub [[R R-e1] [r r-e1]]);
118                          R3' := (sub-redex R3);
119                          r3' := (sub-redex r3);
120                          redex-e1 := conclude
121                            (next-redex e1 = SOME (pr R-e1 r-e1))
122                              (!left-and e1-redex-unique);
123                          redex-e := (!chain<- [
124                                  (next-redex e = (SOME (pr R3' r3')))
125                              <== redex-e1                 [next-redex-def]]);
126                          unique := (!uniqueM e R3' r3' redex-e);
127                          _ := (!both redex-e unique)
128                         }
129                     (!egen* existsP [R3' r3'])
130               )}
131        (!chain<- [(premise e) <== existsP  [prop-taut]])
132     }
133 by-induction unique-redex-lemma {
134   (e as (Var v))         => (!eVal e)
135 | (e as (Lambda v e'))  => (!eVal e)
136 | (e as (App e1 e2))    => (!e1-e2-cases e e1 e2
137                               Hole (AppR (aexp->val e1) (aexp->val e2))
138                               (AppCL (aexp->val e1) R) r
139                               (AppCR R (aexp->ae e2)) r)
140 | (e as (Br e1 e2 e3))  => (!e1-cases e e1
141                               Hole (BrR (aexp->val e1) (aexp->ae e2)
142                                         (aexp->ae e3))
143                               (BrC R (aexp->ae e2) (aexp->ae e3)) r)
144 | (e as (LetActor v e1 e2)) => (!e1-cases e e1
145                               Hole (LetActorR v (aexp->val e1) (aexp->ae e2))
146                               (LetActorC v R (aexp->ae e2)) r)
147 | (e as (Send e1 e2))   => (!e1-e2-cases e e1 e2
148                               Hole (SendR (aexp->val e1) (aexp->val e2))
149                               (SendCL (aexp->val e1) R) r
150                               (SendCR R (aexp->ae e2)) r)
151 | (e as (Become e1))    => (!e1-cases e e1
152                               Hole (BecomeR (aexp->val e1))
153                               (BecomeC R) r)
154 | (e as (Ready e1))     => (!e1-cases e e1
155                               Hole (ReadyR (aexp->val e1))
156                               (ReadyC R) r)
157 }
```

Lines 133–157 consider actor expressions in all their possible syntactic forms, and use the following Athena methods—the equivalent of lambda abstractions for deduction—to conclude the premise:

- `eVal`, for value expressions,
- `e1-cases`, for expressions that can only contain the reduction context in one argument position, and
- `e1-e2-cases`, for expressions that can contain the reduction context in two possible argument positions.

The `eVal` method (lines 6–9), concludes the premise for value expressions by using a backward implication chain, where the two steps are justified by the `isValue` definition axioms, and propositional tautology—namely, $p \rightarrow (p \lor q)$. We use the `premise` Athena procedure, (lines 1–5) to produce the first-order logic sentence desired for each individual actor expression.[4]

[4] Notice that a *procedure* producing a first-order logic sentence, such as `premise`, does not check whether it follows from the axioms; in contrast, a *method*, such as `eVal`, only succeeds to produce a "theorem", when it can deduce the expression logically from known axioms.

The `e1-cases` method (lines 23–63) takes the actor expression `e`, its sub-expression `e1` for which we are going to assume by inductive hypothesis that the premise is true, and the pairs $(R1, r1)$ and $(R2, r2)$ to consider the two decompositions of `e` for the cases on whether `e1` is a value (lines 30–41), or not (lines 42–61). If `e1` is a value, we can deduce that `(next-redex e1 = NONE)` by the `val-no-redex` lemma (lines 31–33), and that allows us to deduce that $R1[r1]$ is a valid decomposition of `e` by the `next-redex` definition (lines 35–37). We also deduce that this decomposition is unique by using the `uniqueM` method (line 38). Finally, we combine the decomposition existence, and its uniqueness (line 39), and use existential generalization to conclude the premise (line 41). If `e1` is not a value, we know `e1-noval`, the second disjunct of the inductive hypothesis (lines 42 45). We pick witnesses `R-e1` and `r-e1` for this existentially quantified sentence, as the unique reduction context and redex, respectively for `e1`. Following, we can take the context and redex given in arguments `R2` and `r2` and substitute our witnesses for `R` and `r` getting `R2'` and `r2'` (lines 49–51). Subsequently, we prove that `e` can be decomposed into $R2'[r2']$ by using `next-redex`'s definition (lines 52–57). We also deduce that this decomposition is unique by using the `uniqueM` method (line 58). Finally, we combine the decomposition existence, and its uniqueness (line 59), and use existential generalization to conclude the premise (line 61). By propositional tautology, we can finally deduce the premise for `e` from the second disjunct named `existsP` (line 63).

The `uniqueM` method (lines 10–22) shows that for any decomposition of `e`, $R'[r']$, it must be the case that $R' = R$ and $r' = r$, assuming that $e = R[r]$ (`redex-e`). This method's deduction is justified by transitivity of equality (`combine-equations` method in lines 17–21), and datatype axioms for pairs and options, particularly, their *no-confusion* axioms (lines 13–16).

The `e1-e2-cases` method (lines 64–131) considers three possible decompositions for actor expression `e`: $R1[r1]$, $R2[r2]$, and $R3[r3]$. The method also takes advantage of the inductive hypotheses for sub-expressions `e1` and `e2` (lines 65–66). The deduction follows the cases of `e1` being a value (lines 72–109), or not (lines 110–130). Inside the case of `e1` being a value, we consider for `e2` being a value (lines 76–87), or not (lines 88–108). When both `e1` and `e2` are values, we can demonstrate that `R1` and `r1` are the context and redex satisfying the unique decomposition property, in a similar way to the `e1-cases` method's `e1-val` case. When `e1` is a value, and `e2` is not a value, we pick witnesses `R-e2` and `r-e2` for the existentially quantified inductive hypothesis for `e2`, and use them to prove the unique decomposition $R2'[r2']$. Finally, when `e1` is not a value, we demonstrate the unique decomposition of `e` into $R3'[r3']$. In these last two cases, we substitute the witness reduction context and redex into the arguments given to the method to create the respective unique reduction context and redex for `e`.

The `val-no-redex` lemma—used in lines 31–33, 73–74, and 79–80—is proven by structural induction on actor expressions. The first two forms `Var` and `Lambda` are values, and by the `next-redex` definition return `NONE`. The remaining (non-value) forms, use the `isValue` definition and contradiction (`from-complements` method) to prove the consequent of the implication.

