Praxis

A logic-based DSL for modeling social practices

Demo

- Show Marriage Proposal from two angles
- Show Dinner Party, choosing two different characters

Versu is…

- Real-time
- Multiplayer
- Text-based
- Simulation
- Set in Jane Austen's Regency England

The Simulator

- The world is a set of facts
- The dynamic elements are social practices and agents

The Simulator

- The world is a set of facts
- The dynamic elements are social practices and agents

Social Practices

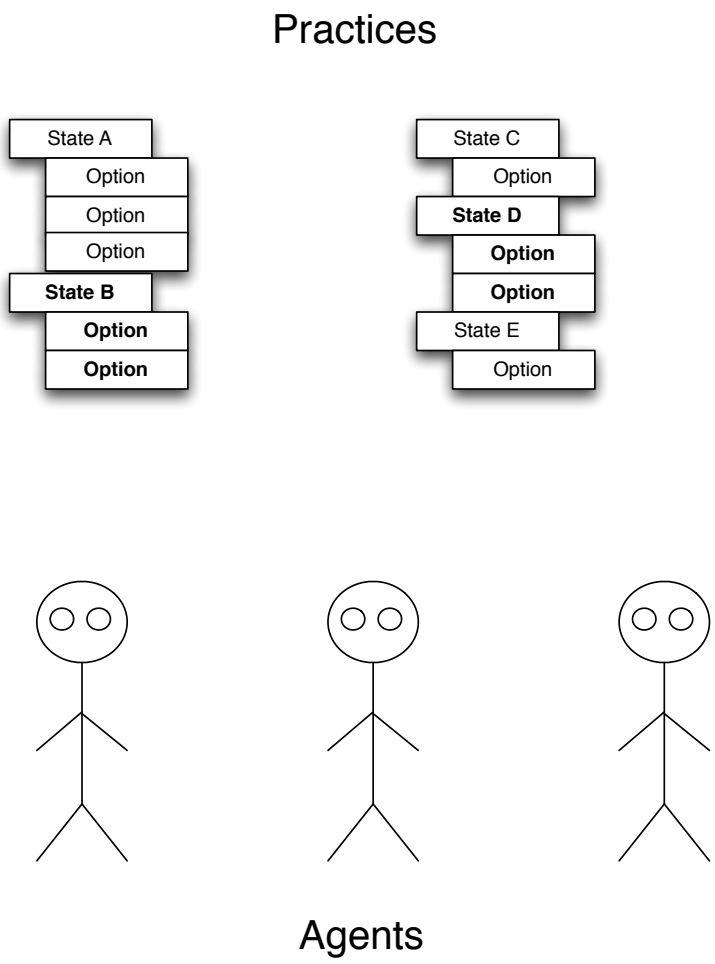
The Need for Social Practices

- The Sims 1
- My Sim invited his boss over for dinner.
- When he arrived, my Sim let him in - *but then he went to have a bath!*
- He didn't understand that certain things were expected of him as a host.

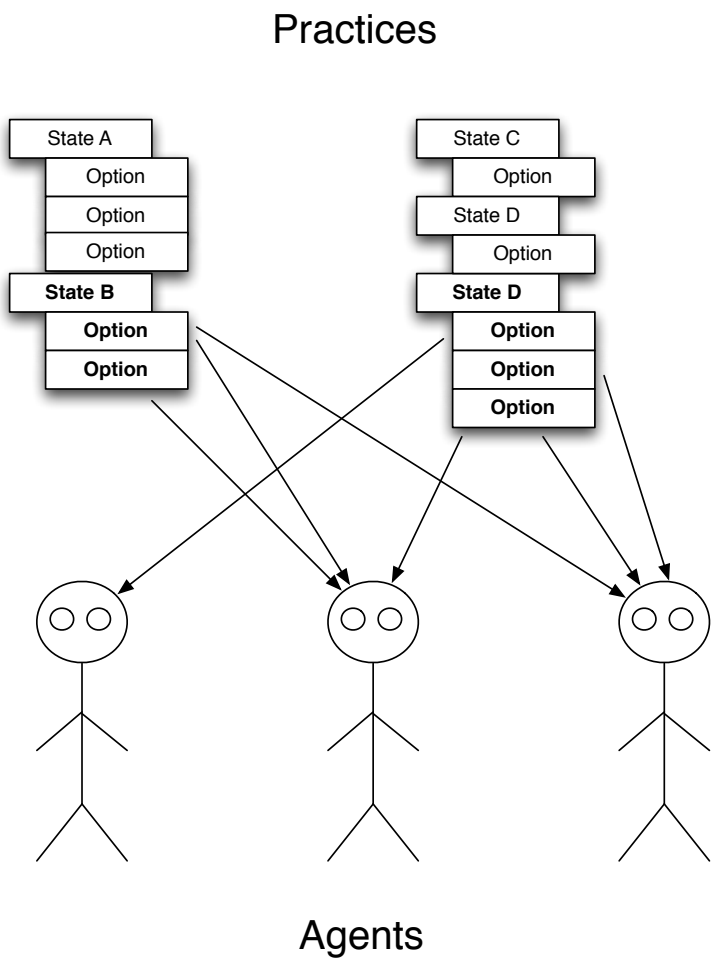
What is a Social Practice?

- It describes what agents *can* do in a social situation
- It also says what agents *should* do

