Intelligent Agents
Representations for Classical Planning

Ute Schmid

Cognitive Systems, Applied Computer Science, Bamberg University

slides partially by Dana S. Nau
CMSC 722, AI Planning
University of Maryland, Spring 2008

last change: 25. April 2016
Quick Review: The DWR Domain
Representation of a Planning Problem

To compute a plan you need to have a representation of the environment, an initial state and a set of goals to be reached (objectives)
Outline

- Representation schemes
  - Classical representation
  - Set-theoretic representation
  - State-variable representation
  - Examples: DWR and the Blocks World
  - Comparison of representation formats
Classical Representation

- Start with a \textit{function-free} first-order language
  - Finitely many predicate symbols and constant symbols, but no function symbols

- Example: the DWR domain
  - Locations: \( l_1, l_2, \ldots \)
  - Containers: \( c_1, c_2, \ldots \)
  - Piles: \( p_1, p_2, \ldots \)
  - Robot carts: \( r_1, r_2, \ldots \)
  - Cranes: \( k_1, k_2, \ldots \)
Classical Representation

- **Atom**: predicate symbol and args
  - Use these to represent both fixed and dynamic relations
    - adjacent(l,l')
    - occupied(l)
    - loaded(r,c)
    - holding(k,c)
    - in(c,p)
    - top(c,p)
    - attached(p,l)
    - at(r,l)
    - unloaded(r)
    - empty(k)
    - on(c,c')
    - top(pallet,p)

- **Ground** expression: contains no variable symbols - e.g., in(c1,p3)
- **Unground** expression: at least one variable symbol - e.g., in(c1,x)
- **Substitution**: \( \theta = \{ x_1 \leftarrow v_1, x_2 \leftarrow ..., x_n \leftarrow v_n \} \)
  - Each \( x_i \) is a variable symbol; each \( v_i \) is a term
- **Instance of** \( e \): result of applying a substitution \( \theta \) to \( e \)
  - Replace variables of \( e \) simultaneously, not sequentially
**States**

- **State**: a set $s$ of ground atoms
  - The atoms represent the things that are true in one of $\Sigma$’s states
  - Only finitely many ground atoms, so only finitely many possible states

$$s_1 = \{\text{attached}(p1, \text{loc}1), \text{in}(c1, p1), \text{in}(c3, p1), \text{top}(c3, p1), \text{on}(c3, c1), \text{on}(c1, \text{pallet}), \text{attached}(p2, \text{loc}1), \text{in}(c2, p2), \text{top}(c2, p2), \text{on}(c2, \text{pallet}), \text{belong}(\text{crane}1, \text{loc}1), \text{empty}(\text{crane}1), \text{adjacent}(\text{loc}1, \text{loc}2), \text{adjacent}(\text{loc}2, \text{loc}1), \text{at}(r1, \text{loc}2), \text{occupied}(\text{loc}2), \text{unloaded}(r1)\}$$
Operators

- **Operator**: a triple \( o=(name(o), \text{precond}(o), \text{effects}(o)) \)
  - \( name(o) \) is a syntactic expression of the form \( n(x_1, \ldots, x_k) \)
    - \( n: \text{operator symbol} - \) must be unique for each operator
    - \( x_1, \ldots, x_k: \) variable symbols (parameters)
      - must include every variable symbol in \( o \)
  - \( \text{precond}(o): \text{preconditions} \)
    - literals that must be true in order to use the operator
  - \( \text{effects}(o): \text{effects} \)
    - literals the operator will make true

\textit{take}(k, l, c, d, p)

;; crane \( k \) at location \( l \) takes \( c \) off of \( d \) in pile \( p \)

\begin{align*}
\text{precond:} & \quad \text{belong}(k, l), \text{attached}(p, l), \text{empty}(k), \text{top}(c, p), \text{on}(c, d) \\
\text{effects:} & \quad \text{holding}(k, c), \neg\text{empty}(k), \neg\text{in}(c, p), \neg\text{top}(c, p), \neg\text{on}(c, d), \text{top}(d, p)
\end{align*}
**Actions**

