Improving Dynamic Procedural Knowledge Learning

Mazin Assanie
University of Michigan
mazina@umich.edu

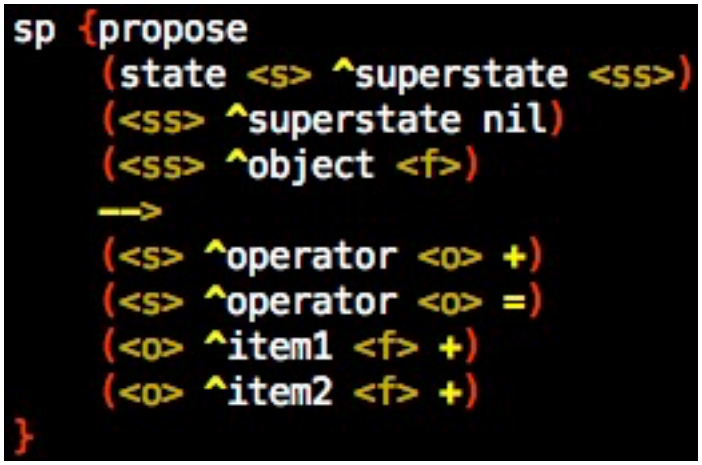
Motivation

- Chunking was a core initial capability of Soar that has potential to drastically improve an agent's future performance as it gains experience solving problems in the world
- *Yet seldom used in real-world systems*

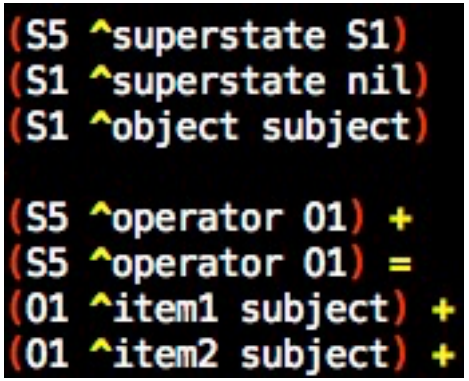
- Our goal:
  1. Identify problems that chunking currently has
  2. Diagnose underlying source
  3. Implement an improved algorithm to address

Chunking Terminology

- **Rule**: Human-written production
- **Chunk**: Rule learned from Soar's rule learning mechanism
- **Instantiation**: Data structure created by Soar to record a specific rule firing



Match



Rule

Instantiation

Chunking Terminology

- **Rule**: Human-written production
- **Chunk**: Rule learned from Soar's rule learning mechanism
- **Instantiation**: Data structure created by Soar to record a specific rule firing

```
sp {propose
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil)
    (<ss> ^object <f>)
    -->
    (<s> ^operator <o> +)
    (<s> ^operator <o> =)
    (<o> ^item1 <f> +)
    (<o> ^item2 <f> +)
}
```

Match

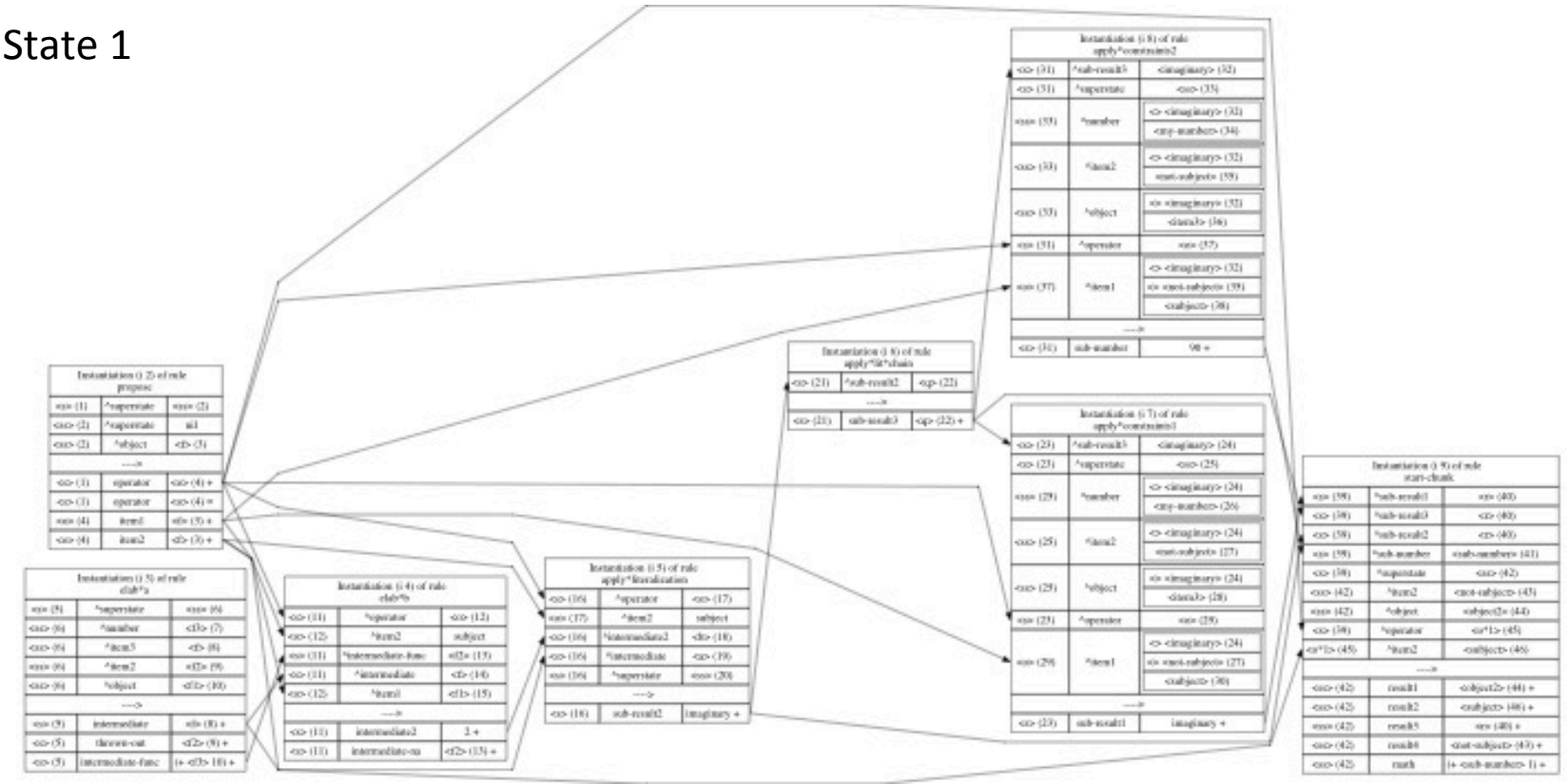
Instantiation (i 2) of rule propose		
S5 (1)	^superstate	S1 (2)
S1 (2)	^superstate	nil
S1 (2)	^object	subject (3)
---->		
S5	operator	O1 +
S5	operator	O1 =
O1	item1	subject +
O1	item2	subject +

Rule

Instantiation

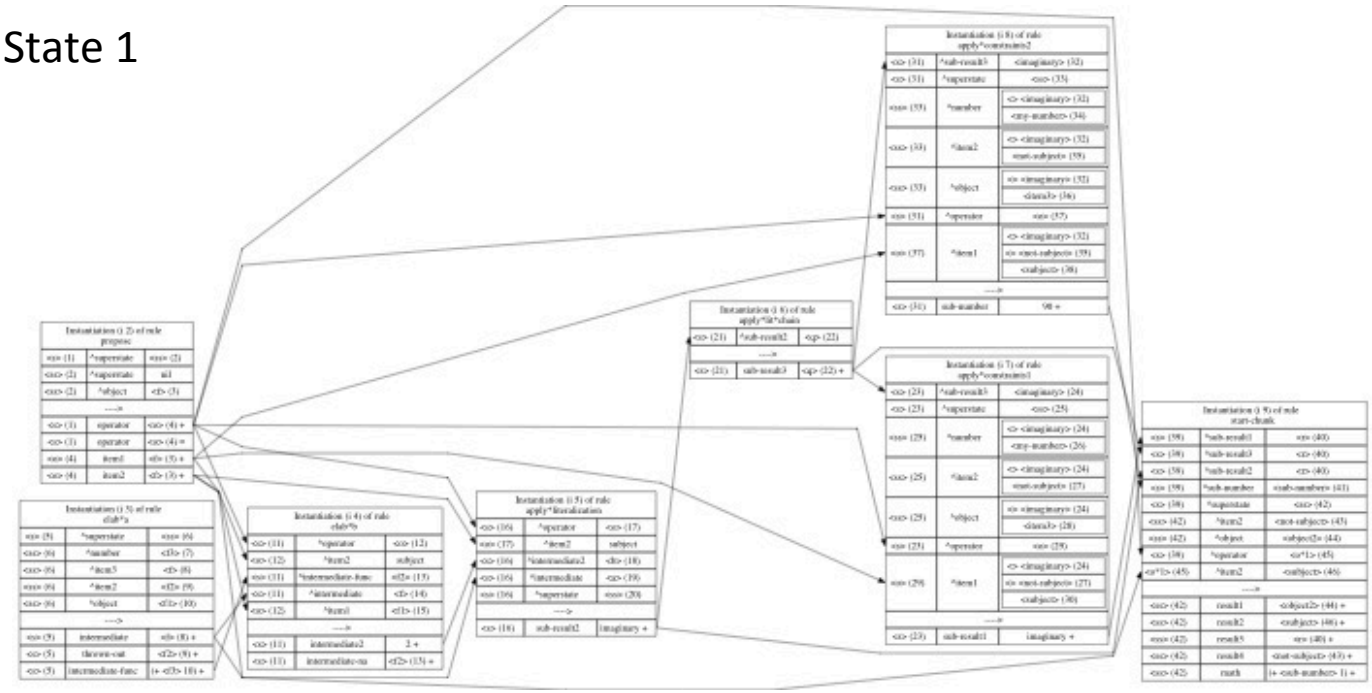
Problem-Solving in State Occurs

State 1



Impasse Occurs

State 1

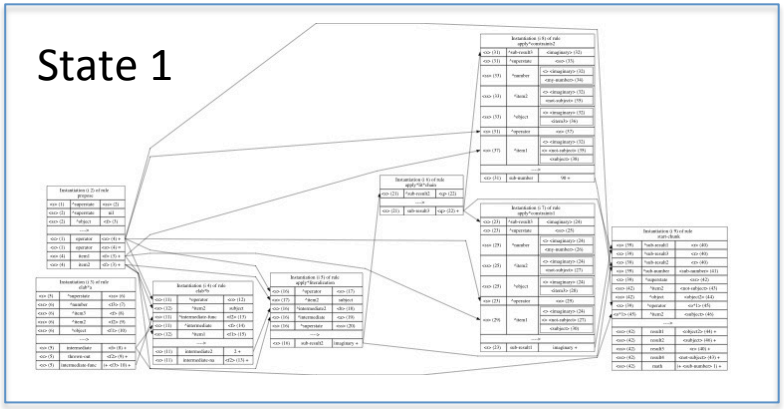


- Agent does not have enough information to make further progress in goal

  (no rule can fire or no operator can be selected)

New Sub-state Created

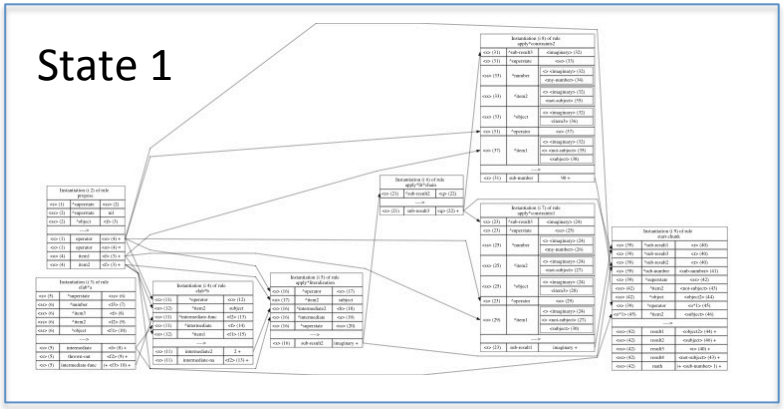
- Soar creates a subgoal to resolve that lack of information.

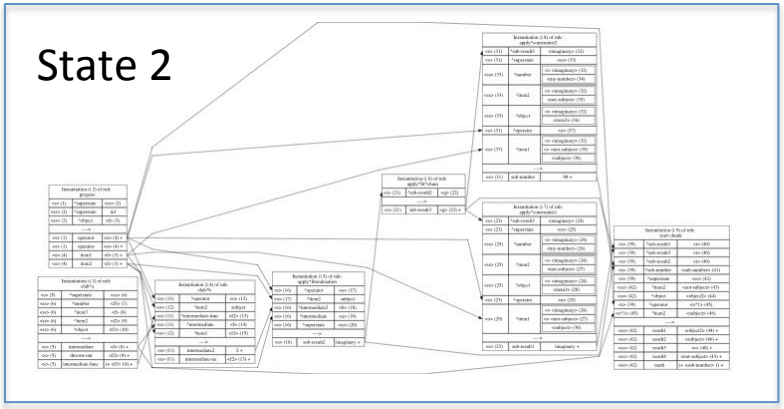


State 2

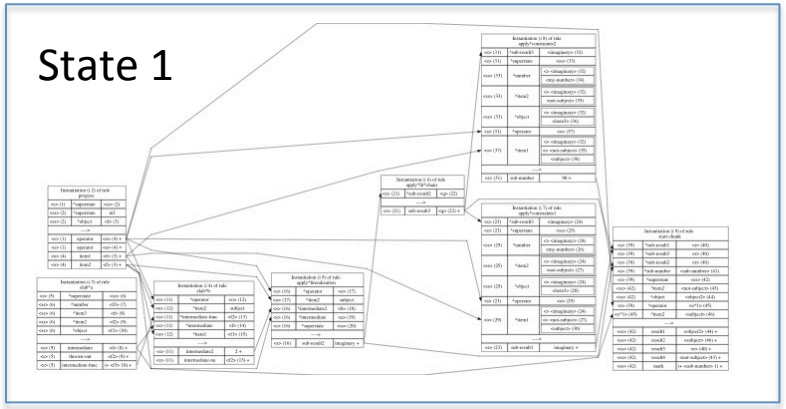
Rules Fire in Subgoal

- Problem-solving in the sub-goal attempts to generate super-state knowledge to resolve the impasse

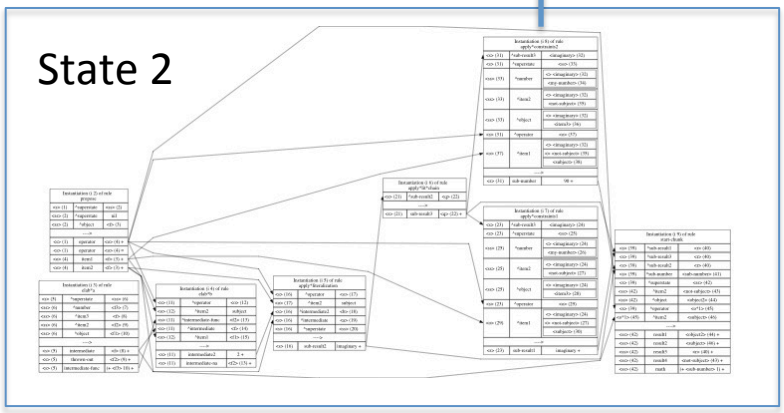




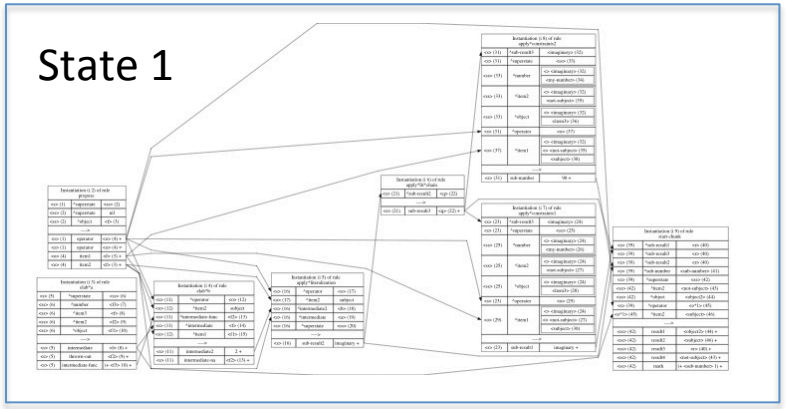
Result Created



New WMEs in State 1



Chunking Happens

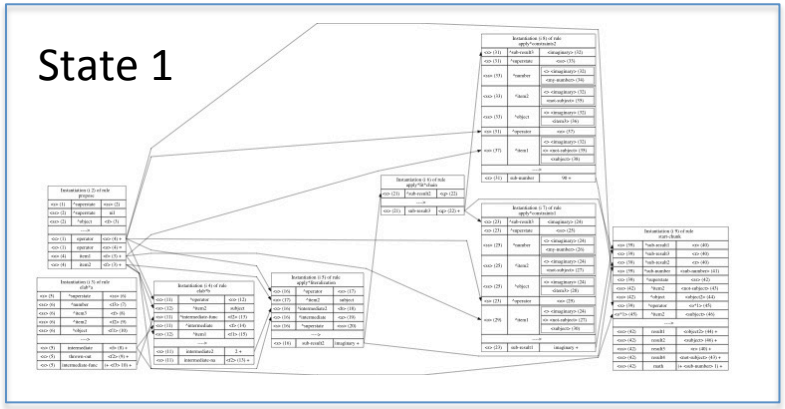


New WMEs in State 1

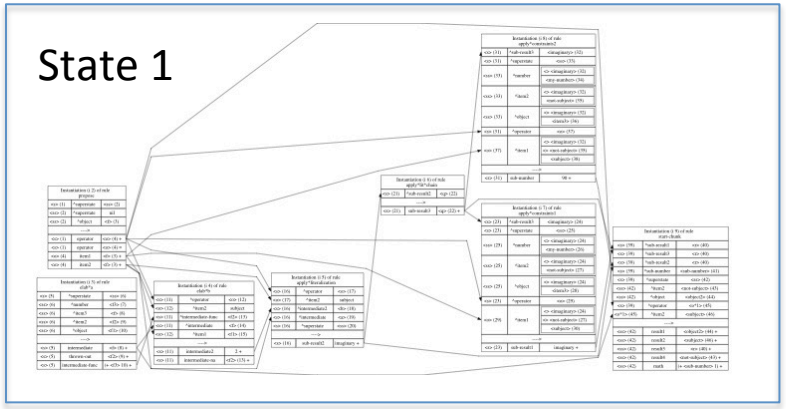
State 2

Analyzes trace of rule firings to learn new rule

Future Situation



Future Situation



Chunk

<s> (39)	^sub-result1	<z> (40)
<s> (39)	^sub-result3	<z> (40)
<s> (39)	^sub-result2	<z> (40)
<s> (39)	^sub-number	<sub-number> (41)
<s> (39)	^superstate	<ss> (42)
<ss> (42)	^item2	<not-subject> (43)
<ss> (42)	^object	<object2> (44)
<s> (39)	^operator	<o*1> (45)
<o*1> (45)	^item2	<subject> (46)
	--->	
<ss> (42)	result1	<object2> (44) +

New WMEs in State 1

Processing Avoided

How Chunking Originally Built Rules

- Three main mechanisms:
  1. Instantiation tracking
  2. Dependency analysis
  3. Variablization

Dependency Analysis

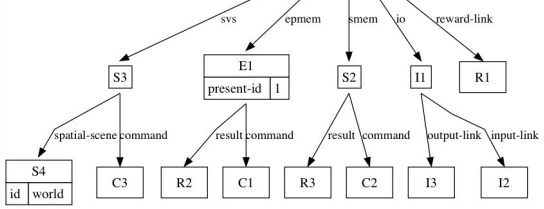
- Soar determines the set of *necessary working memory elements in the superstate* that were needed for those rules to match.
- How?
  - Starts with rules that create super-goal knowledge
  - Traverses back through a trace of substate instantiated rules that fired
  - Anything that matches supergoal WMEs is added to the set of necessary WMEs

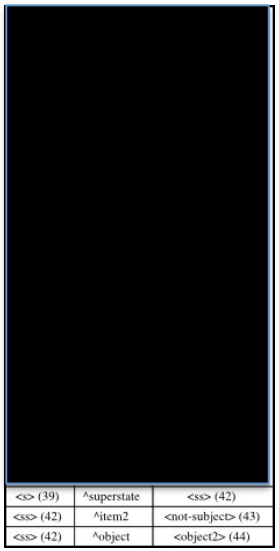
Dependency Analysis

Working Memory

State 1 WM

S1	
superstate	nil
number	23
item3	apple
math	91
item2	not-subject
object	subject
result3	imaginary
type	state
result4	not-subject
result2	subject
result1	subject



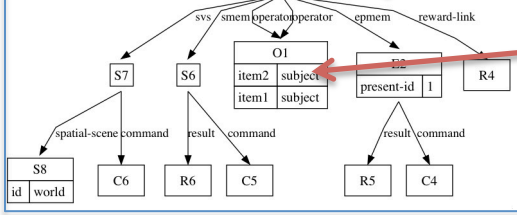


<s> (39)	^superstate	<ss> (42)
<ss> (42)	^item2	<not-subject> (43)
<ss> (42)	^object	<object2> (44)

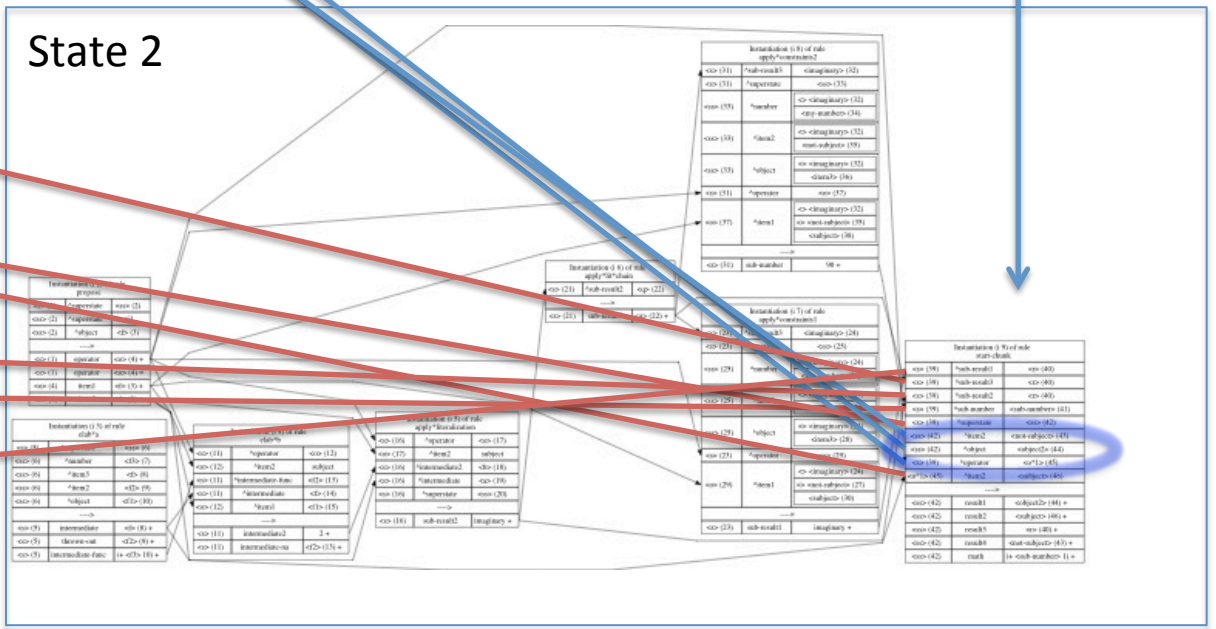
Rule that creates the result

State 2 WM

S5	
quiescence	t
superstate	S1
type	state
intermediate-na	33
intermediate-func	33
intermediate2	2
sub-result3	imaginary
sub-result2	imaginary
impasse	no-change
thrown-out	not-subject
sub-number	90
attribute	state
intermediate	apple
choices	none
sub-result1	imaginary



