A Tour of the Lambda Calculus

Adam Houser

Formal methods guest lecture

28 Nov 2016

λ-calculus, defined

- Model of computation (i.e., a formal system for defining computable functions) that is *Turing complete.*

- POWER: computation through simplest possible functions, with focus on transformation rules for those functions.

- Underpins *functional programming* languages (ex: Haskell, Coq, Elm).

- Note: no Newtonian calculus is involved.

Fundamentals

Names	letters	$a, b, c, .. z$
Functions	transformations	$\lambda x. x$ $f(x) = x$
Applications	expressions applied to functions	$(\lambda x. x) y$
Expressions	names, functions, or applications	a $\lambda x. x$ $(\lambda x. x) y$

Fundamentals, continued

- *Everything* in pure (untyped) λ-calculus is a function; only concern is computation via name substitution.
- All names are local to expressions.
- There are only a few different strategies used to evaluate functions, which we will cover in a bit.

Evaluation

- When λ-calculus functions are evaluated, they are *resolved.*
- Resolution goes as far as it can by using substitution.
- Completion occurs when there are no more functions to evaluate. This simplest form is called *normal form.*
- Next, we look at some resolution rules and strategies.

Evaluation rule: binding

Variables in λ-calculus can be either *bound* or *free*

- Bound = variable bound to λ and within the scope of the λ operator's expression
- Free = variable not bound to λ and beyond the scope of the λ operator's expression

$(\lambda x.\, xy)$ x is *bound*, y is *free*

$(\lambda x.\, x)(\lambda y.\, yx)$ left x is bound, right y is bound

- POP QUIZ: in the second expression above, is the x name consistent across both functions?

Evaluation strategy: β-reduction

Purpose: simplifying expressions down to normal form.

$$(\lambda x.\, \lambda z.\, x\ z)\, y$$

$$\left(\lambda x.\, \left(\lambda z.\, (x\ z)\right)\right) y$$

$$\left(\lambda y.\, \left(\lambda z.\, (y\ z)\right)\right) y$$

$$\left(\lambda z.\, (y\ z)\right)$$

Evaluation strategy: β-reduction, with practice

Simplify the following expression to normal form.

$$(\lambda y.x(yz))(ab)$$

$$\left(\lambda ab.x\left((ab)z\right)\right)(ab)$$

$$x((ab)z) \quad \text{or} \quad x(ab\ z)$$

Evaluation strategy: α-conversion

Purpose: reassigning names to prevent incorrect or “illegal” reductions.

$$(\lambda x.(\lambda y.xy))y \qquad (\lambda x.(\lambda y.xy))y \rightarrow_{\alpha} (\lambda x.(\lambda t.xt))y$$

$$(\lambda y.(\lambda y.yy))y \qquad (\lambda y.(\lambda t.yt))y$$

$$(\lambda y.yy) \qquad (\lambda t.yt)$$

Church numerals

- Numbers have to be encoded as functions, because untyped λ-calculus doesn't provide any vocabulary for talking about numbers.

- Critical to Church numerals: the *successor* function, which adds 1 to any existing function:

$0 = \lambda sz.\, z$

$1 = \lambda sz.\, s(z)$

$2 = \lambda sz.\, s\big(s(z)\big)$

$3 = \lambda sz.\, s\Big(s\big(s(z)\big)\Big)$

Additional capabilities

Arithmetic operations: 2 + 3

$$\left(\lambda sz.\, s\left(s(z)\right)\right)\left(\lambda wyx.\, y(wyx)\right)\left(\lambda tu.\, t\left(t(t(u)\right)\right)$$

Booleans

- $T \equiv \lambda xy.\, x$
- $F \equiv \lambda xy.\, y$

Logical operations and conditionals

- AND, OR, NOT.
- If > then > else.

Implementation

“A formal approach to modeling and analyzing human taskload in simulated air traffic scenarios.”

Background:

- Using simulation and formal methods synergistically to discover interesting, potentially dangerous human taskload conditions.
- This particular WMC scenario used one ATC, three aircraft agents in a simulated landing at an airport.
- NextGen Air Traffic Control system: shifting authority and autonomy allocation of current air navigation paradigm.
- As actions are executed by all agents during landing, can we use formal methods to find problematic corner cases in the simulation trace?