- **Action:**
  ground instance (via substitution) of an operator

  **Operator** (i.e., an abstract schema of actions, containing variable)

  \[
  \text{take}(k, l, c, d, p) ;\text{ crane } k \text{ at location } l \text{ takes } c \text{ off of } d \text{ in pile } p
  \]
  precond: belong\((k, l)\), attached\((p, l)\), empty\((k)\), top\((c, p)\), on\((c, d)\)
  effects: holding\((k, c)\), ¬empty\((k)\), ¬in\((c, p)\), ¬top\((c, p)\), ¬on\((c, d)\), top\((d, p)\)

- **Action** (i.e., an instantiated operator which can be applied to some specific states)

  \[
  \text{take}(\text{crane1}, \text{loc1}, c3, c1, p1) ;\text{ crane crane1 at location loc1 takes c3 off of c1 in pile p1}
  \]
  precond: belong\((\text{crane1,loc1})\), attached\((p1,\text{loc1})\),
  empty\((\text{crane1})\), top\((c3,p1)\), on\((c3,c1)\)
  effects: holding\((\text{crane1, c3})\), ¬empty\((\text{crane1})\), ¬in\((c3,p1)\),
  ¬top\((c3,p1)\), ¬on\((c3,c1)\), top\((c1,p1)\)
Let $S$ be a set of literals. Then
- $S^+ = \{ \text{atoms that appear positively in } S \}$
- $S^- = \{ \text{atoms that appear negatively in } S \}$

More specifically, let $a$ be an operator or action. Then
- $precond^+(a) = \{ \text{atoms that appear positively in } a's \text{ preconditions} \}$
- $precond^-(a) = \{ \text{atoms that appear negatively in } a's \text{ preconditions} \}$
- $effects^+(a) = \{ \text{atoms that appear positively in } a's \text{ effects} \}$
- $effects^-(a) = \{ \text{atoms that appear negatively in } a's \text{ effects} \}$

$take(k, l, c, d, p)$

;; crane $k$ at location $l$ takes $c$ off of $d$ in pile $p$
precond: $\text{belong}(k, l), \text{attached}(p, l), \text{empty}(k), \text{top}(c, p), \text{on}(c, d)$
effects: $\text{holding}(k, c), \lnot \text{empty}(k), \lnot \text{in}(c, p), \lnot \text{top}(c, p), \lnot \text{on}(c, d), \text{top}(d, p)$

Let $S$ be a set of literals. Then
- $effects^+(take(k, l, c, d, p)) = \{ \text{holding}(k, c), \text{top}(d, p) \}$
- $effects^-(take(k, l, c, d, p)) = \{ \text{empty}(k), \text{in}(c, p), \text{top}(c, p), \text{on}(c, d) \}$
Classical Representation

Applicability

- An action \( a \) is *applicable* to a state \( s \) if \( s \) satisfies \( \text{precond}(a) \), i.e., if \( \text{precond}^+(a) \subseteq s \) and \( \text{precond}^-(a) \cap s = \emptyset \).

- Here are an action and a state that it's applicable to:

\[
s_1 = \{ \text{attached}(p1, loc1), \text{in}(c1, p1), \\
\text{in}(c3, p1), \text{top}(c3, p1), \text{on}(c3, c1), \\
\text{on}(c1, \text{pallet}), \text{attached}(p2, loc1), \\
\text{in}(c2, p2), \text{top}(c2, p2), \text{on}(c2, \text{palet}), \\
\text{belong}(\text{crane1, loc1}), \text{empty}(\text{crane1}), \\
\text{adjacent}(\text{loc1, loc2}), \text{adjacent}(\text{loc2, loc1}), \\
\text{at}(r1, \text{loc2}), \text{occupied}(\text{loc2}), \\
\text{unloaded}(r1) \}
\]