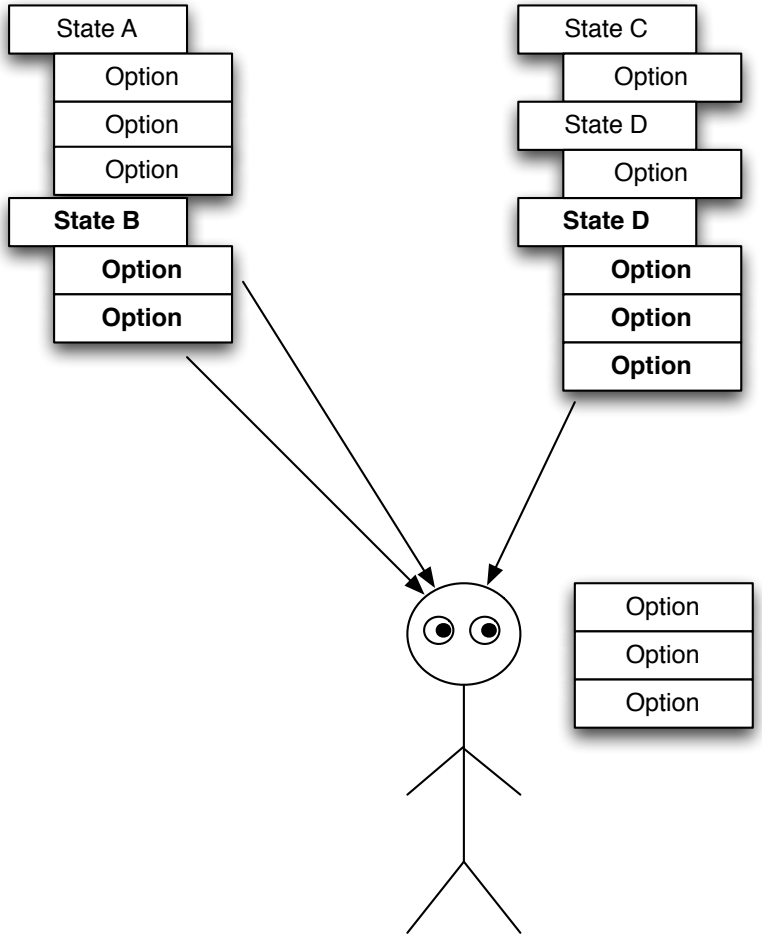
Social Practices



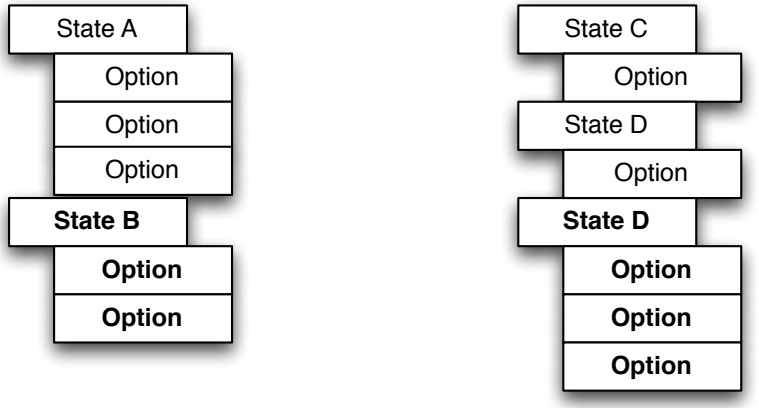
Social Practices

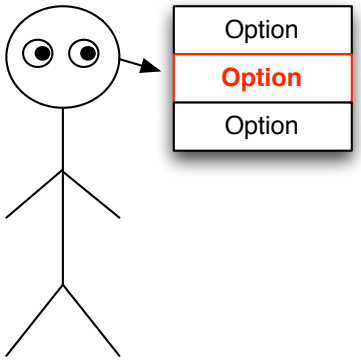


Social Practices

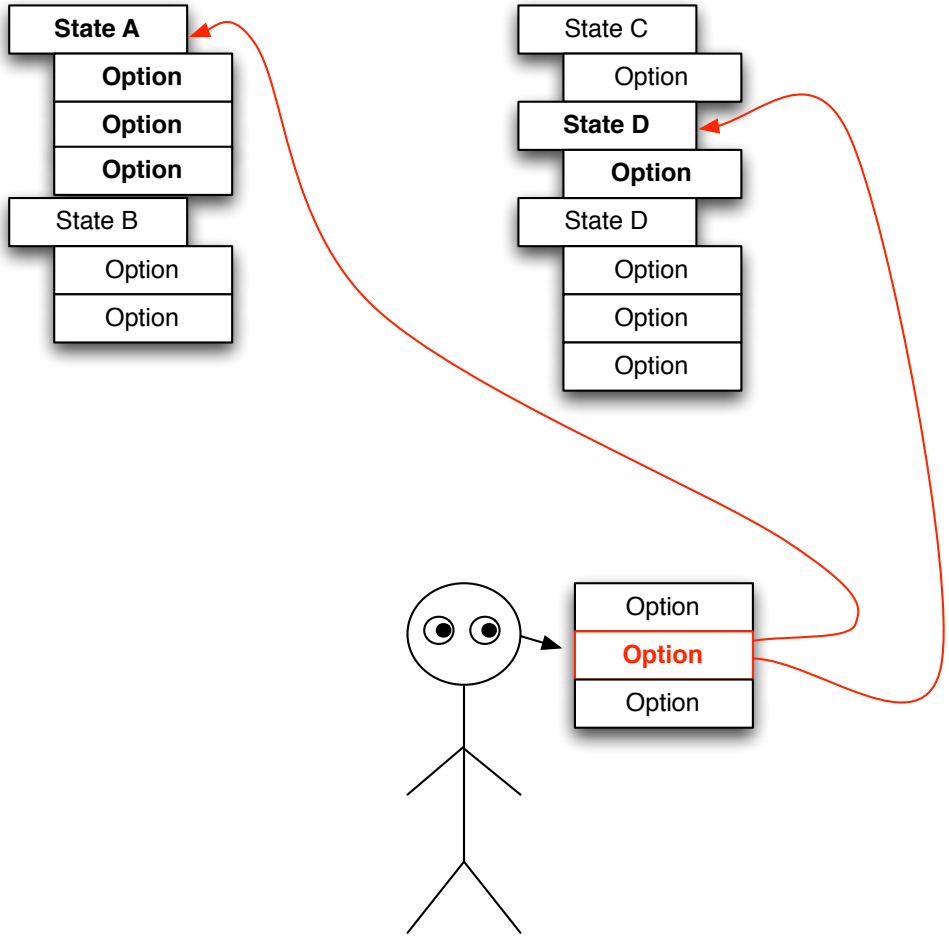


Social Practices





Social Practices



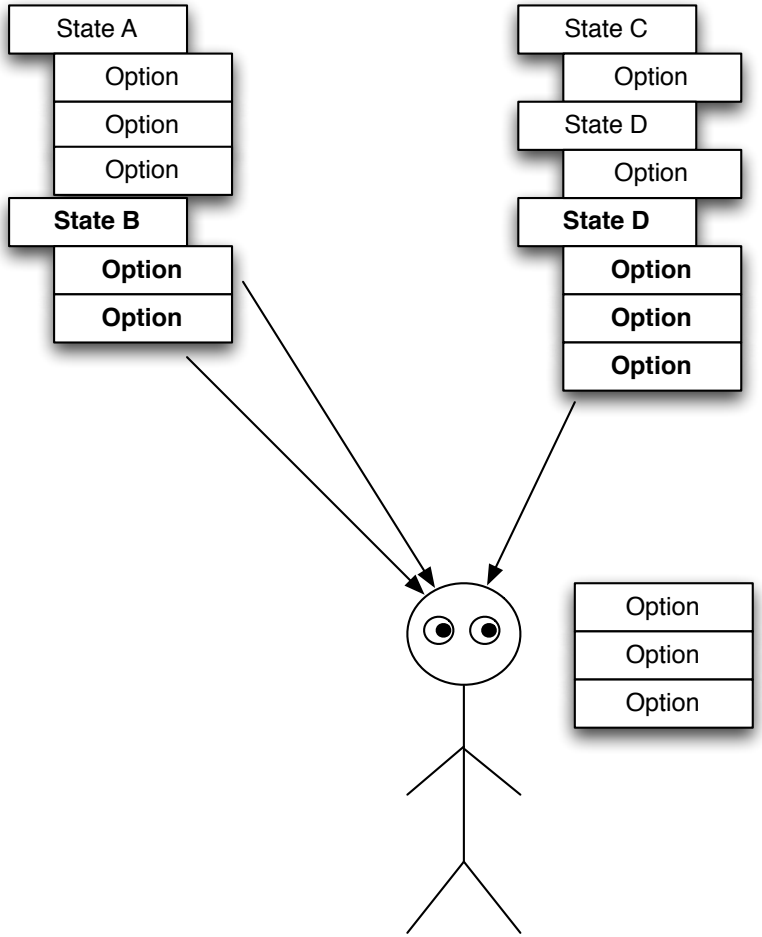
What is a Social Practice?

- It issues different requests in different circumstances
- It issues different requests to different people
- It notices when requests are satisfied or confounded

Demo

- Show an example of norm-violation.