```
define val-no-redex :=
  (forall e . ((isValue e) ==> (next-redex e = NONE)))

let { eVal := method (e)
                assume (isValue e)
                  (!chain<- [    (next-redex e = NONE)
                             <== true                    [next-redex-def]]);
      eNoVal := method (e)
                  assume (isValue e)
                    (!from-complements
                      (next-redex e = NONE)
                      (isValue e)
                      (!chain<- [(~ isValue e) <== true  [isValue-def]]))}
by-induction val-no-redex {
  (e as (Var v))            => (!eVal e)
| (e as (Lambda v e))       => (!eVal e)
| (e as (App e1 e2))        => (!eNoVal e)
| (e as (Br e1 e2 e3))      => (!eNoVal e)
| (e as (LetActor v e1 e2)) => (!eNoVal e)
| (e as (Send e1 e2))       => (!eNoVal e)
| (e as (Become e'))        => (!eNoVal e)
| (e as (Ready e'))         => (!eNoVal e)
}
```

Following are the theorems that Athena outputs corresponding to the `val-no-redex` and `unique-redex-lemma` deductions above:

```
Sentence Actors.val-no-redex defined.

Theorem:
(forall ?e:AExp
  (if (Actors.isValue ?e:AExp)
      (= (Actors.next-redex ?e:AExp)
         NONE:(Option (Pr Actors.ARC Actors.ARedex)))))

Sentence Actors.unique-redex-lemma defined.

