Formal Model vs. Representation Language

Why a Separate Representation?

The formal model is "**<u>as simple as possible</u>**"

Few concepts involved: unstructured **<u>states</u>** & **<u>actions</u>** + **<u>transition function</u>**

Sufficient for any classical planning problem

No additional concepts are *required*!

Easier to *understand*
Easier to *analyze*

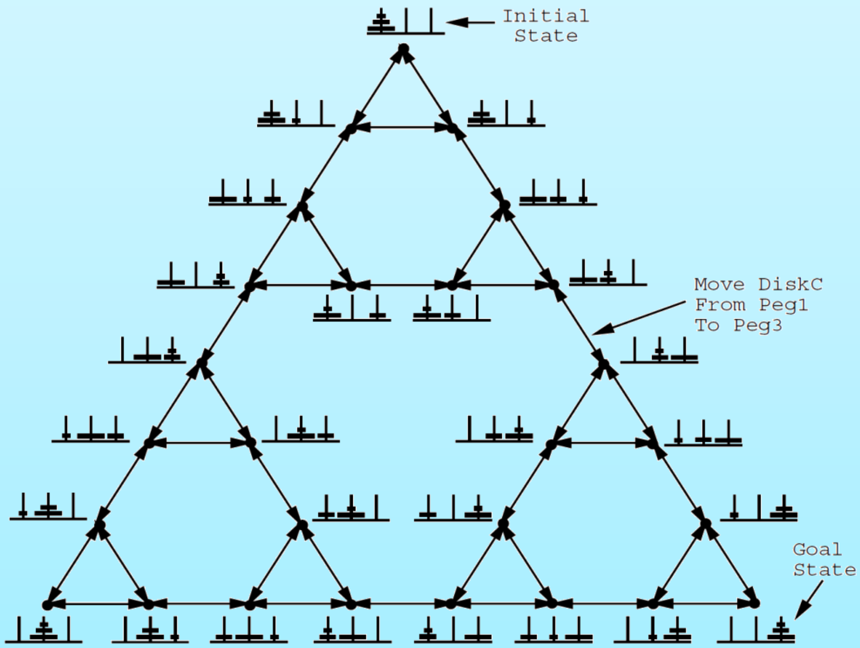
➔ We can analyze algorithms
relative to the model,
prove them correct,
and be certain of our conclusions

Inconvenient
for actual problem specifications!

Without additional structure,
each transition
[state, action] ➔ state
has to be defined separately!

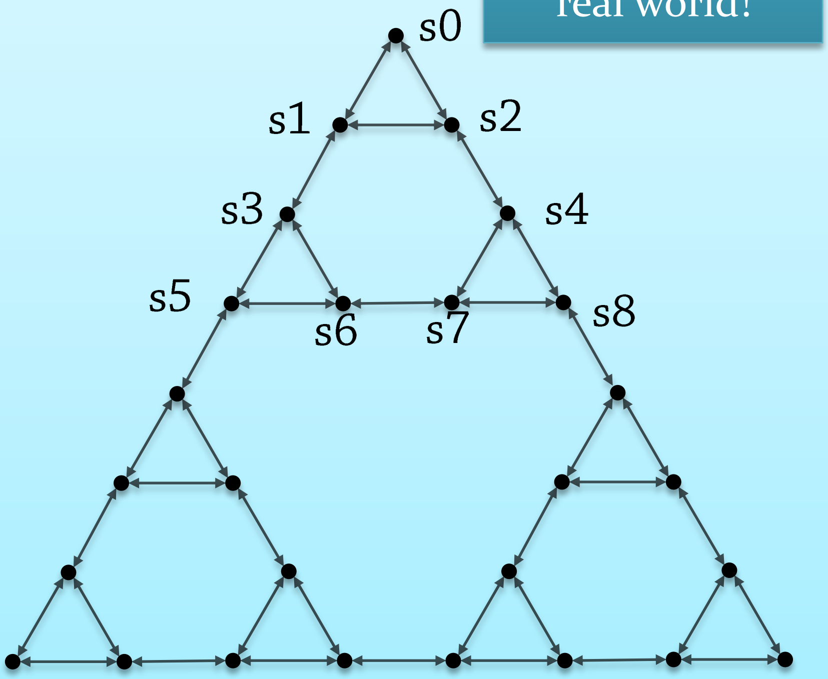
Why a Separate Representation? (2)

Instead of this…



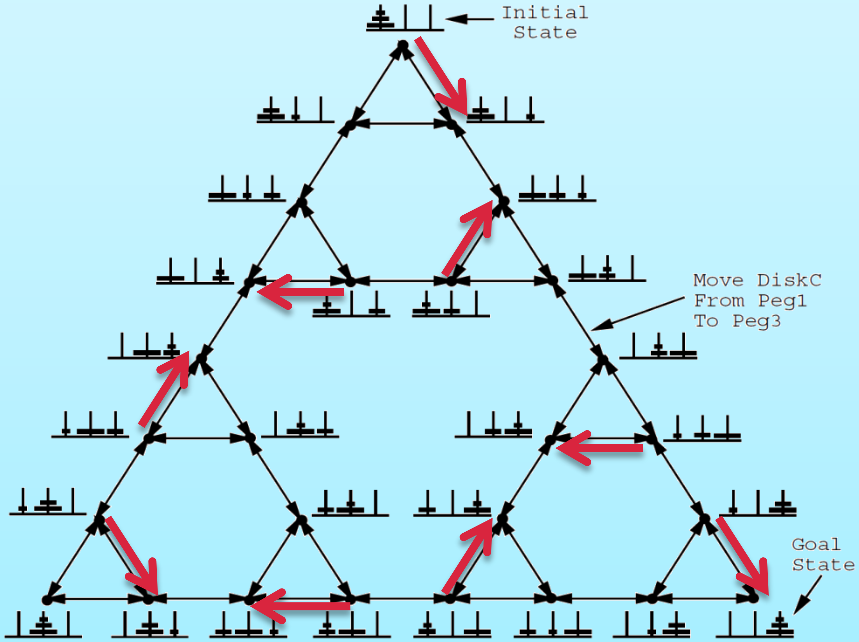
The STS really contains this:

No information about what s0 ”means” in the real world!



Why a Separate Representation? (3)

If all red arrows should be
“move diskA from Peg1 to Peg3”…



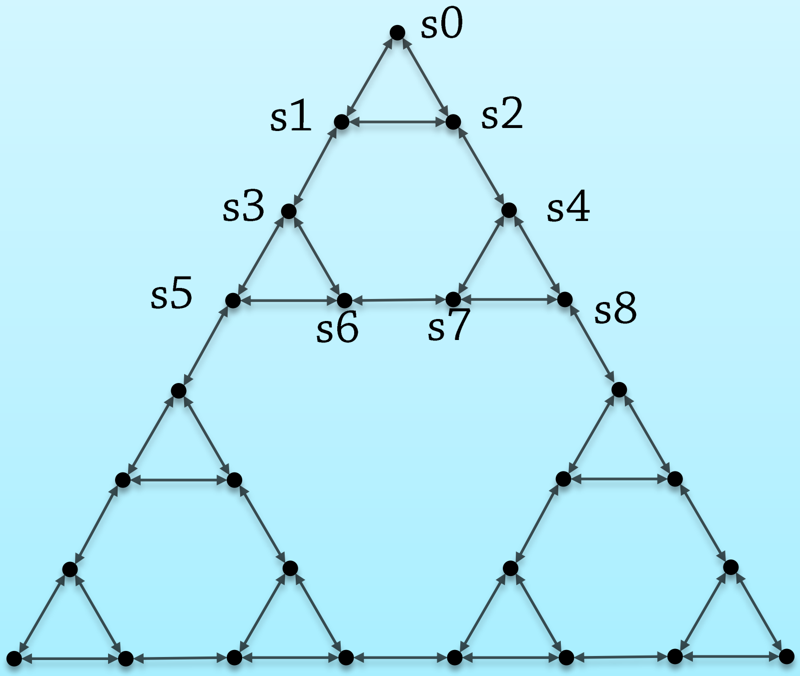
You have to specify this:

$\gamma(s0, A1) = s2$

$\gamma(s6, A1) = s5$

$\gamma(s7, A1) = s4$

…



We want **structure** – for convenience (now), and for problem analysis (later)!

Representations

”Mathematical” notation

Three variations in the book

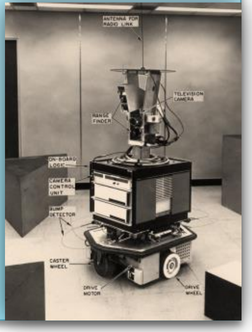
Set-theoretic representation
for classical problems:
Builds on propositions and set theory,
easy to define/analyze

Classical representation
for classical problems:
Builds on first-order logical predicates,
more convenient for problem specs

State variable representation
for classical problems:
Adds non-boolean functions,
same actual expressivity

”Practical” notation

- **PDDL**: Planning Domain Definition Language
  - Most common language today
  - General; many expressivity levels
- Lowest level of expressivity: **STRIPS** (from the 1969 planner)
  - Quite restrictive input language
  - Pioneered some concepts that we today associate with classical planning
  - In general, “STRIPS planning” ≈ “classical planning”



Running Example

- Running example (from the book): **Dock Worker Robots**

Containers shipped
in and out of a harbor

Cranes move containers
between ”piles” and robotic trucks

Representation Languages for Classical Planning: The STRIPS level of PDDL

Planning Domain Definition Language

PDDL: Domain and Problem Definition

- **PDDL** uses a Lisp-like syntax
  - Domains are **named**, associated with **expressivity requirements**

```
(define (domain dock-worker-robots)
        (:requirements
                :strips  ;; Standard level of expressivity
                ...)
        ;; Remaining domain information goes here!
)
```

Warning:
Many planners' parsers *ignore* expressivity specifications

- Problem instances are also named, associated with a specific domain

```
(define (problem dwr-problem-1)
        (:domain dock-worker-robots)
        ...
)
```