Multiple Concurrent Practices



300+ Social Practices in Versu

- Dinner Party
- Conversation
- Debate
- Games
- Death

Demo

- Evaluate process.X in dinner party and whist game
- Show sub-tree of process.whist

Implementation

- A Social Practice is a set of sentences in Exclusion Logic

The Simulator

- The world is a set of facts
- The dynamic elements are social practices and agents

Agents

Implementation

- An agent is just a set of sentences in Exclusion Logic
  - Beliefs
  - Desires
  - Personality quirks
  - Backstory

Demo

- Show sub-terms of brown in the Dinner Party

Agents

- An agent has a set of wants
- He uses utility-based decision-making

Demo

- Show sub-terms of brown.wants in the Dinner Party
- Show the actions Brown is considering, sorted by score

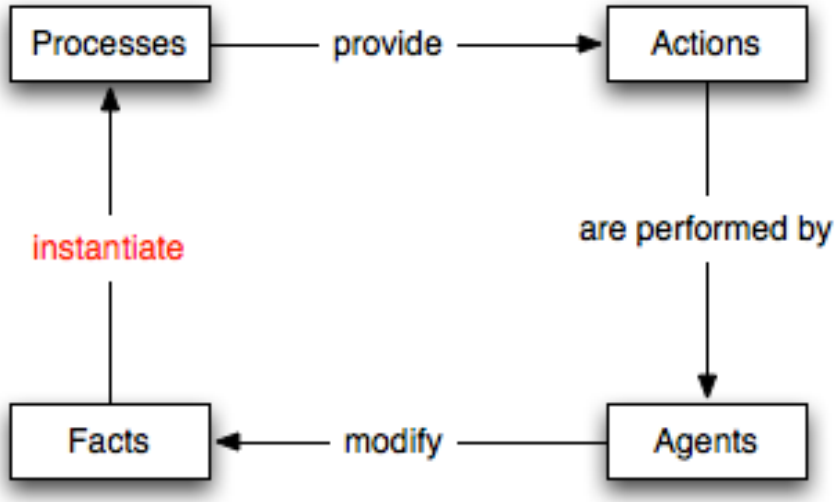
The Simulator

- The world is a set of facts
- The dynamic elements are social practices and agents
- The dynamic elements supervene on the facts

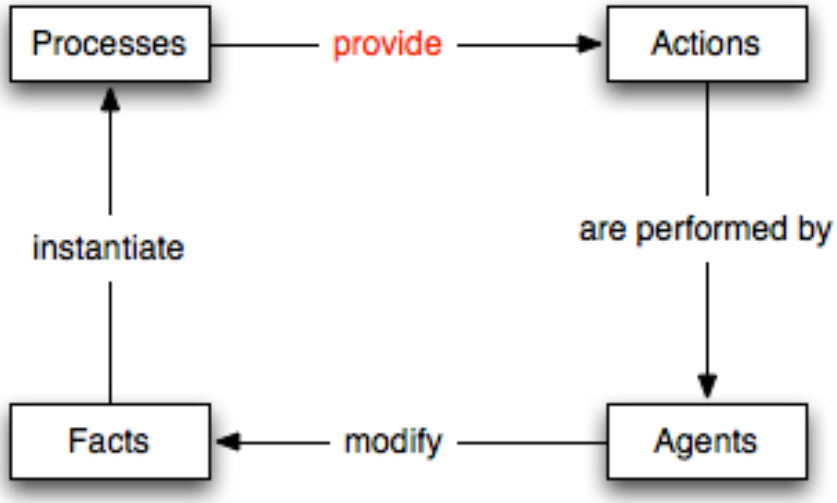
The Simulator

- The world is a set of facts
- The dynamic elements are social practices and agents
- The dynamic elements supervene on the facts

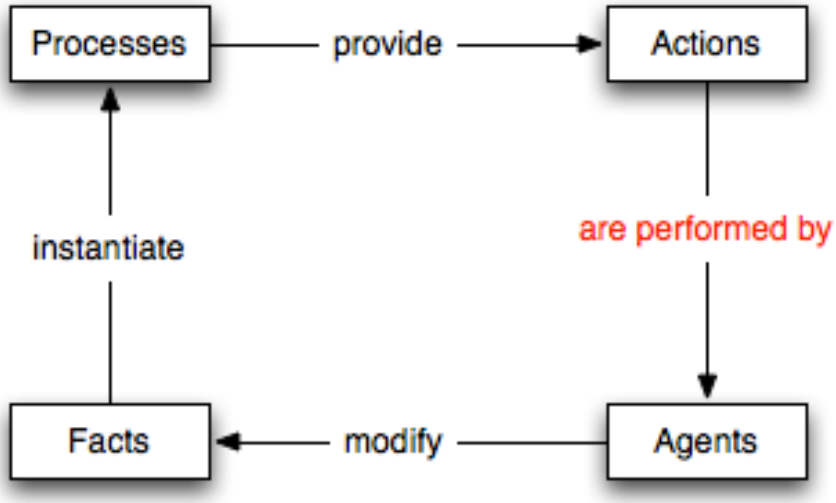
Facts Instantiate Processes



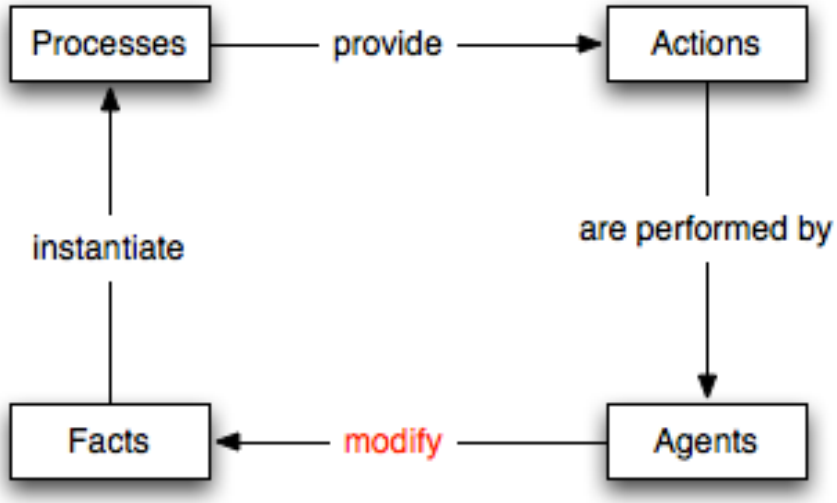
Processes Provide Actions



Agents Perform Actions



Performance Modifies Facts



Exclusion Logic

Praxis is based on a new modal logic called Exclusion Logic

Elementary Propositions

- Jack fell
- Jack likes Jill

Propositional Logic

$$\text{“}Jack\ \ likes\ \ Jill'' \longrightarrow p$$

- We cannot infer “Jack likes someone”

Predicate Logic

$$\text{“}Jack\ \ likes\ \ Jill'' \longrightarrow Likes(Jack, Jill)$$

$$Likes(Jack, Jill) \vdash (\exists x) Likes(Jack, x)$$

- In Predicate Logic, there are no logical relations between elementary propositions

$$Likes(Jack, Jill) \vdash (\exists x) Likes(Jack, x)$$

- In Predicate Logic, there are no logical relations between elementary propositions

$$Likes(Jack, Jill) \vdash (\exists x) Likes(Jack, x)$$

Logical Relations between Elementary Propositions

- “Jack is male” is incompatible with “Jack is female”
- “Jack walks quickly” entails “Jack walks”

Exclusion Logic

- A logic which supports logical relations between elementary propositions

Wittgenstein

- “There are rules for the truth functions which also deal with the *elementary* part of the proposition”

Elementary Propositions

Propositional Logic	An indivisible atomic sentence
Predicate Logic	Supports inferential relations with *compound* sentences
Exclusion Logic	Supports inferential relations with other *elementary* propositions