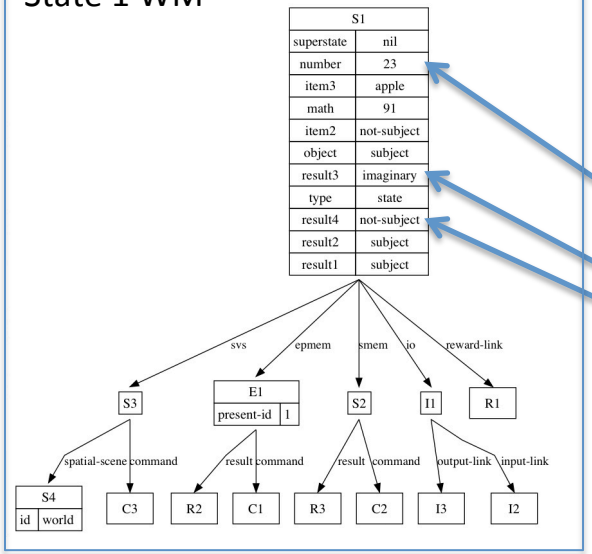
State 2

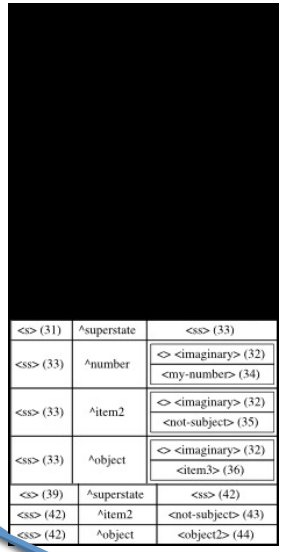


Dependency Analysis

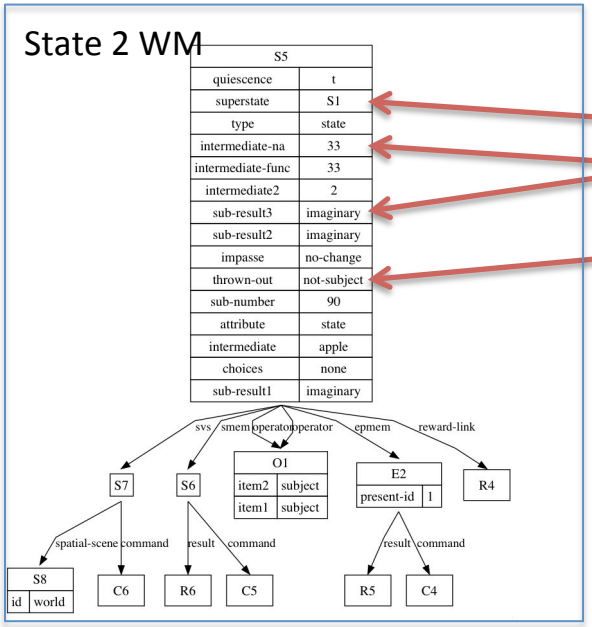
Working Memory

State 1 WM

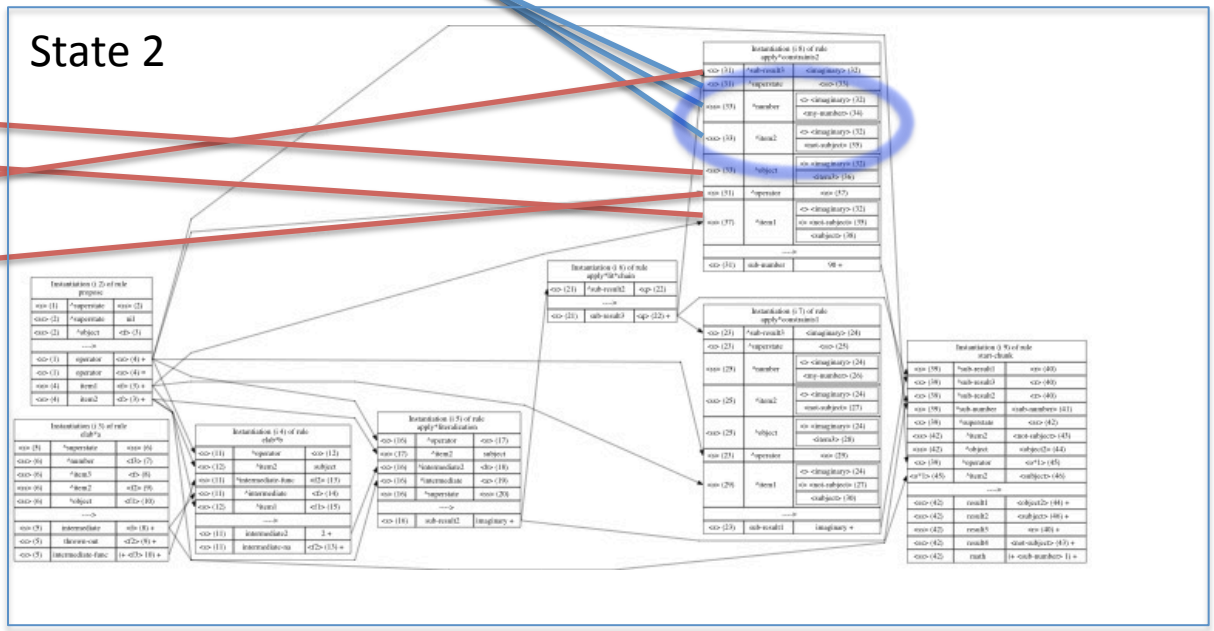




State 2 WM



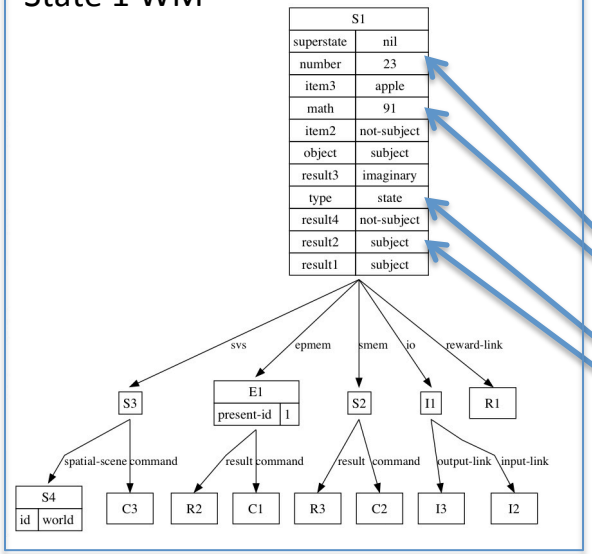
State 2

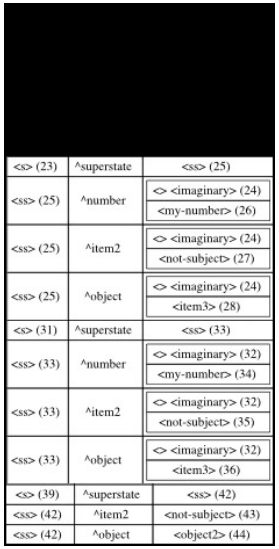


Working Memory

Dependency Analysis

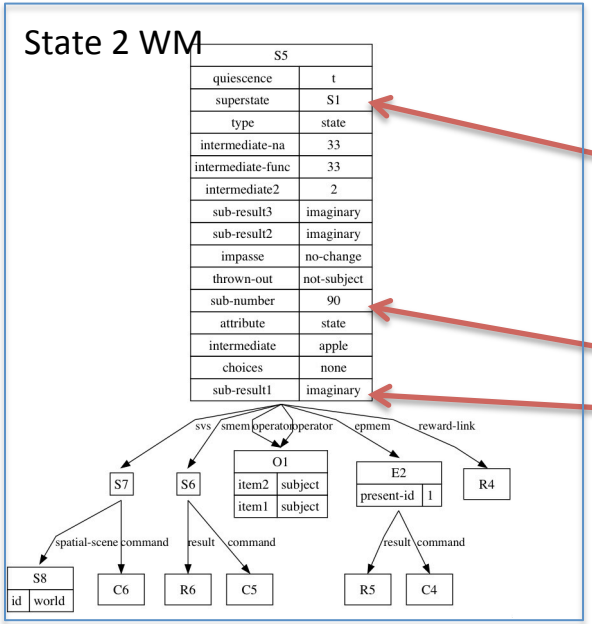
State 1 WM



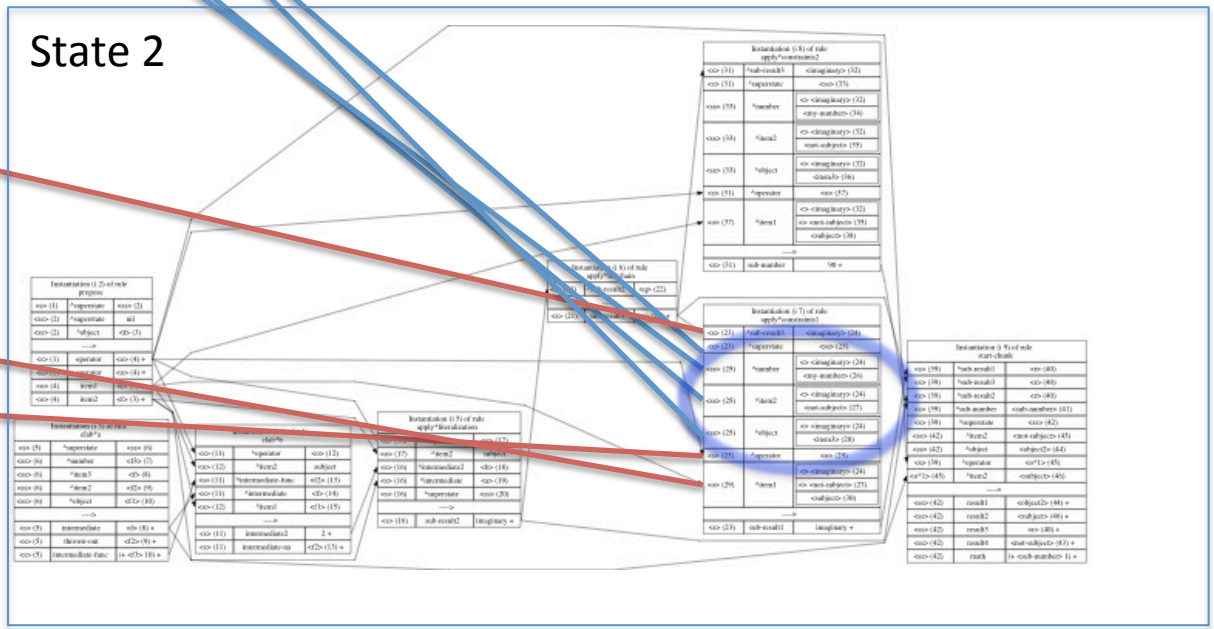


<s> (23)	^superstate	<ss> (25)
<ss> (25)	^number	<> <imaginary> (24) <my-number> (26)
<ss> (25)	^item2	<> <imaginary> (24) <not-subject> (27)
<ss> (25)	^object	<> <imaginary> (24) <item3> (28)
<s> (31)	^superstate	<ss> (33)
<ss> (33)	^number	<> <imaginary> (32) <my-number> (34)
<ss> (33)	^item2	<> <imaginary> (32) <not-subject> (35)
<ss> (33)	^object	<> <imaginary> (32) <item3> (36)
<s> (39)	^superstate	<ss> (42)
<ss> (42)	^item2	<not-subject> (43)
<ss> (42)	^object	<object2> (44)

State 2 WM



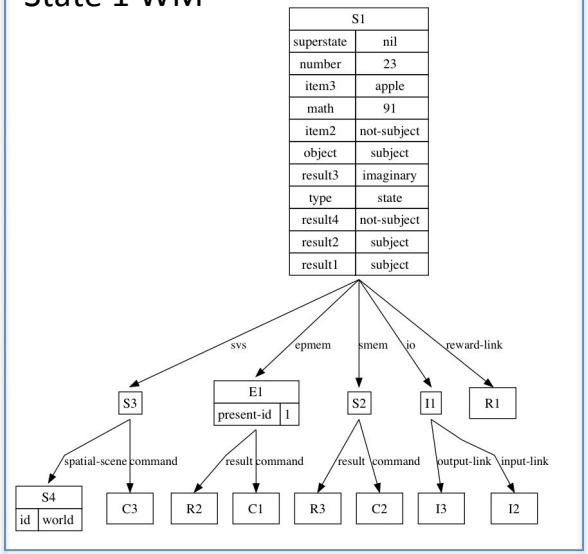
State 2

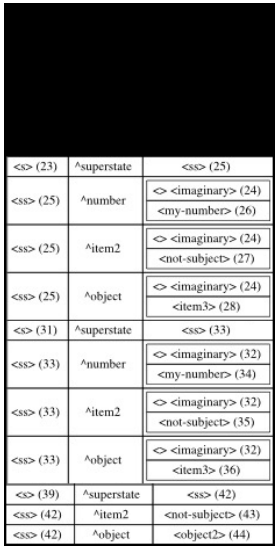


Working Memory

Dependency Analysis

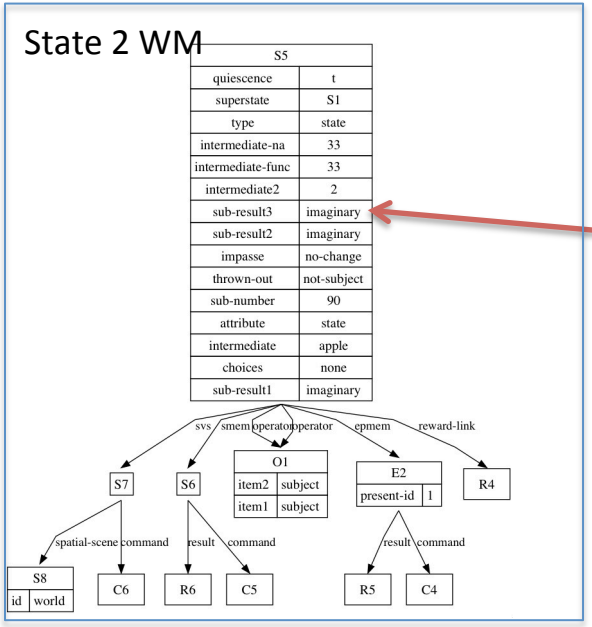
State 1 WM



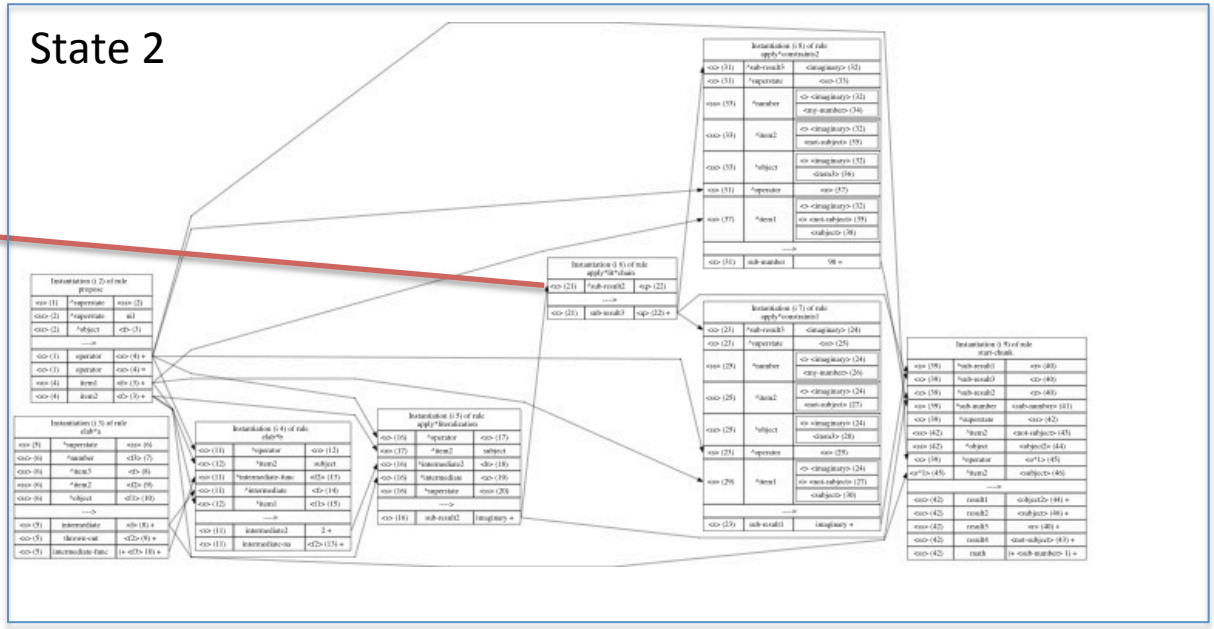


<s> (23)	^superstate	<ss> (25)
<ss> (25)	^number	<> <imaginary> (24) <my-number> (26)
<ss> (25)	^item2	<> <imaginary> (24) <not-subject> (27)
<ss> (25)	^object	<> <imaginary> (24) <item3> (28)
<s> (31)	^superstate	<ss> (33)
<ss> (33)	^number	<> <imaginary> (32) <my-number> (34)
<ss> (33)	^item2	<> <imaginary> (32) <not-subject> (35)
<ss> (33)	^object	<> <imaginary> (32) <item3> (36)
<s> (39)	^superstate	<ss> (42)
<ss> (42)	^item2	<not-subject> (43)
<ss> (42)	^object	<object2> (44)