\textit{take}(\text{crane1, loc1, c3, c1, p1})

;; crane crane1 at location loc1 takes c3 off of c1 in pile p1

precond: belong(\text{crane1,loc1}), \text{attached}(p1,\text{loc1}),

\text{empty}(\text{crane1}), \text{top}(c3,p1), \text{on}(c3,c1)

effects: \text{holding}(\text{crane1, c3}), \neg \text{empty}(\text{crane1}), \neg \text{in}(c3,p1),

\neg \text{top}(c3,p1), \neg \text{on}(c3,c1), \text{top}(c1,p1)
Classical Representation

Result of Performing an Action

- If a is applicable to s, the result of performing it is

\[ \gamma(s, a) = s - \text{effects}^- (a) \cup \text{effects}^+ (a) \]

- Delete the negative effects, and add the positive ones

\[ \text{take}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}) \]

`; crane crane1 at location loc1 takes c3 off of c1 in pile p1

precond: belong(crane1,loc1), attached(p1,loc1),
empty(crane1), top(c3,p1), on(c3,c1)

effects: holding(crane1, c3), \neg empty(crane1), \neg in(c3,p1),
\neg top(c3,p1), \neg on(c3,c1), top(c1,p1)
Classical Representation

\[ s_1 = \{ \text{attached}(p_1, \text{loc}_1), \text{in}(c_1, p_1), \text{in}(c_3, p_1), \text{top}(c_3, p_1), \text{on}(c_3, c_1), \\
\text{on}(c_1, \text{pallet}), \text{attached}(p_2, \text{loc}_1), \text{in}(c_2, p_2), \text{top}(c_2, p_2), \text{on}(c_2, \text{palet}), \\
\text{belong}(\text{crane}_1, \text{loc}_1), \text{empty}(\text{crane}_1), \text{adjacent}(\text{loc}_1, \text{loc}_2), \\
\text{adjacent}(\text{loc}_2, \text{loc}_1), \text{at}(r_1, \text{loc}_2), \text{occupied}(\text{loc}_2), \text{unloaded}(r_1) \} \]

take(\text{crane}_1, \text{loc}_1, c_3, c_1, p_1) \text{ with} \\
\text{precond: belong(}\text{crane}_1, \text{loc}_1), \text{attached}(p_1,\text{loc}_1), \text{empty(}\text{crane}_1), \text{top}(c_3,p_1), \text{on}(c_3,c_1) \\
is \text{applicable to } s_1: \\
\{ \text{belong(}\text{crane}_1, \text{loc}_1), \text{attached}(p_1,\text{loc}_1), \text{empty(}\text{crane}_1), \text{top}(c_3,p_1), \text{on}(c_3,c_1) \subseteq s_1 \} = \\
\{ \text{attached}(p_1, \text{loc}_1), \text{in}(c_1, p_1), \text{in}(c_3, p_1), \text{top}(c_3, p_1), \text{on}(c_3, c_1), \\
\text{on}(c_1, \text{pallet}), \text{attached}(p_2, \text{loc}_1), \text{in}(c_2, p_2), \text{top}(c_2, p_2), \text{on}(c_2, \text{palet}), \\
\text{belong}(\text{crane}_1, \text{loc}_1), \text{empty}(\text{crane}_1), \text{adjacent}(\text{loc}_1, \text{loc}_2), \\
\text{adjacent}(\text{loc}_2, \text{loc}_1), \text{at}(r_1, \text{loc}_2), \text{occupied}(\text{loc}_2), \text{unloaded}(r_1) \} \\
\cup \text{holding(}\text{crane}_1, c_3), \text{top}(c_1,p_1) \}

