Naming State Variables in IOA
Stephen J. Garland
December 15, 2000

In this note, we describe and discuss the notations IOA allows for state variables in automaton definitions, invariants, and simulation relations. To keep the discussion from getting overly abstract, we introduce several automata that model simple games to illustrate these notations.

Sample automata

We consider a family of games that involve two players, A and B, who take turns removing some number of stones—between one and a fixed limit—from a single pile. In the game LastWins, the player removing the last stone wins; in LastLoses, this player loses; in EvenWins, which starts with an odd number of stones in the pile, the player removing an even number wins.

The following automaton `Player` models the actions of a single player (A or B) in any of these games.[1] The automaton's parameters identity the player, the initial number of stones, and the limit on the number of stones that can be removed in a single turn.

```
automaton Player(IamA: Bool, sizeOfPile: Int, limit: Int)
  signature
    output move(const IamA, takes: Int) where 0 < takes /\ takes <= limit
    input move(const ~IamA, takes: Int) where 0 < takes /\ takes <= limit
  states
    left: Int := sizeOfPile,
    myTurn: Bool := IamA
  transitions
    output move(IamA, takes)
      pre myTurn; takes <= left
      eff left := left - takes;
          myTurn := false
    input move(~IamA, takes)
      eff left := left - takes;
          myTurn := true
```

A separate automaton, `Judge`, for LastWins and LastLoses observes the actions of the players and, at the end of the game, announces which has won.

[1] Language design questions: How can we extend the definition of the IOA language to treat `left` as a shared state variable? How would we indicate its initial value?

```
automaton Judge(sizeOfPile: Int, lastWins: Bool)
    signature
      input move(b: Bool, takes: Int)
      output Awins, Bwins
    states
      left: Int := sizeOfPile,
      AmovesNext: Bool := true
    transitions
      input move(b, takes)
        eff AmovesNext := ~AmovesNext;
            left := left - takes
      output Awins
        pre left = 0; AmovesNext = ~lastWins
        eff left := -1
      output Bwins
        pre left = 0; AmovesNext = lastWins
        eff left := -1
```

Finally, two composite automata model the games LastWins and LastLoses. For the sake of variety in discussing how state variables are named, we define these two automata in slightly different fashions, one with parameterized components and one without.

```
automaton LastLoses
  components
    A: Player(true, 10, 3); B: Player(false, 10, 3);
    Ref: Judge(10, false)

automaton LastWins
  components
    Who[p: Bool]: Player(p, 10, 3);
    Ref: Judge(10, true)
```

Exercise: Define an automaton to serve as a judge for EvenWins.

Sample invariants

IOA provides several notations for state variables that can be used to express invariants. For example, the notations for state variables that appear in the definition of an automaton can also appear in the statement of an invariant:

```
invariant of Player:
  sizeOfPile > limit => left >= 0
```

The definition of a composite automaton does not contain explicit references to its state variables, which it inherits from its components. IOA treats the states of such an automaton with unparameterized components as records containing the states of these components. Hence it provides a record-like notation for state variables inherited from component automata.

```
invariants of LastLoses:
  Ref.AmovesNext = A.myTurn;
  A.myTurn = ~B.myTurn;
  Ref.left = A.left;
  A.left = B.left
```

The situation for an automaton with a parameterized component is slightly more complicated, because there may not be fixed names for all possible values of the component's parameters. Hence IOA treats the states of such an automaton as records that contain the states of unparameterized components, together with arrays of states for parameterized components:

```
invariants of LastWins:
  Who[Ref.AmovesNext].myTurn;
  \A p: Bool (Ref.left = Who[p].left)
```

Abbreviations are allowed when no ambiguity can arise. For example, `AmovesNext` is the name of a state variable in a single component of `Judge`. Hence invariants involving `Ref.AmovesNext` need not mention the name `Ref` of that component:

```
invariant of LastLoses: AmovesNext = A.myTurn
```

```
invariant of LastWins: Who[AmovesNext].myTurn
```

Sample simulation relation

Record-like notations can be used to express simulation relations as well as invariants. Consider, for example, the following alternative description of the automaton `Player`:

```
automaton ModifiedPlayer(IamA: Bool, sizeOfPile: Int, limit: Int)
  signature
    output move(const IamA, takes: Int) where 0 < takes /\ takes <= limit
    input move(const ~IamA, takes: Int) where 0 < takes /\ takes <= limit
    internal update
  states
    removed: Int := 0,
    myTurn: Bool := IamA
 transitions
    output move(IamA, takes)
      pre myTurn; removed + takes <= sizeOfPile
      eff removed := removed + takes;
          myTurn := false
    input move(~IamA, takes)
      eff removed := removed + takes;
          myTurn := true
```