Colon before many keywords,
to avoid collisions
when new keywords are added

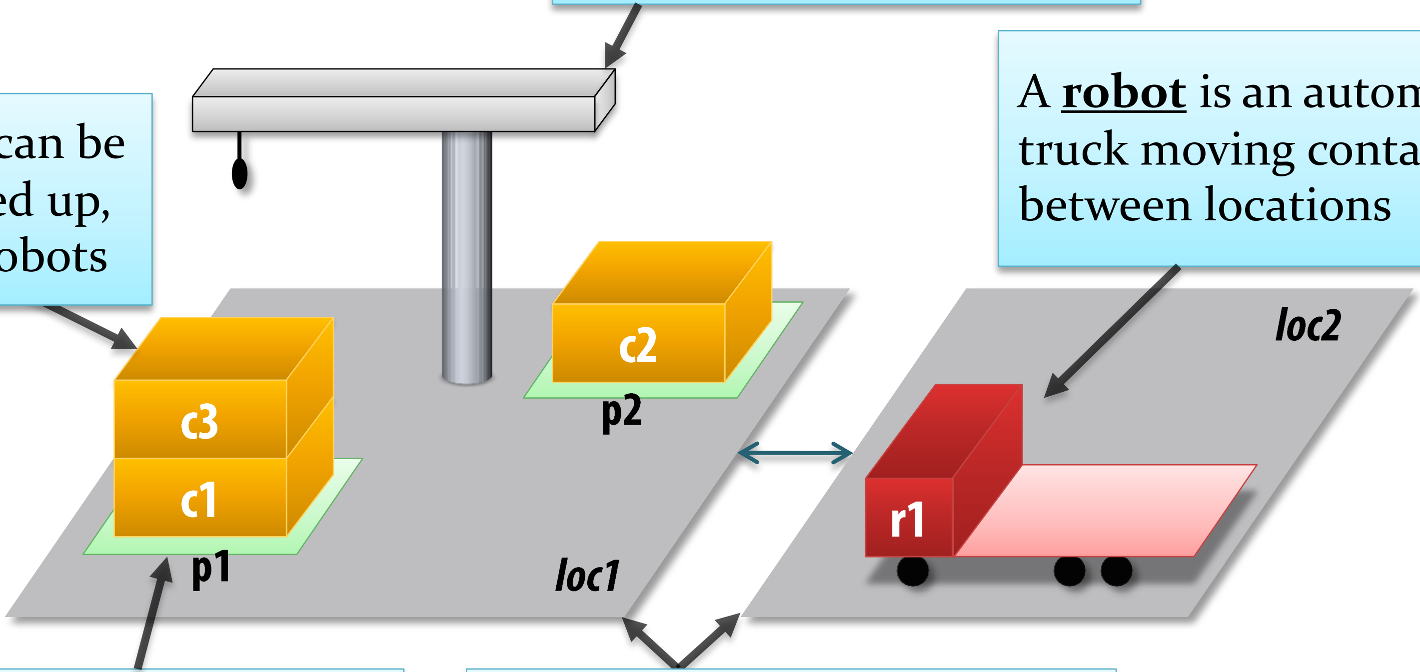
Objects and Object Types

Objects 1

- In the **classical representation** of planning problems:
  - The world contains a **finite** number of objects
  - Buildings, cards, aircraft, people, trucks, pieces of sheet metal, …

Objects 2: Dock Worker Robots

- **Dock Worker Robots**



A **crane** moves containers between piles and robots

A **container** can be stacked, picked up, loaded onto robots

A **robot** is an automated truck moving containers between locations

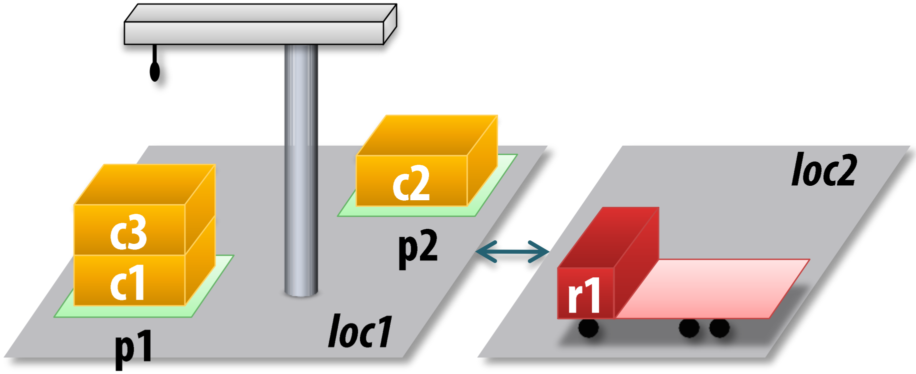
A **pile** is a stack of containers – at the bottom, there is a **pallet**

A **location** is an area that can be reached by a single crane. Can contain several piles, at most one robot.

Objects 3: Types

- Most planners (but not all) support distinct **object types**
  - Part of the domain – identical for all problem instances
    - (**define** (**domain** dock-worker-robots)
      (:**requirements** :**strips** :**typing**)
      (:**types**
      location ; there are several connected locations in the harbor
      pile ; attached to a location, holds a pallet + a stack of containers
      robot ; holds at most 1 container, only 1 robot per location
      crane ; belongs to a location to pickup containers
      container)
)
  - **Subtypes** can be defined
    - (:**types**
      ; containers and robots
      ; are movable objects
      container robot – movable
      …)

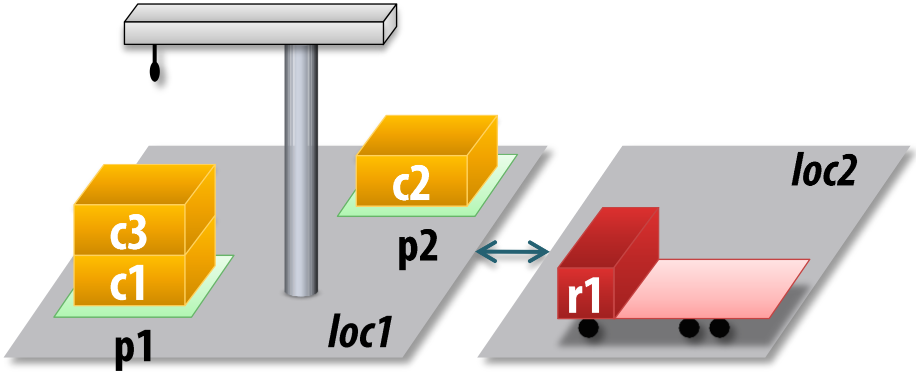
Predefined ”topmost supertype”: object



Objects 4: Object Definitions

- **Objects** are generally specified in the **problem instance**

  -
    ```
    (define  (problem dwr-problem-1)
             (:domain dock-worker-robot)
             (:objects
                      r1          – robot
                      loc1 loc2   – location
                      k1          – crane
                      p1 p2       – pile
                      c1 c2 c3 pallet – container)
    ```



Objects 5: ”Constants”

- But PDDL also supports ”constants” declared in the domain

```
(define (domain woodworking) (:requirements :typing)
 (:types
   acolour awood woodobj machine surface treatmentstatus aboardsize apartsize – object
   highspeed-saw glazer grinder immersion-varnisher planer saw spray-varnisher – machine
   board part - woodobj)
 (:constants
        verysmooth smooth rough – surface
        varnished glazed untreated colourfragments – treatmentstatus
        natural – acolour
        small medium large - apartsize)

(:action do-immersion-varnish
  :parameters (?x - part ?m - immersion-varnisher ?newcolour - acolour ?surface - surface)
  :precondition (and
       …
       (treatment ?x untreated))
  :effect (and
       (not (treatment ?x untreated))    (treatment ?x varnished)
       (not (colour ?x natural))         (colour ?x ?newcolour)) …)
```

Define *once* – use in *all* problem instances

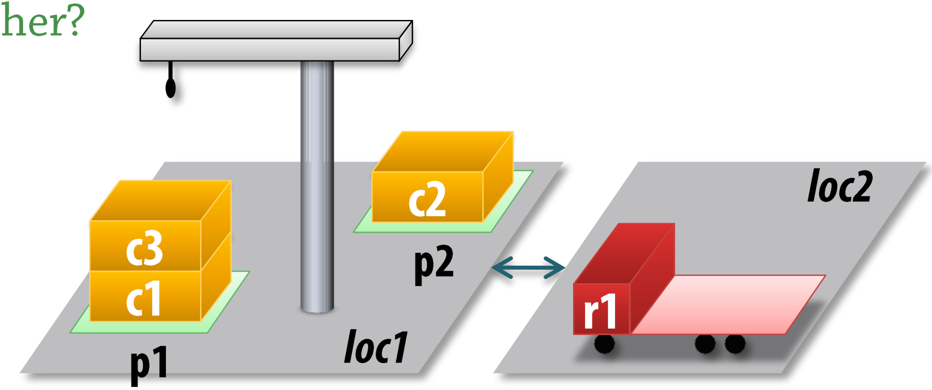
Use in the domain definition as well!

Properties of the World

Properties

- We are often interested in **properties** of objects
  - **Location** of a **card**, whether we **have a picture** of a **building** or not, …

- The **classical representation** uses boolean **predicates**
  - Any **fact** is represented as a (logical) **atom**: Predicate + arguments
  - Example: Some **fixed** predicates (cannot be changed by actions)
    - ;; Can you move from one loc to another?
      adjacent(location, location)
    - ;; Is the pile in the given location?
      attached(pile, location)
    - ;; Is the crane in the given location?
      belong(crane, location)



Predicates: Dynamic

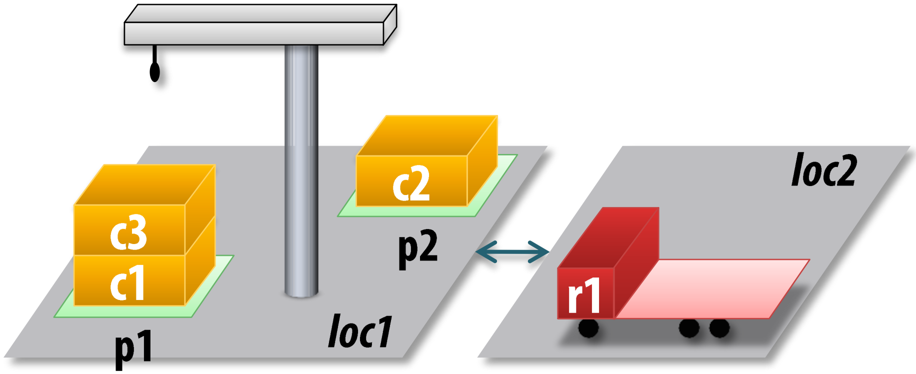
- **Dynamic** predicates can change (through actions)
  - **at**(robot, location) – the given robot is at the location
  - **occupied**(location) – there is a robot (truck) at the location
  - **loaded**(robot, container) – the robot is loaded with the given container
  - **unloaded**(robot) – the robot has no container
  - **holding**(crane, container) – the crane is holding the given container
  - **empty**(crane) – the crane is not holding anything
  - **in**(container, pile) – the container is somewhere in the given pile
  - **on**(container, container) – the first container is on the second one
  - **top**(container, pile) – the container is at the top of the given pile
  - **top**(pallet, pile) – the pallet is at the top (the pile is empty)

Atom: predicate symbol + args,
empty (crane1)

Literal: atom or ¬atom

Ground expression: No variables

Unground expression: Has variables



Predicates: In PDDL

- PDDL: **classical** (**predicate**) *representation*, Lisp-like *syntax*

Variables are prefixed with "?"