Exclusion Logic

$$E ::= S \mid S.E \mid S!E$$

$$C ::= E \mid \neg C \mid C \wedge C$$

$$E ::= S \mid S.E \mid S!E$$

- The "." and "!" operators are used to build up trees of information
- *S.E* means that *E* is *one* of the ways in which *S* is true
- *S!E* means that *E* is the *only way* in which *S* is true

$$E ::= S \mid S.E \mid S!E$$

Jack.Fell	One of the properties of Jack is that he fell
Jack.Likes.Jill	One of the people Jack likes is Jill
Jack.Gender!Male	The (unique!) gender of Jack is male

$$E ::= S \mid S.E \mid S!E$$

- Jack.Likes.Jill
- Jack.Likes.Josie

$$E ::= S \mid S.E \mid S!E$$

- Jack.Gender!Male
- Jack.Gender!Female

Inference Rules

$$X.Y \vdash X$$

$$X!Y \vdash X$$

$$X!Y \wedge X!Z \vdash P$$

Inference Rules

$$X.Y \vdash X$$

$$X!Y \vdash X$$

$$X!Y \wedge X!Z \vdash P$$

$$X.Y \nvdash Y$$

$$X!Y \nvdash Y$$

$$X.Y \wedge X.Z \nvdash P$$

Logical Relations between Elementary Propositions

- “Jack is male” is incompatible with “Jack is female”
- “Jack walks quickly” entails “Jack walks”

Logical Relations between Elementary Propositions

- "Jack is male" is incompatible with "Jack is female"

$$Jack.Gender!Male \vdash \neg Jack.Gender!Female$$

Logical Relations between Elementary Propositions

- "Jack walks quickly" entails "Jack walks"

$$Jack.Walks.Quickly \vdash Jack.Walks$$

Representing Incompatible Predicates in Predicate Logic

$$Gender(Jack) = Male$$

- Requires identity predicate and axiom schema

$$(\forall x, y) \;\; x = y \wedge F(x) \rightarrow F(y)$$

Representing Incompatible Predicates in Predicate Logic

- Brachman and Levesque:

$$(\forall x) Man(x) \rightarrow \neg Woman(x)$$

Representing Incompatible Predicates in Predicate Logic

- Brachman and Levesque:

$$(\forall x) Man(x) \rightarrow \neg Woman(x)$$

$$(\forall x) SupportsArsenal(x) \rightarrow \neg SupportsBarnsley(x) \wedge \neg SupportsFulham(x) \wedge \neg SupportsGrimsby(x) \wedge ...$$

Adverbial Inferences in Predicate Logic

- Davidson analysed "I flew my spaceship to the Morning Star" as:

$$(\exists x) Flew(I, MySpaceship, x)$$
$$\wedge To(x, TheMorningStar)$$

- "I flew my spaceship to the Morning Star" entails "I flew my spaceship"

Adverbial Inferences in Predicate Logic

- "Jack walks"

$$(\exists x) Walks(Jack, x)$$

Predicate Logic vs Exclusion Logic

- Predicate Logic can handle these inferences
- But it can only do so be reinterpreting the sentences as compound
- It uses more complex machinery to get the same results that Exclusion Logic gets *directly*

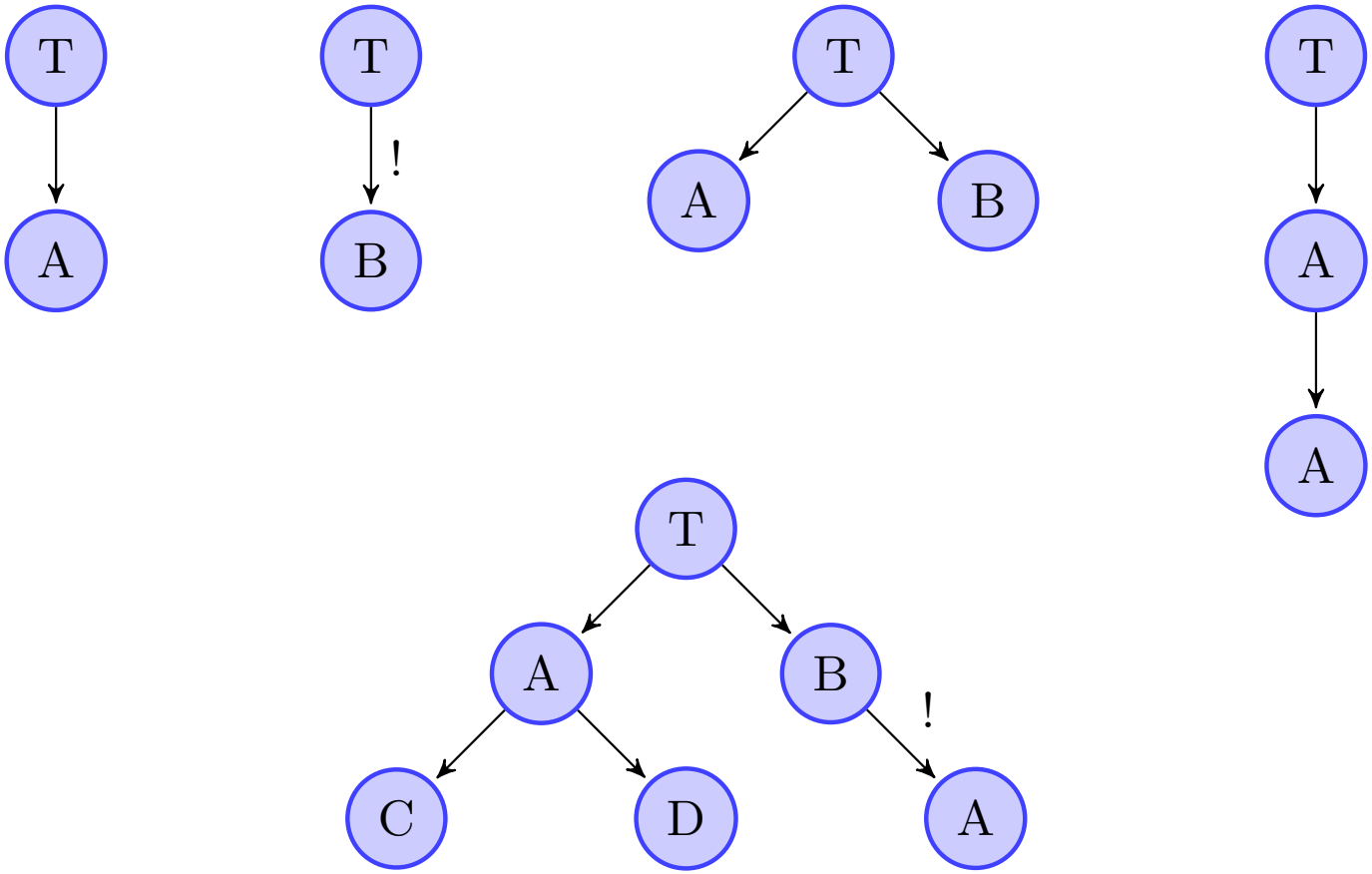
Semantics

- We use a labeled rooted tree
- Every vertex is reachable from a starting vertex T
- Each vertex is labeled with a symbol from S
- Each edge is labeled with either ! or *