Representing action queues efficiently

Assigning actions to queues (explained in a bit) is computationally intensive for SAL:

- Agent modules must keep track of all actions in their Active and Inactive sets, with the goal of identifying violations (e.g., exceeding maxInactive)
- Action modules must determine (based on whether their status is *doing, assigning, or shuffling*) if their state is set to *active*, *interrupted*, or *delayed*.
- This must be computed for each simulation tick, maintained in memory, then manipulated and stored in memory during the following tick.

Solution: using λ-calculus sets with queues

Much more efficient way to handle complex operations.

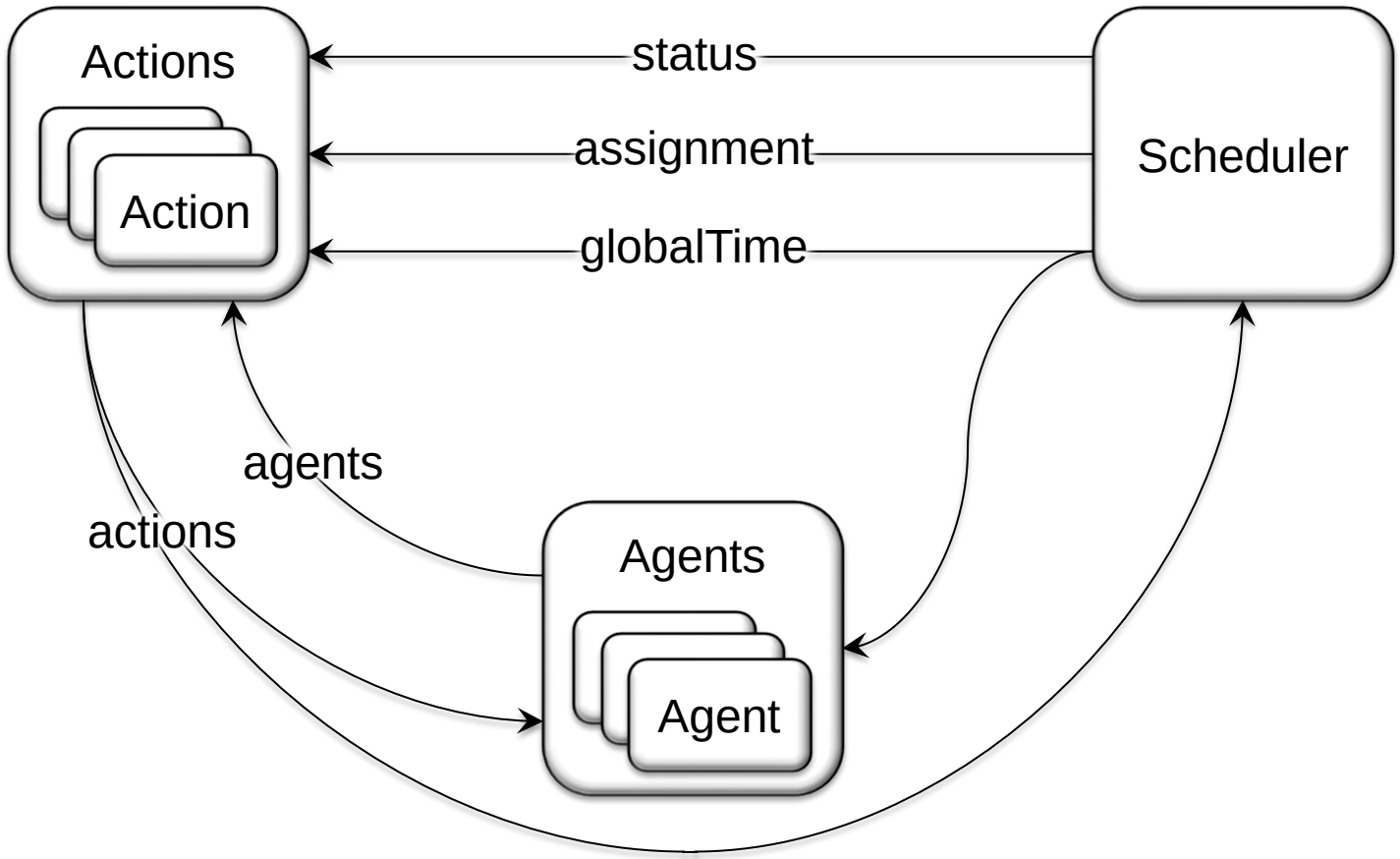
Maps elements to Boolean values:

- True = in the set
- False = not in the set

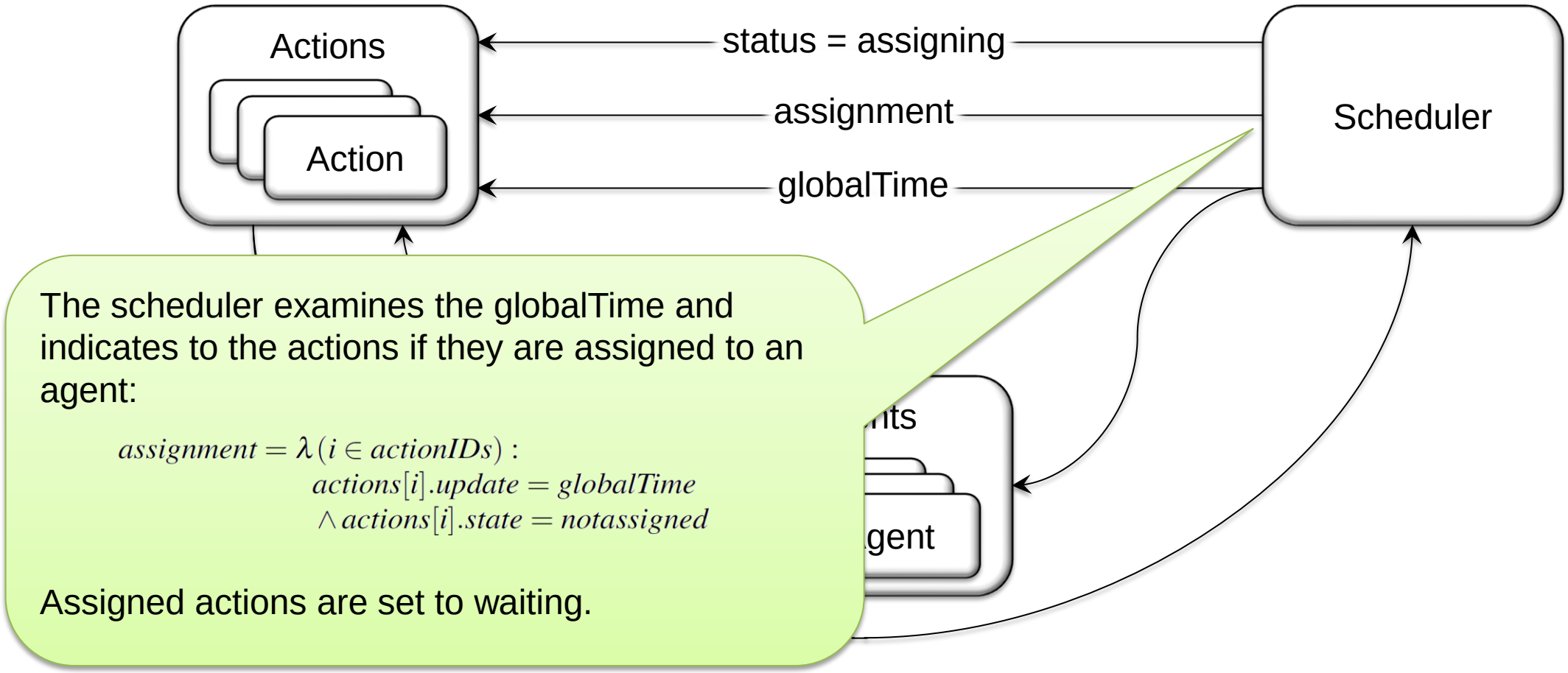
Efficiency improvement results from set mapping: concern is now about set membership, rather than computing arrays or manipulating array data.

Formal model

This is the formal model architecture used in our evaluation.