```
(define (domain dock-worker-robots)
  (:requirements ...)
  (:predicates
    (adjacent  ?l1  ?l2 - location)          ; location ?l1 is adjacent to ?l2
    (attached  ?p - pile ?l - location)      ; pile ?p attached to location ?l
    (belong    ?k - crane ?l - location)     ; crane ?k belongs to location ?l

    (at        ?r - robot ?l - location)     ; robot ?r is at location ?l
    (occupied  ?l - location)                ; there is a robot at location ?l
    (loaded    ?r - robot ?c - container )   ; robot ?r is loaded with container ?c
    (unloaded ?r - robot)                    ; robot ?r is empty

    (holding   ?k - crane ?c - container)    ; crane ?k is holding container ?c
    (empty     ?k - crane)                   ; crane ?k is empty

    (in        ?c - container ?p - pile)     ; container ?c is within pile ?p
    (top       ?c - container ?p - pile)     ; container ?c is on top of pile ?p
    (on        ?k1 ?k2 - container )         ; container ?k1 is on container ?k2
  )
```

Predicates: With Type Hierarchies

- Note the many predicates with similar meaning!
  - Due to the example's **flat type structure**
  - Could also use **type hierarchies** – in most planners

```
(define (domain dock-worker-robots)
  (:requirements ...)
  (:types      robot crane container pile – thing
               location
  (:predicates
Before:    (attached  ?p - pile ?l - location)      ; pile ?p attached to location ?l
           (belong    ?k - crane ?l - location)     ; crane ?k belongs to location ?l
           (at        ?r - robot ?l - location)     ; robot ?r is at location ?l