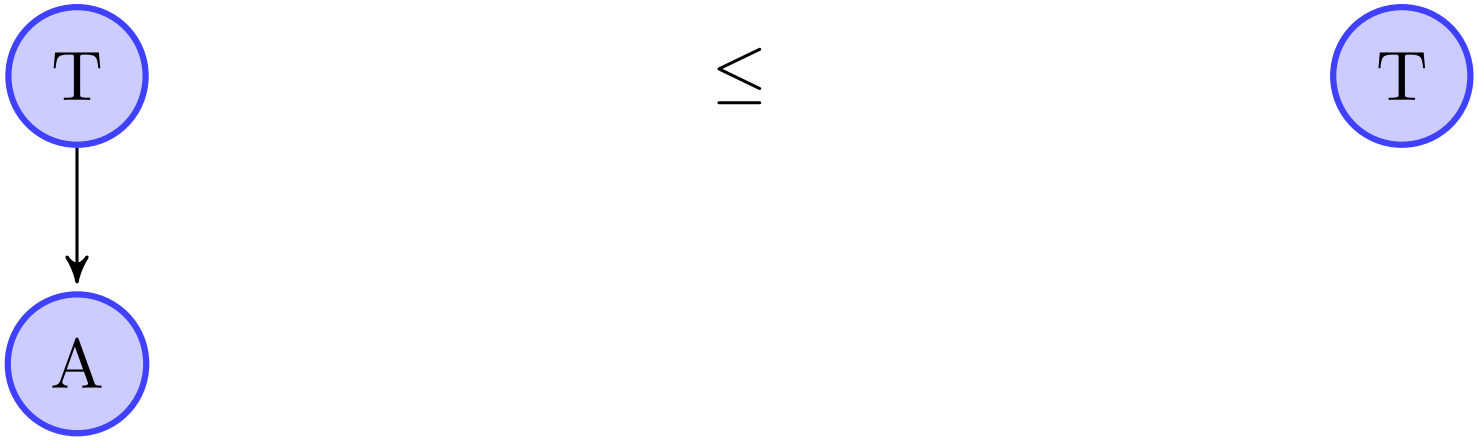
Labeled Rooted Tree

- (V, E, L, M, R) where
- V: set of vertices
- E: set of edges (V_1, V_2)
- L: vertex labeling V -> *S*
- M: edge labeling E -> {*,!}
- R: root, member of V