Application:

\[ s_2 = \{ \text{attached}(p_1, \text{loc}_1), \text{in}(c_1, p_1), \text{in}(c_3, p_1), \text{top}(c_3, p_1), \text{on}(c_3, c_1), \\
\text{on}(c_1, \text{pallet}), \text{attached}(p_2, \text{loc}_1), \text{in}(c_2, p_2), \text{top}(c_2, p_2), \text{on}(c_2, \text{palet}), \\
\text{belong}(\text{crane}_1, \text{loc}_1), \text{empty}(\text{crane}_1), \text{adjacent}(\text{loc}_1, \text{loc}_2), \\
\text{adjacent}(\text{loc}_2, \text{loc}_1), \text{at}(r_1, \text{loc}_2), \text{occupied}(\text{loc}_2), \text{unloaded}(r_1) \\
\cup \text{holding(}\text{crane}_1, c_3), \text{top}(c_1,p_1) \} \]
move\( (r, l, m) \)
   ;; robot \( r \) moves from location \( l \) to location \( m \)
   precond: adjacent\((l,m)\), at\((r,l)\), ¬occupied\((m)\)
   effects: at\((r,m)\), occupied\((m)\), ¬occupied\((l)\), ¬at\((r,l)\)

load\( (k, l, c, r) \)
   ;; crane \( k \) at location \( l \) loads container \( c \) onto robot \( r \)
   precond: belong\((k,l)\), holding\((k,c)\), at\((r,l)\), unloaded\((r)\)
   effects: empty\((k)\), ¬holding\((k,c)\), loaded\((r,c)\), ¬unloaded\((r)\)

unload\( (k, l, c, r) \)
   ;; crane \( k \) at location \( l \) takes container \( c \) from robot \( r \)
   precond: belong\((k,l)\), at\((r,l)\), loaded\((r,c)\), empty\((k)\)
   effects: ¬empty\((k)\), holding\((k,c)\), unloaded\((r)\), ¬loaded\((r)\)

put\( (k, l, c, d, p) \)
   ;; crane \( k \) at location \( l \) puts \( c \) onto \( d \) in pile \( p \)
   precond: belong\((k,l)\), attached\((p,l)\), holding\((k,c)\), top\((d,p)\)
   effects: ¬holding\((k,c)\), empty\((k)\), in\((c,p)\), top\((c,p)\), on\((c,d)\), ¬top\((d,p)\)

take\( (k, l, c, d, p) \)
   ;; crane \( k \) at location \( l \) takes \( c \) off of \( d \) in pile \( p \)
   precond: belong\((k,l)\), attached\((p,l)\), empty\((k)\), top\((c,p)\), on\((c,d)\)
   effects: holding\((k,c)\), ¬empty\((k)\), ¬in\((c,p)\), ¬top\((c,p)\), ¬on\((c,d)\), ¬top\((d,p)\)

Planning domain: language plus operators
- Corresponds to a set of state-transition systems
- Example: operators for the DWR domain
Given a planning domain (language $L$, operators $O$)

*Statement* of a planning problem: a triple $P = (O, s_0, g)$

- $O$ is the collection of operators
- $s_0$ is a state (the initial state)
- $g$ is a set of literals (the goal formula)

The actual *planning problem*: $P = (\Sigma, s_0, S_g)$

- $s_0$ and $S_g$ are as above
- $\Sigma = (S, A, \gamma)$ is a state-transition system
- $S = \{\text{all sets of ground atoms in } L\}$
- $A = \{\text{all ground instances of operators in } O\}$
- $\gamma$ is the state-transition function determined by the operators

I’ll often say “planning problem“ when I mean the statement of the problem
Plan: any sequence of actions $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ such that each $a_i$ is a ground instance of an operator in $O$

The plan is a solution for $P = (O, s_0, g)$ if it is executable and achieves $g$

i.e., if there are states $s_0, s_1, \ldots, s_n$ such that

\[
\begin{align*}
\Rightarrow & \quad \gamma(s_0, a_1) = s_1 \\
\Rightarrow & \quad \gamma(s_1, a_2) = s_2 \\
\Rightarrow & \quad \ldots \\
\Rightarrow & \quad \gamma(s_{n-1}, a_n) = s_n \\
\Rightarrow & \quad s_n \text{ satisfies } g
\end{align*}
\]
Example