Now:       (at        ?t – thing ?l - location)     ; thing ?t is at location ?l

  )
```

States, Initial States, Goal States

States 1: Classical Representation

We know all **predicates** that exist, and their argument types: **adjacent**(location, location), …

We know a set of **objects** for each type, specified in the domain + problem

We assume the objects are **unique**: robot1 != robot3, since their names are different

We assume **domain closure**: No other objects exist than the ones specified in the domain + problem instance

We can calculate all *ground atoms*

adjacent(loc1,loc1)
adjacent(loc1,loc2)
…
adjacent(loc7,loc7)
attached(pile1,loc1)
…

These are the *facts* to keep track of!

We can infer all (relevant) states!

A classical state should define which ground atoms are true

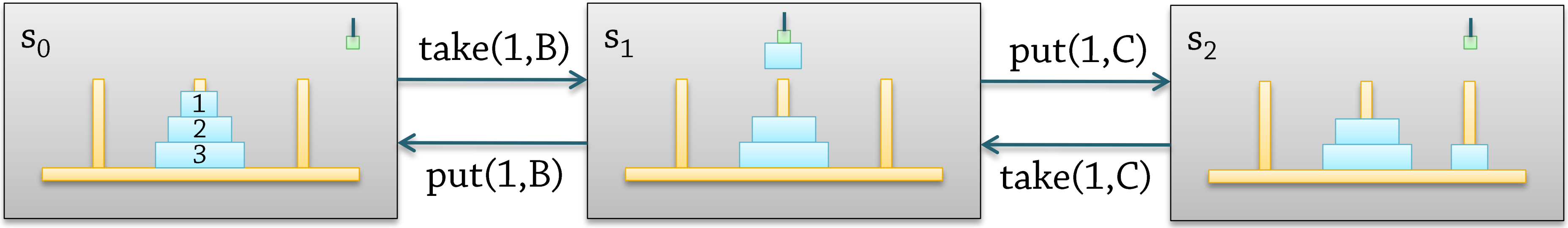
➔ A state *is* an assignment of true/false to all ground atoms!

Number of states: $2^{\text{number of atoms}}$

States 2: Internal Structure

- So: **Classical states** have **internal structure**!
  - Instead of knowing *only* ”this is state s0”,
    we know ”this is a state where top(1) is true, top(2) is false, …”

We can see
the *difference*
between two states



s_0
top(1) is true
top(2) is false
top(3) is false
on(1,1) is false
on(1,2) is true
on(1,3) is false
on(2,1) is false
…

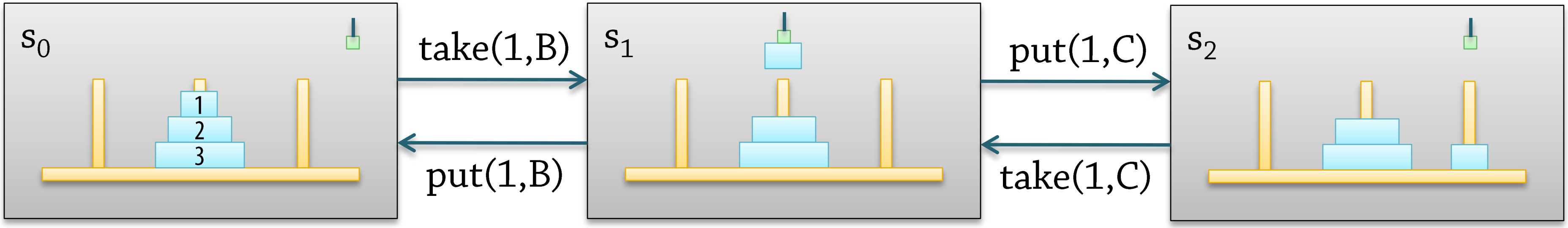
s_1
top(1) is **false**
top(2) is **true**
top(3) is false
on(1,1) is false
on(1,2) is **false**
on(1,3) is false
on(2,1) is false
…

Structure is essential!

We will see later how planners make use of structured states…

States 3: Classical Representation

- **<u>Efficient representation</u>** for a **<u>single state</u>**:
  - Specify **<u>which atoms are true</u>**
    - All other atoms have to be false – what else would they be?
  - ➔ A **<u>classical state</u>** is a **<u>set</u>** of all **<u>variable-free atoms</u>** that are true
    - s_0 = { on(1,2), on(2,3), in(1,B), in(2,B), in(3,B), top(1), bot(3) }



s_0
top(1) in(1,B)
on(1,2) in(2,B)
on(2,3) in(3,B)
bot(3)

s_1
hold(1)
top(2) in(2,B)
on(2,3) in(3,B)
bot(3)

s_2
top(1) in(1,C)
top(2) in(2,B)
on(2,3) in(3,B)
bot(3)
bot(1)

top(1) ∈ s_0 ➔ top(1) is true in s_0
top(2) ∉ s_0 ➔ top(2) is false in s_0

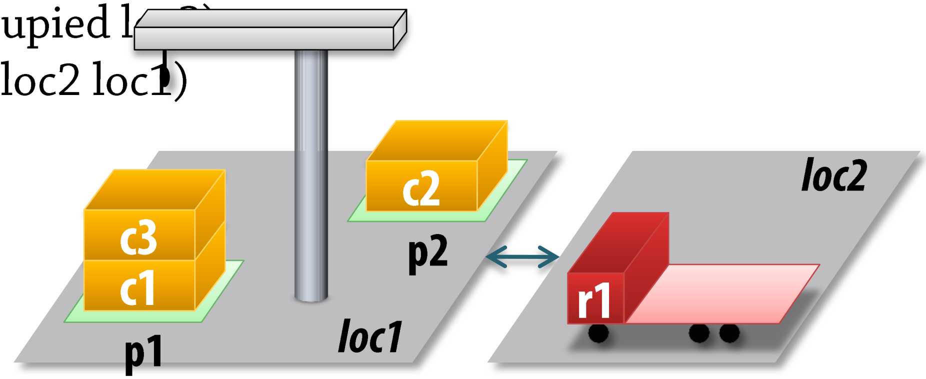
Why not store all ground atoms
that are **<u>false</u>** instead?

States 4: Initial State in PDDL

- Classical planning ➔ *complete information*: **<u>only one possible initial state</u>**
  - Initial state specification in PDDL:
    - Again, only specify atoms that are **<u>true</u>**
    - (**define** (**problem** dwr-problem-1)
      (:**<u>domain</u>** dock-worker-robot)
      (:**<u>objects</u>** …)
      (:**<u>init</u>**
      (attached p1 loc1) (in c1 p1) (on c1 pallet) (in c3 p1) (on c3 c1) (top c3 p1)
      (attached p2 loc1) (in c2 p2) (on c2 pallet) (top c2 p2)
      (belong crane1 loc1) (empty crane1)
      (at r1 loc2) (unloaded r1) (occupied l
      (adjacent loc1 loc2) (adjacent loc2 loc1)
)
)

Complete within the model: We know everything about those predicates and objects we have specified…

Lisp-like notation again: (attached p1 loc), not attached(p1,loc)



States 5: Goal States

- Classical planning ➔
  <u>many possible goal states</u>
  - Ex: We want containers 1—5 in pile 3, but don't care about the order

<u>Formal model</u>:

Arbitrary set of goal states $S_g \subseteq S$:
Must end up in one of these states

$S_g = \{s_{10}, s_{200}, s_{201}, s_{202}, s_{307}, \ldots\}$

<u>Classical representation</u>:

Arbitrary set of *ground goal literals:*
Must end up a state satisfying these

$g = \{ in(c1,p3), \ldots, in(c5,p3), \neg foo\}$
(adds structure to goals)

Not identical in expressivity!

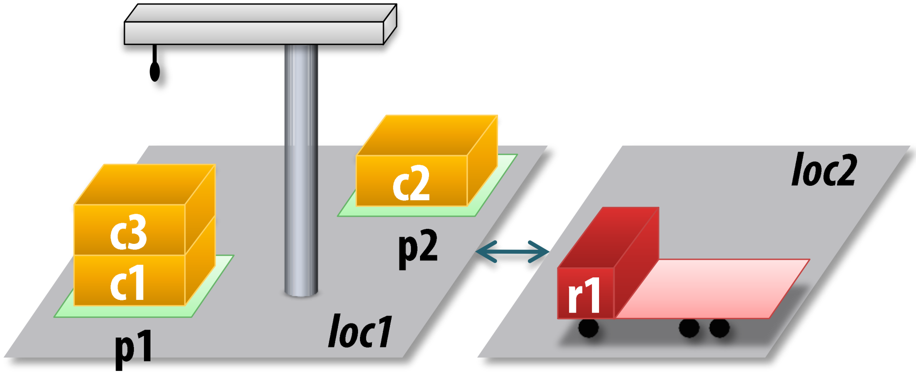
A set of goal literals cannot express arbitrary disjunctions:
All states where "in(c1,p3) or in(c1,p4)" is true

States 6: Classical Goals in PDDL

- PDDL uses a **goal formula**
  - Some planners: **Conjunctions** of **positive literals** of atoms
  - Some planners: **Conjunctions** of **positive and negative literals**
  - Some planners: More expressive (allow disjunctions, etc.)
  - (**define** (**problem** dwr-problem-1)
    (:**domain** dock-worker-robot)
    (:**objects** …)
    (:**goal** (**and** (in c1 p2) (in c3 p2))))
    - Even with only conjunctions, we can easily "ignore" particular facts: We don't care where r1 is

A *non-classical* goal could include:

- Achieving a goal in a certain amount of time
- Visiting interesting states along the way / *not* visiting dangerous states
- …

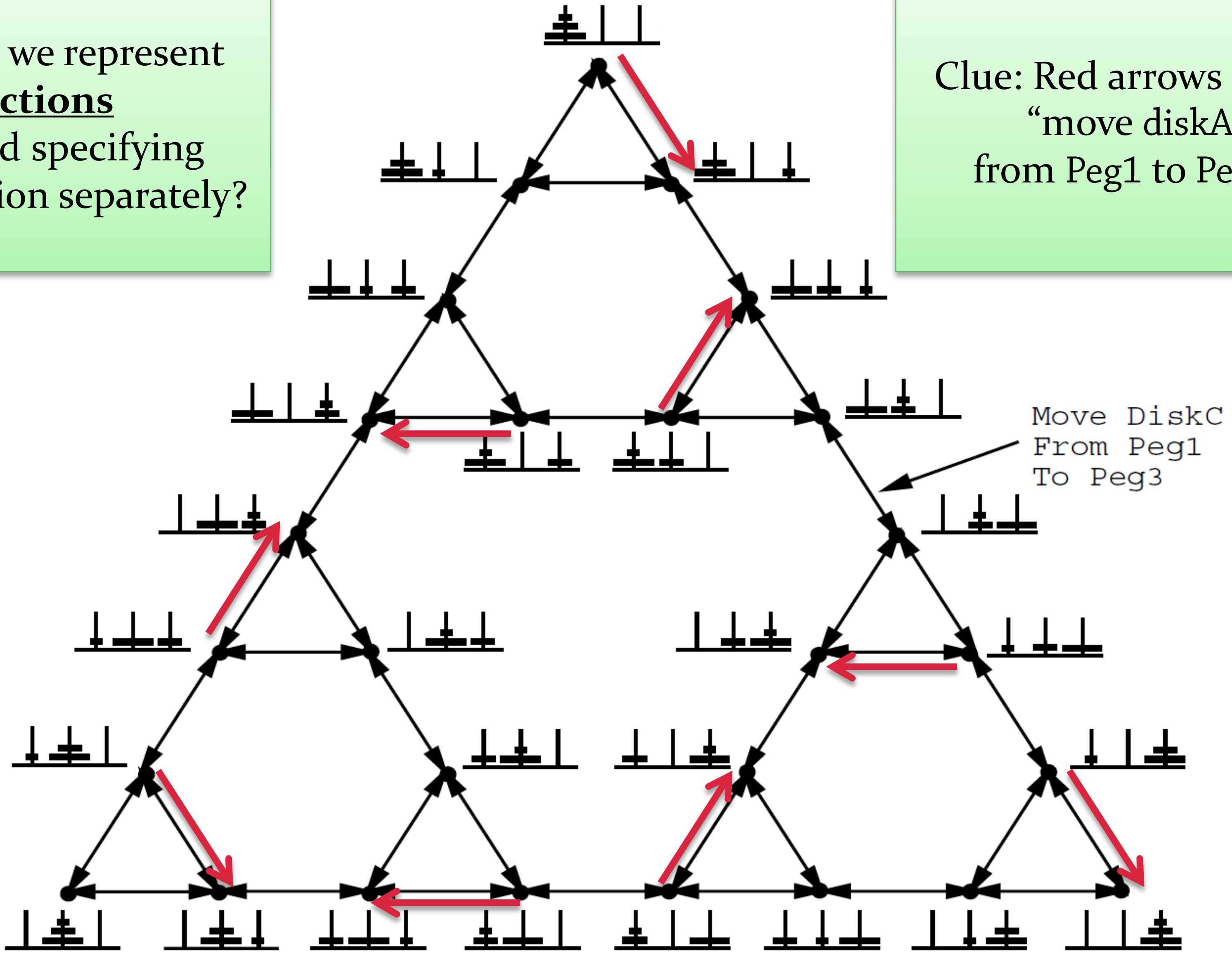


Operators and Actions

Actions in Classical Planning

How do we represent
actions
to avoid specifying
every action separately?

Clue: Red arrows mean
“move diskA
from Peg1 to Peg3”



Operators and Actions

- A parameterized **operator** *o* represents a **set** of actions!
    - ➔ Defines *many* state transitions
        - take(k, l, c, d, p):

;; crane *k* at location *l* takes container *c* off container *d* in pile *p*

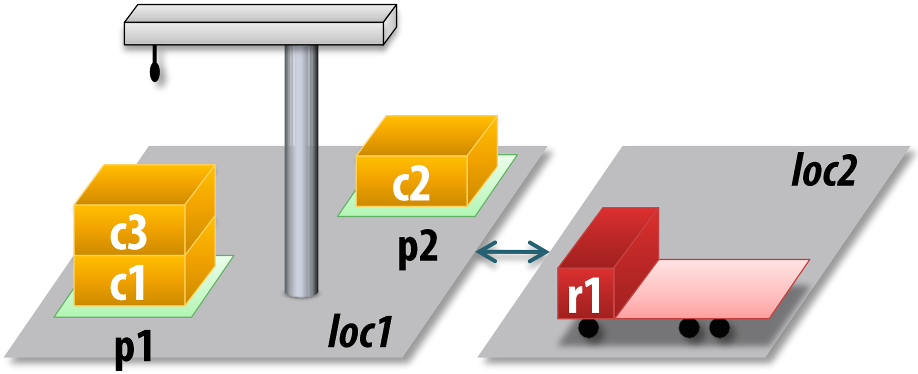
precond: belong(k,l), empty(k), attached(p,l), top(c,p), on(c,d)

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

name(o): Operator symbol + parameters

precond(o): **set** of **literals** (negated or positive atoms) that must hold in the state where the action is started

effects(o): **set** of **literals** (negated or positive atoms) that will be made to hold by the action



Notation

- Notation:
  - If a is an operator or action…
    - precond+(a) = { atoms that appear **positively** in a's preconditions }
    - precond–(a) = { atoms that appear **negatively** in a's preconditions }
    - effects+(a) = { atoms that appear **positively** in a's effects }
    - effects–(a) = { atoms that appear **negatively** in a's effects }
  - Example:
    - take(*k*, *l*, *c*, *d*, *p*):

      ;; crane *k* at location *l* takes container *c* off container *d* in pile *p*

      precond: belong(*k*,*l*), empty(*k*), attached(*p*,*l*), top(*c*,*p*), on(*c*,*d*)

      effects: holding(*k*,*c*), ¬empty(*k*), ¬in(*c*,*p*), ¬top(*c*,*p*), ¬on(*c*,*d*), top(*d*,*p*)

    - effects+(take(k,l,c,d,p)) = { holding(k,c), top(d,p) }
    - effects–(take(k,l,c,d,p)) = { empty(k), in(c,p), top(c,p), on(c,d) }

Negation disappears!

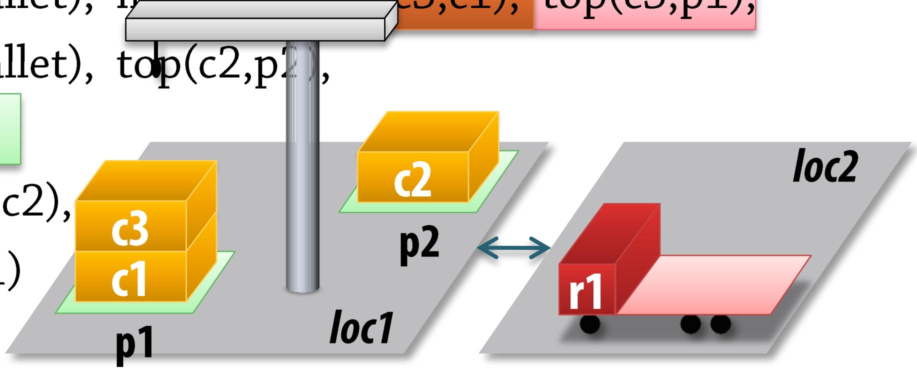
Applicable Actions

- An action *a* is **applicable** in a state *s*…
  - … if precond+(a) $\subseteq$ s and precond–(a) $\cap$ s = $\emptyset$
- Example:
  - take(crane1, loc1, c3, c1, p1):

    ;; crane1 at loc1 takes c3 off c1 in pile p1

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

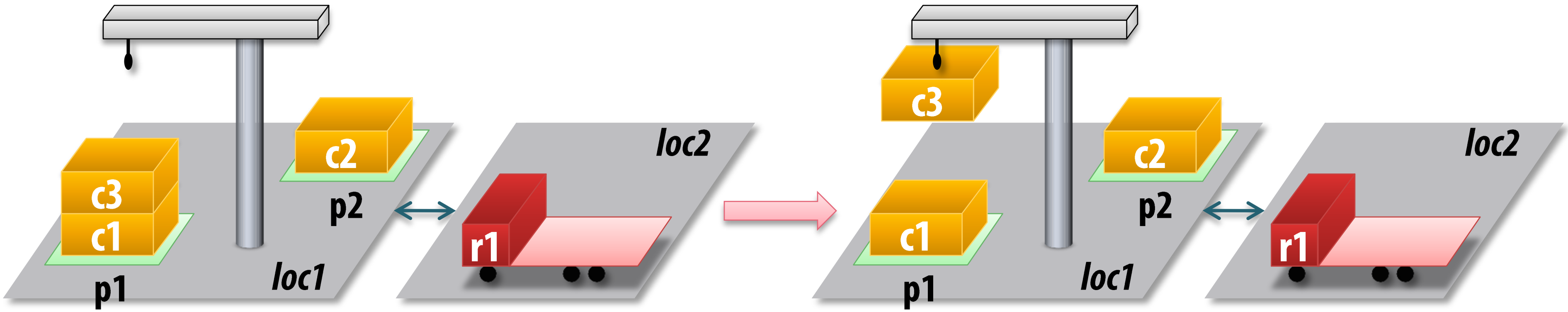
    effects: holding(crane1,c3), ¬empty(crane1),
    ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)
  - s1 = {
    attached(p1,loc1), in(c1,p1), on(c1,pallet), in(c3,p1), on(c3,c1), top(c3,p1),
    attached(p2,loc1), in(c2,p2), on(c2,pallet), top(c2,p2),
    belong(crane1,loc1), empty(crane1),
    at(r1,loc2), unloaded(r1), occupied(loc2),
    adjacent(loc1,loc2), adjacent(loc2,loc1)
    }



Result of Performing an Action

- **<u>Applying</u>** will **<u>add</u>** positive effects, **<u>delete</u>** negative effects
  - If a is applicable in s, then
    the new state is (s – effects–(a)) ∪ effects+(a)
  - This indirectly specifies the transition relation!

  - <u>take</u>(crane1, loc1, c3, c1, p1):
    ;; crane1 at loc1 takes c3 off c1 in pile p1
    <u>precond</u>: belong(crane1,loc1), empty(crane1),
    attached(p1,loc1), top(c3,p1), on(c3,c1)
    <u>effects</u>: holding(crane1,c3), top(c1,p1),
    ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1)



Operators in PDDL

- **Operators** are called **actions** in PDDL, for some reason…

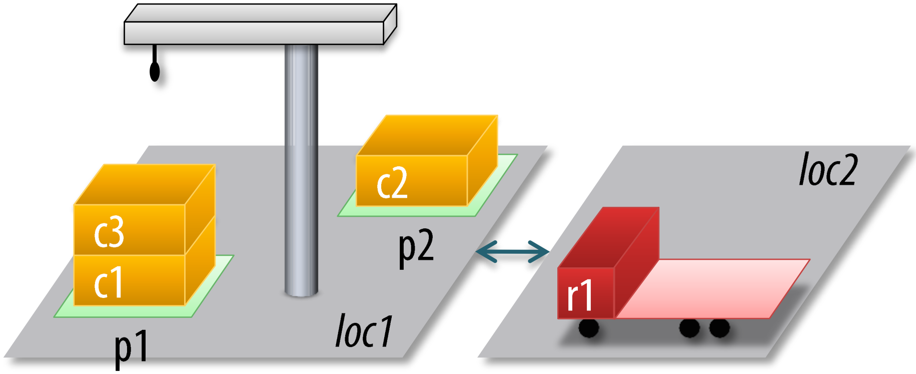
  -
    ```
    (define (domain dock-worker-robots) …
            (:action move
                    :parameters     (?r – robot
                                     ?from ?to - location)