Semantics

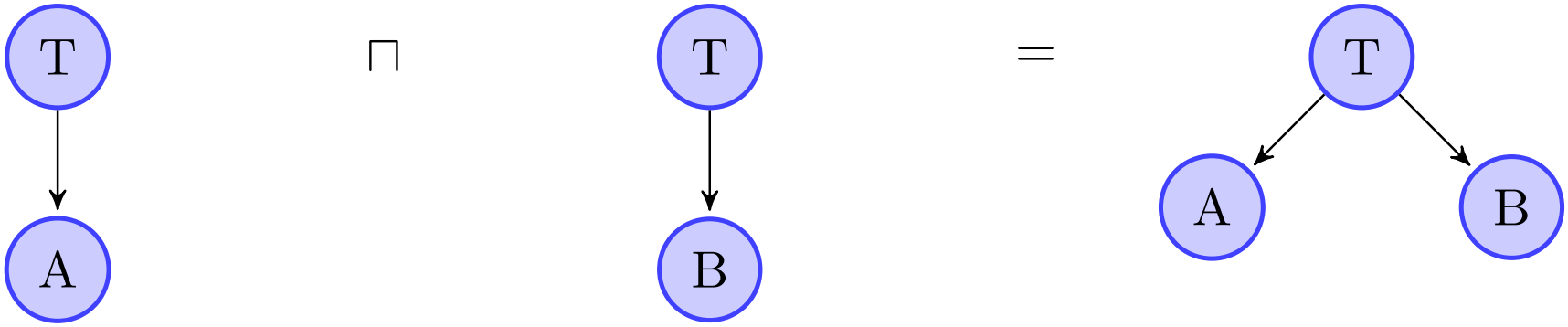


A Partial Ordering on LRTs

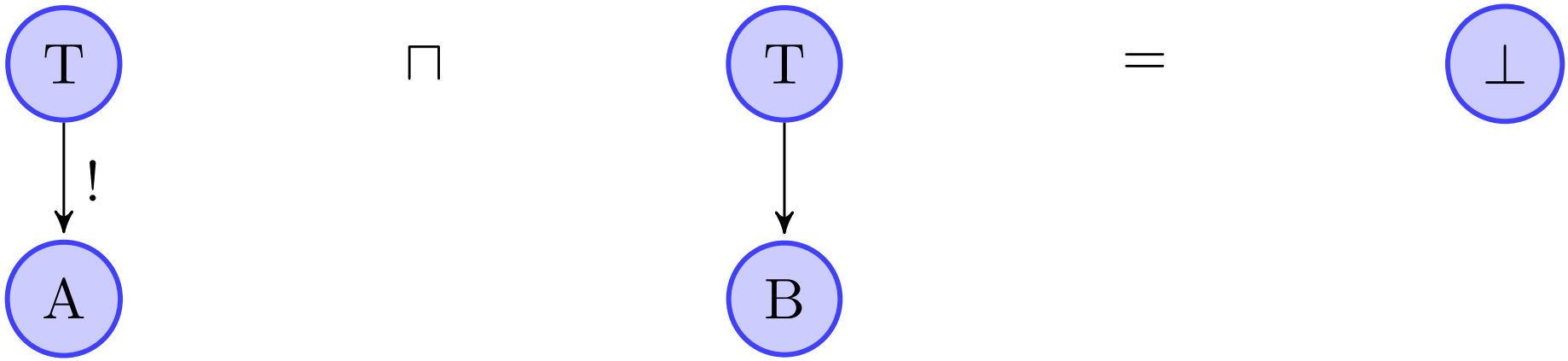


A Partial Ordering on LRTs

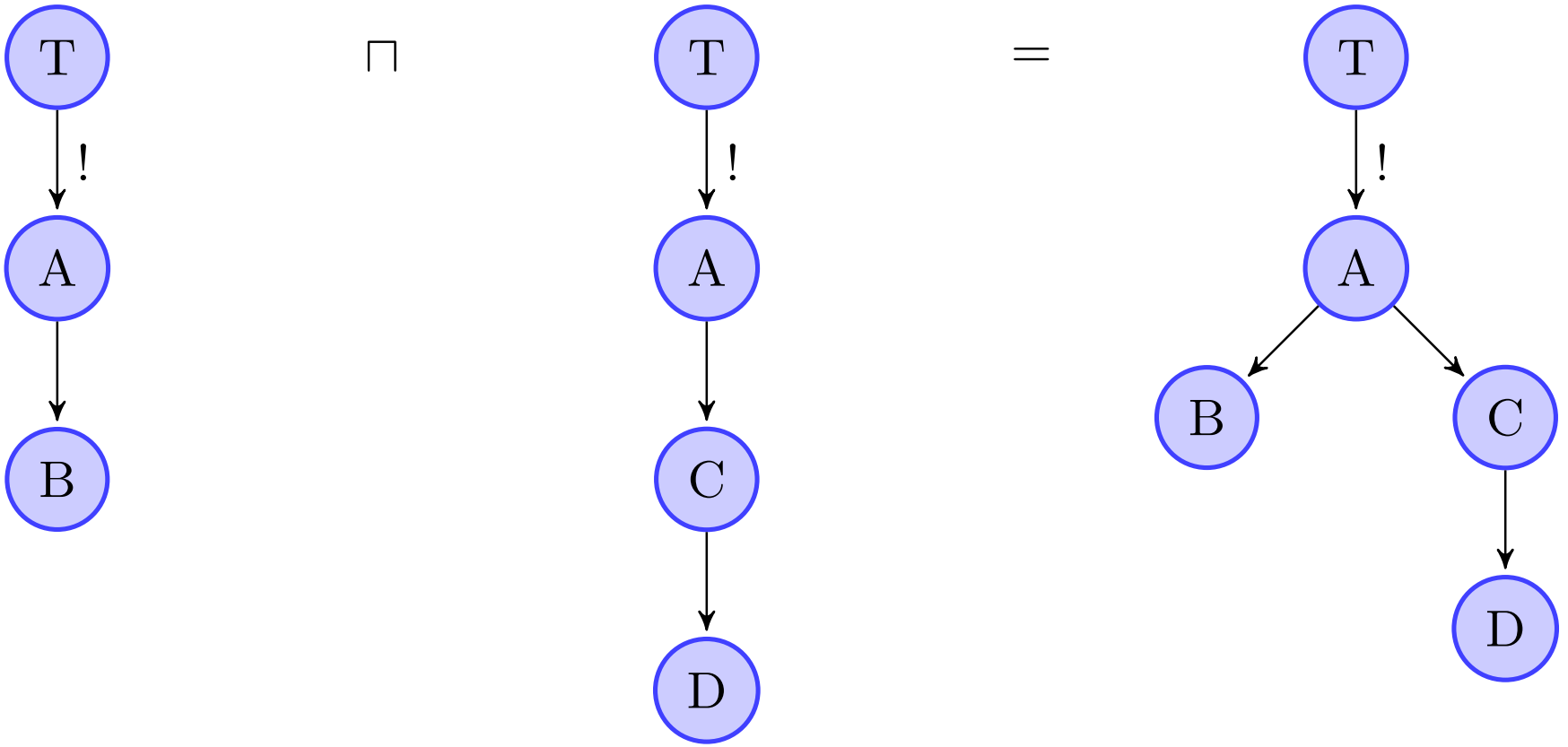
Greatest Lower Bound



Greatest Lower Bound



Greatest Lower Bound



Satisfaction

$$Sat(X, v, L, S) \text{ iff } \exists v' : (v, v') \in E_X \; L_X(v') = S \text{ and } M_X(v, v') = L$$

$$Sat(X, v, L, S!E) \text{ iff } \exists v' : (v, v') \in E_X \; L_X(v') = S \; M_X(v, v') = L \text{ and } Sat(X, v', !, E)$$

$$Sat(X, v, L, S.E) \text{ iff } \exists v' : (v, v') \in E_X \; L_X(v') = S \text{ and } M_X(v, v') = L \text{ and } Sat(X, v', *, E)$$

Decision Procedure

- Define *[x]* as the set of LRTs which satisfy *x*

$$[x] = \{M \mid \models_M x\}$$

- Because the LRTs form a lattice, this set has a least upper bound:

$$\bigsqcup [x]$$

Decision Procedure

$$X \models Y \text{ iff } \forall M \ \models_M X \Rightarrow \models_M Y$$

$$\text{iff } [X] \subseteq [Y]$$

$$\text{iff } \bigsqcup [X] \leq \bigsqcup [Y]$$

Computing the LUB

$$m(A \wedge B) = m(A) \sqcap m(B)$$

$$m(A{:}B) = (V_{m(A)} \cup \{v\}, E_{m(A)} \cup \{(v', v)\}, L_{m(A)} \cup (v, B), M_{m(A)} \cup \{((v', v), *)\})$$

$$m(A.B) = (V_{m(A)} \cup \{v\}, E_{m(A)} \cup \{(v', v)\}, L_{m(A)} \cup (v, B), M_{m(A)} \cup \{((v', v), !)\})$$

Hennessy-Milner Logic

- Let *A* be a set of constants
- Let *B* = {*,!} be a two-point set

$$C ::= \langle \alpha, b \rangle C \mid C \wedge C \mid \top$$
$$where \;\; \alpha \in A, b \in B$$

Hennessy-Milner Logic

- A model is a rooted graph where transitions are labeled with constants from *A*
- Satisfaction in a graph T rooted at r:

$$T \models \langle \alpha, * \rangle C \ if \ \exists t, r \xrightarrow{\alpha} t \wedge T(t) \models C$$

$$T \models \langle \alpha, ! \rangle C \ if \ \exists t, r \xrightarrow{\alpha} t \wedge T(t) \models C \wedge out(t) = 1$$

$$T \models A \wedge B \ if \ T \models A \wedge T \models B$$

Praxis

Using Exclusion Logic as a
Logic Programming Language

Praxis: Evolution

1) Roll-my-own procedural language
   - Spent a lot of time implementing basic language features
   - No debugger; no visualisation of state
2) Thin DSL on top of LUA
   - Untyped
3) Coded practices directly in C#
   - Verbose, error-prone
4) Practices encoded in Deontic Logic
5) Praxis

The Query Language

$$E ::= T \mid T.E \mid T!E$$

$$Q ::= E \mid \neg Q \mid Q \wedge Q \mid Q \vee Q \mid$$

$$Q \rightarrow Q \mid \forall X, Q \mid \exists X, Q$$

Typing

- Praxis is strongly typed and statically typed
- It has sub-typing
- It uses type-inference

Type Inference

```
function define_characters
    if global.playable.N!X
    then
        insert global.is_playing.X
        insert X.at!front_yard
        ...
```

```
global.playable.Index(number)!Agent(agent)
```

Instantiating Practices

```
process.greet.X(agent).Y(agent)
    action "Greet"
        preconditions
            Actor = X
            Actor.in!L
        postconditions
            text "[X] says 'hullo' to [Y obj]" if Recipient.in!L
            call update_conversation.L.Actor.greet.Y.respond_to_greet
            insert process.respond_to_greet.Y.X
            delete Self
end
```

Instantiating Practices

```
process.greet.X(agent).Y(agent)
    action "Greet"
        preconditions
            Actor = X
            Actor.in!L
        postconditions
            text "[X] says 'hullo' to [Y obj]" if Recipient.in!L
            call update_conversation.L.Actor.greet.Y.respond_to_greet
            insert process.respond_to_greet.Y.X
            delete Self
end
```

process.greet.jack.jill

Instantiating Practices

```
process.greet.X(agent).Y(agent)
    action "Greet"
        preconditions
            Actor = X
            Actor.in!L
        postconditions
            text "[X] says 'hullo' to [Y obj]" if Recipient.in!L
            call update_conversation.L.Actor.greet.Y.respond_to_greet
            insert process.respond_to_greet.Y.X
            delete Self
end
```