Theorem:
(forall ?e:AExp
  (or (Actors.isValue ?e:AExp)
      (exists ?R:Actors.ARC
        (exists ?r:Actors.ARedex
          (and (= (Actors.next-redex ?e:AExp)
                  (SOME (pr ?R:Actors.ARC ?r:Actors.ARedex)))
               (forall ?R':Actors.ARC
                 (forall ?r':Actors.ARedex
                   (if (= (Actors.next-redex ?e:AExp)
                          (SOME (pr ?R':Actors.ARC ?r':Actors.ARedex)))
                       (and (= ?R:Actors.ARC ?R':Actors.ARC)
                            (= ?r:Actors.ARedex ?r':Actors.ARedex)))))))))))
```

We can now proceed to define the operational semantics of AMST as a reduction relation for actor configurations—see Fig. 2.

The first two rules labeled `(fun a)`, consider functional progress within an actor `a`, when its redex is either an `AppR` or a `BrR`. They rely on an auxiliary `beta` function to perform β-reduction (the main operational semantics rule from the lambda calculus) extended with branching, as follows:

```
declare beta: [AExp] -> AExp

assert* beta-def :=
  [(beta (App (Lambda v e1) e2) = (sub e1 e2 v))
   (beta (Br e e1 e2)           = e1            <== (e = True))
   (beta (Br e e1 e2)           = e2            <== (e = False))]
```

```
# given a config and a label produces the next config
declare reduce : [AConfig Label] -> AConfig

assert* reduce-def :=
  [((reduce (kfg amap n) (fun a)) =
           (kfg ([a (fill-in R
                                 (beta (App (val->aexp val)
                                            (val->aexp val'))))] ++
                  (amap - a))
                 n)
           <== (next-redex (option-val (amap at a)) =
                             (SOME pr R (AppR val val'))))
   ((reduce (kfg amap n) (fun a)) =
           (kfg ([a (fill-in R
                         (beta (Br (val->aexp val)
                                   (ae->aexp ae) (ae->aexp ae'))))] ++
                  (amap - a))
                 n)
           <== (next-redex (option-val (amap at a)) =
                             (SOME pr R (BrR val ae ae'))))
   ((reduce (kfg amap n) (new a af)) = # af fresh
           (kfg ([af (Ready (sub (val->aexp bv) (Var af) a'))] ++
                  [a (fill-in R (sub (ae->aexp ae) (Var af) a'))] ++
                  (amap - a))
                 n)
           <== ((next-redex (option-val (amap at a)) =
                             (SOME pr R (LetActorR a' bv ae))) &
                 (~ af in (dom amap))))
   ((reduce (kfg amap n) (snd a)) =
           (kfg ([a (fill-in R Nil)] ++ (amap - a))
                 ((msg a' (val->aexp cv)) ++ n))
           <== ((next-redex (option-val (amap at a)) =
                             (SOME pr R (SendR (VarV a') cv)))))
   ((reduce (kfg amap n) (rcv a e')) =
           (kfg ([a (App (val->aexp bv) e')] ++ (amap - a))
                 (n - (msg a e')))
           <== (((msg a e') in n) &
                 (next-redex (option-val (amap at a)) =
                             (SOME pr R (ReadyR bv)))))
   ((reduce (kfg amap n) (bec a a')) = # a' fresh
           (kfg ([a (Ready (val->aexp bv))] ++
                  [a' (fill-in R Nil)] ++
                  (amap - a))
                 n)
           <== ((next-redex (option-val (amap at a)) =
                             (SOME pr R (BecomeR bv))) &
                 (~ a' in (dom amap))))
   ]
```

Fig. 2. AMST actor language operational semantics.

which itself relies on alpha-equivalence of expressions—to compare `e` to `True` and `False`, and the substitution function:

```
#  (sub e1 e2 v) denotes e1{e2/v}
declare sub: [AExp AExp Ide] -> AExp

assert* sub-def :=
  [((sub (Var v) e v)             = e)
   ((sub (Var v) e v2)            = (Var v) <== v =/= v2)
   ((sub (Lambda v e) e2 v)       = (Lambda v e))
   ((sub (Lambda v e) e2 v2)      = (Lambda v (sub e e2 v2)) <==  v =/= v2)
   ((sub (App e1 e2) e v)         = (App (sub e1 e v) (sub e2 e v)))
```