                    :precondition (and (adjacent ?from ?to)
                                       (at ?r ?from)
                                       (not (occupied ?to)))

                    :effect (and (at ?r ?to) (not (occupied ?from))
                            (occupied ?to) (not (at ?r ?from))
    ```

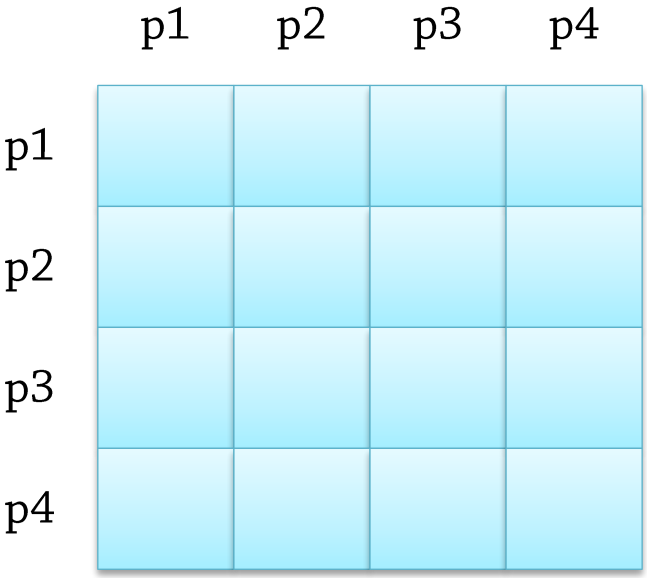
Written as logical conjunctions instead of sets! PDDL supports more expressive preconds and effects than the pure classical representation (but not all planners do).

With STRIPS expressivity, you **must** use a simple conjunctive precondition.



Repeating Arguments

- Warning: Repeating Arguments
  - Some planners **<u>refuse</u>** to **<u>use the same argument</u>** twice in an action
  - Avoids trying pointless actions such as move(robot, locA, locA)
    - No point in moving to the same location
  - But there are also cases where you **<u>want</u>** the same argument
    - Represent coordinates in a grid as p1, p2, p3, p4, …
    - Why does the planner never do move-to(p2,p2) in my 2-dimensional grid???
  - Possible solution: Duplicating objects
    - move-to(p2,q2)



What Is a Precondition?

- Usual assumption in **domain-independent planning**:
  - Preconditions should have to do with *executability*, not *suitability*
    - Weakest constraints under which the action *can* be executed

take(crane1, loc1, c3, c1, p1):
;; crane1 at loc1 takes c3 off c1 in pile p1
precond: belong(crane1,loc1), empty(crane1),
attached(p1,loc1), top(c3,p1), on(c3,c1)
effects: holding(crane1,c3), top(c1,p1),
¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1),

These are *physical* requirements for taking a container!

- The *planner* chooses which actions are suitable, using heuristics (etc.)
- If you add explicit “suitability preconditions”,
you are in the realm of *domain-configurable planning*
  - ”Only pick up a container if there is a truck on which the crane can put it”
  - ”Only pick up a container if it *needs* to be moved according to the goal”

Plan Structure

Plan Structure

- **Assumption A5: Sequential plans**
  - No concurrency
  - No if-then conditions
  - …

Plan

- Move disk 1 from B to A
- Move disk 2 from B to C
- Move disk 1 from A to C
- …

A simple sequence!

Plans and Solutions

- There are some disagreements about **terminology**...
  - In the book: **Any** sequence of actions $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ is a **plan**
    - Does not have to be executable
  - If it **is** executable, it is called... an **executable plan**!
    - There exist states $s_0, s_1, \ldots, s_n$ such that
    - $\gamma(s_0, a_1) = \{ s_1 \}$
    - $\gamma(s_1, a_2) = \{ s_2 \}$
    - …
    - $\gamma(s_{n-1}, a_n) = \{ s_n \}$
    - Some others only consider executable plans to be plans
  - A plan is a **solution** if it is executable
    and ends in a state s_n satisfying the goal

In the exam, we will make clear which variation we mean!

Representations Revisited: Alternatives to the Classical Representation

Representations

- **<u>First-order</u>** vs. **<u>propositional</u>** representations:
  - ”First-order” = we explicitly model **<u>objects</u>**
    - Compare:

Propositional logic:	facts are propositions,	p, q, r
First-order logic:	facts are atoms,	on(A,B), at(rob1, loc44)

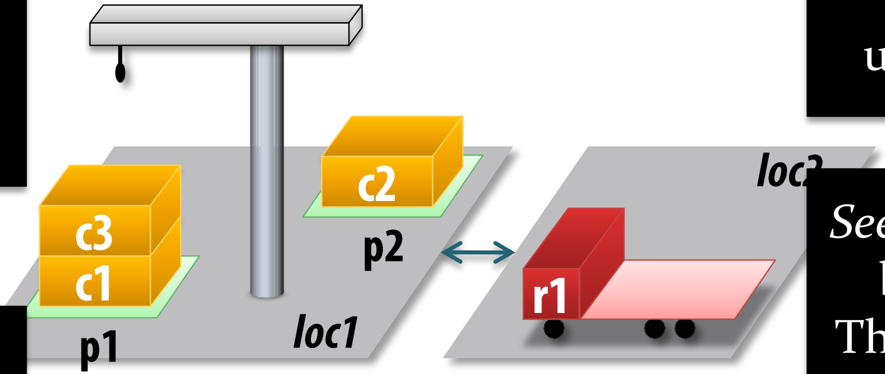
  - The **<u>set-theoretic</u>** representation is *propositional*
    - Useful for analysis, less important for practical planners
  - The **<u>classical</u>** and **<u>state-variable</u>** representations are **<u>first-order</u>**

Classical and State-Var Representation

- **Classical planning** with **classical representation**
  - A state defines the values of **logical atoms** (boolean)
    - adjacent(location, location) – can you go directly from one loc to another?
    - attached(pile, location) – is the pile in the given location?

Can be *wasteful*:
Can represent a pile
being in *many* locations,
which never happens

Can be *convenient*,
space-efficient
➔ often used internally!