process.greet.jack.jill

Jack/X, Jill/Y

Practices are HFSMs

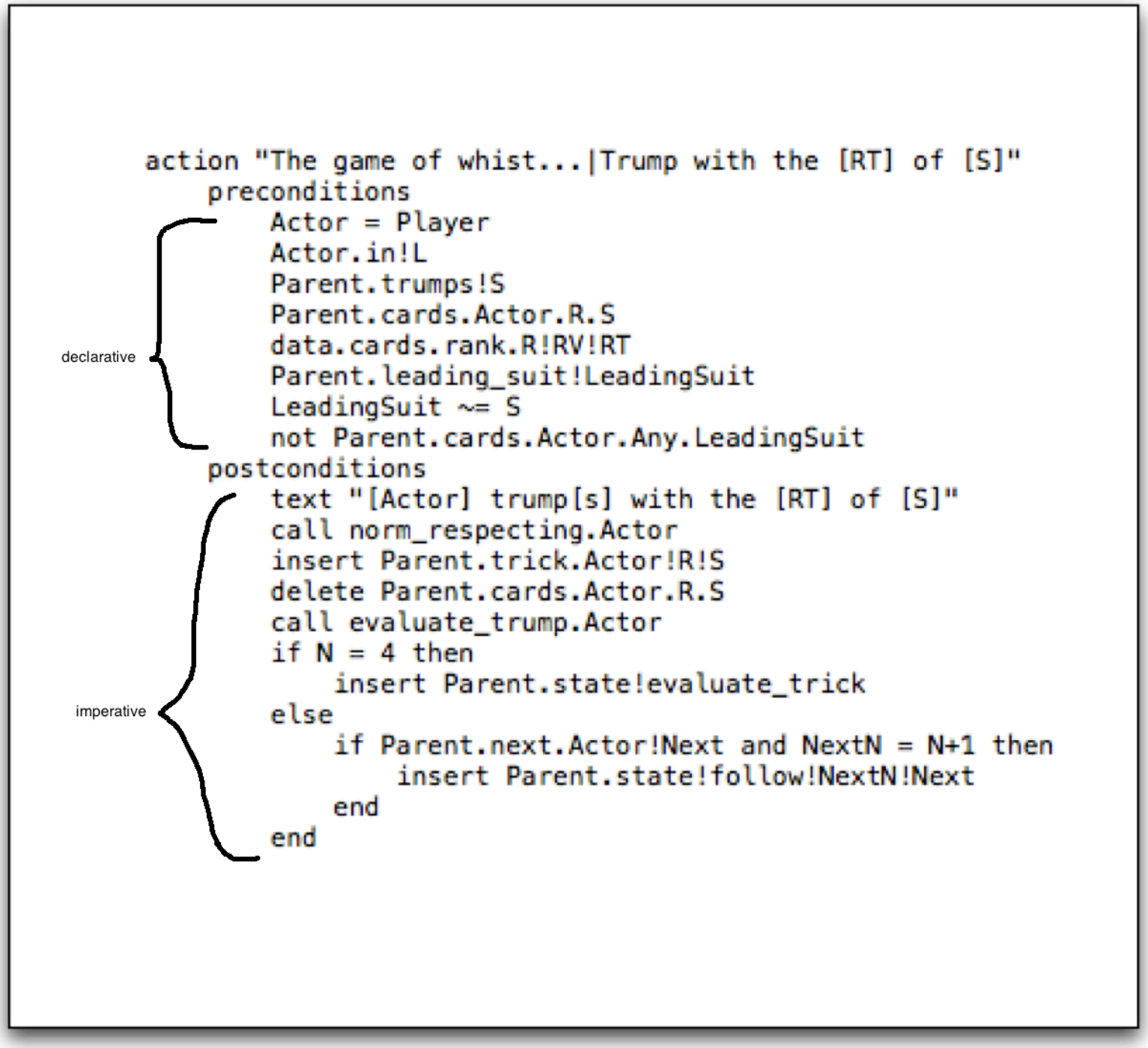
```
process.ticTacToe.Player1(agent).Player2(agent)
    ...
    state!whoseMove!Mover(agent)!Other(agent)
        action "Tic Tac Toe | Row [R] | Place [Piece] at [C],[R]"
            preconditions
                Actor = Mover
                Parent.board.C.R!empty
                Parent.piece.Mover!Piece
                Parent.piece.Other!OtherPiece
            postconditions
                text "[Mover] place[s] an [Piece] at [C], [R]." if Par
                insert Parent.board.C.R!Piece
                call updateBoardOnMove.Mover.Other.C.R.Piece.OtherPiec
                insert Parent.state!whoseMove!Other!Mover
                ...
```

Practices have constructors

```
process.ticTacToe.Player1(agent).Player2(agent)
    start
        insert Self.participants.Player1
        insert Self.participants.Player2
        insert Self.viewers.Player1
        insert Self.viewers.Player2
        text "You are playing 'X'" to Player1
        text "You are playing 'O'" to Player2
        insert Self.piece.Player1!x
        insert Self.piece.Player2!o
        ...
```

Practices provide actions

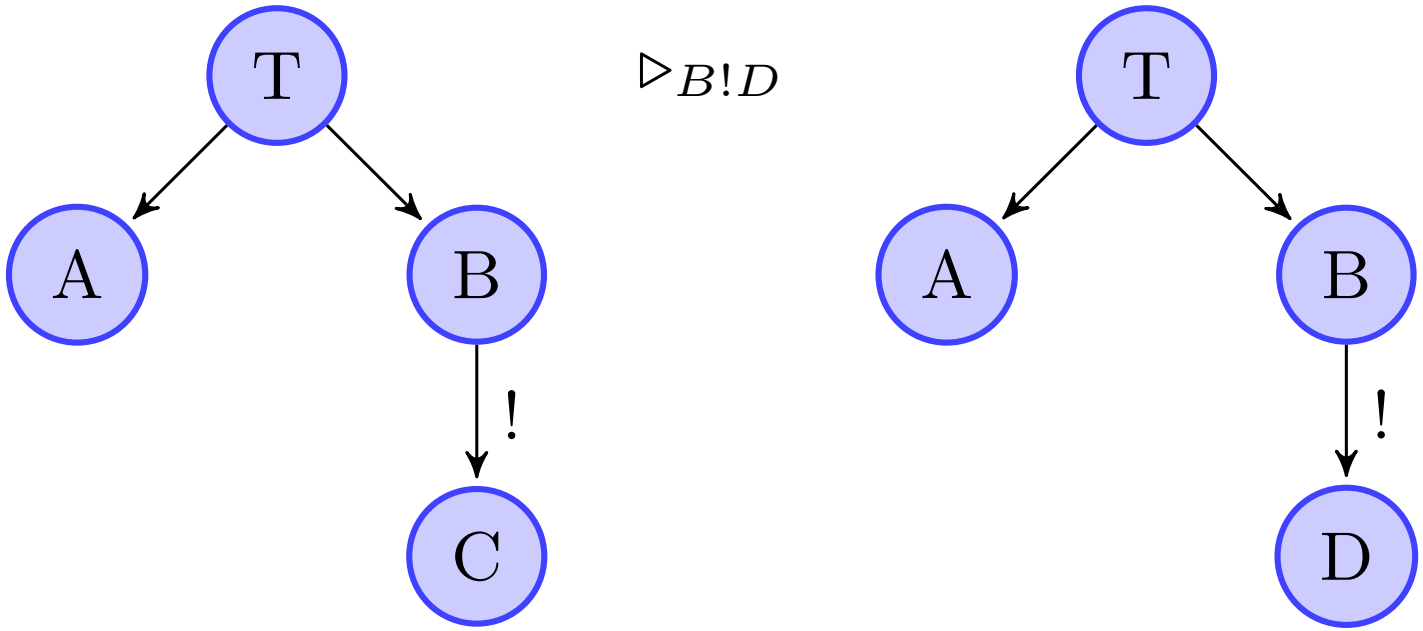
```
action "The game of whist...|Trump with the [RT] of [S]"
    preconditions
        Actor = Player
        Actor.in!L
        Parent.trumps!S
        Parent.cards.Actor.R.S
        data.cards.rank.R!RV!RT
        Parent.leading_suit!LeadingSuit
        LeadingSuit ~= S
        not Parent.cards.Actor.Any.LeadingSuit
    postconditions
        text "[Actor] trump[s] with the [RT] of [S]"
        call norm_respecting.Actor
        insert Parent.trick.Actor!R!S
        delete Parent.cards.Actor.R.S
        call evaluate_trump.Actor
        if N = 4 then
            insert Parent.state!evaluate_trick
        else
            if Parent.next.Actor!Next and NextN = N+1 then
                insert Parent.state!follow!NextN!Next
            end
        end
```