```
((sub (Br e1 e2 e3) e v)        = (Br (sub e1 e v) (sub e2 e v) (sub e3 e v)))
((sub (LetActor v e1 e2) e v)   = (LetActor v e1 e2))
((sub (LetActor v e1 e2) e v2) = (LetActor v (sub e1 e v2) (sub e2 e v2))
                                  <== v =/= v2)
((sub (Send e1 e2) e v)         = (Send (sub e1 e v) (sub e2 e v)))
((sub (Become e1) e v)          = (Become (sub e1 e v)))
((sub (Ready e1) e v)           = (Ready (sub e1 e v)))
]
```

The next four actor redexes are as follows:

The `LetActorR` redex reduction rule creates an actor with fresh name `af`, ready to process messages with behavior `bv` and lets the parent actor fill in the `LetActor` reduction context with expression `ae`. We substitute the fresh actor name `af` for `a'` in both the new actor's behavior and the continuation `ae`, so that both the actor and its parent know about this new actor name. This rule only is enabled if the fresh actor name in the label `(new a af)` is actually fresh, *i.e.*, af $\notin$ (dom amap).

The `SendR` redex reduction rule adds a new message directed to actor `a'` to the network, and lets the sending actor `a` continue immediately with its continuation—*i.e.*, its reduction context filled in with `Nil`.

The `ReadyR` redex reduction rule is enabled when there is a message directed to `a` in the network with content `e'`. Actor `a` applies its behavior (a lambda abstraction) to the incoming message's content, and one occurrence of the message is removed from the network (using multiset difference). Notice that the reduction context `R` is not used.

The `BecomeR` redex reduction rule makes the actor in focus `a`, `Ready` to receive subsequent messages with behavior `bv`, while creating an anonymous actor (with fresh and unknown name `a'`), to carry out its continuation. This rule only is enabled if the fresh actor name in the label `(bec a a')` is actually fresh, *i.e.*, $a' \notin$ (dom amap).

Definition 1 ((enabled k l)). *A transition* `l` *is* enabled *in the configuration* `k`*, written* `(enabled k l)`*, iff there is a configuration* `k'` *such that* `k' = (reduce k l)`*. Otherwise, the transition is* disabled.

We define an `enabled` function, using a `redexes` function that computes all the enabled transitions in a given configuration:

```
# given a config and a list of possible fresh names, produce multi-set
# of enabled transitions
declare redexes: [AConfig (List Ide)] -> (MSet.MSet Label)

assert* redexes-def :=
[((redexes (kfg amap n) fvars) = (redexes_dm (kfg amap n) (dom amap) fvars))]

declare enabled : [AConfig (List Ide) Label] -> Boolean

assert* enabled-def :=
[((enabled k fvars l) <==> l in (redexes k fvars))]
```

where the auxiliary `redexes_dm` function ensures that fresh names in labels do not conflict with existing actor names:

```
# given a config, a set of actor names, and a list of possible fresh
# names, produce multi-set of enabled transitions
declare redexes_dm: [AConfig (Set.Set Ide) (List Ide)] -> (MSet.MSet Label)