There is an obvious simulation relation between this automaton and `Player` itself, which can be expressed as follows.

```
forward simulation from Player to ModifiedPlayer:
  Player.left + ModifiedPlayer.removed = sizeOfPile;
  Player.myTurn = ModifiedPlayer.myTurn
```

Futhermore, since `left` is a state variable of `Player`, but not `ModifiedPlayer`, and `removed` is a state variable of `ModifiedPlayer`, but not `Player`, this description of the simulation relation can be abbreviated unambiguously as

```
forward simulation from Player to ModifiedPlayer:
  left + removed = sizeOfPile;
  Player.myTurn = ModifiedPlayer.myTurn
```

The references to the state variable `myTurn` in the two automata cannot be abbreviated, however, without introducing an ambiguity.

States of basic automata

For defining a basic automaton and for expressing its invariants, IOA provides a constant for each of the automaton's state variables. Thus, it provides two constants for each of the basic automata we have considered:

```
Player               Judge                   ModifiedPlayer
------               -----                   --------------
left:   -> Int       left:       -> Int      removed: -> Int
myTurn: -> Bool      AmovesNext: -> Int      myTurn:  -> Bool
```

In contexts that involve more than a single automaton, these constants may be ambiguous. To provide satisfactory notations in all contexts, IOA treats the states of a basic automaton as tuples of values for its state variables:

```
type States[Player]         = tuple of left: Int, myTurn: Bool
type States[ModifiedPlayer] = tuple of removed: Int, myTurn: Bool
type States[Judge]          = tuple of left: Int, AmovesNext: Bool
```

In accordance with this treatment, IOA provides selection operators (in all contexts) for extracting the values of state variables from states, e.g.,

```
__.left:       States[Player]         -> Int
__.myTurn:     States[Player]         -> Bool

__.left:       States[Judge]          -> Int
__.AmovesNext: States[Judge]          -> Bool

__.removed:    States[ModifiedPlayer] -> Int
__.myTurn:     States[ModifiedPlayer] -> Bool
```

In contexts that involve a single instance of an automaton (i.e., in its definition, invariants, and simulation relations), IOA provides a constant denoting an arbitrary state of that automaton, e.g.,

```
Player:         -> States[Player]
Judge:          -> States[Judge]
ModifiedPlayer: -> States[ModifiedPlayer]
```

Together, such constants and selection operators provide notions for expressing simulation relations:

```
forward simulation from Player to ModifiedPlayer:
  Player.left + ModifiedPlayer.removed = sizeOfPile;
  Player.myTurn = ModifiedPlayer.myTurn
```

Furthermore, IOA enables us to abbreviate the first expression in this simulation relation unambiguously as

```
  left + removed = sizeOfPile
```

by providing constants

```
left:    -> Int
removed: -> Int
```

(which are unambiguous in all contexts that involve a single instance of each of the automata `Player` and `ModifiedPlayer`). IOA does not introduce an overloaded constant `myTurn:->Bool` in addition to `left` and `removed`, because this constant would be ambiguous and would not enable us to express the simulation relation as

```
Player.myTurn = ModifiedPlayer.myTurn
```

any more succinctly.

States of composite automata (with no parameterized components)

IOA treats the states of a composite automaton that has no parameterized components (e.g., `LastLoses)` as tuples of states of its component automata.

```
type States[LastLoses] = tuple of A, B: States[Player], Ref: States[Judge]
```

Again in accordance with this treatment, IOA provides selection operators for extracting the values of state variables from states, e.g.,

```
__.A:   States[LastLoses] -> States[Player]
__.B:   States[LastLoses] -> States[Player]
__.Ref: States[LastLoses] -> States[Judge]
```

and (in contexts involving a single instance of `LastLoses`) a constant

```
LastLoses: -> States[LastLoses]
```

denoting an arbitrary state of that automaton. These notations, together with the selection operators introduced for the state variables of the `Player` and `Judge` automata, enable us to express the invariants of `LastLoses` in an unambiguous, although somewhat long-winded fashion:

```
invariants of LastLoses:
  LastLoses.Ref.AmovesNext = LastLoses.A.myTurn;
  LastLoses.A.myTurn = ~LastLoses.B.myTurn;
  LastLoses.Ref.left = LastLoses.A.left;
  LastLoses.A.left = LastLoses.B.left
```