State 2 WM



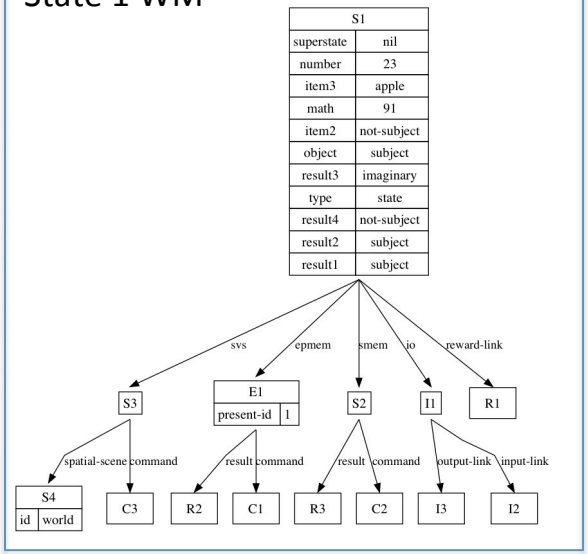
State 2

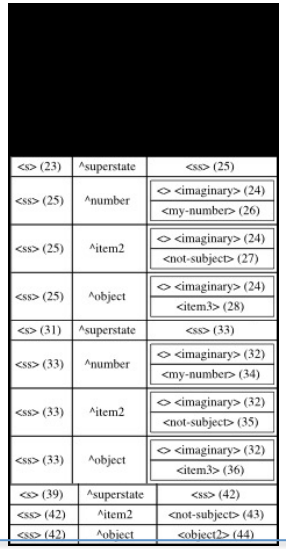


Working Memory

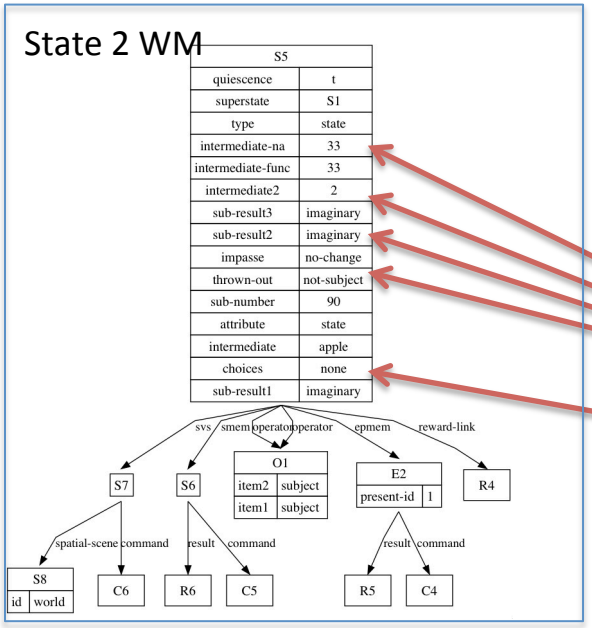
Dependency Analysis

State 1 WM

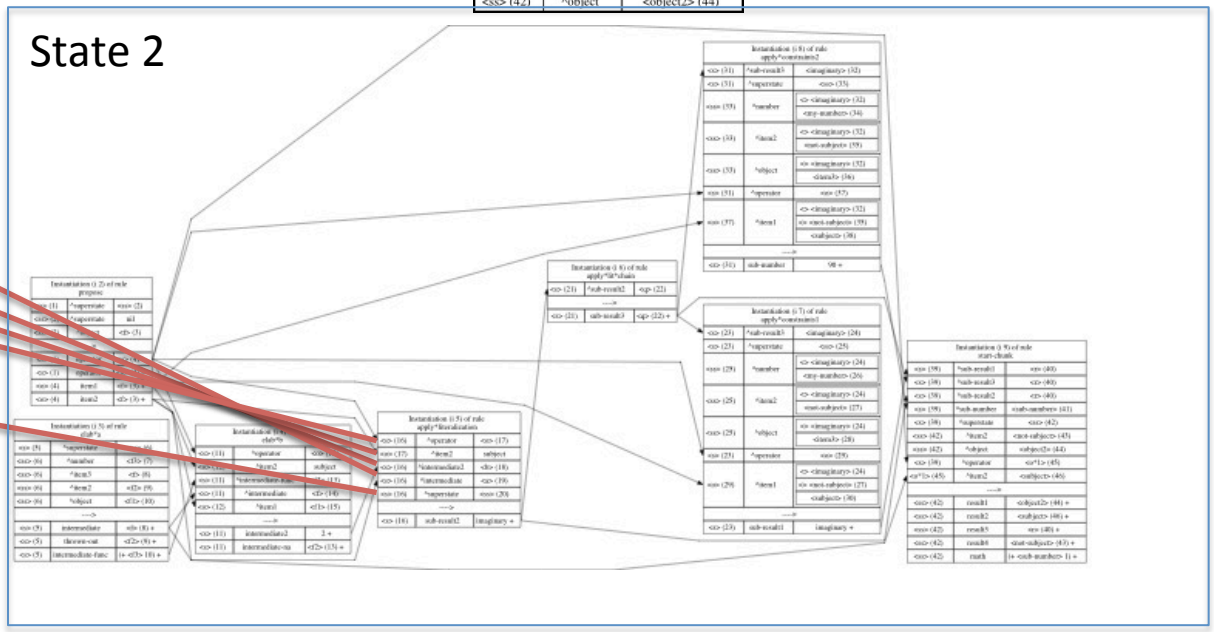




State 2 WM



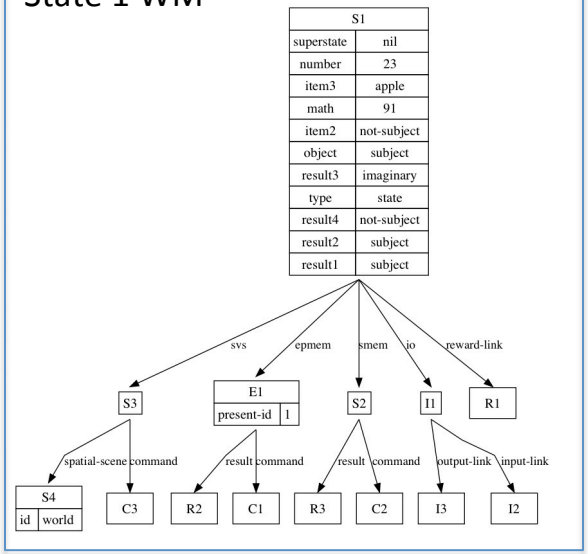
State 2

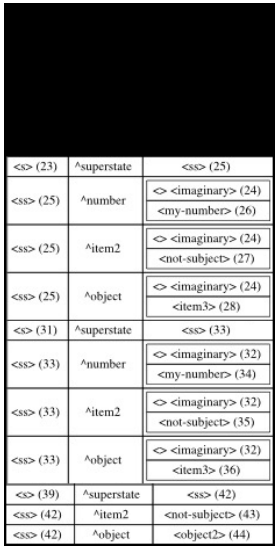


Dependency Analysis

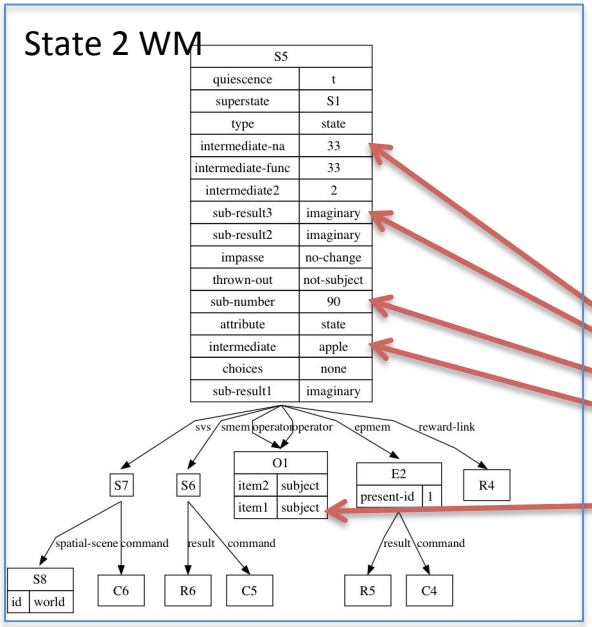
Working Memory

State 1 WM

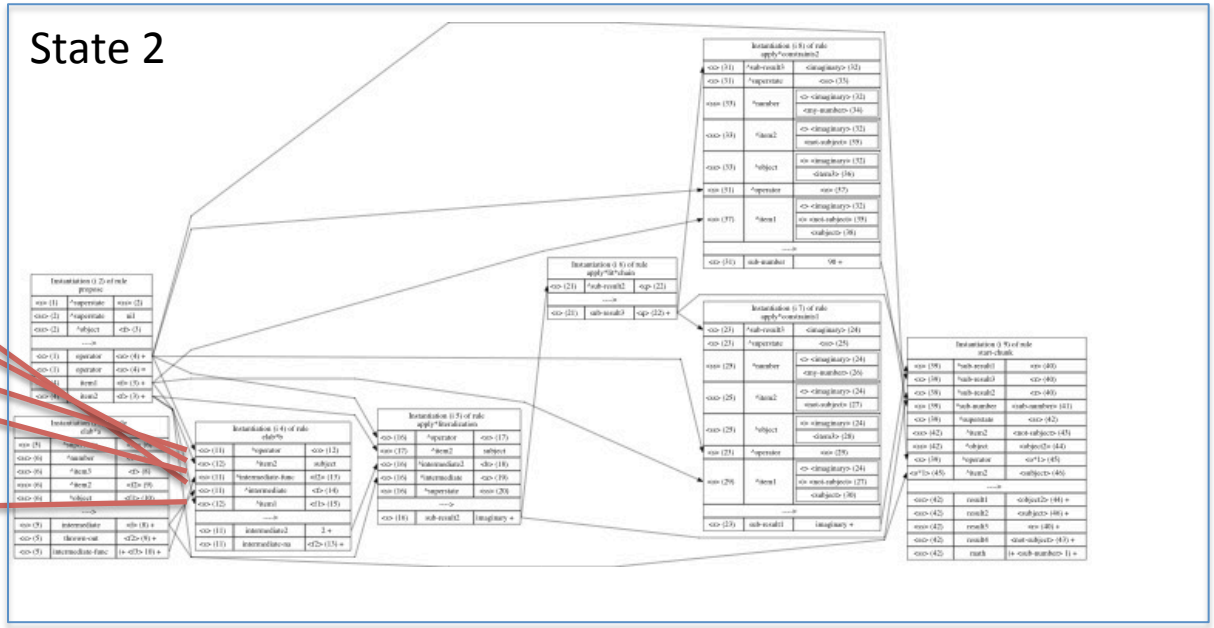




State 2 WM



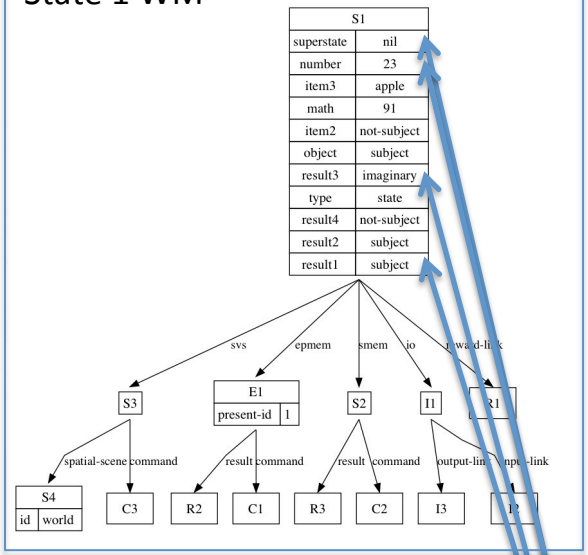
State 2



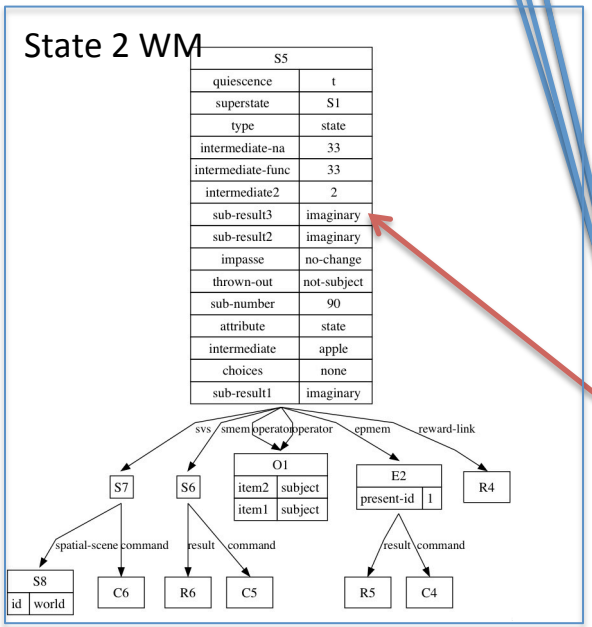
Dependency Analysis

Working Memory

State 1 WM

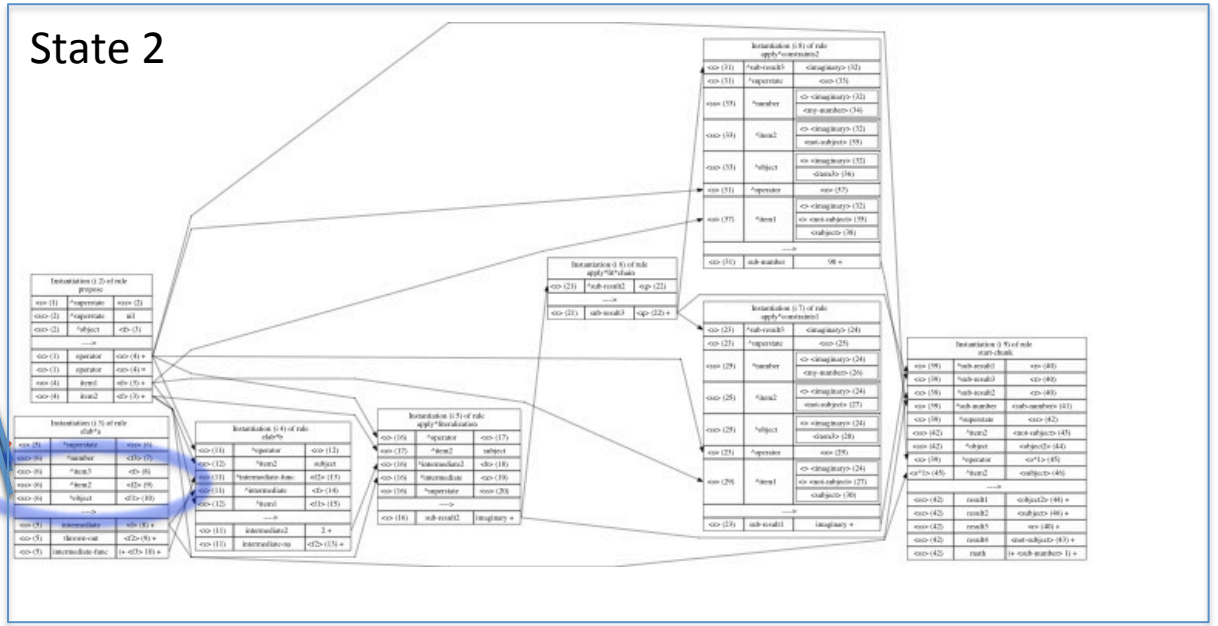


State 2 WM



<ss> (6)	^number	<f3> (7)
<ss> (6)	^item3	<f> (8)
<ss> (6)	^item2	<f2> (9)
<ss> (6)	^object	<f1> (10)
<s> (23)	^superstate	<ss> (25)
<ss> (25)	^number	<> <imaginary> (24) <my-number> (26)
<ss> (25)	^item2	<> <imaginary> (24) <not-subject> (27)
<ss> (25)	^object	<> <imaginary> (24) <item3> (28)
<s> (31)	^superstate	<ss> (33)
<ss> (33)	^number	<> <imaginary> (32) <my-number> (34)
<ss> (33)	^item2	<> <imaginary> (32) <not-subject> (35)
<ss> (33)	^object	<> <imaginary> (32) <item3> (36)
<s> (39)	^superstate	<ss> (42)
<ss> (42)	^item2	<not-subject> (43)
<ss> (42)	^object	<object2> (44)

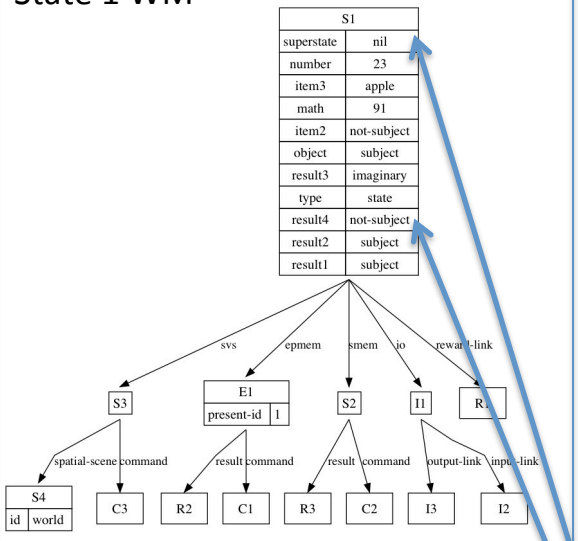
State 2



Dependency Analysis

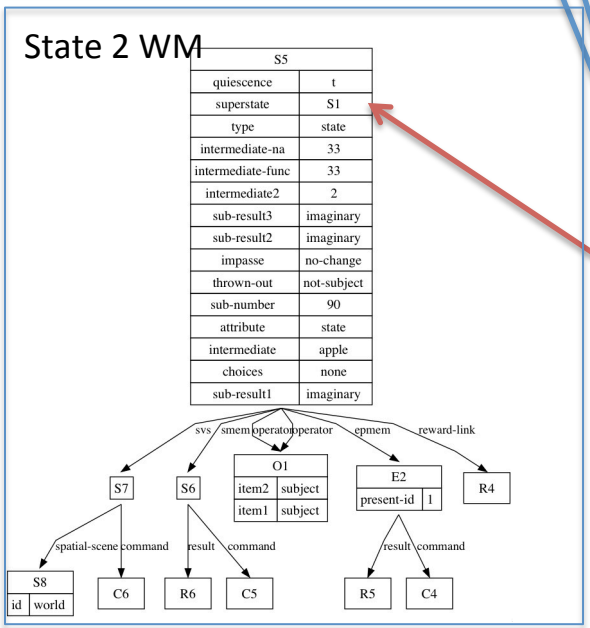
Working Memory

State 1 WM

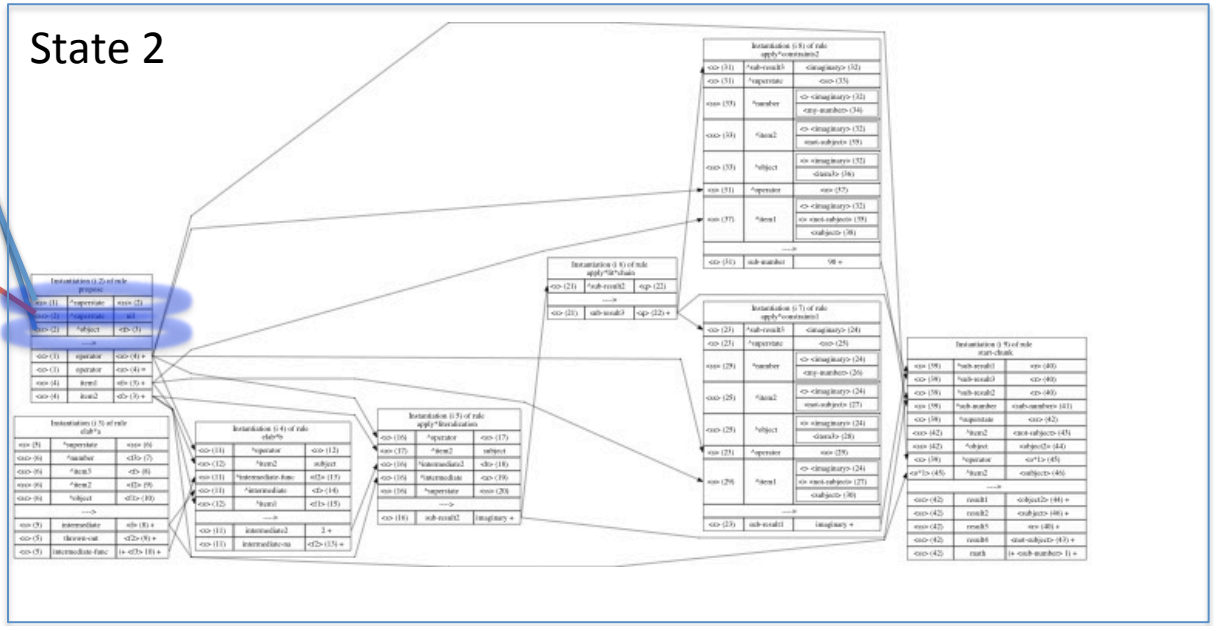


<ss> (2)	^superstate	nil
<ss> (2)	^object	<f> (3)
<ss> (6)	^number	<f3> (7)
<ss> (6)	^item3	<f> (8)
<ss> (6)	^item2	<f2> (9)
<ss> (6)	^object	<f1> (10)
<s> (23)	^superstate	<ss> (25)
<ss> (25)	^number	<> <imaginary> (24) <my-number> (26)
<ss> (25)	^item2	<> <imaginary> (24) <not-subject> (27)
<ss> (25)	^object	<> <imaginary> (24) <item3> (28)
<s> (31)	^superstate	<ss> (33)
<ss> (33)	^number	<> <imaginary> (32) <my-number> (34)
<ss> (33)	^item2	<> <imaginary> (32) <not-subject> (35)
<ss> (33)	^object	<> <imaginary> (32) <item3> (36)
<s> (39)	^superstate	<ss> (42)
<ss> (42)	^item2	<not-subject> (43)
<ss> (42)	^object	<object2> (44)

State 2 WM

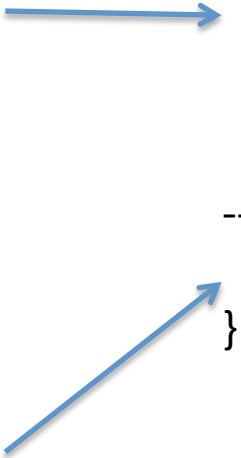


State 2



What does chunking have now?

- Conditions:
  - Working memory elements that were collected during the traversal
- Actions:
  - Working memory elements that were created in the super-goal

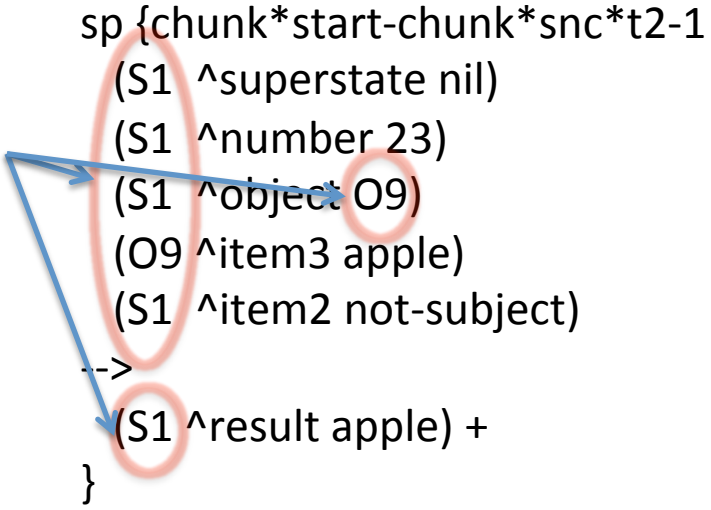


```
sp {chunk*start-chunk*snc*t2-1
   (S1  ^superstate nil)
   (S1  ^number 23)
   (S1  ^object O9)
   (O9 ^item3 apple)
   (S1  ^item2 not-subject)
-->
   (S1 ^result apple) +
}
```

Completely Specific!

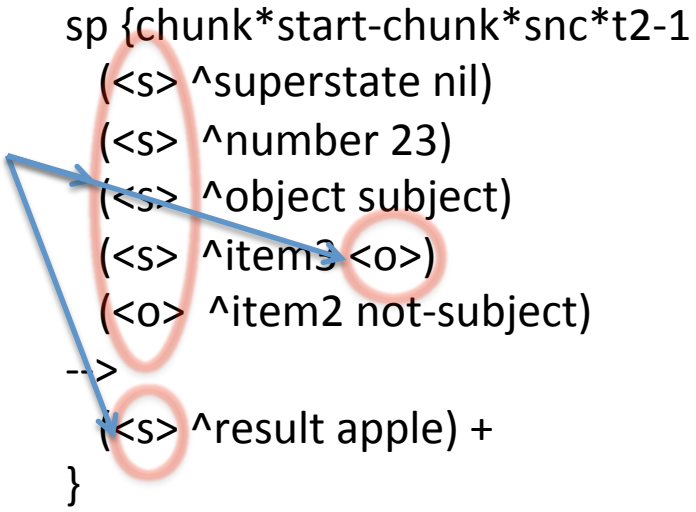
Generalizing the Learned Rule

- To make the rule more general, chunking replaces soar identifiers with variables



Generalizing the Learned Rule

- To make the rule more general, chunking replaces Soar identifiers with variables



Why is it not used?

Learned rules have issues with:

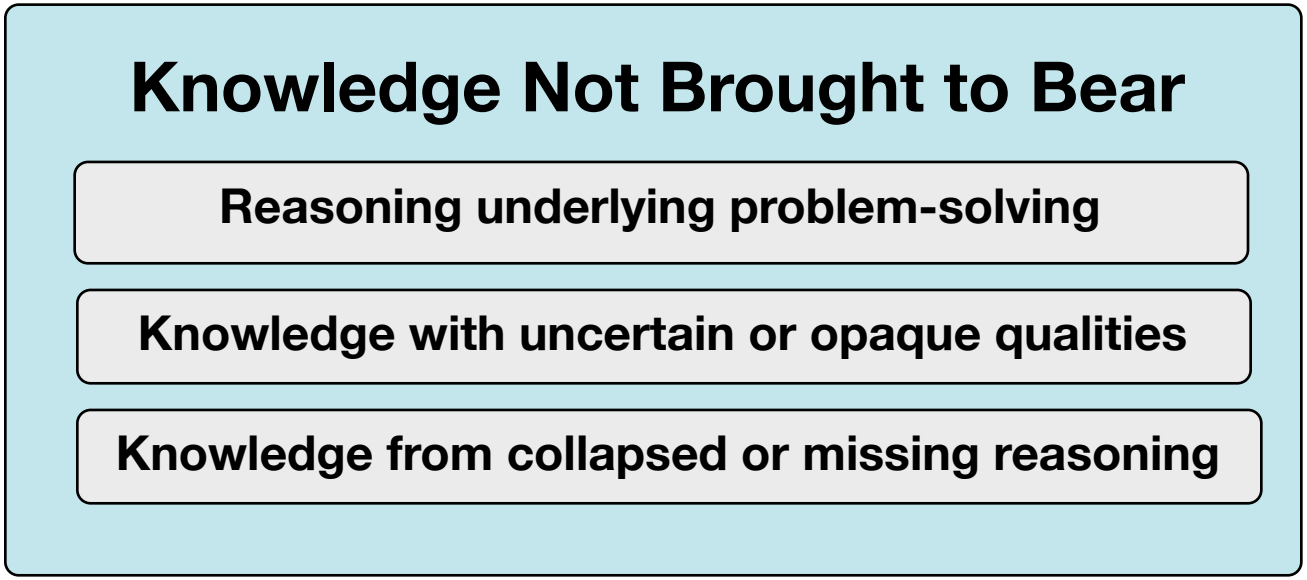
- **Correctness**: Learned rules don't always effect the same inferences as the original problem solving would have in the same situation.

- **Generality**: Learned rules don't correctly apply to a broad range of similar situations.

Both the *amount of engineering to avoid generality issues* and the *lack of confidence in the correctness of the rules* causes agent engineers to avoid chunking.

Diagnosis

- *Current algorithms do not capture and take advantage of all the knowledge available about the problem-solving that occurs*.