action "The game of whist...|Trump with the [RT] of [S]"
preconditions
Actor = Player
Actor.in!L
Parent.trumps!S
Parent.cards.Actor.R.S
data.cards.rank.R!RV!RT
Parent.leading_suit!LeadingSuit
LeadingSuit ~= S
not Parent.cards.Actor.Any.LeadingSuit
declarative
postconditions
text "[Actor] trump[s] with the [RT] of [S]"
call norm_respecting.Actor
insert Parent.trick.Actor!R!S
delete Parent.cards.Actor.R.S
call evaluate_trump.Actor
if N = 4 then
insert Parent.state!evaluate_trick
else
if Parent.next.Actor!Next and NextN = N+1 then
insert Parent.state!follow!NextN!Next
end
end
imperative

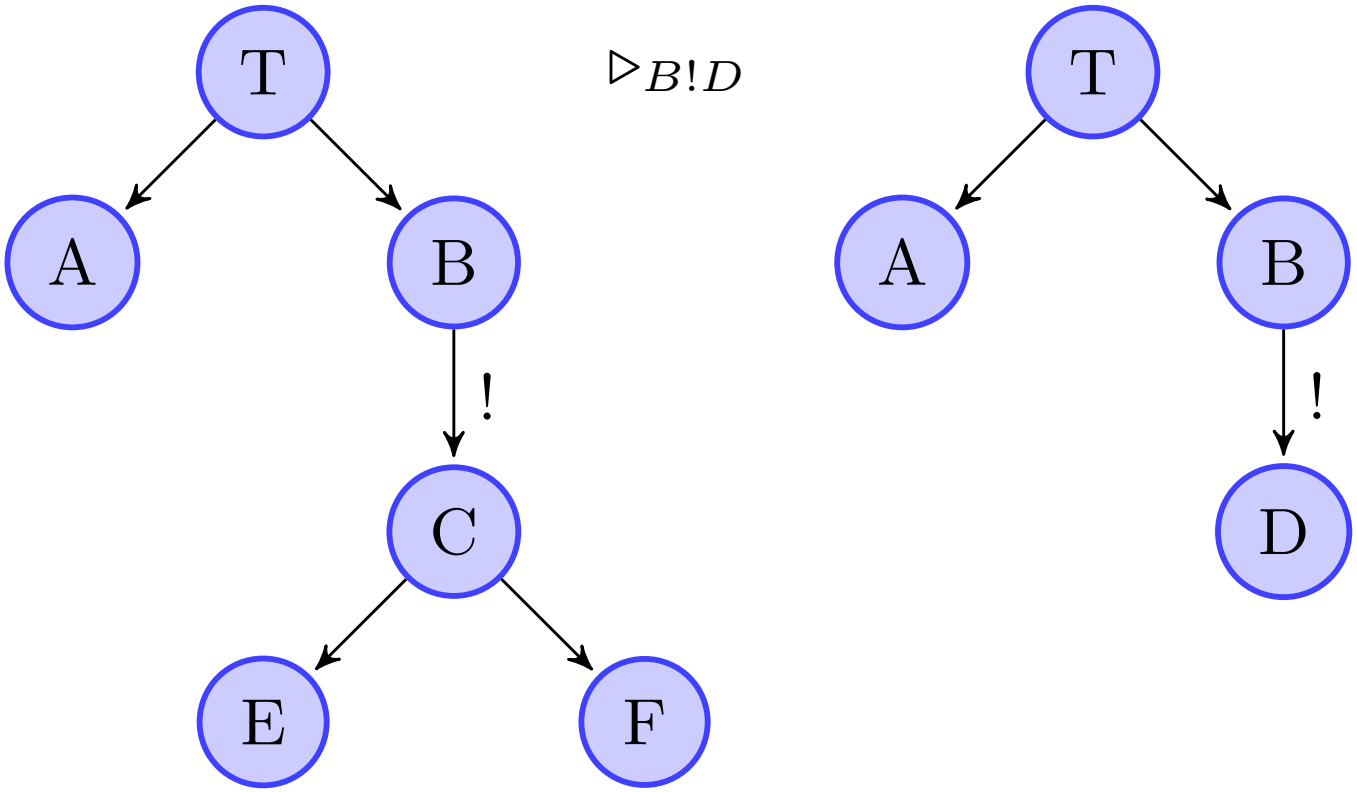
Updating the Database

- When adding a sentence *p* to the database, *we first remove all information which is incompatible with p*
- This is a non-monotonic update

Updating the Database



Updating the Database



Using Exclusion Logic as a KRL

An Object is a Sub-Tree

- `brown.sex!male`
- `brown.class!upper`
- `brown.in!dining_room`
- `brown.relationship.lucy.evaluation.attractive!40`
- `brown.relationship.lucy.evaluation.humour!20`

An Object is a Sub-Tree

- Specify the life-time of a piece of data by placing it in the right part of the tree

- `brown.relationship.lucy.evaluation.attractive!40`
- `process.whist.data.whose_move!brown`

Garbage Collection

- An FSM has two states **a** and **b**
- State **a** has two bits of data: x and y
- We are in state **a**:
- fsm.state!a.x /\ fsm.state!a.y
- Now insert fsm.state!b
- The data (a.x /\ a.y) is removed automatically

Simpler Postconditions

```
action move(A, X, Y)
    preconditions
        at(A, X)
    postconditions
        add at(A, Y)
        remove at(A, X)
```

Simpler Postconditions

```
action move(A, X, Y)
    preconditions
        A.at!X
    postconditions
        add A.at!Y
```

Simpler Queries

$$Married(Bride, Groom, Place, Time, Official)$$

Who is Jill married to?

$$(\exists g, p, t, f)\ Married(Jill, g, p, t, f)$$

$$Married.Jill$$

Exclusion is Typing Information

- `A(agent).sex!G(gender)`
- `brown.sex.male`
- Bad typing in brown.sex.male in line 65
- The first problem appears to be with “male”

Improvements to Praxis

Exclusion Logic

$$E ::= S \mid S.E \mid S!E$$

$$C ::= E \mid \neg C \mid C \wedge C$$

Extended Exclusion Logic

$$E ::= T \mid T.E \mid T!E \mid E \wedge E$$

$$A.(B \wedge C) = A.B \wedge A.C$$

$$A!(B \wedge C) \neq A!B \wedge A!C$$

Extended Exclusion Logic

$$A.(B \wedge C) \models A.(C \wedge B)$$
$$A.(B.D \wedge C) \models A.(B \wedge C)$$
$$A.(B \wedge C) \models A.B \wedge A.C$$

$$A!(B \wedge C) \models A!(C \wedge B)$$
$$A!(B.D \wedge C) \models A!(B \wedge C)$$
$$A!(B \wedge C) \models \neg A!B \wedge \neg A!C$$

Improving Praxis

- Data abstraction
- Hindley-Milner type system

Compiling Praxis

- Warren Abstract Machine?
- Or Mercury-style compilation?
  - Explicit mode declarations for predicates
    - append(in, in, out)
    - append(out, out, in)
  - Separate procedures generated for each mode declaration