Let \( P_1 = (O, s_1, g_1) \), where

- \( O \) is the set of operators given earlier

\[
O = \left\{ \text{loaded}(r_1, c_3), \ \at(r_1, \text{loc}2) \right\}
\]

\[
s_1 = \left\{ \text{attached}(p_1, \text{loc}1), \ \text{in}(c_1, p_1), \ \text{in}(c_3, p_1), \ \text{top}(c_3, p_1), \ \text{on}(c_3, c_1), \ \text{on}(c_1, \text{pallet}), \ \text{attached}(p_2, \text{loc}1), \ \text{in}(c_2, p_2), \ \text{top}(c_2, p_2), \ \text{on}(c_2, \text{pallet}), \ \text{belong}(\text{crane}1, \text{loc}1), \ \text{empty}(\text{crane}1), \ \text{adjacent}(\text{loc}1, \text{loc}2), \ \text{adjacent}(\text{loc}2, \text{loc}1), \ \at(r_1, \text{loc}2), \ \text{occupied}(\text{loc}2), \ \text{unloaded}(r_1) \right\}
\]
Here are three solutions for $P_1$:

- $\langle \text{take}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move}(\text{r1}, \text{loc2}, \text{loc1}), \text{move}(\text{r1}, \text{loc1}, \text{loc2}), \text{move}(\text{r1}, \text{loc2}, \text{loc1}), \text{load}(\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move}(\text{r1}, \text{loc1}, \text{loc2}) \rangle$

- $\langle \text{take}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{move}(\text{r1}, \text{loc2}, \text{loc1}), \text{load}(\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move}(\text{r1}, \text{loc1}, \text{loc2}) \rangle$

- $\langle \text{move}(\text{r1}, \text{loc2}, \text{loc1}), \text{take}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1}), \text{load}(\text{crane1}, \text{loc1}, \text{c3}, \text{r1}), \text{move}(\text{r1}, \text{loc1}, \text{loc2}) \rangle$

Each of them produces the state shown here:
Example (continued)

- The first is *redundant*: can remove actions and still have a solution
  - \( \langle \text{take}(\text{crane1}, \text{loc}1, c3, c1, p1), \text{move}(r1, \text{loc}2, \text{loc}1), \text{move}(r1, \text{loc}1, \text{loc}2), \text{move}(r1, \text{loc}2, \text{loc}1), \text{load}(\text{crane1}, \text{loc}1, c3, r1), \text{move}(r1, \text{loc}1, \text{loc}2) \rangle \)
  - \( \langle \text{take}(\text{crane1}, \text{loc}1, c3, c1, p1), \text{move}(r1, \text{loc}2, \text{loc}1), \text{load}(\text{crane1}, \text{loc}1, c3, r1), \text{move}(r1, \text{loc}1, \text{loc}2) \rangle \)
  - \( \langle \text{move}(r1, \text{loc}2, \text{loc}1), \text{take}(\text{crane1}, \text{loc}1, c3, c1, p1), \text{load}(\text{crane1}, \text{loc}1, c3, r1), \text{move}(r1, \text{loc}1, \text{loc}2) \rangle \)

- The 2nd and 3rd are *irredundant* and *shortest*
Set-Theoretic Representation

- Like classical representation, but restricted to propositional logic

- States:
  - Instead of a collection of ground atoms . . .
    \{on(c1,pallet), on(c1,r1), on(c1,c2), . . ., at(r1,l1), at(r1,l2), . . .\}
  - . . . use a collection of propositions (boolean variables):
    \{on-c1-pallet, on-c1-r1, on-c1-c2, . . ., at-r1-l1, at-r1-l2, . . .\}
Instead of operators like this one,