assert* redexes_dm-def :=
[((redexes_dm (kfg empty-map n) dm fvars) = empty-mset)
 ((redexes_dm (kfg ([a e] ++ amap) n) dm fvars) = (redexes_dm (kfg amap n) dm fvars)
          <== ((next-redex e) = NONE))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm fvars) =
           ((fun a) ++ (redexes_dm (kfg amap n) dm fvars))
          <== ((next-redex e) = (SOME pr R (AppR val1 val2))))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm fvars) =
           ((fun a) ++ (redexes_dm (kfg amap n) dm fvars))
          <== ((next-redex e) = (SOME pr R (BrR val ae ae'))))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm (af :: fvars)) =
           ((new a af) ++ (redexes_dm (kfg amap n) dm fvars))
          <== (((next-redex e) = (SOME pr R (LetActorR a' val ae'))) &
              (~ af in dm)))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm (af :: fvars)) =
           (redexes_dm (kfg ([a e] ++ amap) n) dm fvars)
          <== (((next-redex e) = (SOME pr R (LetActorR a' val ae'))) &
              (af in dm)))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm fvars) =
           ((snd a) ++ (redexes_dm (kfg amap n) dm fvars))
          <== ((next-redex e) = (SOME pr R (SendR val1 val2))))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm (a' :: fvars)) =
           ((bec a a')
           ++ (redexes_dm (kfg amap n) dm fvars))
          <== (((next-redex e) = (SOME pr R (BecomeR val))) &
              (~ a' in dm)))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm (a' :: fvars)) =
           (redexes_dm (kfg ([a e] ++ amap) n) dm fvars)
          <== (((next-redex e) = (SOME pr R (BecomeR val))) &
              (a' in dm)))
 ((redexes_dm (kfg ([a e] ++ amap) n) dm fvars) =
           ((msgs n a) MSet.\/ (redexes_dm (kfg amap n) dm fvars))
           <== ((next-redex e) = (SOME pr R (ReadyR val))))
]
```

which itself depends on a `msgs` function to compute all labels corresponding to incoming messages for an actor `a`:

```
# for each message in network, create a label
declare msgs: [(MSet.MSet Message) Ide] -> (MSet.MSet Label)

assert* msgs-def :=
[((msgs empty-mset a) = empty-mset)
 ((msgs ((msg a  e') ++ n) a) = ((rcv a e') ++ (msgs n a)))
 ((msgs ((msg a' e') ++ n) a) = (msgs n a) <== (a =/= a'))]
```

Computation Sequences and Paths

Definition 2 (Computation trees). *If* `k` *is a configuration, then we define the* computation tree *for* `k`, `(ctree k)`*, to be the set of all finite sequences of labeled transitions of the form* $[l_i \mid k_{i+1} = (reduce\ k_i\ l_i) \wedge i < n]$ *for some* $n \in \mathbb{N}$*, with* $k = k_0$*. We call such sequences* computation sequences.

```
# computation sequence as a list of labels
define-sort CSequence := (List Label)
# computation tree as a set of computation sequences
define-sort CTree := (Set.Set CSequence)

# shortcut for variable names in different domains
```

```
define [cs rcs l rl] :=
       [?cs:CSequence ?rcs:CTree ?l:Label ?rl:(MSet.MSet Label)]

# +++ takes a Label l and a Computation Tree and prepends the label l
# to all the sequences in the set.
declare +++ : [Label CTree] -> CTree

assert* +++-def :=
[ (l +++ empty-set = (l :: nil) ++ empty-set)
  (l +++ (cs ++ rcs) = ((l :: cs) ++ (l +++ rcs)))
 ]

# computation tree as a set of computation sequences
declare ctree : [AConfig (List Ide)] -> CTree

# computation tree given a multiset of valid next labels
declare ctreel : [AConfig (MSet.MSet Label) (List Ide)] -> CTree

assert* ctree-def :=
[((ctree k fvars) = (ctreel k (redexes k fvars) fvars))]

assert* ctreel-def :=
[((ctreel k empty-mset fvars) = empty-set)
 ((ctreel k (l ++ rl) fvars) = ((l +++ (ctree (reduce k l) fvars))
                                 Set.\/ (ctreel k rl fvars)))
 ]
```

Lemma 2 (Anonymity). *If* `k' = (reduce k (bec a a'))` *and* `cs` *is any computation sequence in the computation tree* `(ctree k')`*, then* `cs` *contains no transitions with label of the form* `(rcv a' e)`*.*

```
define anonymity-lemma :=
  (forall k a a' cs fvars .
    ((cs in (ctree (reduce k (bec a a')) fvars)) ==>
        (forall e . (~ ((rcv a' e) in cs)))))
```

Definition 3 (Computation paths). *The sequences of a computation tree are partially ordered by the initial segment relation. A* computation path *from* `k` *is a maximal linearly ordered set of computation sequences in the computation tree* `(ctree k)`*. It can be* infinite *or* finite*.*