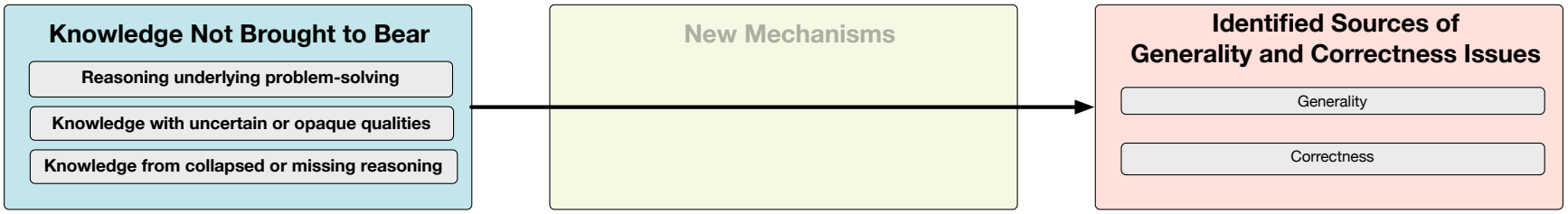
Hypothesis

- Bringing this knowledge to bear will:
  - Improve generality and correctness of learned rules
  - Eliminate need for special knowledge engineering
  - Allow chunks to consistently improve the performance of our Soar agents

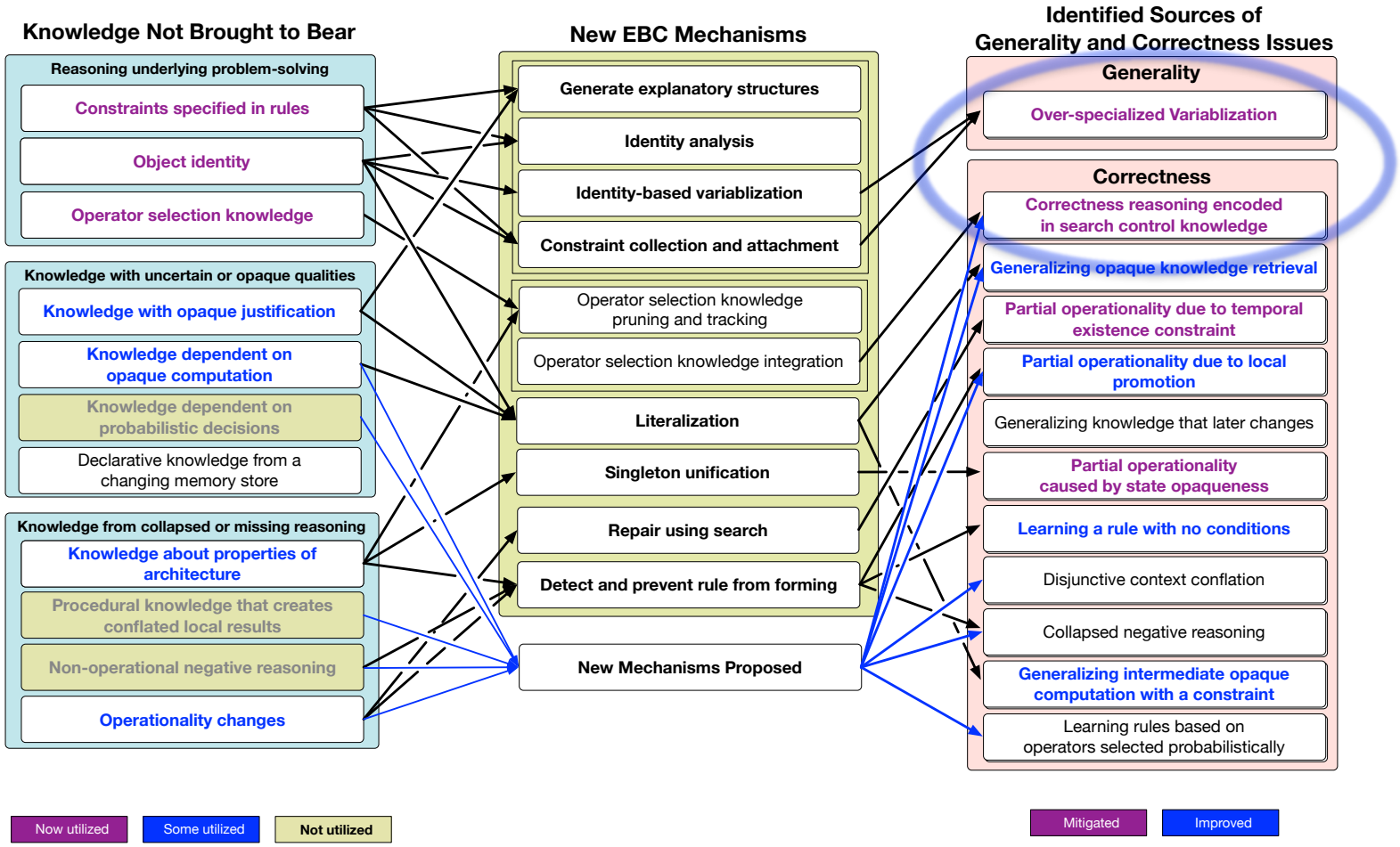
Agenda

1. Describe two generality and correctness issues
   - Show 9.4 chunk
2. Describe knowledge not integrated
   - Show explanation-based chunk
3. High-level overview of the mechanism

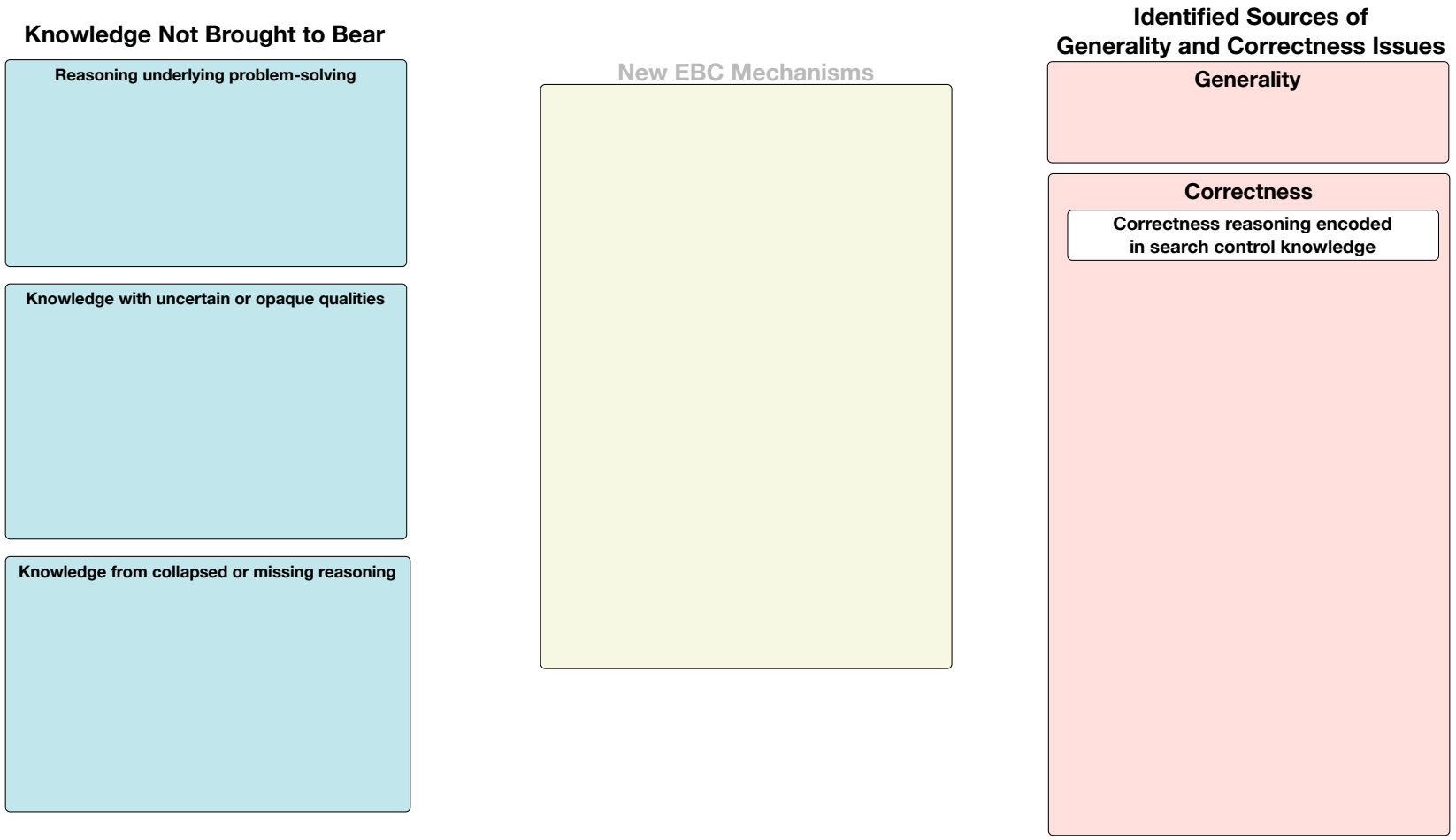
We'll try to do two items…



Preview


Knowledge Not Brought to Bear
Reasoning underlying problem-solving
Constraints specified in rules
Object identity
Operator selection knowledge
Knowledge with uncertain or opaque qualities
Knowledge with opaque justification
Knowledge dependent on opaque computation
Knowledge dependent on probabilistic decisions
Declarative knowledge from a changing memory store
Knowledge from collapsed or missing reasoning
Knowledge about properties of architecture
Procedural knowledge that creates conflated local results
Non-operational negative reasoning
Operationality changes
New EBC Mechanisms
Generate explanatory structures
Identity analysis
Identity-based variablization
Constraint collection and attachment
Operator selection knowledge pruning and tracking
Operator selection knowledge integration
Literalization
Singleton unification
Repair using search
Detect and prevent rule from forming
New Mechanisms Proposed
Identified Sources of Generality and Correctness Issues
Generality
Over-specialized Variablization
Correctness
Correctness reasoning encoded in search control knowledge
Generalizing opaque knowledge retrieval
Partial operationality due to temporal existence constraint
Partial operationality due to local promotion
Generalizing knowledge that later changes
Partial operationality caused by state opaqueness
Learning a rule with no conditions
Disjunctive context conflation
Collapsed negative reasoning
Generalizing intermediate opaque computation with a constraint
Learning rules based on operators selected probabilistically
Now utilized
Some utilized
Not utilized
Mitigated
Improved

Operator Selection Knowledge



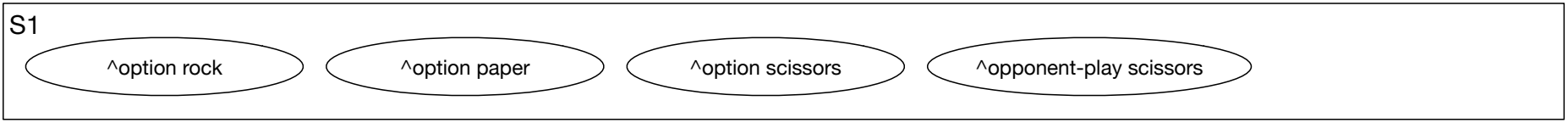
Learning Operator Selection Knowledge (OSK)

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

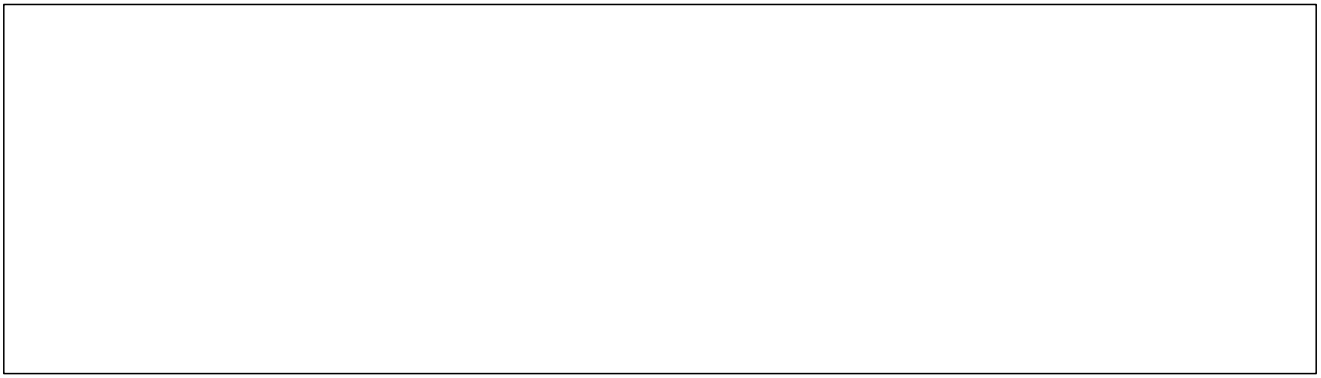
Learning Operator Selection Knowledge (OSK)

- Knowledge in OSK rules are different than other rules:
  - Does not directly lead to removal or creation of WMEs
  - Not clear whether a rule firing plays a role in an operator being selected.
  - Dependent on:
    - Other OSK rules that also fired
    - Soar's operator preference resolution logic

Working Memory



Rule Instantiations



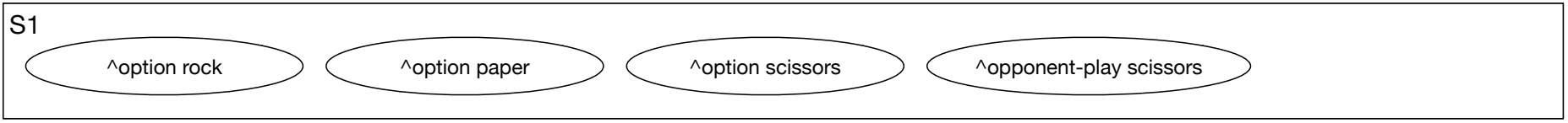
Chunk Formed

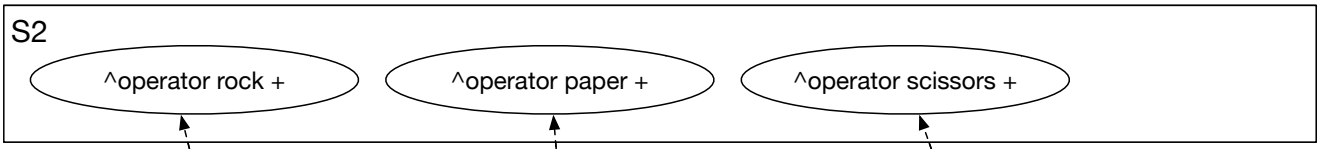
```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

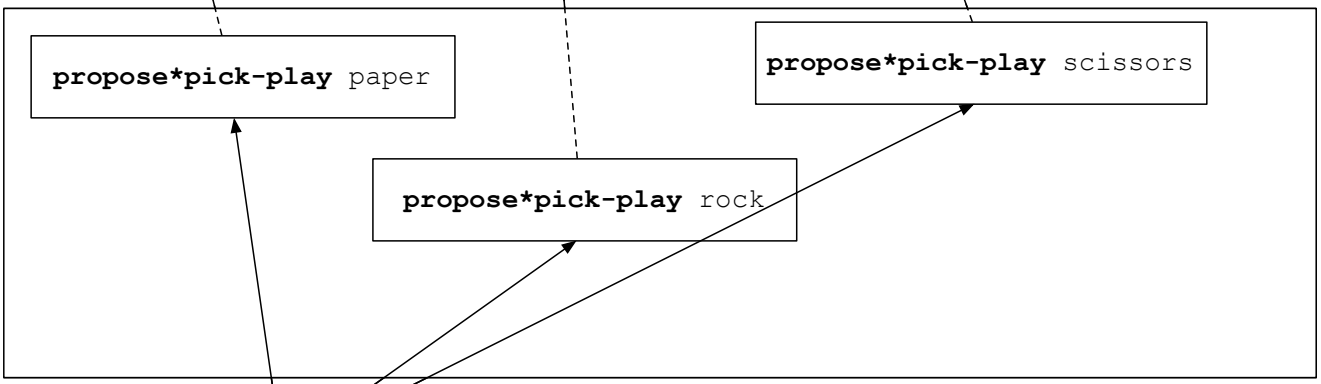
```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory





Rule Instantiations



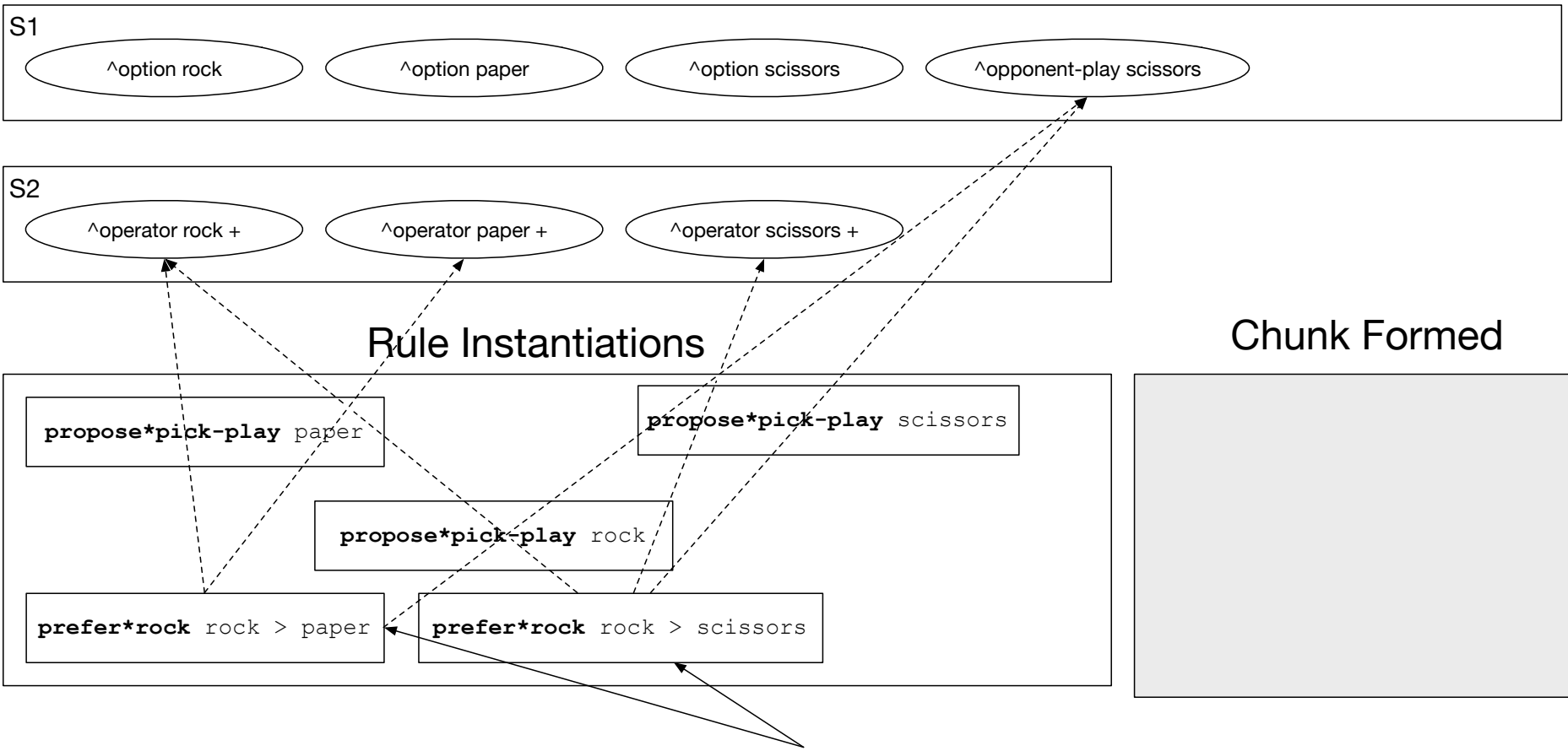
Chunk Formed

```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory

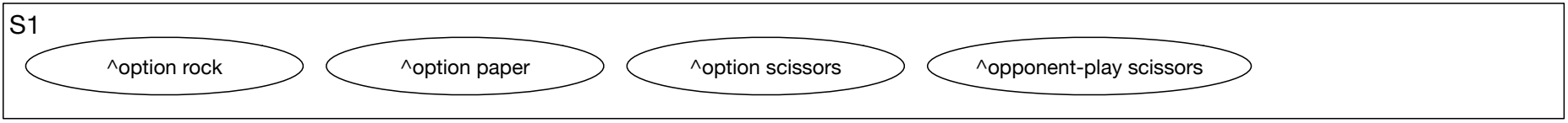


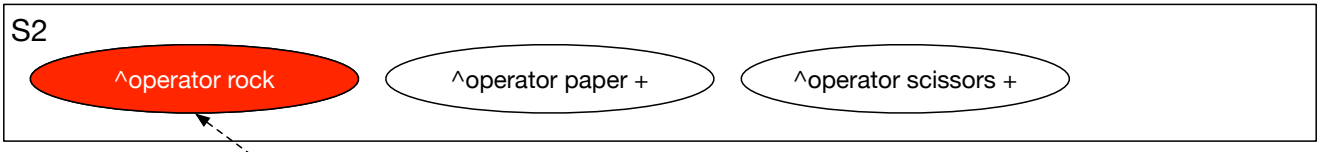

```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

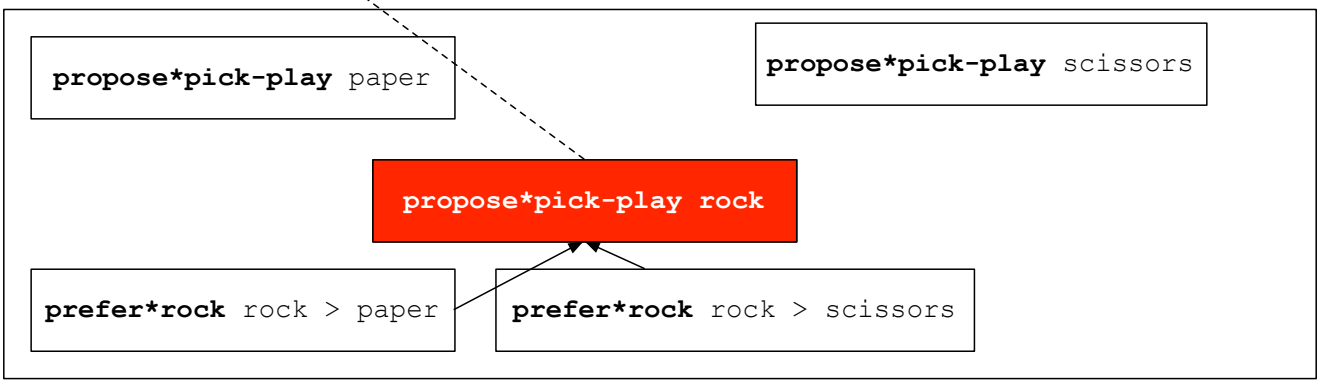
```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory





Rule Instantiations

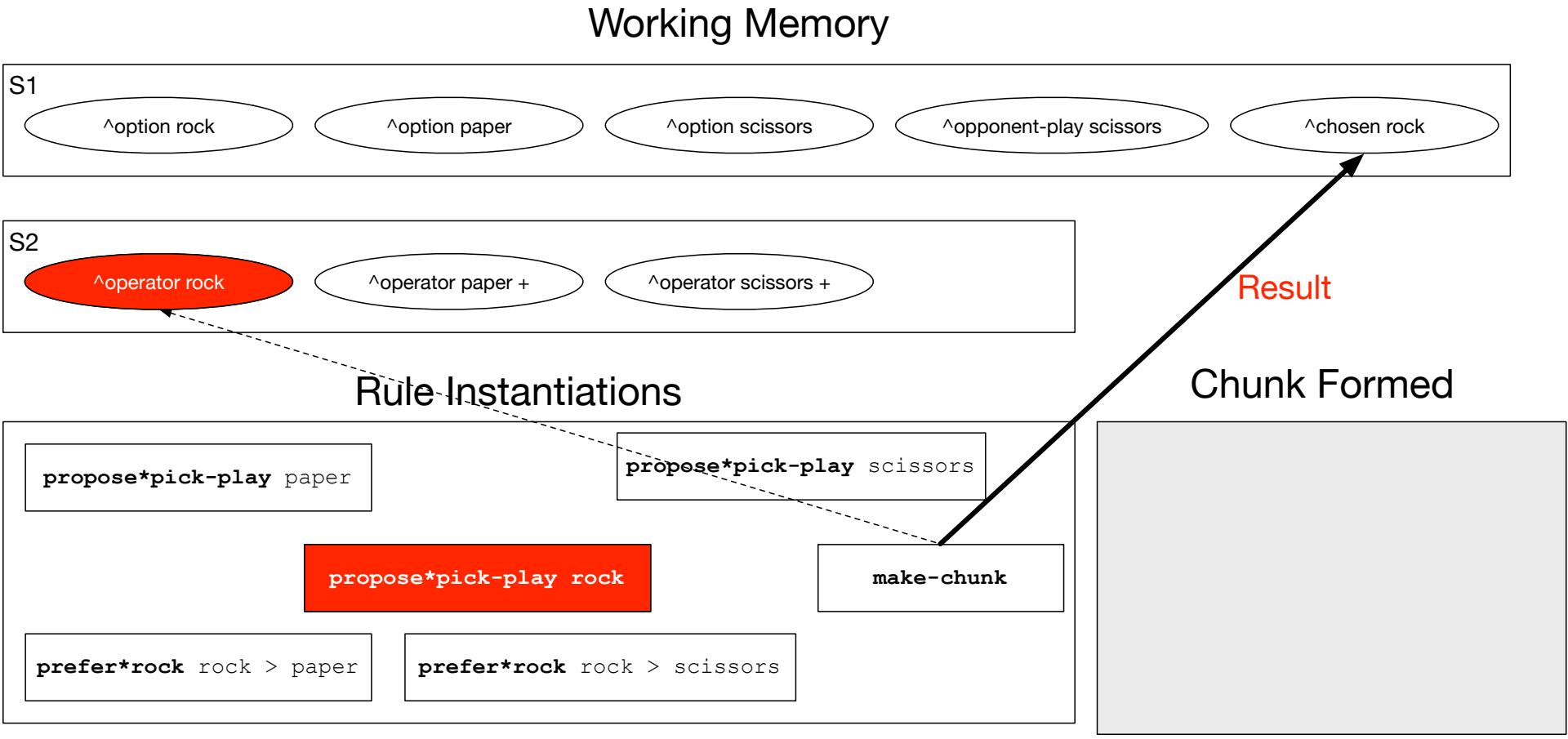


Chunk Formed