Formal Model

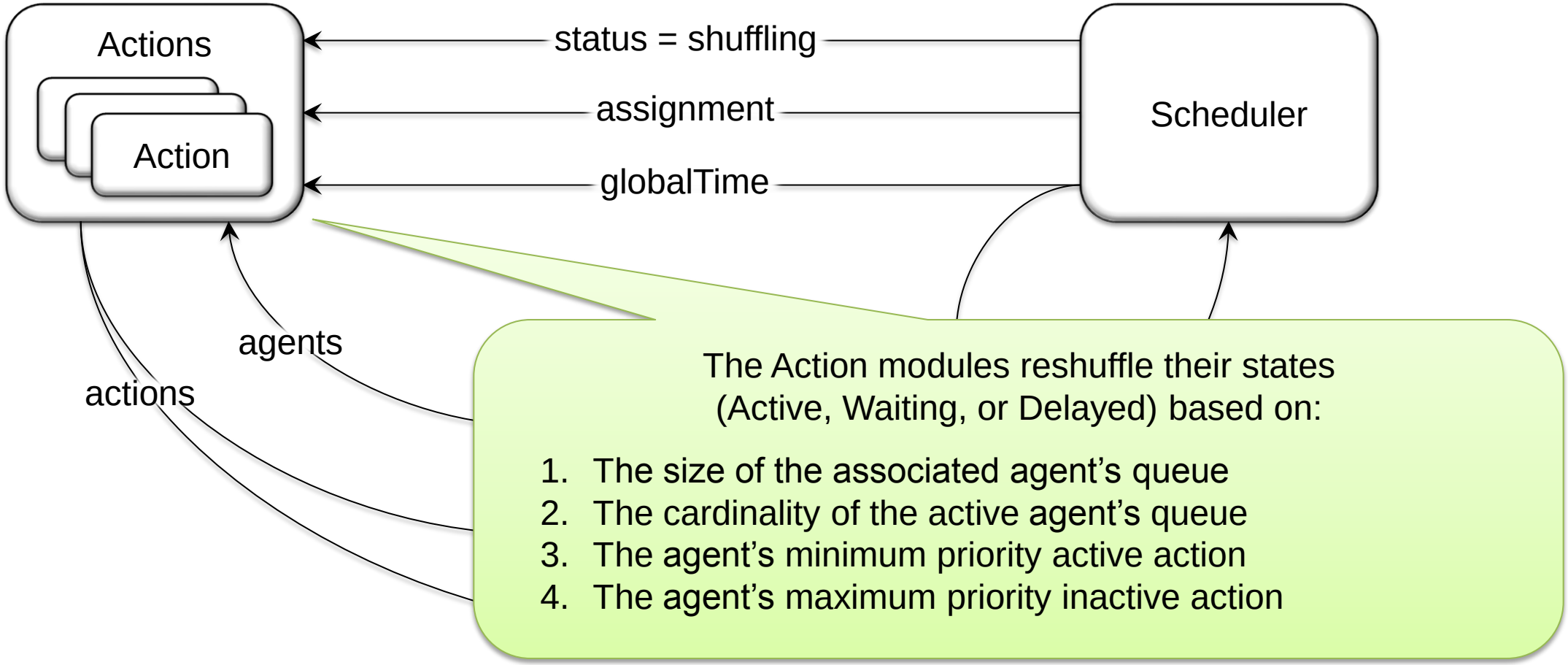


The scheduler examines the globalTime and indicates to the actions if they are assigned to an agent:

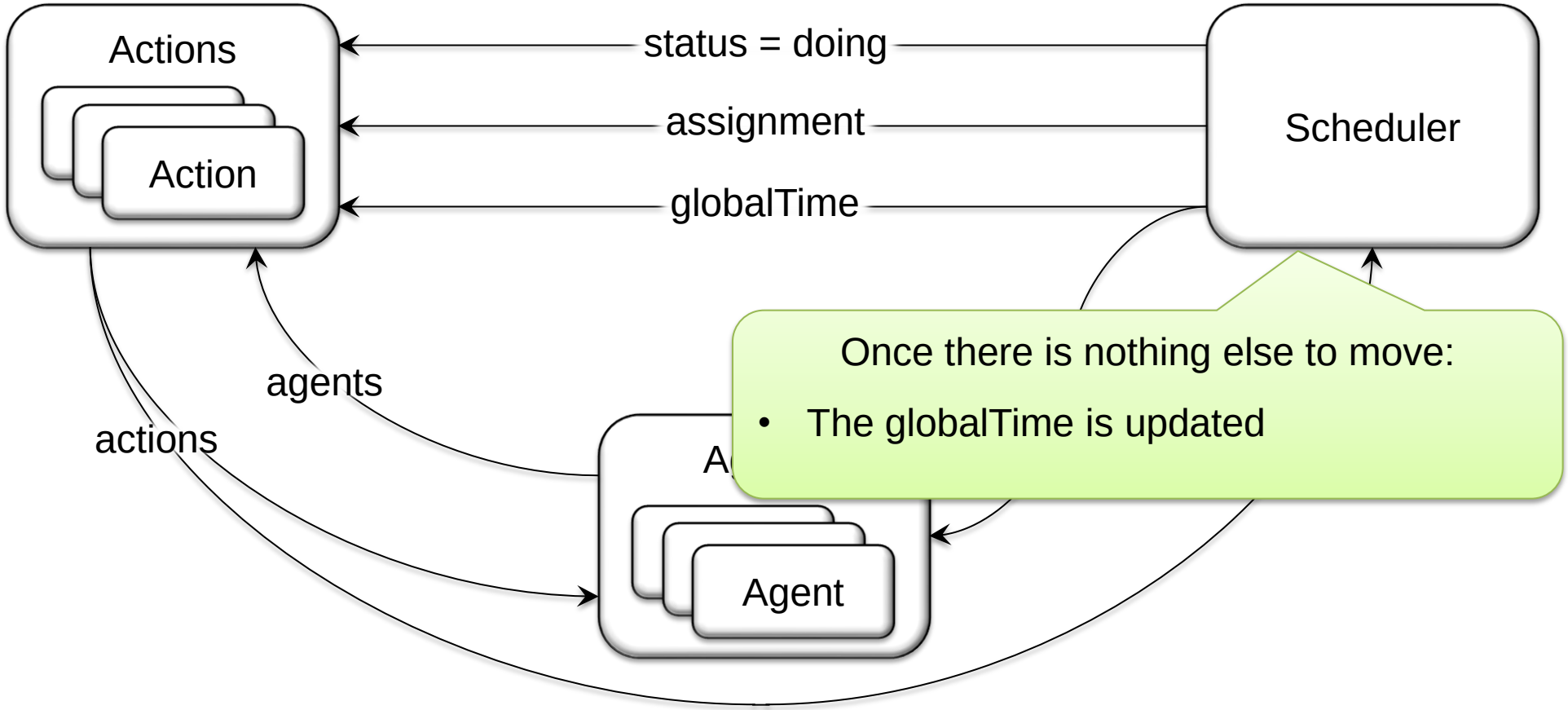
$$assignment = \lambda(i \in actionIDs) : \\ actions[i].update = globalTime \\ \wedge actions[i].state = notassigned$$

Assigned actions are set to waiting.