The less long-winded statement of these invariants shown earlier (page 2) is possible because `A`, `B`, and `Ref` name unambiguous components of `LastLoses`. Hence, IOA can provide constants

```
A:   -> States[Player]
B:   -> States[Player]
Ref: -> States[Judge]
```

thereby enabling abbreviated notations for the state variables of `LastLoses`:

```
LastLoses.A.left             A.left
LastLoses.A.myTurn           A.myTurn
LastLoses.B.left             B.left
LastLoses.B.myTurnB          B.myTurn
LastLoses.Ref.left           Ref.left
LastLoses.Ref.AmovesNext     Ref.AmovesNext
```

Furthermore, since `LastLoses` contains a single instance of the `Judge` automaton, IOA can provide notations

```
__.AmovesNext: States[LastLoses] -> Bool
AmovesNext:                      -> Bool
```

which enable two additional abbreviations, `LastLoses.AmovesNext` and `AmovesNext`, for the last state variable of `LastLoses`. Similar abbreviations are not available for `left` and `myTurn` in the invariant of `LastLoses`, because state variables with these names occur in more than one component of `LastLoses`.

States of composite automata (with parameterized components)

IOA treats the states of a composite automaton that has parameterized components (e.g., `LastWins)` as records containing both the states of its unparameterized components and, for each of its parameterized components, a mapping from the values of its parameters to the states of that components.

```
type States[LastWins] =
     tuple of Who: Mapping[Bool, States[Player]], Ref: States[Judge]
```

As before, IOA provides selection operators for extracting elements from tuples

```
__.Who: States[LastWins] -> Mapping[Bool, States[Player]]
__.Ref: States[LastWins] -> States[Judge]
```

thereby enabling us to express the invariants of LastWins again somewhat long-windedly:

```
invariants of LastWins:
  LastWins.Who[LastWins.Ref.AmovesNext].myTurn;
  \A p: Bool (LastWins.Ref.left = LastWins.Who[p].left)
```

The less long-winded statement of these invariants shown above (page 3) is possible because `Who[p]` and `Ref` name unambiguous components of `LastWins`. Hence, IOA can provide constants

```
Who: -> Mapping[Bool, States[Player]]
Ref: -> States[Judge]
```

thereby enabling abbreviated notations for the state variables of `LastWins`:

```
LastWins.Who[true].left             Who[true].left
LastWins.Who[true].myTurn           Who[true].myTurn
LastWins.Who[false].left            Who[false].left
LastWins.Who[false].myTurn          Who[false].myTurn
LastWins.Ref.left                   Ref.left
LastWins.Ref.AmovesNext             Ref.AmovesNext
```

As before, since `LastWins` contains a single instance of the `Judge` automaton, IOA can provide notations

```
__.AmovesNext: States[LastWins] -> Bool
AmovesNext:                     -> Bool
```

which enable two additional abbreviations, `LastWins.AmovesNext` and `AmovesNext`, for the last state variable of `LastWins`. Similar abbreviations are not available for `left` and `myTurn` in the invariant of `LastWins`, because state variables with these names occur in more than one component of `LastWins`.

Additional examples

Editorial note: Words need to be written about the general rules for generating names for the state variables of composite automata. The following two examples illustrate how to handle some of the finer points.

Example 1: How to handle `where` clauses

```
automaton A(n: Int)
  signature ...
  states s: Int

automaton B
  components C(i: Int): A(i) where p(i);
             D(i: Int): A(i) where q(i);
```

Proof obligation: `~(p(i) /\ q(i))`

Composition:

```
type States[B] = tuple of C: Mapping[Int, States[A]],
                          D: Mapping[ Int, States[B]]
```

where

```
Mapping(D, R): trait
  introduces __[__}:  Mapping[D, R], D    -> R
             update:  Mapping[D, R], D, R -> R
             defined: Mapping[D, R], D    ->  Bool
```

and

```
defined(B.C, i) <=> p(i)
defined(B.D, i) <=> q(i)
```

Example 2: How to handle type parameters

```
automaton A(T1, ..., Tn: type, c1:S1, ..., ck:Sk)
```

The notation for the states of `A` must allow for instantiating formal types `T1, ..., Tn` with actual types.

Proposal:

```
automaton A(T: type, c: T)
automaton B
  components C(n: Int): A(Int, n);
             D(c: Char): A(Char, '0')

type States[B] = tuple of C: Mapping[Int -> States[A, Int]],
                          D: Mapping[ Int -> States[A, Char]]
```