```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

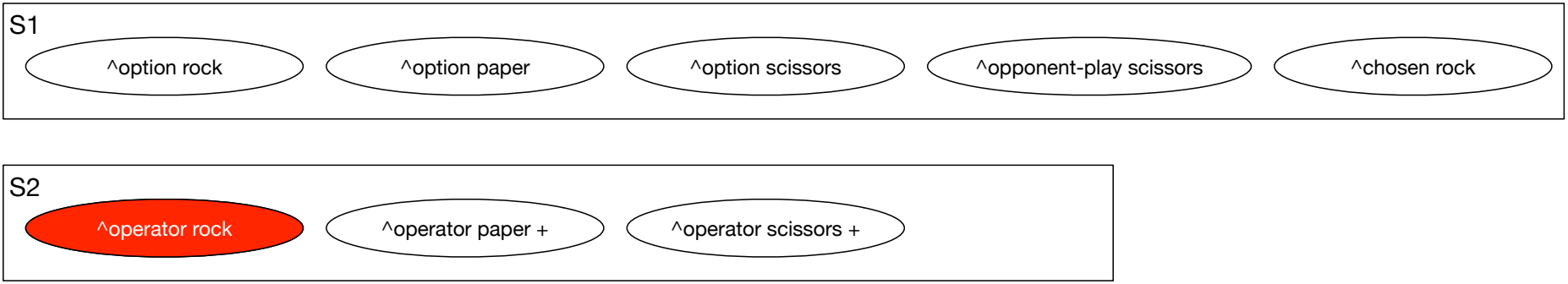



```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory



Rule Instantiations

propose*pick-play paper

propose*pick-play scissors

propose*pick-play rock

make-chunk

prefer*rock rock > paper

prefer*rock rock > scissors

Chunk Formed

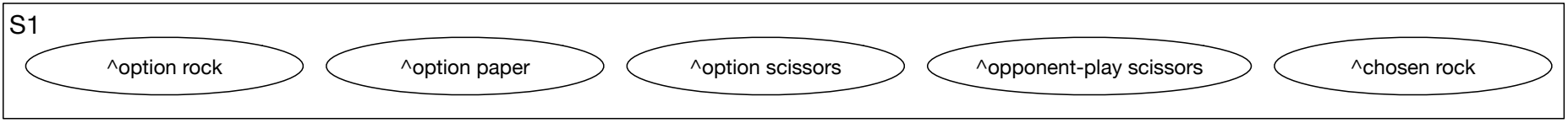
```
sp {chunk*make-chunk*snc*t2-1
    (S1 ^superstate nil)
    (S1 ^option rock)
    -->
    (S1 ^chosen rock +)
}
```

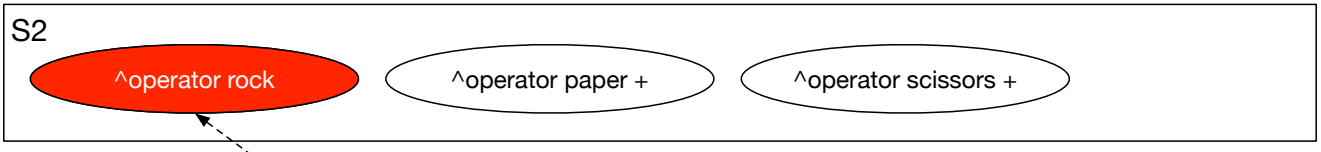
```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

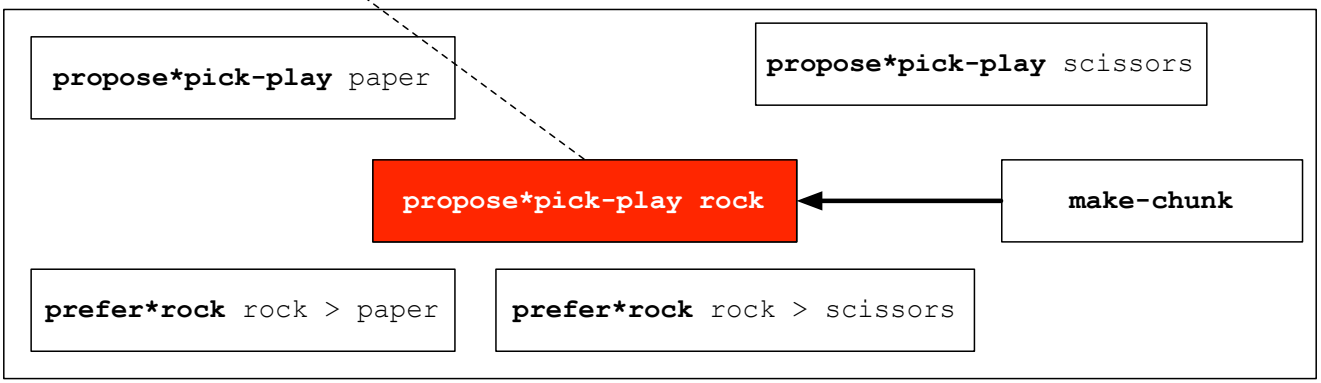
```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory





Rule Instantiations



Chunk Formed

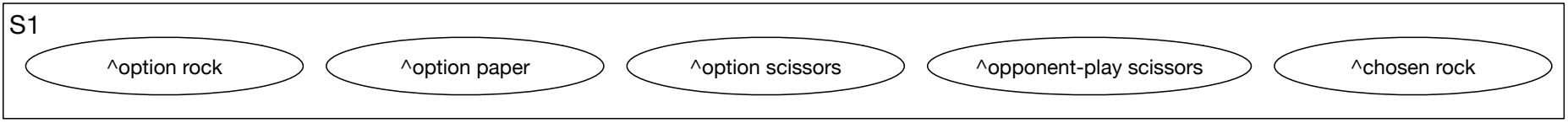
```
sp {chunk*make-chunk*snc*t2-1
    (S1 ^superstate nil)
    (S1 ^option rock)
    -->
    (S1 ^chosen rock +)
}
```

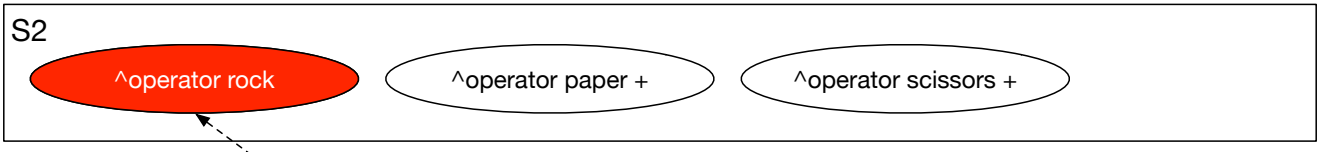
```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

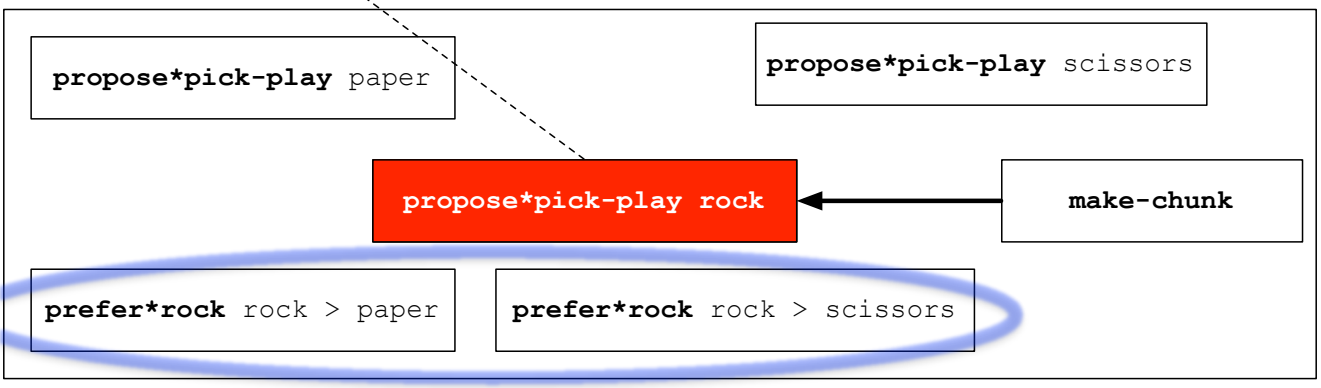
```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory





Rule Instantiations



Chunk Formed

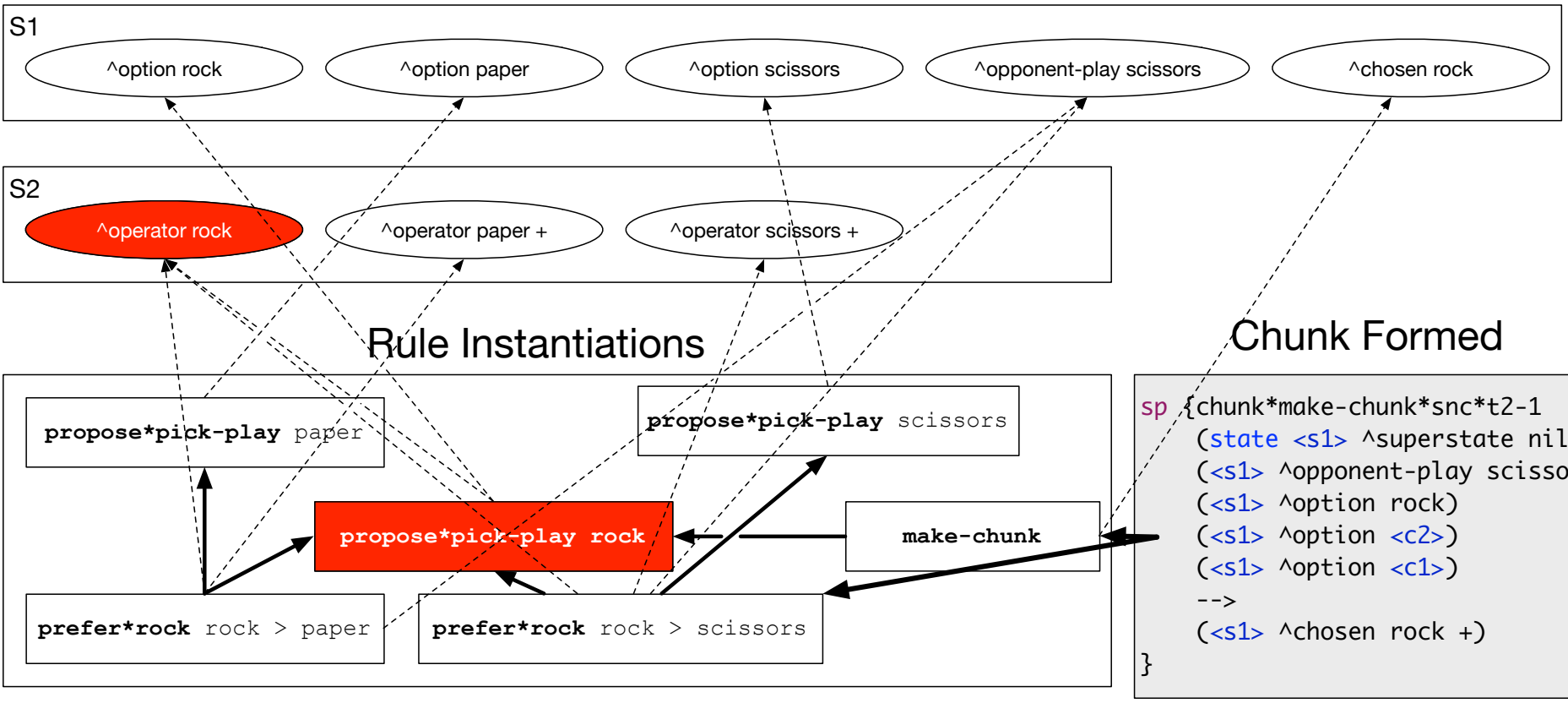
```
sp {chunk*make-chunk*snc*t2-1
    (S1 ^superstate nil)
    (S1 ^option rock)
    -->
    (S1 ^chosen rock +)
}
```

```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

Working Memory

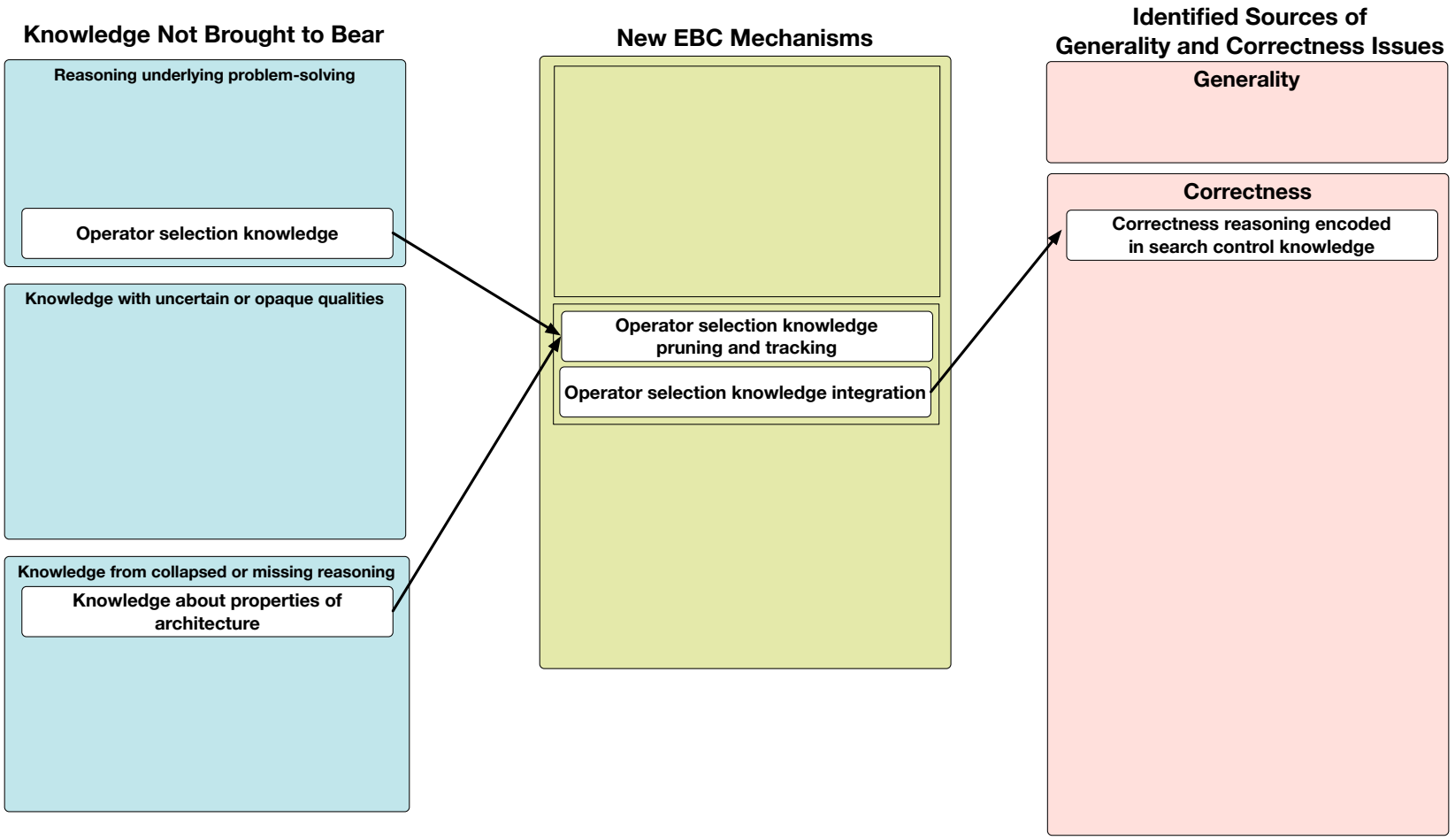



```
sp {propose*pick-play
    (state <s> ^superstate <ss>)
    (<ss> ^superstate nil
          ^option <play>)
-->
    (<s> ^operator <o> +)
    (<o> ^name <play>) }
```

```
sp {prefer*rock
    (state <s> ^operator <o> +
               ^operator {<> <o> <o2>} +
               ^superstate <ss>)
    (<o> ^name rock)
    (<ss> ^opponent-play scissors)
-->
    (<s> ^operator <o> > <o2>) }
```

```
sp {make-chunk
    (state <s> ^superstate <ss>
               ^operator <o>)
    (<o> ^name <play>)
-->
    (<ss> ^chosen <play>) }
```

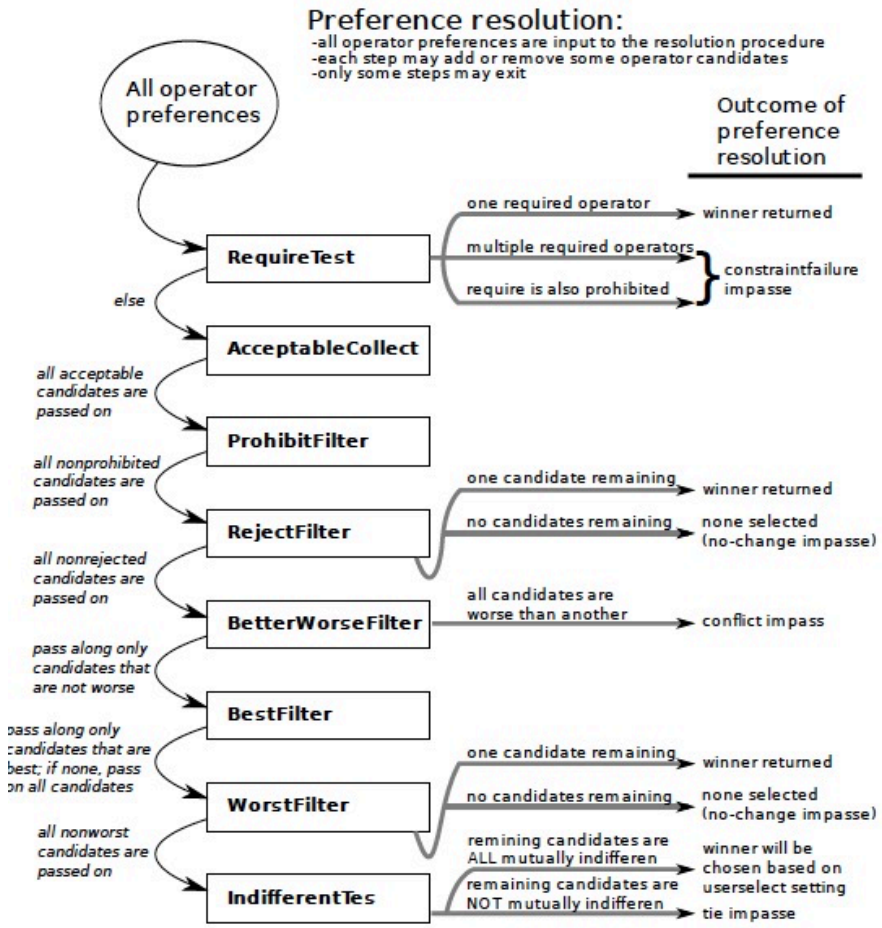
Utilizing
Operator Selection Knowledge



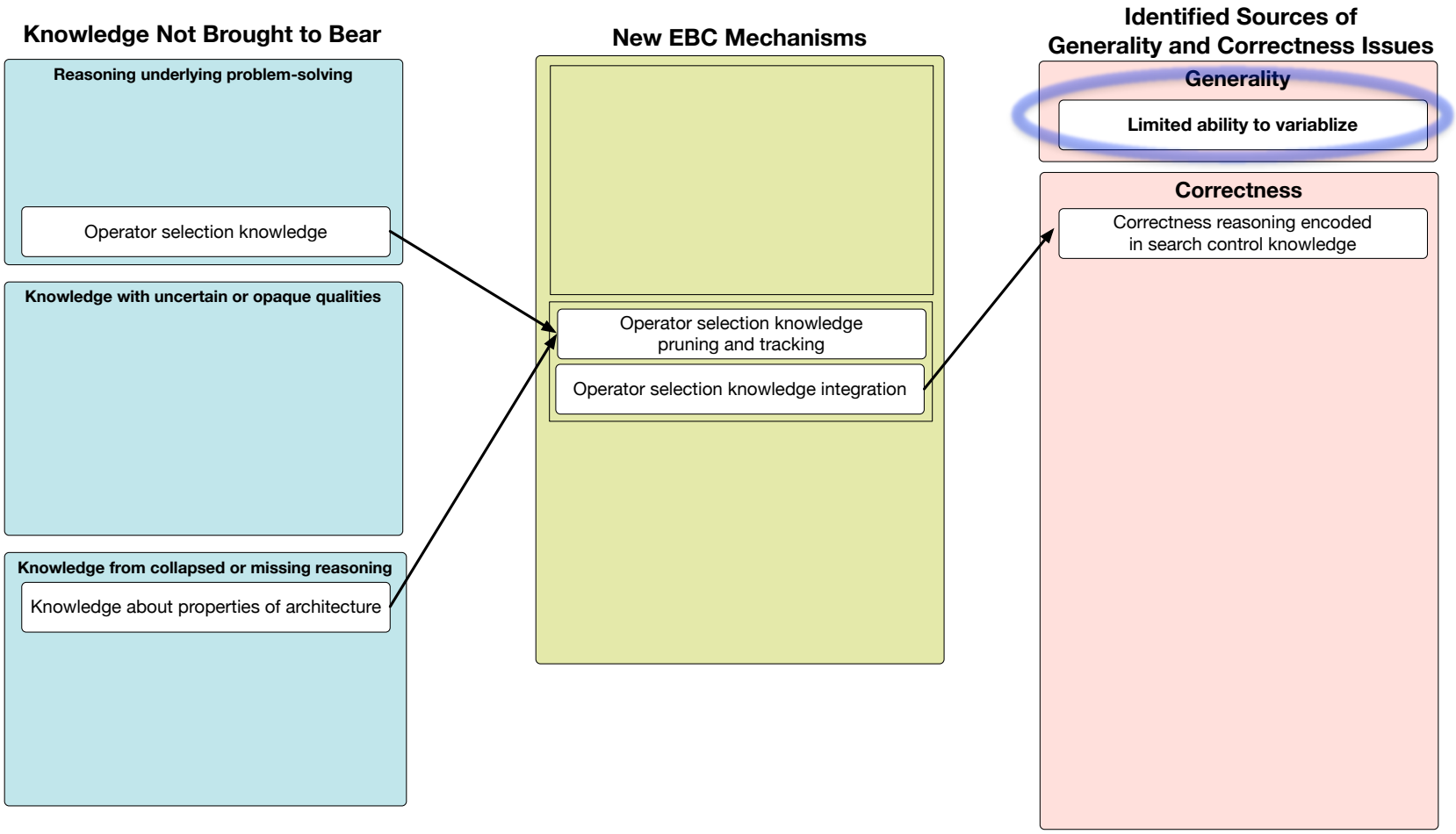
Utilizing Operator Selection Knowledge

- When new operator gets selected in substate:
  - Cache operator preference knowledge
  - Prune out irrelevant preferences
- When a chunk is created
  - Determines all operators tested in trace
  - Backtraces through rules that created the OSK preferences that was cached for those operators
    - This creates additional conditions in chunk

Architectural Knowledge Needed



Limited Ability to Variablize



Water-Jug Chunk in Soar 9.4

```
sp {chunk*9.4.0
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name fill)
    (<o1> ^fill-jug <f1>)
    (<f1> ^filled-jug yes)
    (<f1> ^picked-up yes)
    (<f1> ^volume 5)
    (<f1> ^contents 3)
    -->
    (<f1> ^picked-up yes -)
    (<f1> ^filled-jug yes -)
    (<f1> ^contents 5 +)
    (<f1> ^contents 3 -)}
```

Water-Jug Chunk in Soar 9.4

```
sp {chunk*9.4.0
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name fill)
    (<o1> ^fill-jug <f1>)
    (<f1> ^filled-jug yes)
    (<f1> ^picked-up yes)
    (<f1> ^volume 5)
    (<f1> ^contents 3)
    -->
    (<f1> ^picked-up yes -)
    (<f1> ^filled-jug yes -)
    (<f1> ^contents 5 +)
    (<f1> ^contents 3 -)}
```

Chunk Comparison

```
sp {chunk*9.4.0
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name fill)
    (<o1> ^fill-jug <f1>)
    (<f1> ^filled-jug yes)
    (<f1> ^picked-up yes)
    (<f1> ^volume 5)
    (<f1> ^contents 3)
    -->
    (<f1> ^picked-up yes -)
    (<f1> ^filled-jug yes -)
    (<f1> ^contents 5 +)
    (<f1> ^contents 3 -)
    (<f1> ^rhs 8 +)}
```

Chunk learned in Soar 9.4.0

```
sp {chunk*9.5
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name <c3>)
    (<o1> ^fill-jug <i1>)
    (<i1> ^filled-jug yes)
    (<i1> ^picked-up yes)
    (<i1> ^volume {> <c2> <c1>})
    (<i1> ^contents <c2>)
    -->
    (<i1> ^picked-up yes -)
    (<i1> ^filled-jug yes -)
    (<i1> ^contents <c1> +)
    (<i1> ^contents <c2> -)
    (<f1> ^rhs (+ <c1> <c2>) +)}
```