\[
\text{take}(k, l, c, d, p) \\
\text{;; crane } k \text{ at location } l \text{ takes } c \text{ off of } d \text{ in pile } p
\]

precond: \( \text{belong}(k,l), \text{attached}(p,l), \text{empty}(k), \text{top}(c,p), \text{on}(c,d) \)

effects: \( \text{holding}(k,c), \neg \text{empty}(k), \neg \text{in}(c,p), \neg \text{top}(c,p), \neg \text{on}(c,d), \neg \text{top}(d,p) \)

and rewrite ground atoms as propositions

\[
\text{take}(\text{crane1}, \text{loc1}, c3, c1, p1) \\
\text{;; crane } \text{crane1} \text{ at location } \text{loc1} \text{ takes } c3 \text{ off of } c1 \text{ in pile } p1
\]

precond: \( \text{belong}(\text{crane1},\text{loc1}), \text{attached}(p1,\text{loc1}), \text{empty}(\text{crane1}), \text{top}(c3,p1), \text{on}(c3,c1) \)

effects: \( \text{holding}(\text{crane1}, c3), \neg \text{empty}(\text{crane1}), \neg \text{in}(c3,p1), \neg \text{top}(c3,p1), \neg \text{on}(c3,c1), \text{top}(c1,p1) \)

\[
\text{take} - \text{crane1} - \text{loc1} - c3 - c1 - p1
\]

precond: \( \text{belong-crane1-loc1}, \text{attached-p1-loc1}, \text{empty-crane1}, \text{top-c3-p1}, \text{on-c3-c1} \)

delete: \( \text{empty-crane1}, \text{in-c3-p1}, \text{top-c3-p1}, \text{on-c3-p1} \)

add: \( \text{holding-crane1-c3}, \text{top-c1-p1} \)
Remarks on Set-Theoretic Representation

- A set-theoretic representation is equivalent to a classical representation in which all of the atoms are ground.

- Exponential blow-up
  - If a classical operator contains $n$ atoms and each atom has arity $k$, then it corresponds to $c^{nk}$ actions where $c = |\{\text{constant symbols}\}|$.

- Set-theoretic representation is applied in practice for SAT-solvers and model-checking. For these applications typically, a set-theoretical representation is algorithmically generated from a classical representation.

- Set-theoretic representation is a very useful concept for theoretical analyses of planning (using an explicit representation of all states of a problem).
State-Variable Representation

- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to state variables
  - Like fields in a record structure
- Classical and state-variable representations take similar amounts of space
  - Each can be translated into the other in low-order polynomial time

move(r,l,m)

;; robot r at location l moves to an adjacent location m
precond: rloc(r)=l, adjacent(l,m)
effects: rloc(r) ← m

{top(p1)=c3, cpos(c3)=c1, cpos(c1)=pallet, holding(crane1)=nil, rloc(r1)=loc2, loaded(r1)=nil,…}
The engineering view rather than the logic view

Especially useful when a new value which is assigned to a variable has to be calculated by some procedure

Example: Depots Domain with Fluents in PDDL
  - Predefined functions `assign` and `increase`

In general, arbitrary user-defined methods can be used:

\[
\text{fuel}(r) \leftarrow \text{calcDecrease}(r, l, m)
\]
(define (domain Depot-object-fluents)
   (:requirements :typing :equality :fluents)
   (:types place locatable - object
depot distributor - place
truck hoist surface - locatable
pallet crate - surface)

   (:constants no-crate - crate)
   (:predicates (clear ?s - surface))
   (:functions
      (load-limit ?t - truck)
      (current-load ?t - truck)
      (weight ?c - crate)
      (fuel-cost) - number
      (position-of ?l - locatable) - place
      (crate-held ?h - hoist) - crate
      (thing-below ?c - crate) - (either surface truck))

   (:action drive
      :parameters (?t - truck ?p - place)
      :effect (and (assign (position-of ?t) ?p)
      (increase (fuel-cost) 10)))
Example: The Blocks World