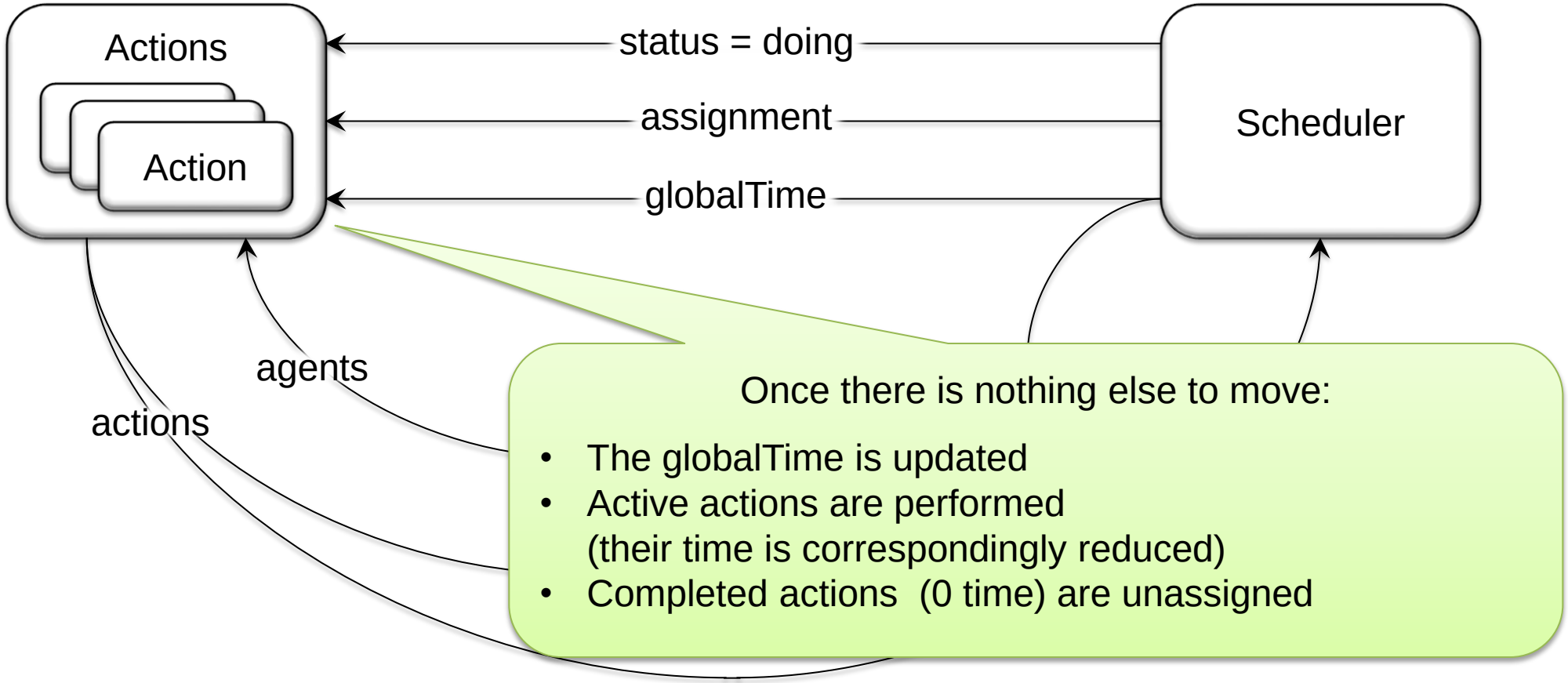
Formal Model



Formal Model



Formal Model

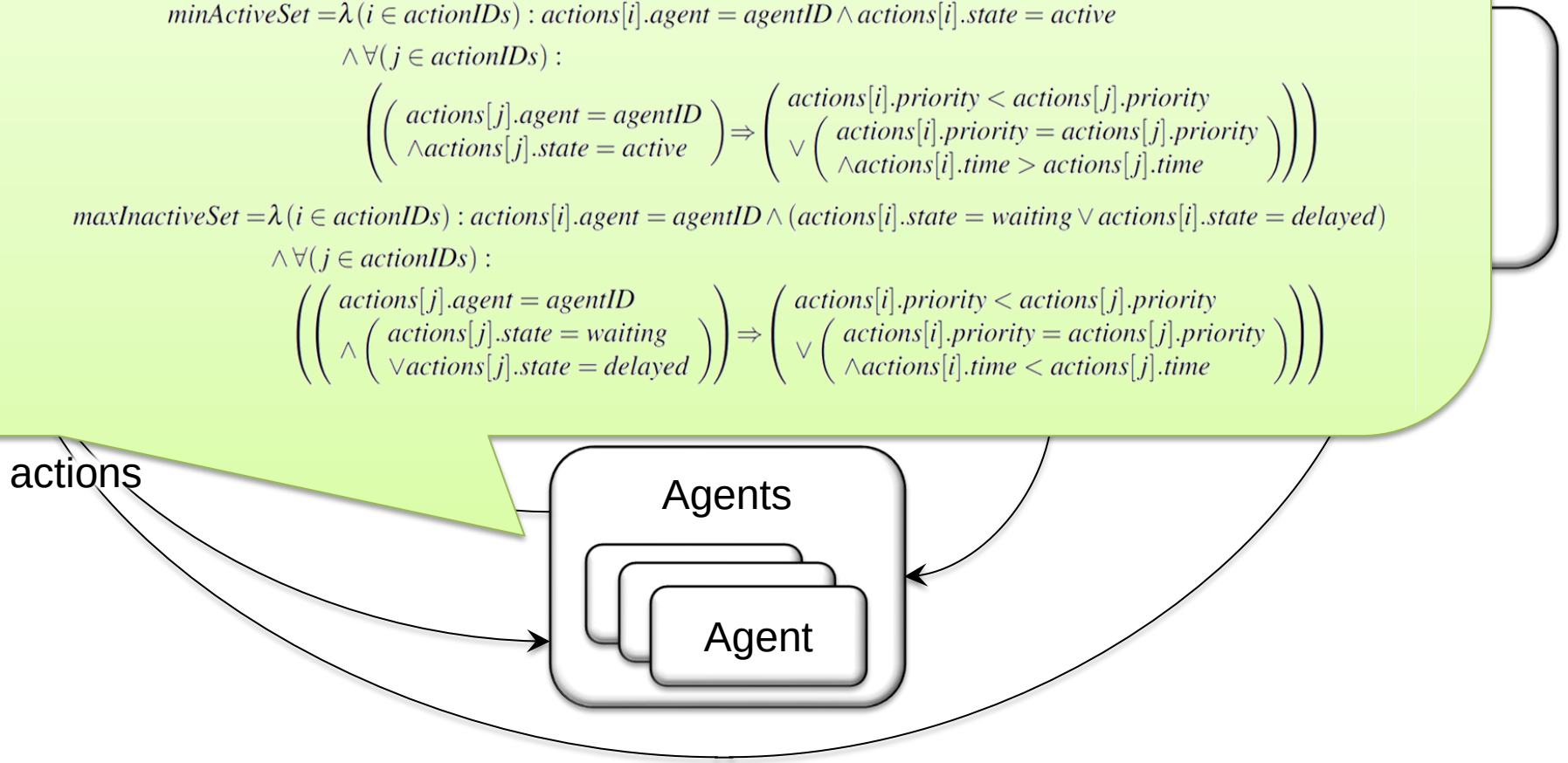


The agent dynamically updates queue metrics based on the shuffling, performance, and completion of actions:

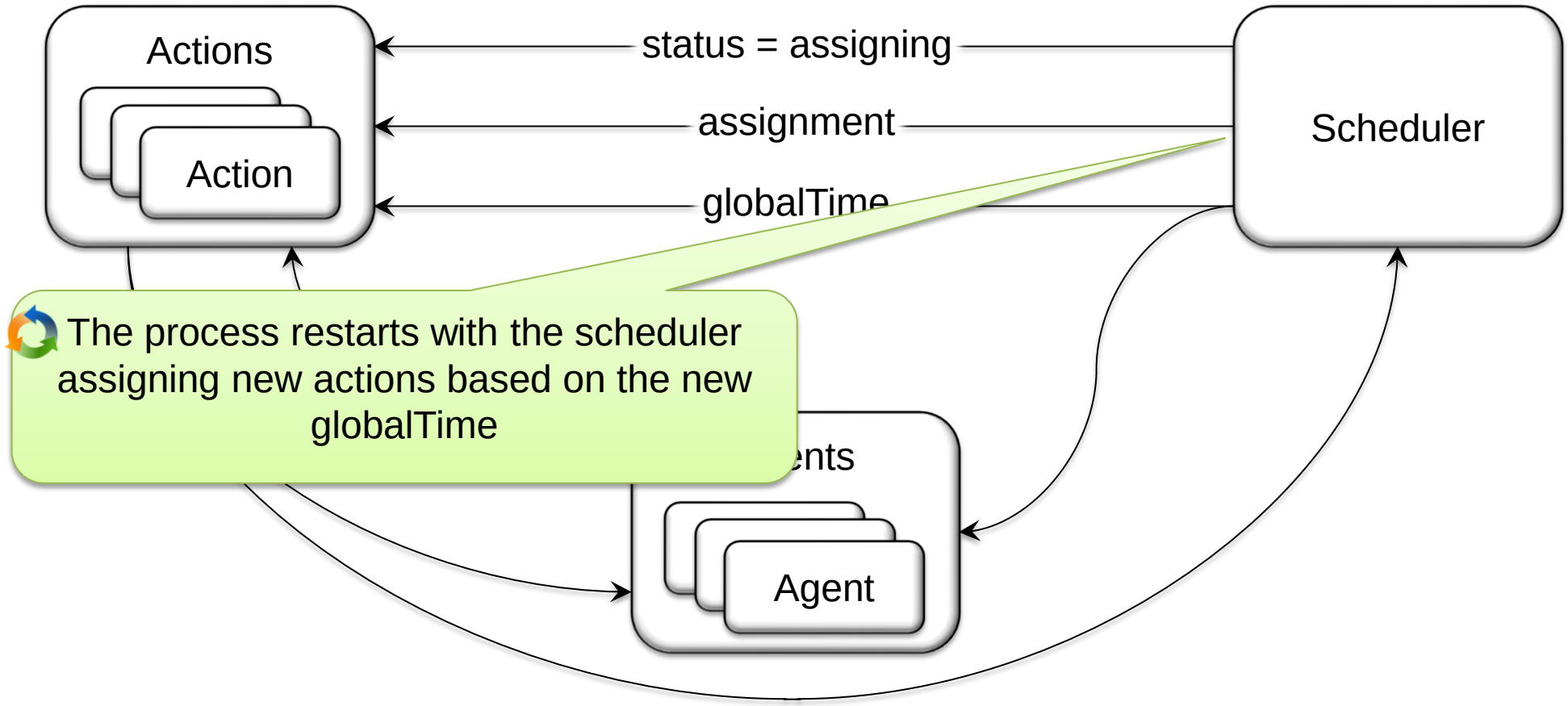
- Cardinality of active and inactive priority queues
- Minimum priority active actions
- Maximum priority inactive actions

$$minActiveSet = \lambda(i \in actionIDs) : actions[i].agent = agentID \wedge actions[i].state = active$$
$$\wedge \forall(j \in actionIDs) :$$
$$\left(\begin{pmatrix} actions[j].agent = agentID \\ \wedge actions[j].state = active \end{pmatrix} \Rightarrow \begin{pmatrix} actions[i].priority < actions[j].priority \\ \vee \begin{pmatrix} actions[i].priority = actions[j].priority \\ \wedge actions[i].time > actions[j].time \end{pmatrix} \end{pmatrix}\right)$$

$$maxInactiveSet = \lambda(i \in actionIDs) : actions[i].agent = agentID \wedge (actions[i].state = waiting \vee actions[i].state = delayed)$$
$$\wedge \forall(j \in actionIDs) :$$
$$\left(\begin{pmatrix} actions[j].agent = agentID \\ \wedge \begin{pmatrix} actions[j].state = waiting \\ \vee actions[j].state = delayed \end{pmatrix} \end{pmatrix} \Rightarrow \begin{pmatrix} actions[i].priority < actions[j].priority \\ \vee \begin{pmatrix} actions[i].priority = actions[j].priority \\ \wedge actions[i].time < actions[j].time \end{pmatrix} \end{pmatrix}\right)$$



Formal Model



Useful resources

http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf

http://worrydream.com/AlligatorEggs/

https://zeroturnaround.com/rebellabs/what-is-lambda-calculus-and-why-should-you-care/

http://people.eecs.berkeley.edu/~gongliang13/lambda/#firstPage

https://www.cs.umd.edu/class/spring2012/cmsc330/lectures/22-lambda.pdf

Bonus Slides

A successor function:

$$S \equiv (\lambda wyx.\, y(wyx))$$

Apply it to zero, with the hopes of getting 1:

$$S_0 \equiv (\lambda wyx.\, y(wyx))(\lambda sz.\, z)$$
$$\lambda yx.\, y((\lambda sz.\, z)yx)$$
$$\lambda yx.\, y((\lambda z.\, z)x)$$
$$(\lambda yx.\, y(x))$$

$$\lambda sz.\, s(z)$$
$$S_0 \equiv 1$$

Bonus Slides