What we want

Chunk Comparison

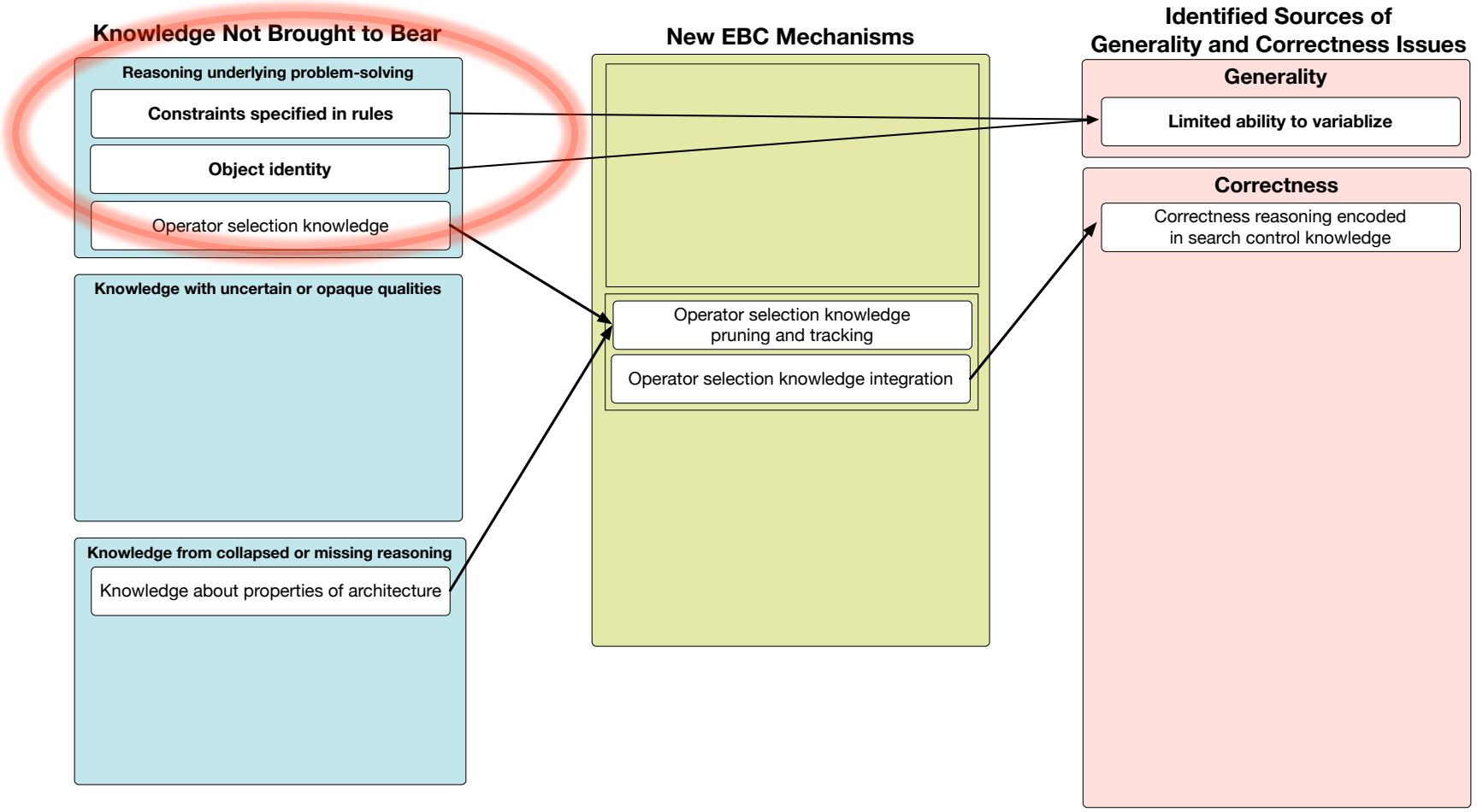
```
sp {chunk*9.4.0
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name fill)
    (<o1> ^fill-jug <f1>)
    (<f1> ^filled-jug yes)
    (<f1> ^picked-up yes)
    (<f1> ^volume 5)
    (<f1> ^contents 3)
    -->
    (<f1> ^picked-up yes -)
    (<f1> ^filled-jug yes -)
    (<f1> ^contents 5 +)
    (<f1> ^contents 3 -)
    (<f1> ^rhs 8 +)}
```

Chunk learned in Soar 9.4.0

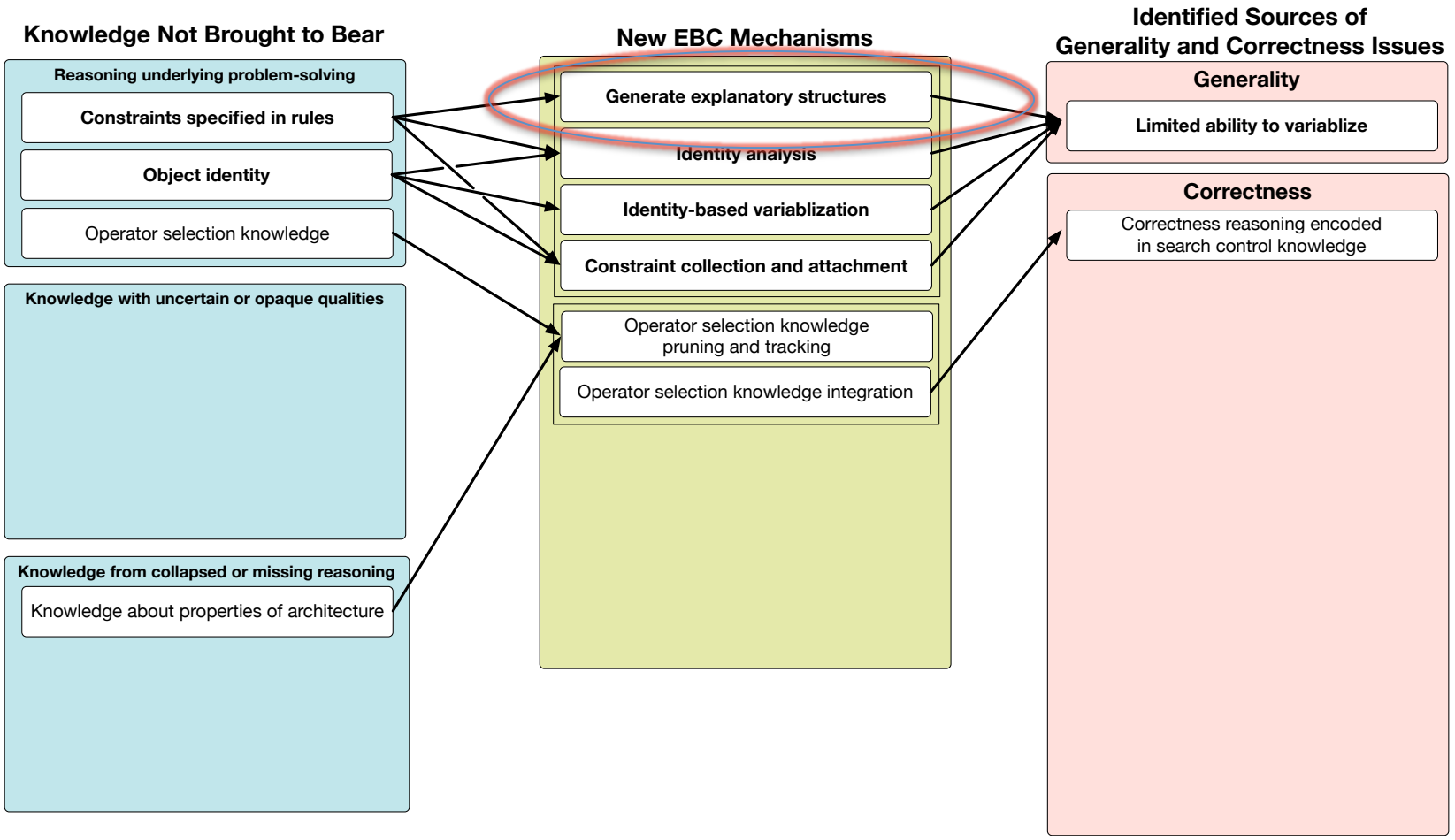
```
sp {chunk*9.5
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name <c3>)
    (<o1> ^fill-jug <i1>)
    (<i1> ^filled-jug yes)
    (<i1> ^picked-up yes)
    (<i1> ^volume {> <c2> <c1>})
    (<i1> ^contents <c2>)
    -->
    (<i1> ^picked-up yes -)
    (<i1> ^filled-jug yes -)
    (<i1> ^contents <c1> +)
    (<i1> ^contents <c2> -)
    (<f1> ^rhs (+ <c1> <c2>) +)}
```

What we want

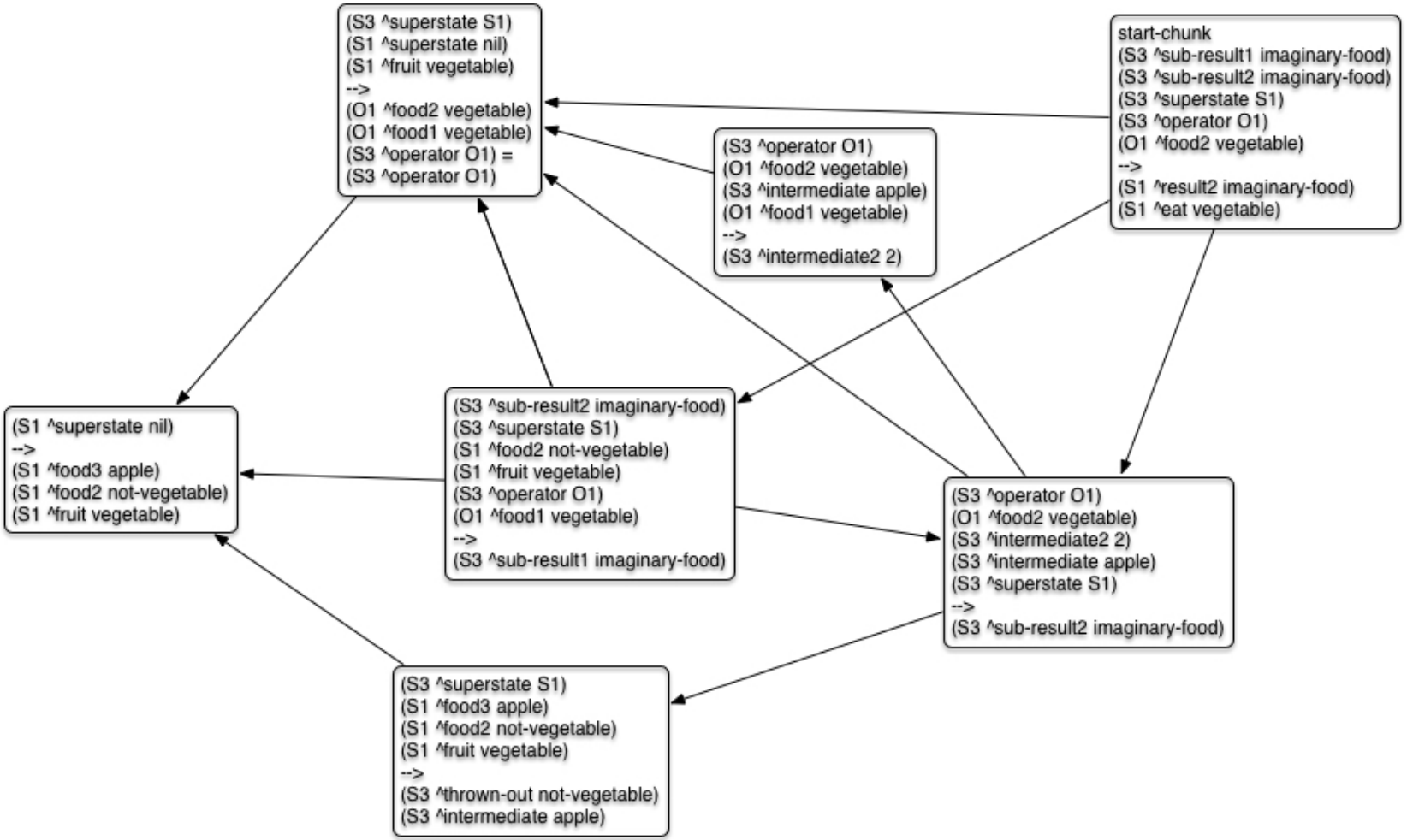
Knowledge Needed



More General Variablization

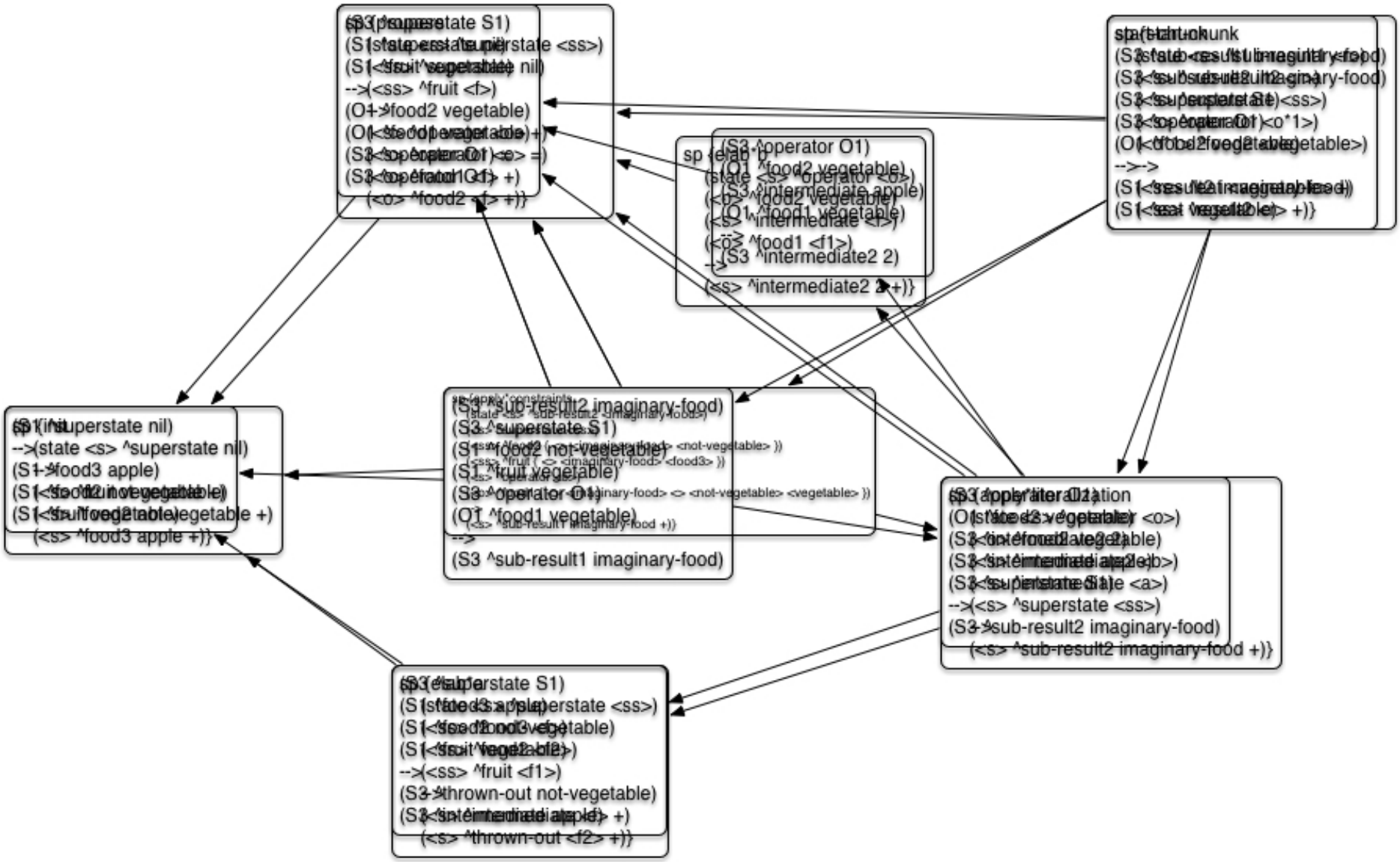


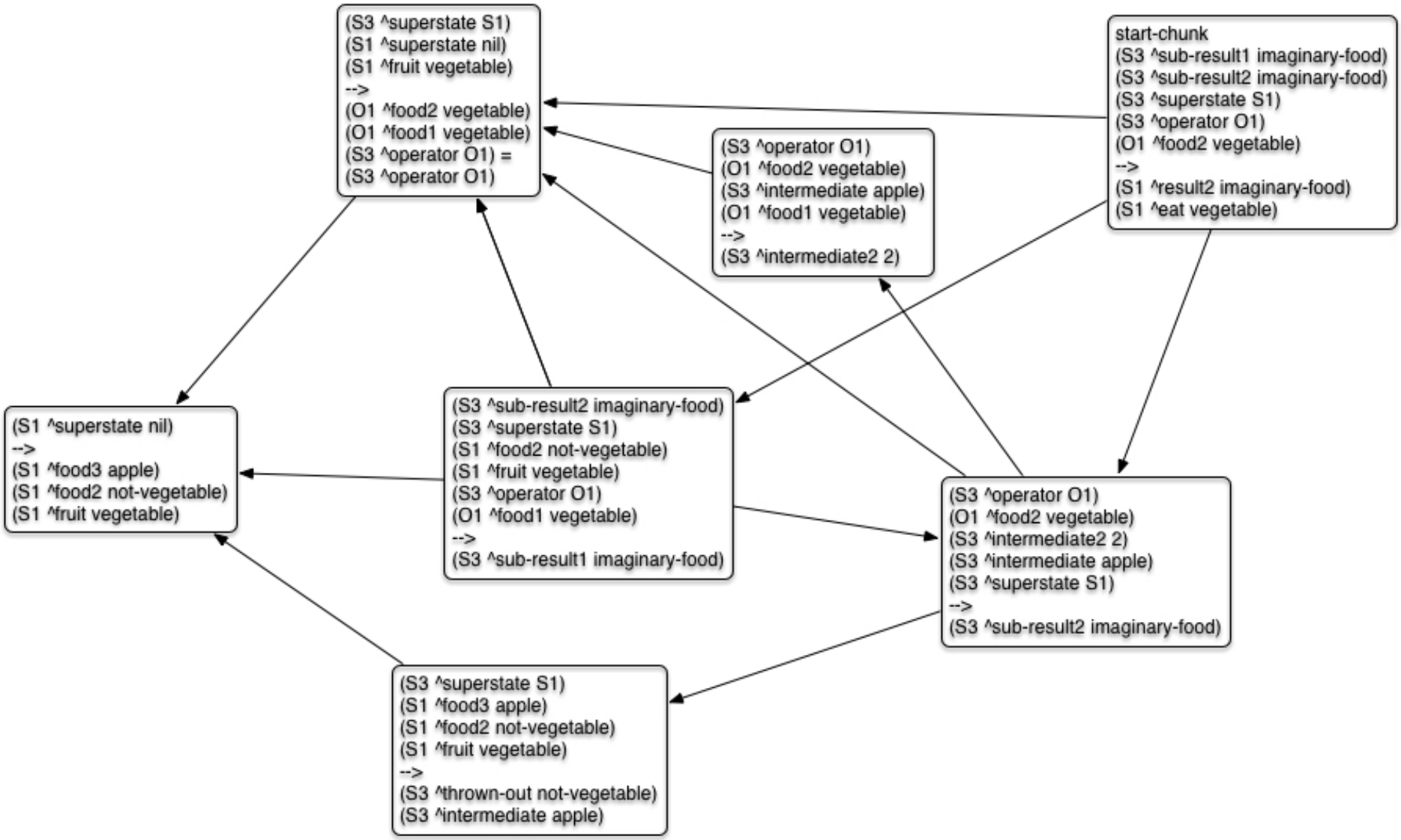
- Chunking learned all its knowledge purely by analyzing the *working memory trace.*

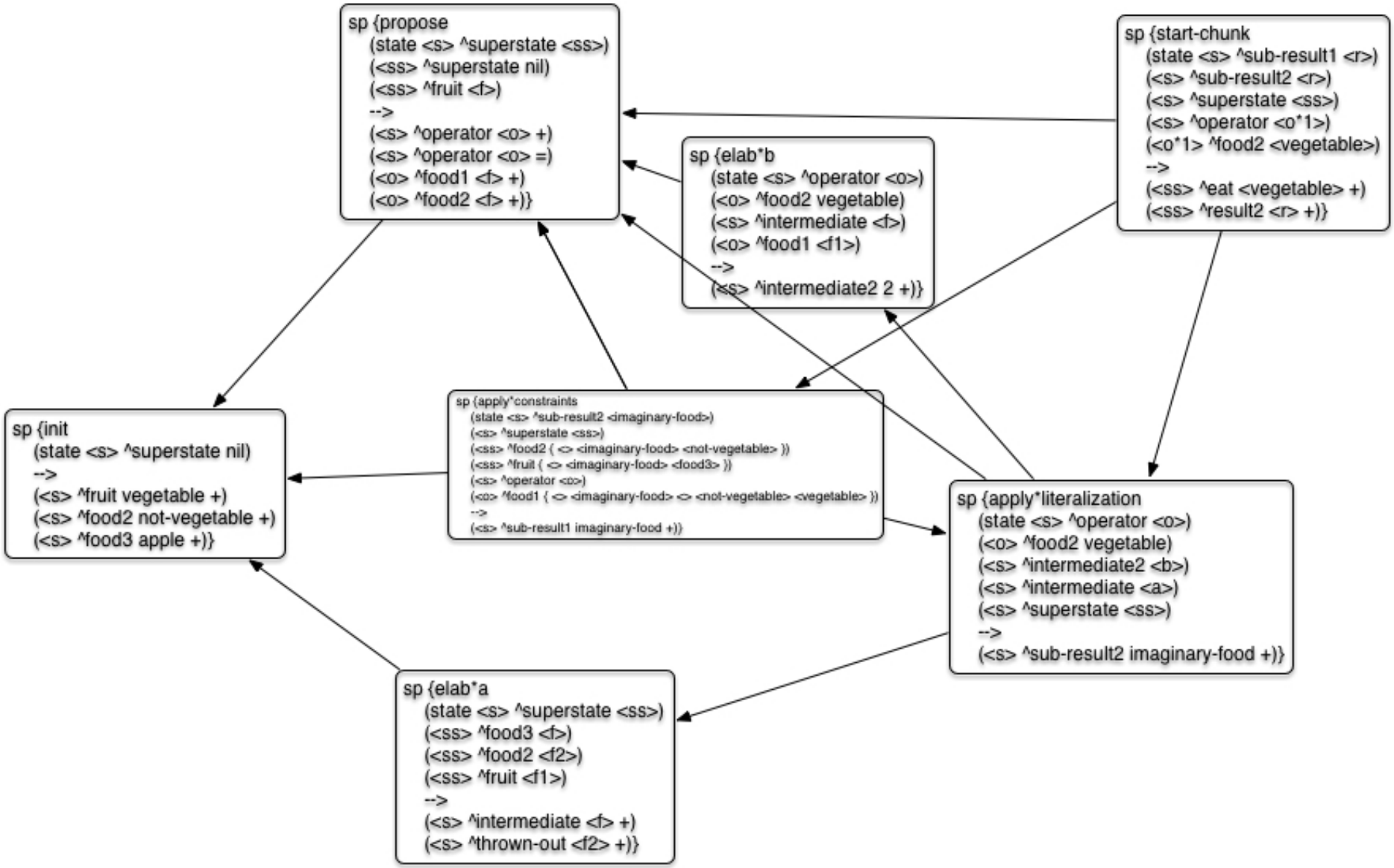


How does EBC differ from chunking?