- Infinitely wide table, finite number of children’s blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- Want to move blocks from one configuration to another
  - e.g.,
    
    initial state
    
    goal

- Can be expressed as a special case of DWR
  - But the usual formulation is simpler
- I’ll give classical, set-theoretic, and state-variable formulations
  - For the case where there are five blocks
Classical Representation: Symbols

- **Constant symbols:**
  - The blocks: \( a, b, c, d, e \)

- **Predicates:**
  - \( ontable(x) \): block \( x \) is on the table
  - \( on(x,y) \): block \( x \) is on block \( y \)
  - \( clear(x) \): block \( x \) has nothing on it
  - \( holding(x) \): the robot hand is holding block \( x \)
  - \( handempty \): the robot hand isn’t holding anything
### Classical Operators

**unstack(x,y)**
- **Precond:** on(x,y), clear(x), handempty
- **Effects:** \(\neg\text{on}(x,y), \neg\text{clear}(x), \neg\text{handempty}, \text{holding}(x), \text{clear}(y)\)

**stack(x,y)**
- **Precond:** holding(x), clear(y)
- **Effects:** \(\neg\text{holding}(x), \neg\text{clear}(y), \text{on}(x,y), \text{clear}(x), \text{handempty}\)

**pickup(x)**
- **Precond:** ontable(x), clear(x), handempty
- **Effects:** \(\neg\text{ontable}(x), \neg\text{clear}(x), \neg\text{handempty}, \text{holding}(x)\)

**putdown(x)**
- **Precond:** holding(x)
- **Effects:** \(\neg\text{holding}(x), \text{ontable}(x), \text{clear}(x), \text{handempty}\)
Set-Theoretic Representation: Symbols

For five blocks, there are 36 propositions

Here are 5 of them:

- `ontable-a` - block a is on the table
- `on-c-a` - block c is on block a
- `clear-c` - block c has nothing on it
- `holding-d` - the robot hand is holding block d
- `handempty` - the robot hand isn’t holding anything
Fifty different actions

Here are four of them:

**unstack-c-a**
- Pre: on-c,a, clear-c, handempty
- Del: on-c,a, clear-c, handempty
- Add: holding-c, clear-a

**stack-c-a**
- Pre: holding-c, clear-a
- Del: holding-c, clear-a
- Add: on-c-a, clear-c, handempty

**pickup-c**
- Pre: ontable-c, clear-c, handempty
- Del: ontable-c, clear-c, handempty
- Add: holding-c

**putdown-c**
- Pre: holding-c
- Del: holding-c
- Add: ontable-c, clear-c, handempty
State-Variable Representation: Symbols

- **Constant symbols:**
  - $a, b, c, d, e$ of type *block*
  - $0, 1, table, nil$ of type *other*

- **State variables:**
  - $pos(x)=y$ if block $x$ is on the block $y$
  - $pos(x)=table$ if block $x$ is on the table
  - $pos(x)=nil$ if block $x$ is being held
  - $clear(x)=1$ if block $x$ has nothing on it
  - $clear(x)=0$ if block $x$ is being held or has another block on it
  - $holding=x$ if the robot hand is holding block $x$
  - $holding=nil$ if the robot hand is holding nothing
**State-Variable Operators**

- **unstack(x : block, y : block)**
  - Precond: pos(x)=y, clear(y)=0, clear(x)=1, holding=nil
  - Effects: pos(x)=nil, clear(x)=0, holding=x, clear(y)=1

- **stack(x : block, y : block)**
  - Precond: holding=x, clear(x)=0, clear(y)=1
  - Effects: holding=nil, clear(y)=0, pos(x)=y, clear(x)=1

- **pickup(x : block)**
  - Precond: pos(x)=table, clear(x)=1, holding=nil
  - Effects: pos(x)=nil, clear(x)=0, holding=x

- **putdown(x : block)**
  - Precond: holding=x
  - Effects: holding=nil, pos(x)=table, clear(x)=1
Functions and Relations cont.