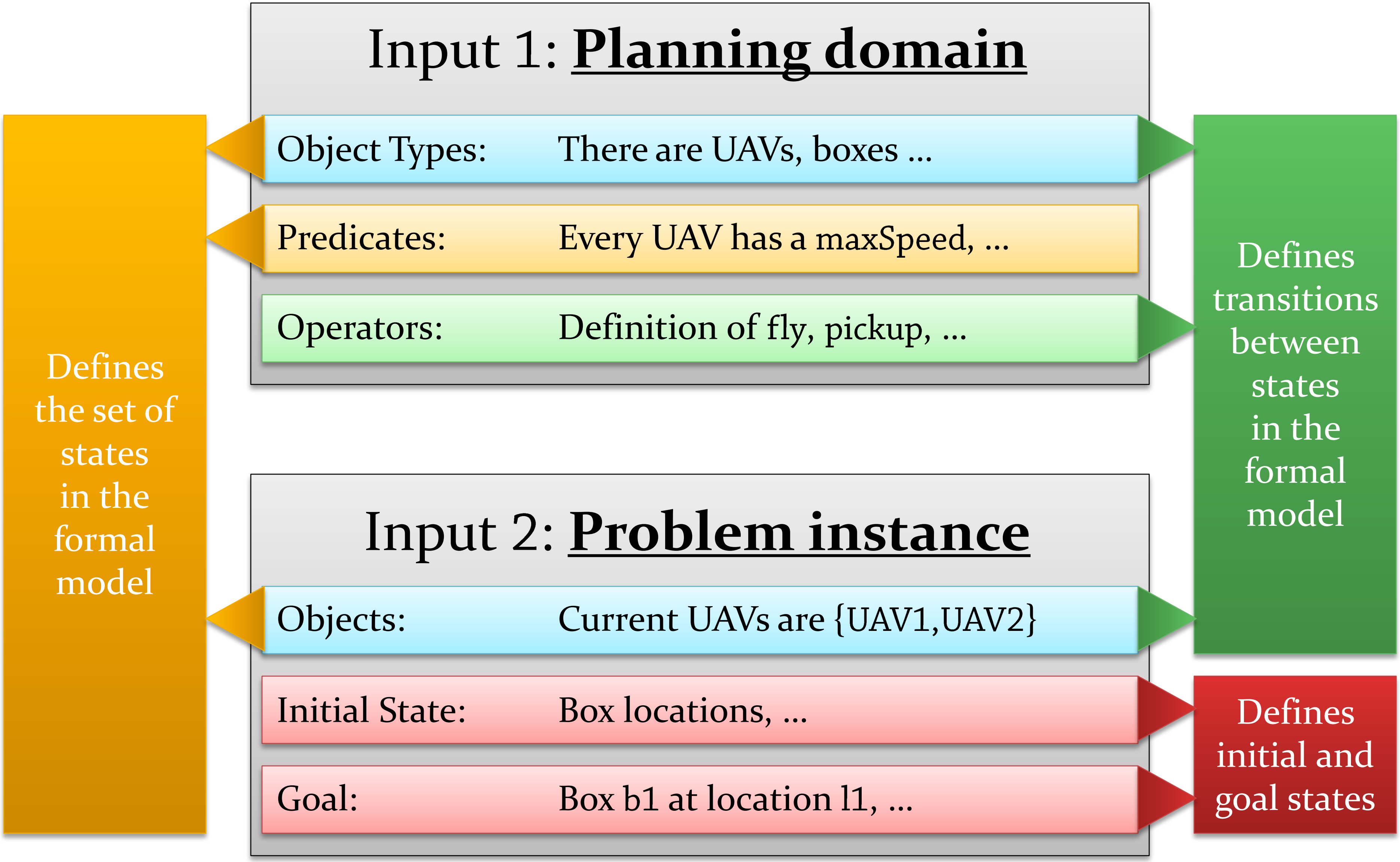
We will continue
using classical rep!

Seems more powerful,
but is equivalent:
This slide exemplifies
how to translate
back and forth…

- **Classical planning** with **state-variable representation**
  - A state defines the values of **arbitrary state variables**
    - boolean adjacent(location, location) ;; still boolean!
    - location ploc(pile) ;; a pile is in exactly one location

Formal World Model vs. Problem Statement in a Representation Language

Domains and Problem Instances



Defines the set of states in the formal model

Input 1: **Planning domain**

Object Types:	There are UAVs, boxes …
Predicates:	Every UAV has a maxSpeed, …
Operators:	Definition of fly, pickup, …

Input 2: **Problem instance**

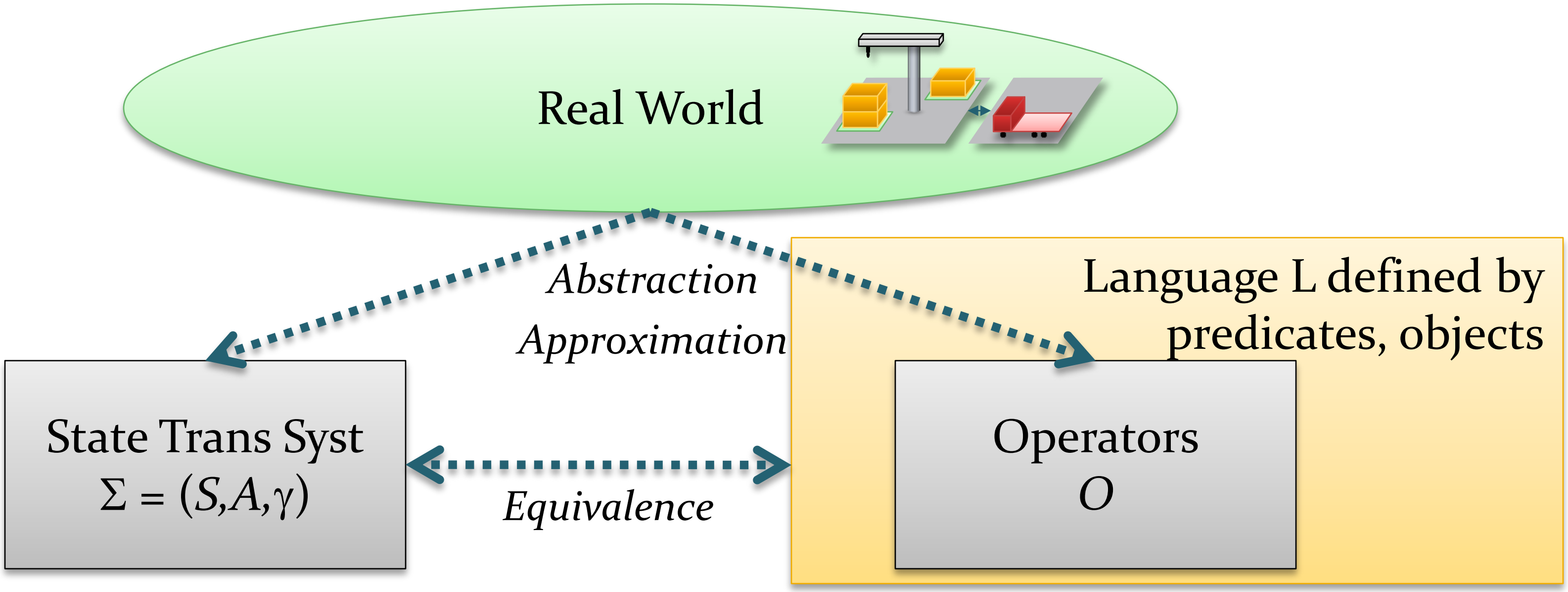
Objects:	Current UAVs are {UAV1,UAV2}
Initial State:	Box locations, …
Goal:	Box b1 at location l1, …

Defines transitions between states in the formal model

Defines initial and goal states

Problems and Problem Statements 1

Formally, the classical representation uses a first-order language L



Real World

Abstraction

Approximation

Language L defined by predicates, objects

State Trans Syst
$\Sigma = (S,A,\gamma)$

Equivalence

Operators
O

States have no internal structure

Actions are unstructured symbols

State transitions are unstructured (γ specified by state / action symbols)

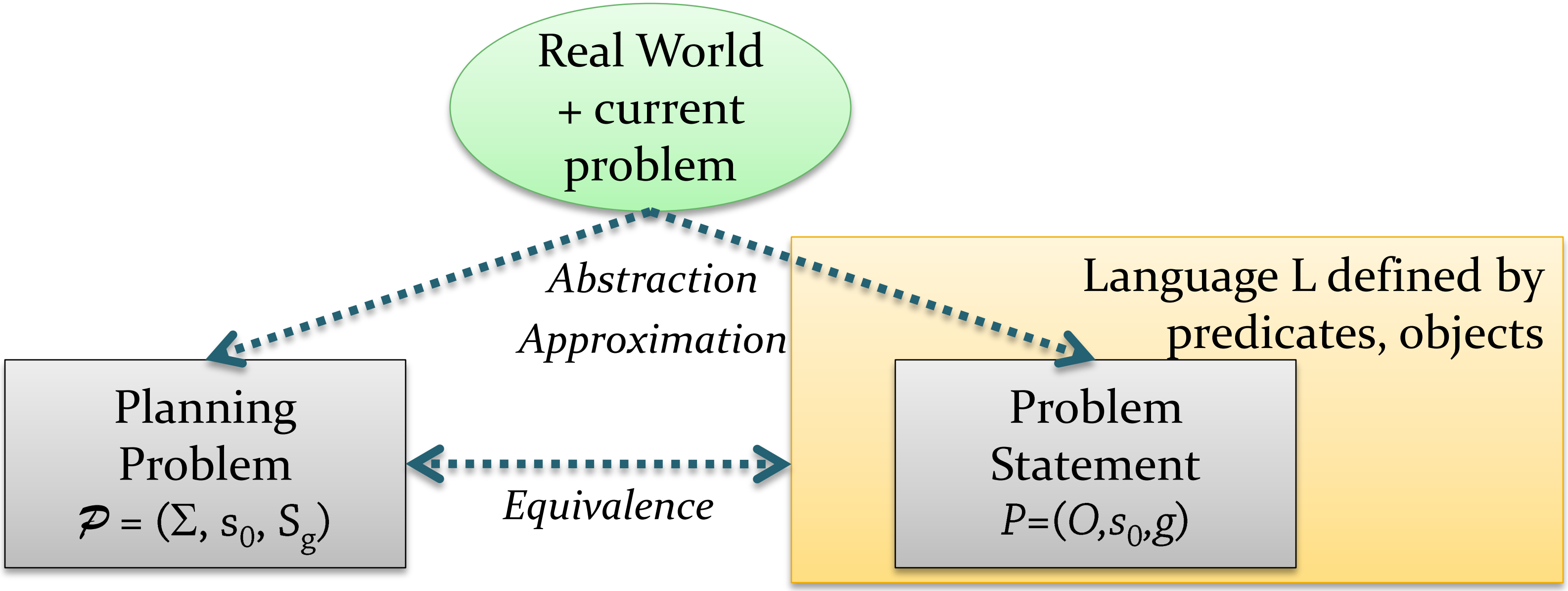
States are **sets of atoms**, induced by the predicates and objects in L