```
# +++p takes a Label l and a Computation Path Tree and prepends the
# label l to all the sequences in the set.
declare +++p : [Label CTree] -> CTree

assert* +++p-def :=
[ (l +++p empty-set = empty-set)
  (l +++p (cs ++ rcs) = ((l :: cs) ++ (l +++p rcs)))
 ]

# computation paths as a set of maximal computation sequences
declare cpaths : [AConfig (List Ide)] -> CTree

# computation paths given a multiset of valid next labels
declare cpathsl : [AConfig (MSet.MSet Label) (List Ide)] -> CTree

assert* cpaths-def :=
[((cpaths k fvars) = (cpathsl k (redexes k fvars) fvars))]

assert* cpathsl-def :=
[((cpathsl k empty-mset fvars) = nil ++ empty-set)
```

```
((cpathsl k (l ++ empty-mset) fvars) = (l +++p (cpaths (reduce k l) fvars)))
((cpathsl k (l ++ rl) fvars) = ((l +++p (cpaths (reduce k l) fvars))
                                  Set.\/ (cpathsl k rl fvars))
                                      <== (rl =/= empty-mset))]
```

The following example illustrates the difference between computation sequences and computation paths:

```
> (eval (ctree (kfg (['a (App I I)] ++ ['b (App I I)] ++ empty-map) empty-mset)
               (gen-fresh 'f 30)))

Term:
(Set.insert (:: (Actors.fun 'a)
                (:: (Actors.fun 'b)
                    nil:(List Actors.Label)))
   (Set.insert (:: (Actors.fun 'a)
                   nil:(List Actors.Label))
      (Set.insert (:: (Actors.fun 'b)
                      (:: (Actors.fun 'a)
                          nil:(List Actors.Label)))
         (Set.insert (:: (Actors.fun 'b)
                         nil:(List Actors.Label))
                     Set.null:(Set.Set (List Actors.Label))))))


> (eval (cpaths (kfg (['a (App I I)] ++ ['b (App I I)] ++ empty-map) empty-mset)
                (gen-fresh 'f 30)))

Term:
(Set.insert (:: (Actors.fun 'a)
                (:: (Actors.fun 'b)
                    nil:(List Actors.Label)))
   (Set.insert (:: (Actors.fun 'b)
                   (:: (Actors.fun 'a)
                       nil:(List Actors.Label)))
      Set.null:(Set.Set (List Actors.Label))))
```

where we notice that the computation tree includes non-maximal sequences, whereas the computation paths only include the two maximal sequences where both transitions in actors `a` and `b` have occurred, albeit in different orders.

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

1. $\mathit{Cfig}(k, L, 0) = k$
2. $\mathit{Cfig}(k, L * [l_i], i + 1) = k'$ *where* $K' = (\mathit{reduce}\ \mathit{Cfig}(k, L, i)\ l_i)$*. Thus, the path* π *determined by* `k,L` *is the sequence*

$$[\mathit{Cfig}(k, L, i) \mid i < \infty]$$

(cfig k L) computes the configuration resulting after executing a finite list of labeled transitions L starting from k:

```
# executes a finite computation sequence following list of labels
declare cfig : [AConfig (List Label)] -> AConfig

assert* cfig-def :=
  [((cfig k nil) = k)
   ((cfig k (l :: rest)) = (cfig (reduce k l) rest))]
```

Notice that the cfig function computes Cfig(k,L) = Cfig(k,L,Len(L)).

Fairness. Not all paths are 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. A path π in the computation tree (ctree k), is *fair*, written (fair π) if each enabled transition eventually happens or becomes permanently disabled.

Definition 5 ((fair π)).

$$(\mathit{fair}\,\pi) \iff (\forall i < \infty)(\exists l)(\mathit{enabled}\ k_i\ l) \implies (((\exists j \geq i)(\mathit{reduce}\ k_j\ l) = k_{j+1}) \vee (\exists j > i)(\forall j' > j)(\neg(\mathit{enabled}\ k_{j'}\ l)))$$