- A relation $R$ is a relationship between sets
- Examples:
  - $R_{\text{succ}} \subset \text{Nat} \times \text{Nat} = \{(0, 1), (0, 2), (0, 3), \ldots\}$
  - $R_{\text{plus}} \subset \text{Nat} \times \text{Nat} \times \text{Nat} = \{(0, 0, 0), (0, 1, 1), \ldots\}$
- Each function can be transformed into a relation but not each relation is a function!

Append in the functional language Lisp:

```lisp
(defun append (L1, L2)
  (cond ((null L1) L2)
        (T (cons (car L1) (append (cdr L1) L2)))))
```

Append in the logical language Prolog:

```prolog
append([], L, L).
append([X|Xs], L2, [X|L]) :- append(Xs, L2, L).
```
Comparison

- Classical representation
  - The most popular for classical planning, partly for historical reasons

- Set-theoretic representation
  - Can take much more space than classical representation
  - Useful in algorithms that manipulate ground atoms directly
    - e.g., planning graphs, satisfiability
  - Useful for certain kinds of theoretical studies

- State-variable representation
  - Without usage of arbitrary user-defined functions equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers
  - Useful in non-classical planning problems as a way to handle numbers, functions, time
Extended Representations

- Typed variables and relations
- Conditional Operators
- Quantified Expressions
- Equality Constraints
- Disjunctive Preconditions
- Function Symbols
- Axiomatic Inference
- Attached Procedures

- are built upon the classical representation
- can be represented in PDDL
- are explored in some planning systems
Example: Equality constraints and conditioned effects

(define (domain blocks-world-domain)
  (:requirements :strips :equality :conditional-effects)
  (:constants Table)
  (:predicates (on ?x ?y)
    (clear ?x)
    (block ?b)
  )
  ;; Define step for placing one block on another.
  (:action puton
    :parameters (?X ?Y ?Z)
                     (not (= ?Y ?Z)) (not (= ?X ?Z))
                     (not (= ?X ?Y)) (not (= ?X Table)))
    :effect
    (and (on ?X ?Y) (not (on ?X ?Z))
         (when (not (= ?Z Table)) (clear ?Z))
         (when (not (= ?Y Table)) (not (clear ?Y)))))
)
Expressive Power

- Any problem that can be represented in one representation can also be represented in the other two.
- Can convert in linear time and space, except when converting to set-theoretic (where we get an exponential blow-up).

Set-theoretic representation
- write all of the ground instances
- trivial

Classical representation
- \( f(x_1, \ldots, x_n) = y \)
- \( P_f(x_1, \ldots, x_n, y) \)

State-variable representation
- \( P(x_1, \ldots, x_n) \)
- \( f_P(x_1, \ldots, x_n) = 1 \)
A function \( f : D^n \rightarrow W \)

is a unique mapping from \((n)\) values of domains to a value of a range

Examples:

\[\text{succ} : \text{Nat} \rightarrow \text{Nat}\]
\[\text{succ}(0) = 1, \text{succ}(1) = 1, \ldots\]

\[\text{plus} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}\]
\[\text{plus}(0, 0) = 0, \text{plus}(0, 1) = 1, \ldots\]
In classical representation a state is a set of ground literals
The problem domain is defined by a set of operators representing the possible actions
An action is a fully instantiated operator
An action can be applied to a state if all positive preconditions hold, e.g. are included in the state (and if no negative precondition holds)
The effect of an action is calculated by deleting literals which no longer hold after the execution of the action and by inserting literals which hold after the execution
This simple way to model state-transitions can be realized because of the closed-world assumption
Set-theoretic representations can be generated by writing all ground instances (exponential blow-up)

This representation is useful because there exist efficient algorithms for manipulating ground atoms (SAT-solvers, model checking)

The state-variable representation works with assignments of values to state-variables. This is useful when dealing with numbers (time, functions, ...)