Operators are **structured**, have preconditions and effects

Each operator specifies part of γ, through its preconds and effects

Problems and Problem Statements 2

A planning **problem** also requires an initial state and a goal



Real World
+ current
problem

Abstraction

Approximation

Language L defined by
predicates, objects

Planning
Problem
$\mathcal{P} = (\Sigma, s_0, S_g)$

Equivalence

Problem
Statement
$P=(O,s_0,g)$

Specifies the **ID** of the initial state:
s_0

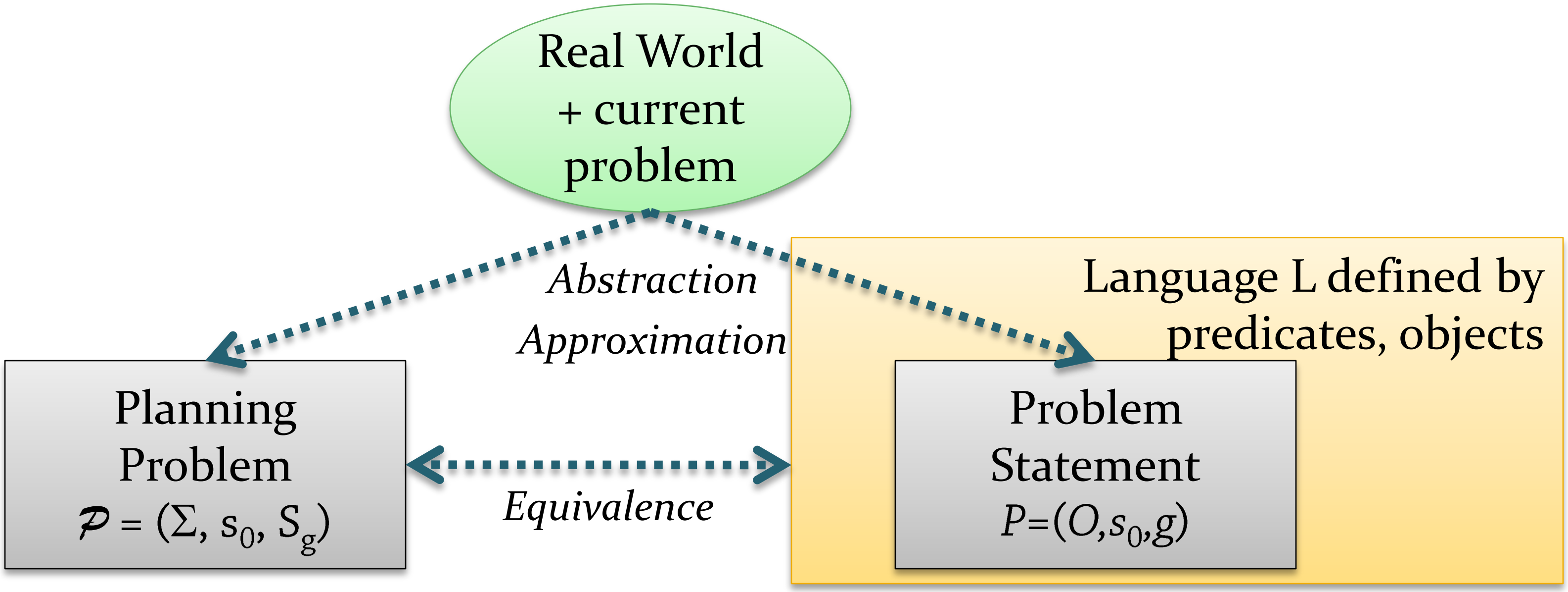
Specifies a set of possible goal **states**:
$S_g = \{ s0, s1, s2, s20, s21, s22, s4912, \dots \}$

Specifies the **true atoms** in the init state:
{ attached(p1,loc1), in(c1,p1), … }

Specifies a set of **literals** that must hold:
g = { in(c1,p2), in(c2,p2) , … }
Often seen as a conjunctive **goal formula**

Problems and Problem Statements 3

Difference in **size**!



Real World
+ current
problem

Abstraction

Approximation

Language L defined by
predicates, objects

Planning
Problem
$\mathcal{P} = (\Sigma, s_0, S_g)$

Equivalence

Problem
Statement
$P=(O,s_0,g)$

Trillions of states in $\Sigma = (S,A,\gamma)$
would be a rather small
planning **problem**

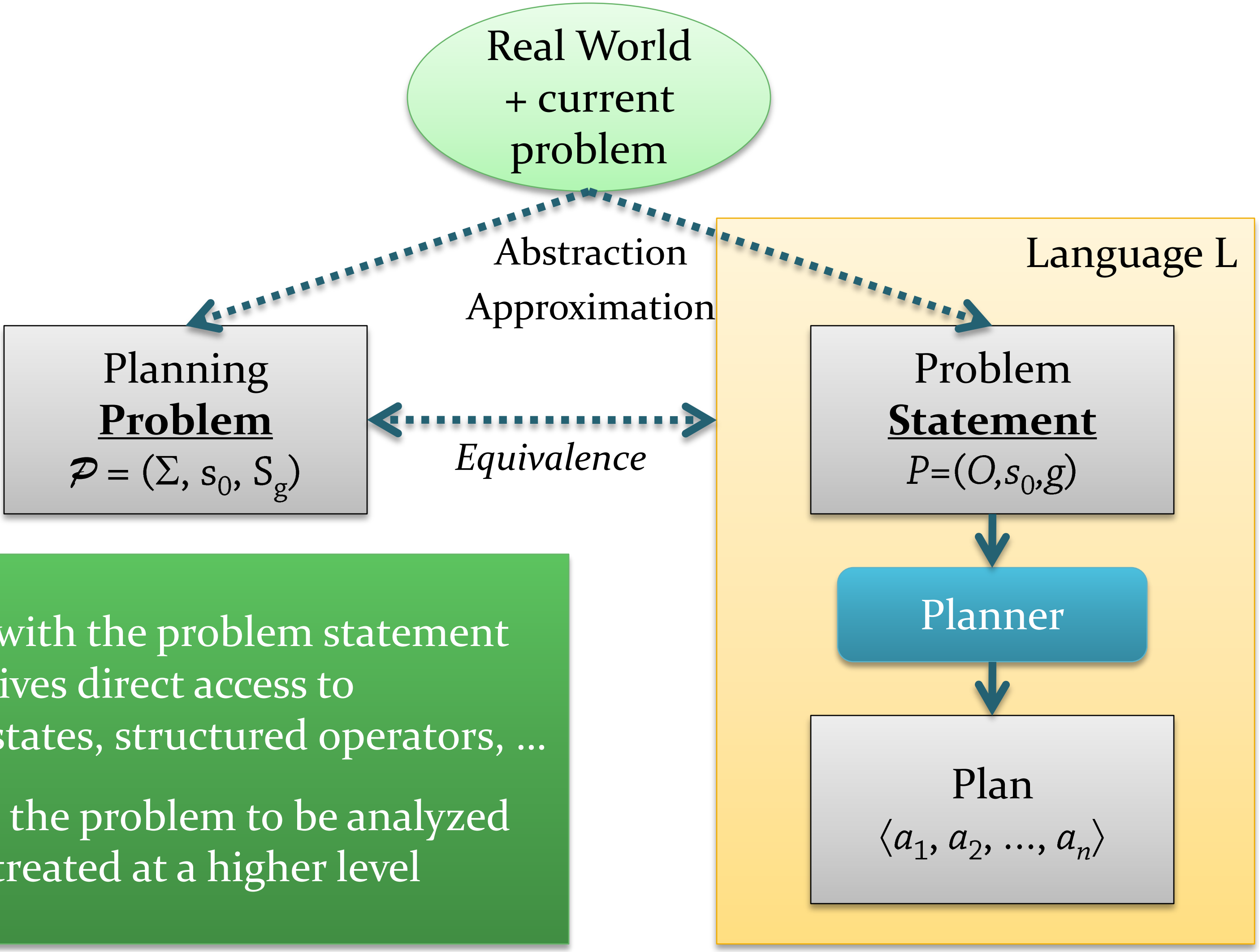
Trillions of state transitions in γ
would also correspond to a small
planning problem

Thousands of constants and predicates
in L would be a rather large
classical planning problem **statement**

Hundreds of operators
would correspond to a very large
classical planning problem statement

Problems and Problem Statements 4

Planning algorithms work with the **problem statement**!



Real World
+ current
problem

Abstraction

Approximation

Language L

Planning
Problem
$\mathcal{P} = (\Sigma, s_0, S_g)$

Equivalence

Problem
Statement
$P=(O,s_0,g)$

Planner

Plan
$\langle a_1, a_2, \ldots, a_n \rangle$

Working with the problem statement
gives direct access to
structured states, structured operators, …

→ Allows the problem to be analyzed
and treated at a higher level

Modeling Classical Planning Problems: Common Issues

Types in Untyped Planners

Untyped Domains

- Some planners lack support for **explicit types**
  - Constants are untyped, operators have untyped parameters, …
  - Consider an untyped operator in the DWR domain:
    - **take**(*k*, *l*, *c*, *d*, *p*): ;; crane *k* at location *l* takes container *c* off container *d* in pile *p*
      precond: belong(*k*,*l*), empty(*k*), attached(*p*,*l*), top(*c*,*p*), on(*c*,*d*)
      effects: holding(*k*,*c*), ¬empty(*k*), ¬in(*c*,*p*), ¬top(*c*,*p*), ¬on(*c*,*d*), top(*d*,*p*)
  - This is a valid instance of that action:
    - **take**(c3, crane1, r1, crane2, r2)
      ;; Container c3 at location crane1 takes robot1 off crane2 in pile robot2

So how do we ensure an *untyped* planner never uses that action?

Untyped Domains (2)

- Standard solution: **Preconditions** use **type predicates**
  - Ordinary predicates that happen to represent types:

    -
      ```
      (:predicates    (OBJ ?x)          (TRUCK ?x)     (LOCATION ?x)
                      (AIRPLANE ?x)     (CITY ?x)      (AIRPORT ?x)
                      (at ?x ?y)        (in ?x ?y)     (in-city ?x ?y))
      ```
  - Initialized in the problem instance:

    -
      ```
      (:init (OBJ package1) (OBJ package2) …
      ```
  - Used as part of preconditions:

    -
      ```
      (:action load-truck
        :parameters (?x ?t ?l)
        :precondition
          (and (OBJ ?x) (TRUCK ?t) (LOCATION ?l)
          (at ?t ?loc) (at ?x ?l))
        :effect …
      ```
  - Since we don't have ”real” types:
    - **load-truck**(truck2, truck3, truck4) is still a valid action
    - But that doesn't matter: Its preconditions can never be satisfied!