- EBC learns more general knowledge by also analyzing the **explanation trace**
  - Original rules written by human knowledge expert are superimposed over the WME trace to create an "explanation trace"




(S3 ^superstate S1)
(S1 ^superstate nil)
(S1 ^fruit vegetable)
-->
(O1 ^food2 vegetable)
(O1 ^food1 vegetable)
(S3 ^operator O1) =
(S3 ^operator O1)
start-chunk
(S3 ^sub-result1 imaginary-food)
(S3 ^sub-result2 imaginary-food)
(S3 ^superstate S1)
(S3 ^operator O1)
(O1 ^food2 vegetable)
-->
(S1 ^result2 imaginary-food)
(S1 ^eat vegetable)
(S3 ^operator O1)
(O1 ^food2 vegetable)
(S3 ^intermediate apple)
(O1 ^food1 vegetable)
-->
(S3 ^intermediate2 2)
(S1 ^superstate nil)
-->
(S1 ^food3 apple)
(S1 ^food2 not-vegetable)
(S1 ^fruit vegetable)
(S3 ^sub-result2 imaginary-food)
(S3 ^superstate S1)
(S1 ^food2 not-vegetable)
(S1 ^fruit vegetable)
(S3 ^operator O1)
(O1 ^food1 vegetable)
-->
(S3 ^sub-result1 imaginary-food)
(S3 ^operator O1)
(O1 ^food2 vegetable)
(S3 ^intermediate2 2)
(S3 ^intermediate apple)
(S3 ^superstate S1)
-->
(S3 ^sub-result2 imaginary-food)
(S3 ^superstate S1)
(S1 ^food3 apple)
(S1 ^food2 not-vegetable)
(S1 ^fruit vegetable)
-->
(S3 ^thrown-out not-vegetable)
(S3 ^intermediate apple)


sp {propose
(state <s> ^superstate <ss>)
(<ss> ^superstate nil)
(<ss> ^fruit <f>)
-->
(<s> ^operator <o> +)
(<s> ^operator <o> =)
(<o> ^food1 <f> +)
(<o> ^food2 <f> +)}
sp {start-chunk
(state <s> ^sub-result1 <r>)
(<s> ^sub-result2 <r>)
(<s> ^superstate <ss>)
(<s> ^operator <o*1>)
(<o*1> ^food2 <vegetable>)
-->
(<ss> ^eat <vegetable> +)
(<ss> ^result2 <r> +)}
sp {elab*b
(state <s> ^operator <o>)
(<o> ^food2 vegetable)
(<s> ^intermediate <f>)
(<o> ^food1 <f1>)
-->
(<s> ^intermediate2 2 +)}
sp {init
(state <s> ^superstate nil)
-->
(<s> ^fruit vegetable +)
(<s> ^food2 not-vegetable +)
(<s> ^food3 apple +)}
sp {apply*constraints
(state <s> ^sub-result2 <imaginary-food>)
(<s> ^superstate <ss>)
(<ss> ^food2 { <> <imaginary-food> <not-vegetable> })
(<ss> ^fruit { <> <imaginary-food> <food3> })
(<s> ^operator <o>)
(<o> ^food1 { <> <imaginary-food> <> <not-vegetable> <vegetable> })
-->
(<s> ^sub-result1 imaginary-food +)}
sp {apply*literalization
(state <s> ^operator <o>)
(<o> ^food2 vegetable)
(<s> ^intermediate2 <b>)
(<s> ^intermediate <a>)
(<s> ^superstate <ss>)
-->
(<s> ^sub-result2 imaginary-food +)}
sp {elab*a
(state <s> ^superstate <ss>)
(<ss> ^food3 <f>)
(<ss> ^food2 <f2>)
(<ss> ^fruit <f1>)
-->
(<s> ^intermediate <f> +)
(<s> ^thrown-out <f2> +)}

Generating Hybrid Explanatory Structure

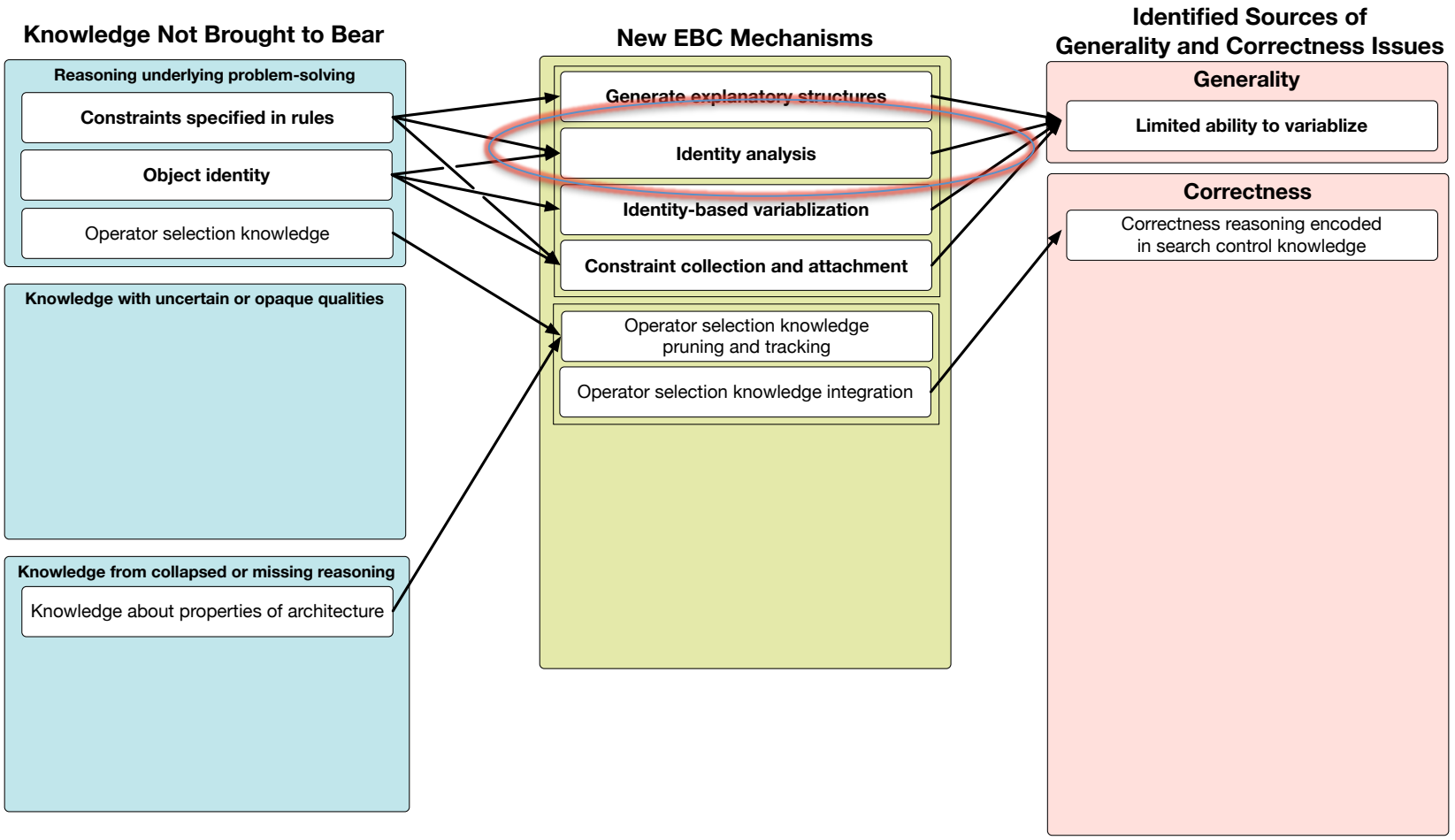
- Working memory trace augmented
  - Any additional constraints in original rules are added
  - All items that matched a variable are assigned a unique variablization ID

Rule condition	**Matched WME**
(<s> ^item <other>)	(S3 ^item food)
(<s> ^foo { <> <other> <bar>})	(S3 ^foo bar)

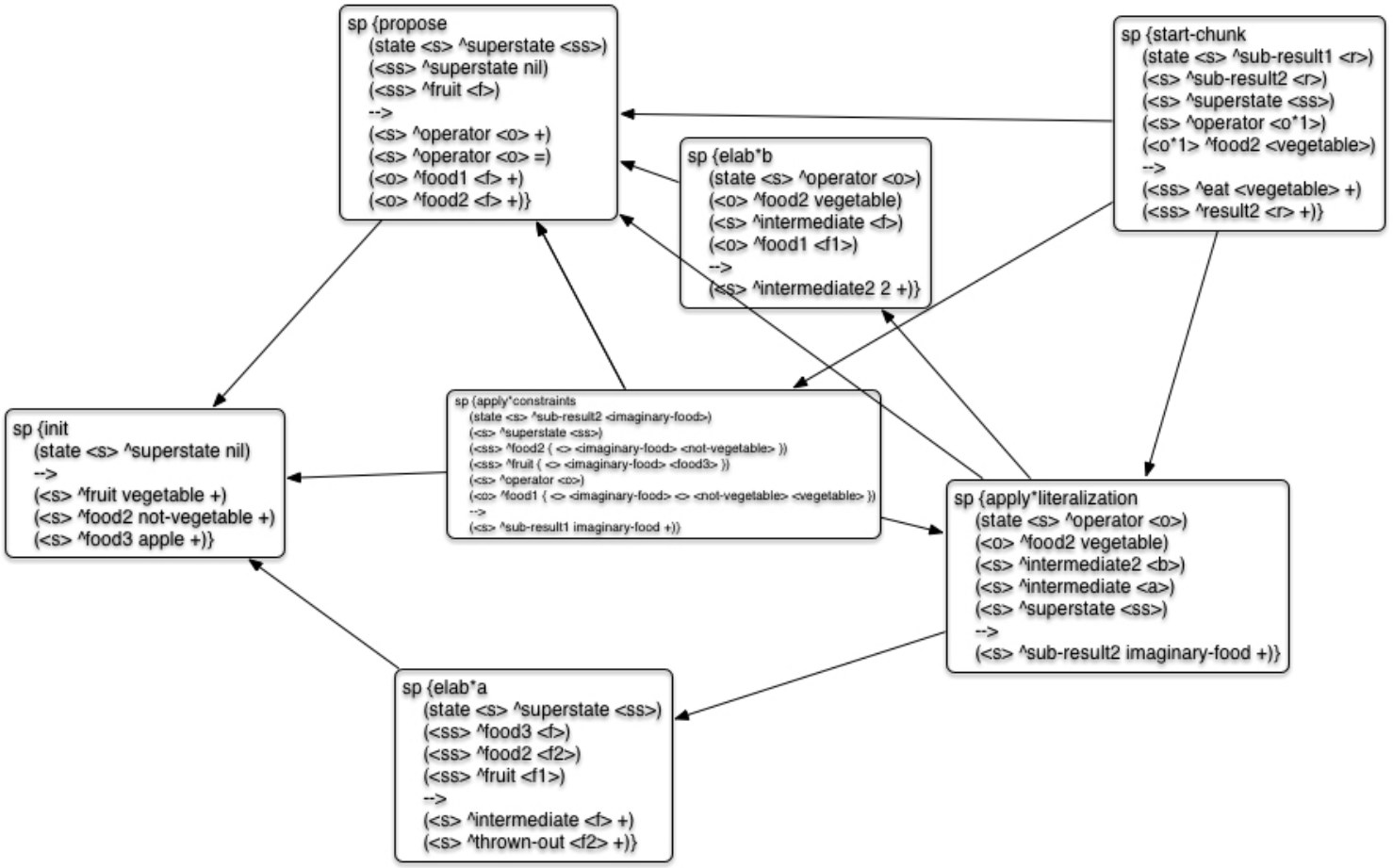
becomes…

```
(S3:1 ^item food:2)
(S3:1 ^foo { <> food:2 bar:3} )
```

More General Variablization

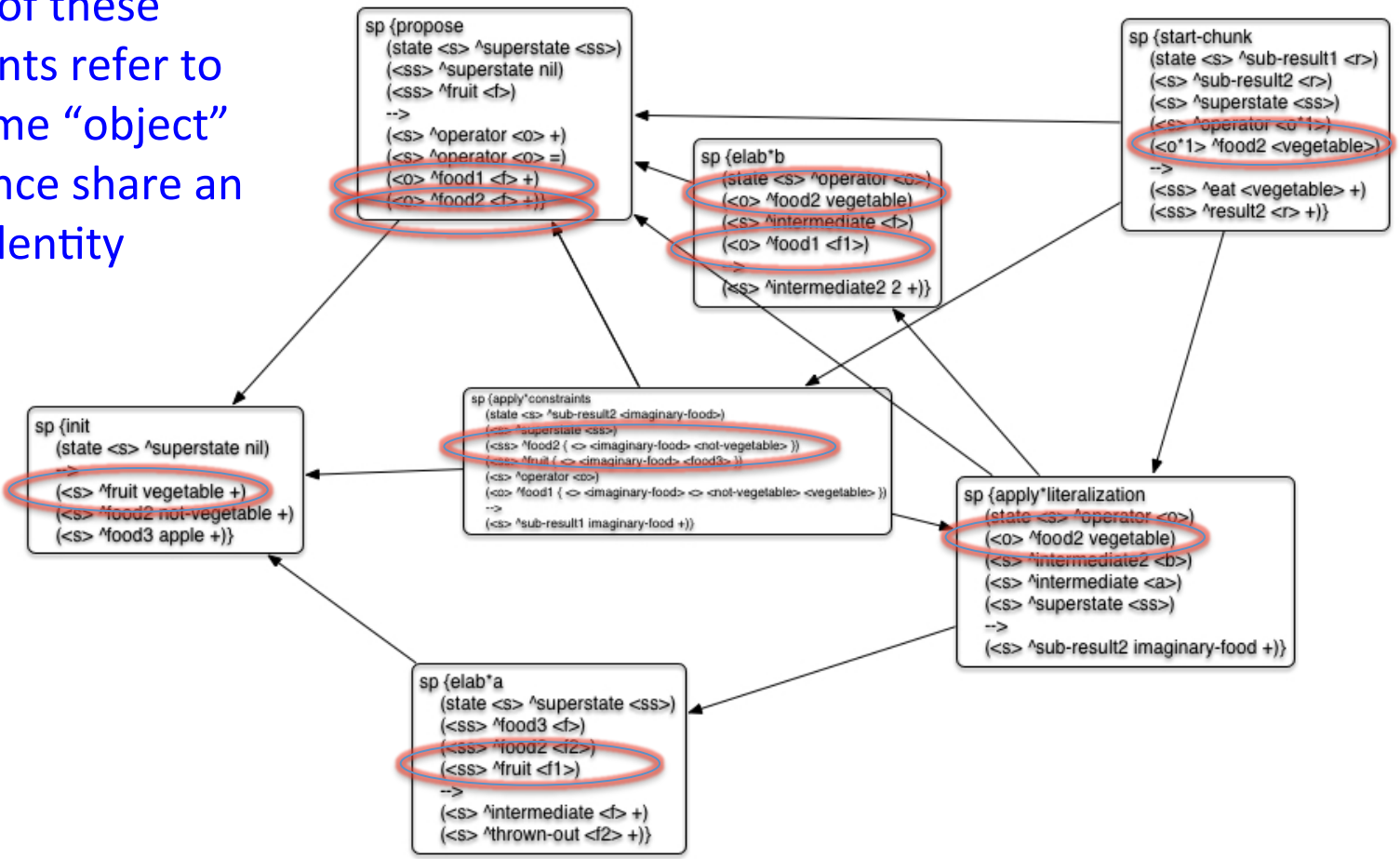


- During backtracing, EBC analyzes how conditions in rules are connected to the actions in other rules



- During backtracing, EBC analyzes how conditions in rules are connected to the actions in other rules

All of these elements refer to the same "object" and hence share an identity



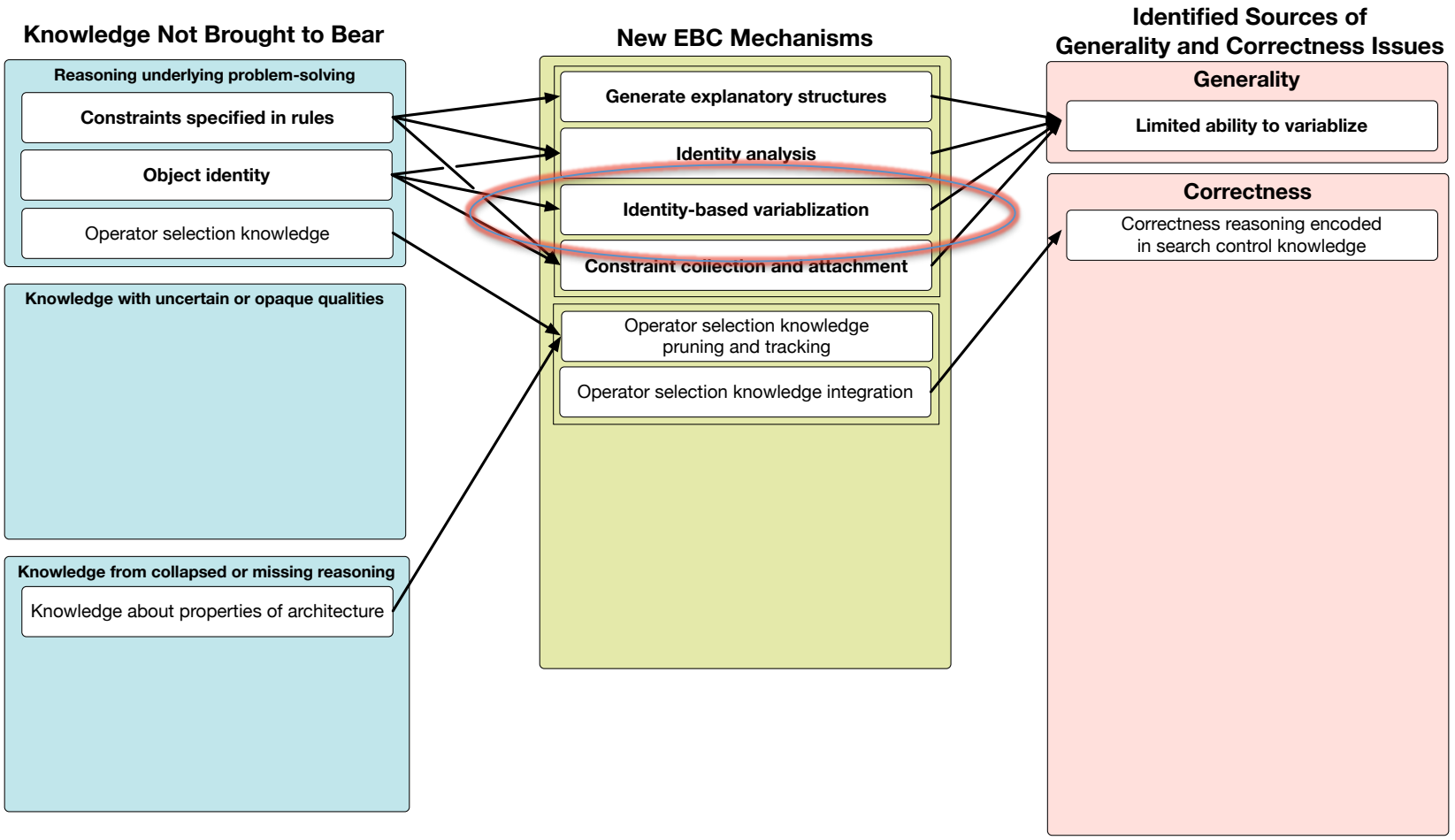
Identity Analysis

- During backtracing:
  - Using special rules, it determines all variablization IDs that share the same "identity"
    - More general form of EGG algorithm [Mooney 86] that maps identities to "identity sets" rather than variables to other variables in the trace
    - Substitution rules extended to accommodate other EBC mechanisms
  - Many chunking mechanisms perform their operations based on identity sets now

Identity Propagation Rules

- - Match is not STI, parent condition has identity, and RHS action has identity that appears on LHS
- - Add mapping from RHS identity to IUS that parent condition is in
- - Add mapping from source identity to target identity unification set
- - Determine target mapping
- - If target IUS is null identity set
- - Target IUS is null identity set
- - Else
- - If mapping from target identity to IUS already exists
- - Target IUS is existing IUS mapping
- - Else
- - Just use passed in target IUS
- - If there is already a mapping for source identity
- - If source identity is already member of null identity set
- - If target IUS is not null identity set
- - Update all mappings that point to target IUS to point to null identity set
- - Else nothing b/c both are members of null identity set
- - Else (already mapped to an IUS)
- - If target IUS is null identity set
- - Update all existing mappings that point to existing mapping to point to null identity set
- - Add a mapping from source identity to null identity set
- - Else
- - Add a mapping from source identity to existing mapping
- - Add a mapping from target identity to existing mapping
- - Update all existing mappings that point to source mapping to point to existing mapping
- - Update all existing mappings that point to target mapping to point to existing mapping
- - Else
- - Add a mapping from source identity to target IUS determine in first step

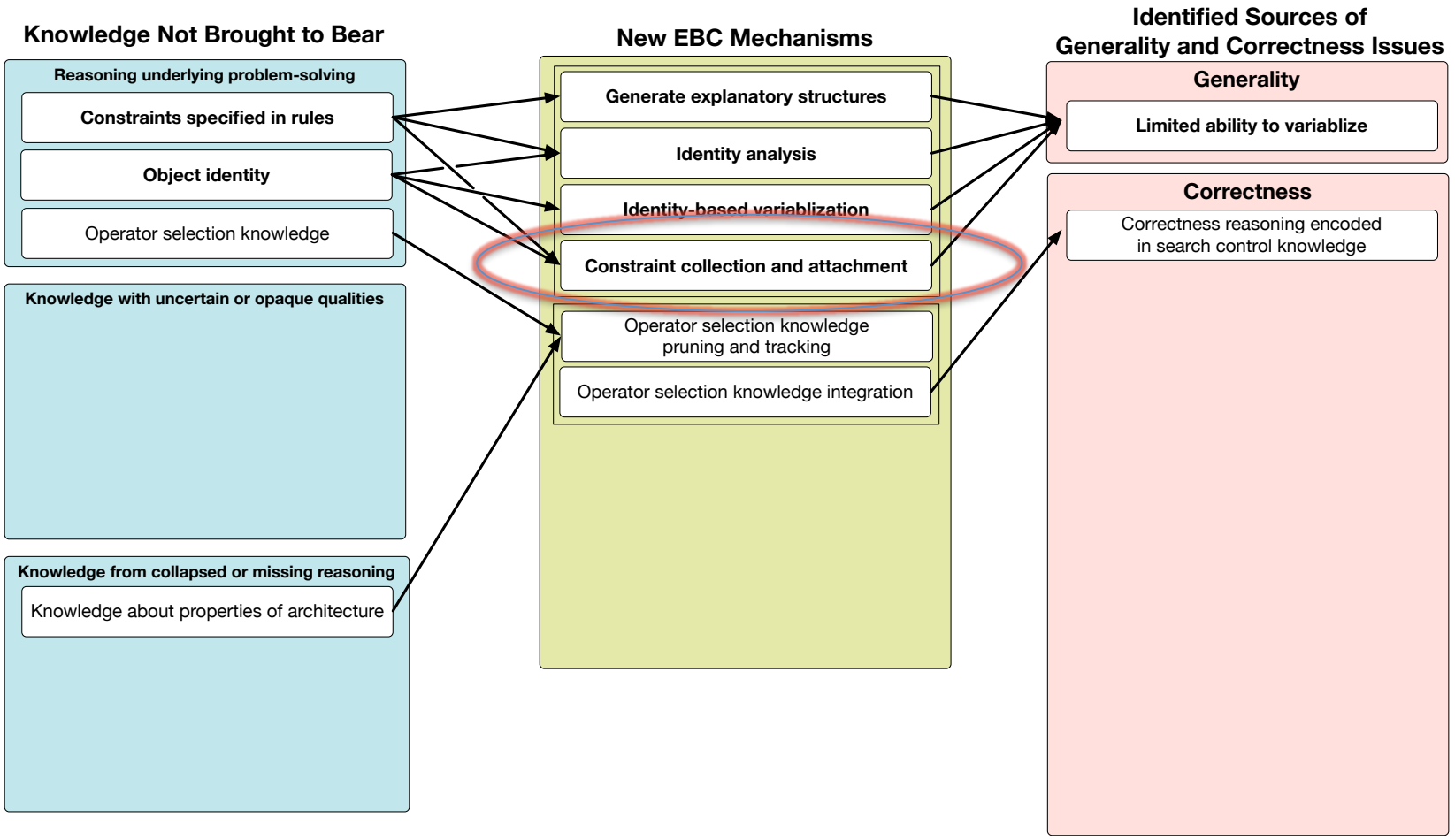
More General Variablization



Identity-Based Variablization

- If an item has no identity value,
  - Keep literal match value
  - Throw out additional constraints
- If an item has an identity value,
  - Look up identity set
  - Replace match value with variable assigned to identity set