```
declare fair : [AConfig CSequence] -> Boolean

assert* fair-def :=
[((fair k cs) <==>
 (cs in (ctree k fvars)) &
  (forall i .
    (exists l . (enabled (reduceN k (take cs i)) fvars l))
    ==> l in (drop cs i) |
        (exists j .
          i < j &
           forall j' . j < j' ==>
                        (~ (enabled (reduceN k (take cs j')) fvars l)))))]
```

Definition 6 (Fair computation trees). *For a configuration* k*, we defined its fair computation tree* (ftree k) *to be the subset of its computation tree that contains fair paths.*

$$(\mathit{ftree}\ k) = \{\pi \in (\mathit{ctree}\ k) \mid (\mathit{fair}\,\pi)\}$$

```
declare ftree: [AConfig (List Ide)] -> CTree
assert* ftree-def :=
  [(cs in (ftree k fvars) <==> (cs in (ctree k fvars) & (fair k cs)))]
```

Lemma 3 (Fair finite paths). *A finite path is fair.*

Proof. If (Len π) $= M$, then by maximality we must have that $\forall l.\neg(\text{enabled Cfig}(k,\pi)\ l)$. Consequently the path is fair, since all enabled transitions have either occurred or become disabled.

4 Discussion

We have presented an executable operational semantics for the Agha, Mason, Smith, and Talcott (AMST) simple lambda-based actor language introduced in [2][5] The theory has been written in the Athena dual computation and deduction language [3] which enables us to not only evaluate expressions in the language but also formally prove properties about it.

The theory is presented as sets of equations that serve two purposes: first, each set of equations is syntactic sugar for a set of universally quantified first-order logic sentences, which are construed as axioms for the theory. We have illustrated the proof of the unique expression decomposition lemma (Lemma 1 in [2]) by structural induction. Second, the set of equations can be viewed as rewrite rules from left to right, and used to convert ground terms into normal forms by Athena's `eval` function. We use this capability to *execute* actor expressions as illustrated by several examples in the paper.

Future work includes fully encoding the actor theory in [2], including compositional aspects (*i.e.*, modeling receptionists and external actors for configurations), equivalence of expressions (may, must, and test equivalences, and the partial collapse to two equivalences under fairness), equivalence of configurations, laws of expression equivalence, and proof methods for verification of actor systems. Athena also can be used to generate code for the theory (using its `fun` form to introduce equation sets) which can be explored to generate executable actor code in different actor languages.

In Honor of Gul Agha

Gul has been a great influence in my academic and non-academic life. When reading his book on Actors [1], I was immediately captivated by the elegance, simplicity, and expressive power of this concurrent computation model. Agha, Mason, Smith, and Talcott presented a simple actor language, its operational semantics, and theory [2]. I have used that language, called here AMST, to introduce concurrency to thousands of students at RPI for over 20 years. In all this time, it would have been great to be able to *execute* programs written in this language. As an honor to Gul and his legacy on the concurrent programming and concurrency theory communities, I decided to make AMST an executable language while keeping the original spirit of developing a theory of actors with laws of expression equivalence. The Athena dual computation and deduction language [3] was a timely choice for developing the executable AMST theory.

Gul Agha has never stopped to be an *academic father* to me. I was extremely lucky to have him as my Ph.D. advisor at UIUC, and over the years, I have grown to increasingly appreciate his philosophy of life, his unwavering quest for understanding and new knowledge, his appreciation for all sentient life on Earth, and his keen sense of humor while diplomatically but firmly pointing out injustices and nonsensical politics at all levels of society. I am truly honored to be his academic descendant.

[5] The AMST theory is available open-source at https://wcl.cs.rpi.edu/amst/.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. **7**, 1–72 (1997)
3. Arkoudas, K., Musser, D.: Fundamental Proof Methods in Computer Science. MIT Press, Cambridge (2017)
4. Church, A.: The Calculi of Lambda Conversion. Princeton University Press, Princeton (1941)
5. Varela, C.A.: Programming Distributed Computing Systems: A Foundational Approach. MIT Press, Cambridge (2013). http://wcl.cs.rpi.edu/pdcs