More General Variablization

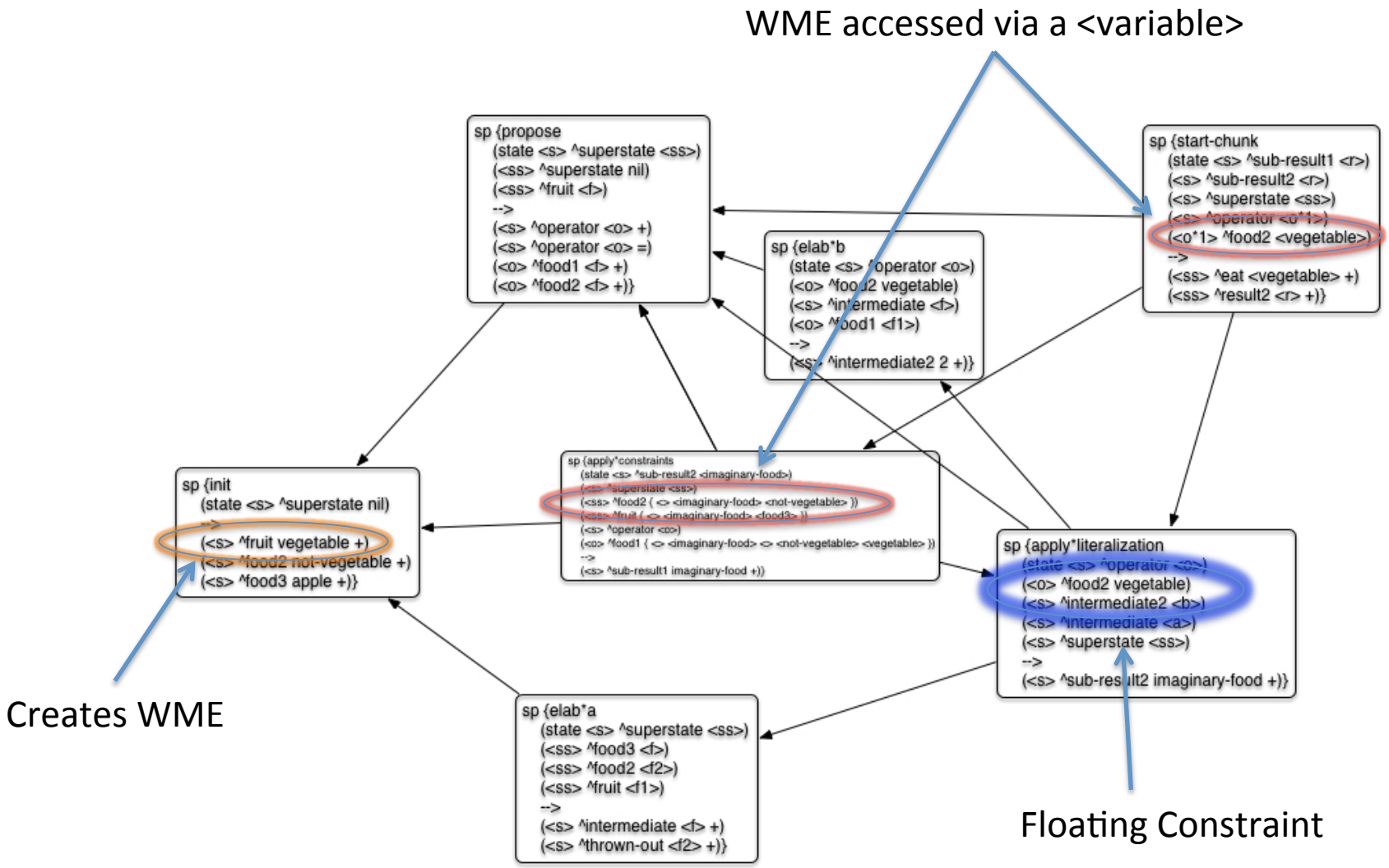


Floating Constraints

- Problem occurs when
  - A rule copies a value from a superstate WME to a substate WME
  - Another substate rules tests that local WME and *places and constraint on its value*
  - ***That constraint applies transitively to original identity***

- Condition with new constraint only test a local WME
  - That means the condition will not appear in the chunk
  - *So a necessary constraint will be lost and rule will be incorrect*

Floating Constraint Example



Constraint Collection and Attachment

- When backtracing:
  - Keeps track of all constraints in the backtrace and the variablization identity that they constrain.
- Before variablization:
  - If identity of constraint exists in chunk and constraint is not already in chunk,
    - Finds another test with the same identity and attaches the constraint to it

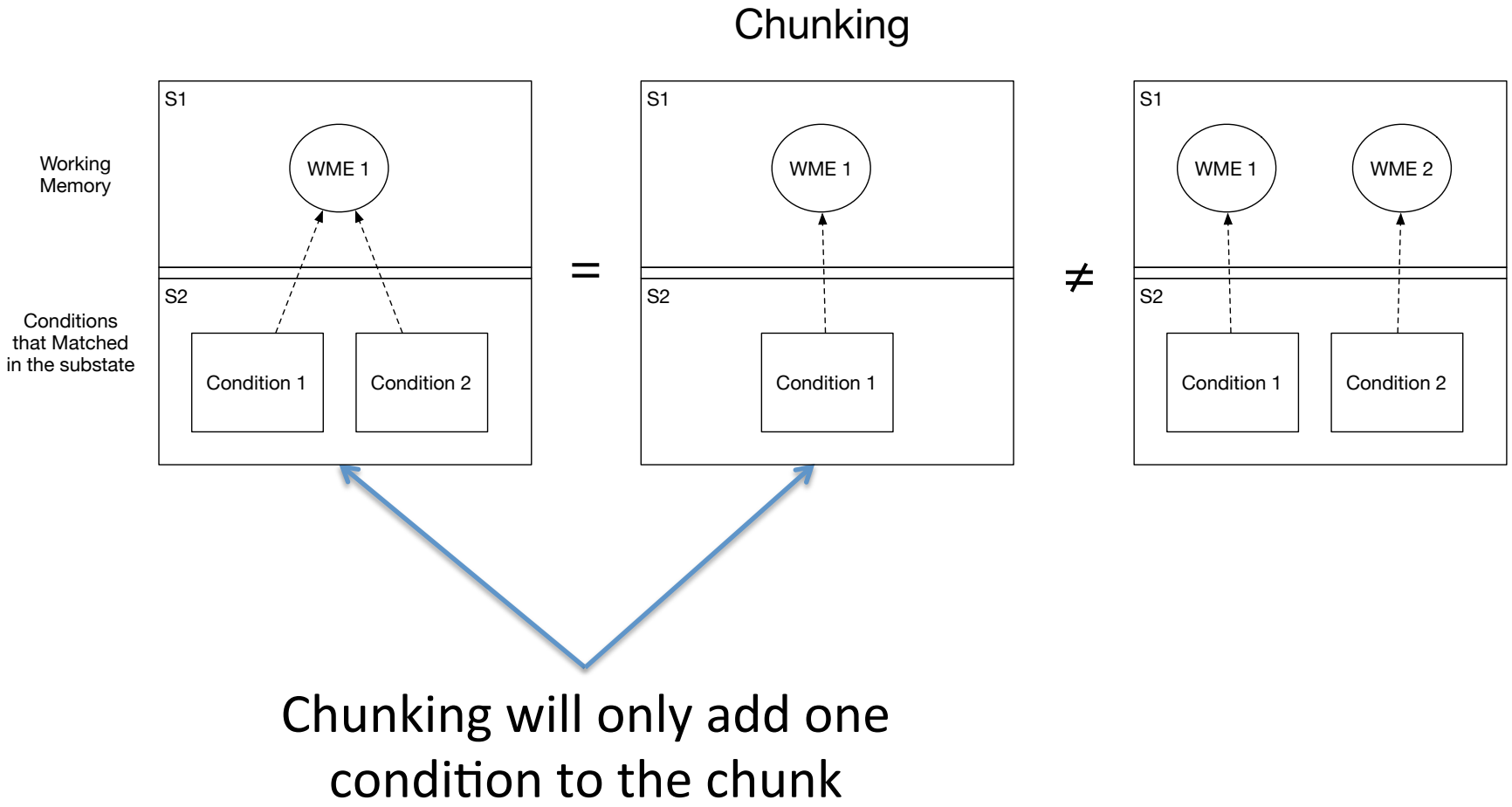
Key Thing To Remember

- When decomposing problems in our agents, we now should be more deliberate and consider how knowledge is delineated:
  - What knowledge is needed
  - How it is copied or linked, and
  - *How it is accessed*

EBC Change That Affects Access

- What happens when 2 conditions match the same WME?
  - Because chunking uses a WM trace…
    - Literally identical
    - Chunking will only adds one condition
  - Because EBC uses an explanation trace
    - Each condition may have different identities and constraints
    - EBC will add both
- In some sense, we say the line between states is opaque to EBC.

Consider State Barrier as Opaque

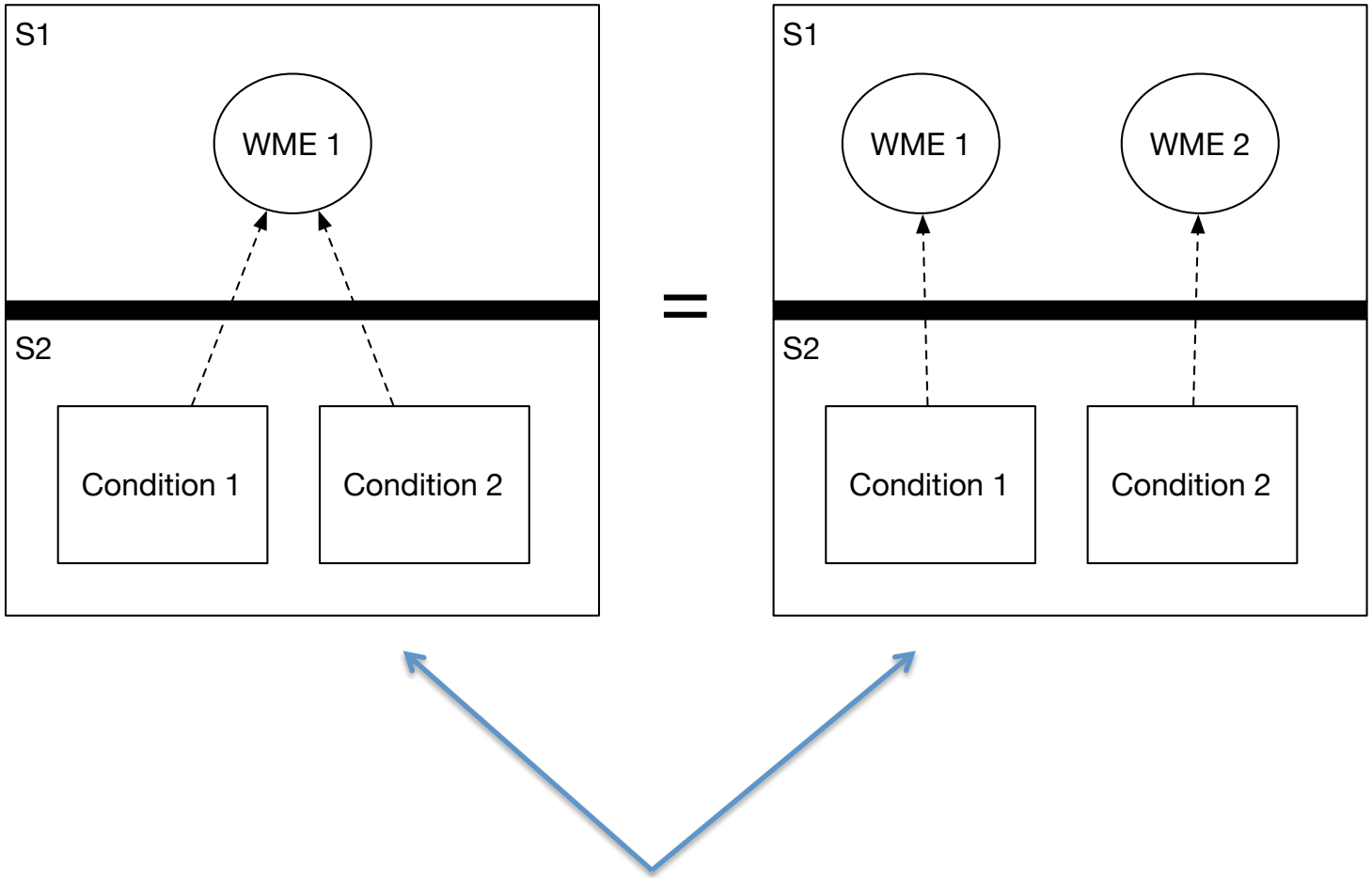


Chunking will only add one condition to the chunk

Consider State Barrier as Opaque

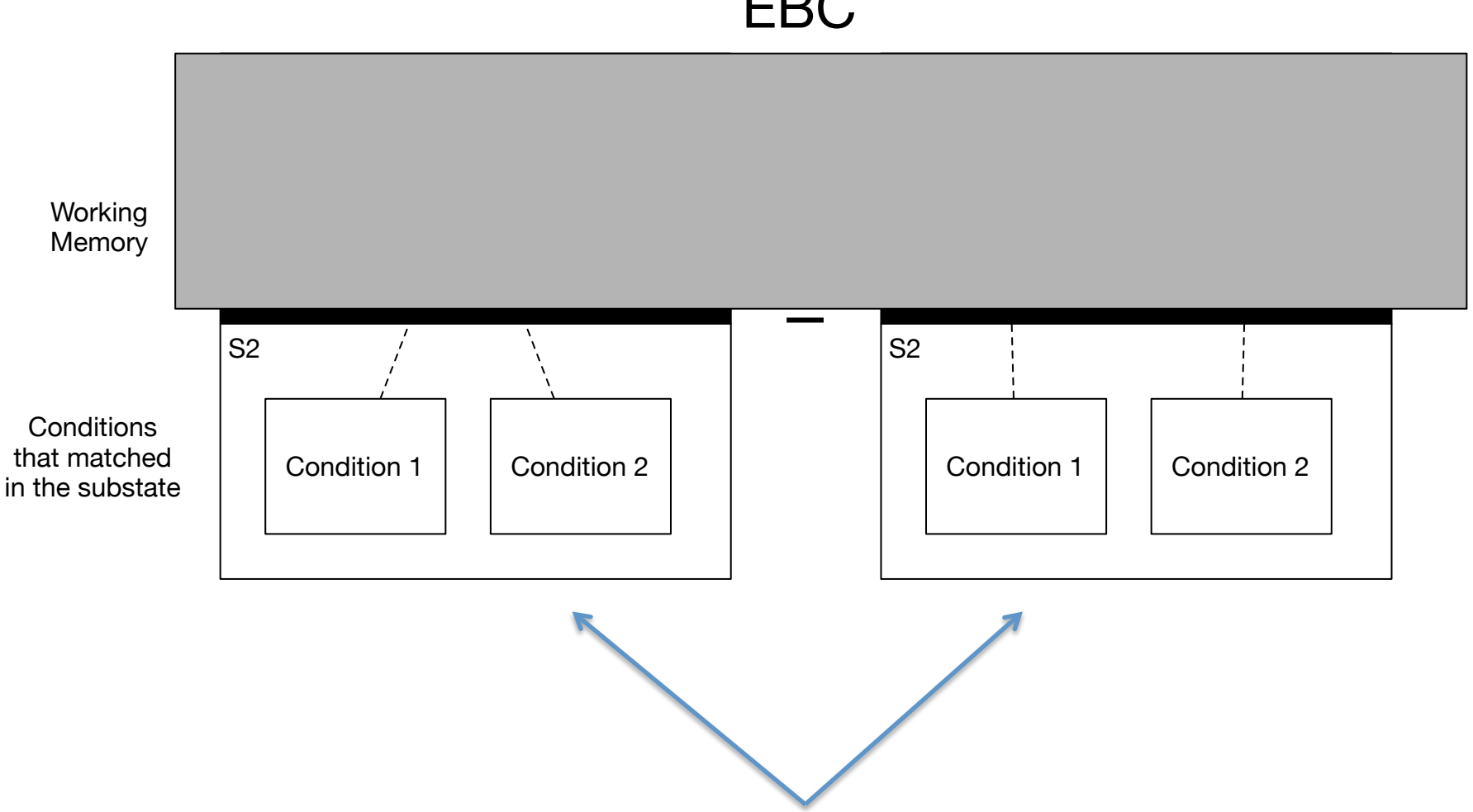
Working Memory

Conditions that matched in the substate



EBC will always add both conditions, and they may end up having different variables

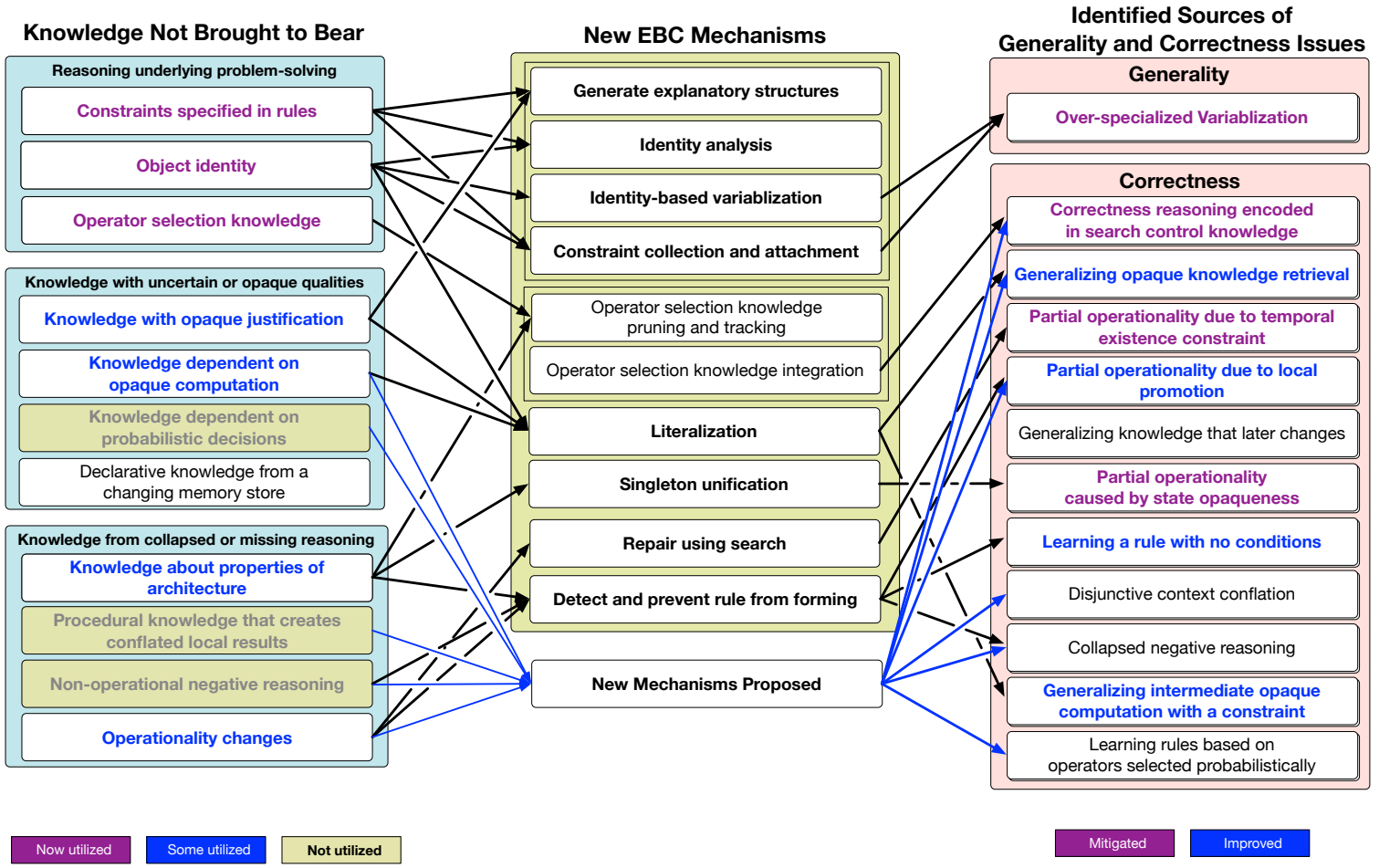
Consider State Barrier as Opaque



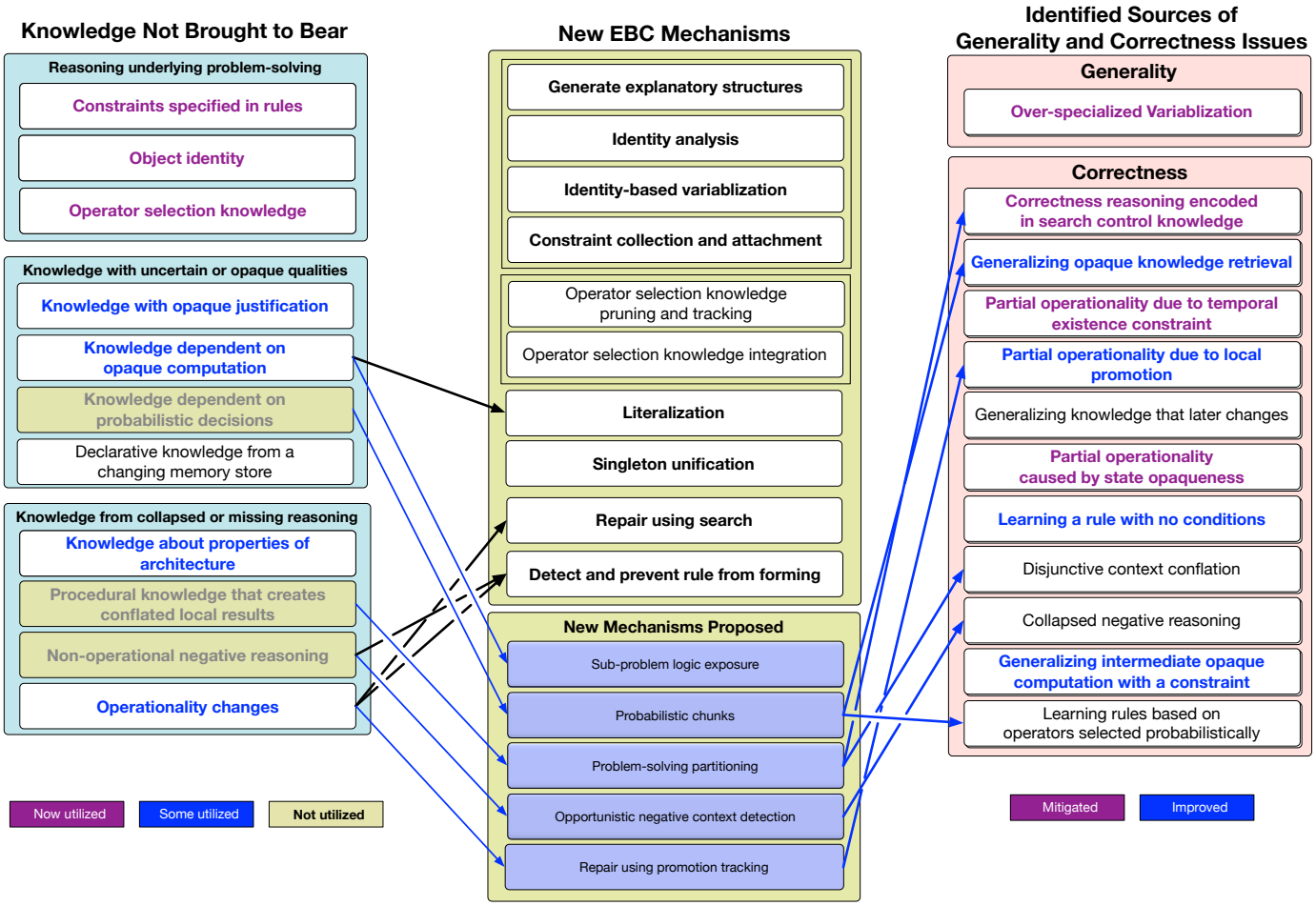
EBC will always add both conditions, and they may end up having different variables

Current Status

Knowledge Chunking Now Uses



Potential New Mechanisms



Nuggets

- Available now.
- Now used by most people in our lab.
  - Tester zero (James) has been using for 1.5 years
- Has been used with agents that do many advanced, convoluted things
- Minimal knowledge engineering required
- 20x - 80x improvement on some agents already

Coal

- Can be difficult to understand how a chunk was formed
  - Real-world examples are way more complex
  - Many aspects not discussed
- Still needs:
  - More formal quantitative evaluation of utility
  - Profiling of memory and CPU usage
  - Documentation