Untyped Domains (3)

- But the DWR example didn't have type predicates!
    - **take**(*k*, *l*, *c*, *d*, *p*): ;; crane k at location l takes container c off container d in pile p
      precond: belong(*k*,*l*), empty(*k*), attached(*p*,*l*), top(*c*,*p*), on(*c*,*d*)
      effects: holding(*k*,*c*), ¬empty(*k*), ¬in(*c*,*p*), ¬top(*c*,*p*), ¬on(*c*,*d*), top(*d*,*p*)

  - What's important: given args of the wrong type, the precondition is false!
    - The precondition requires belong(*k*,*l*)
    - This atom is only true if *k* is a crane
      - This is the case in the initial state (unless we get a "bad" problem instance…)
      - And no action modifies belong()

Finding the value of a property

Property Values 1

- Consider modeling a **<u>”drive” operator</u>** for a truck
  - ”Natural” parameters: The truck and the destination

    -
      ```
      (:action drive
               (:parameters ?t – truck ?dest – location)
               …
      )
      ```
  - ”Natural” effects:
    - The truck ends up at the destination: `(at ?t ?dest)`
    - The truck is no longer where it started: `(not (at ?t …???… ))`
  - How do you find out where the truck is **<u>before</u>** the action?
    - We can **<u>test</u>** whether a truck is at some **<u>specific</u>** location: `(at ?truck ?location)`
    - But there’s no term referring to ”**<u>the place</u>** where the truck started”: `(location-of ?truck)` does not exist

Property Values 2

- Standard solution:
  - Use another parameter to the operator

    -
      ```
      (:action drive
                :parameters (?t – truck ?from – location ?dest – location)
                …
      )
      ```
  - Bind that variable in the precondition
    - **:precond** (and … (at ?t ?from) …)
    - Can only apply those instances of the operator where?from *is* the current location of the truck
  - Now we can define the effects
    - The truck ends up at the destination: (at ?t ?dest)
    - The truck is no longer where it started: (not (at ?t ?from))

Counting

Counting 1

- We often need at least some ”primitive” support for **<u>counting</u>**
  - Elevator domain:
    - Which floor is an elevator at?
    - Which is the **<u>next</u>** floor?
    - Which is the **<u>previous</u>** floor?

  - Few planners support general numeric state variables

Counting 2

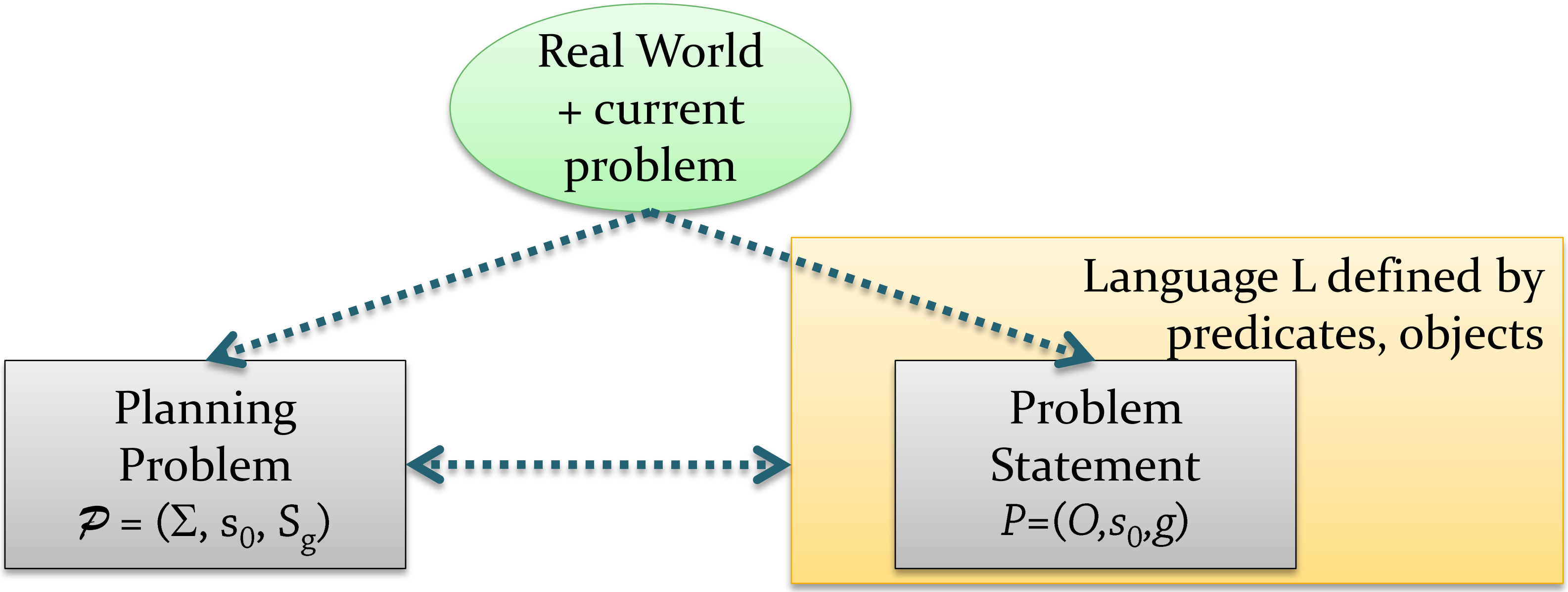
- Standard solution:
  - Create a **type** of ”pseudo-numbers”
    - (:**types** … num …)
  - Define a set of **value objects**
    - (:**objects** … n0 n1 n2 n3 n4 n5 n6 n7 – num)
  - Define the **operations** you need – for example, find the next number
    - (:**predicates** … (next ?numA ?numB – num)
    - (:**init** … (next n0 n1) (next n1 n2) (next n2 n3) … (next n6 n7))
  - Use the value objects as if they were numbers
    - (:**action** move-up

```
          :parameters (?e – elevator   ?from ?to – num)
          :precondition   (and  (at ?elevator ?from)  ;; Where is the elevator?
                                (next ?from ?to) …)   ;; Now ”to” is the next number
          :effects        (and (not (at ?elevator ?from)) (at ?elevator ?to)))
```

There is no ”next” for n7
➔ Won’t be able to move up
from the top floor

Extensions to STRIPS Expressivity – and Workarounds

Extensions: To What?



Real World
+ current
problem

Language L defined by
predicates, objects

Planning
Problem
$\mathcal{P} = (\Sigma, s_0, S_g)$

Problem
Statement
$P=(O,s_0,g)$

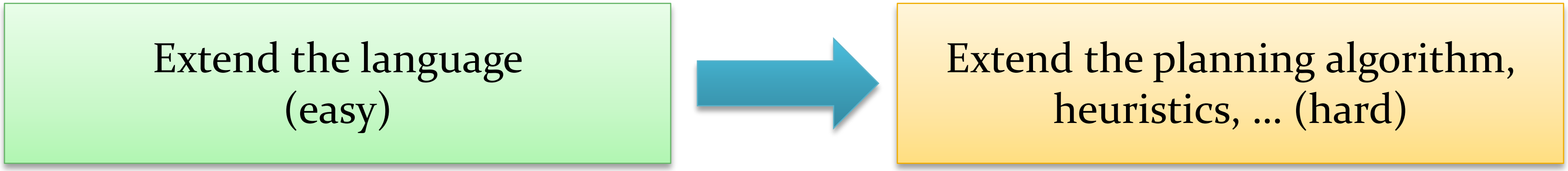
Conceptually simple,
but inconvenient to specify
and lacks detailed structure

More **convenient** and **structured**,
through the addition of new concepts:
Objects, predicates, operators,
precondition formulas, …

The **language** can be made even more convenient
without extending the **formal model**!

Extensions: Limitations

- Extending the language itself is comparatively simple
  - But planners use the representation format *directly*!



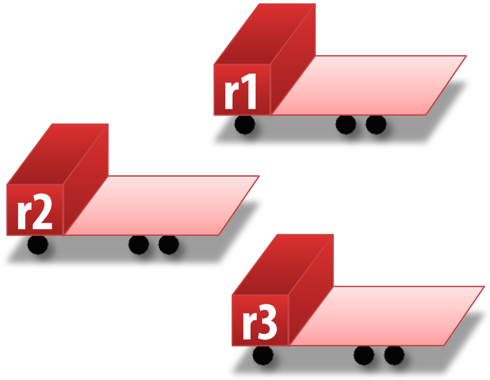
Extend the language (easy)

Extend the planning algorithm, heuristics, … (hard)

  - Many planners do implement such extensions
    - But in others, one needs *workarounds* to stay within standard STRIPS expressivity

Disjunctive Preconditions

Disjunctive Preconditions



- Suppose we have a number of ground robots
  - Can drive between ?from and ?to if there is a **road**, **or** the robot has **all-wheel-drive**
  - Disjunctive representation:
    - (:**requirements** **:disjunctive-preconditions** …)

```
(:action drive
          :parameters (?r - robot ?from ?to - location)
          :precondition (and
                    (or (road-between ?from ?to) (all-wheel-drive ?r))
                    (at ?r ?from))
          :effect (and (at ?r ?to) (not (at ?r ?from)) ))
```

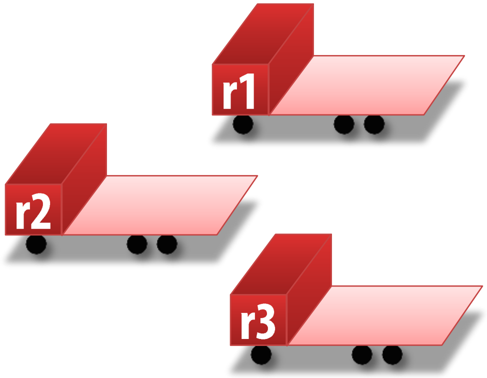
  - The precondition is no longer a **set of literals** that must hold!

Disjunctive Preconditions (2)

- **Disjunctive preconditions:**
  - Convenient
  - Easily supported by the **formal model**
    - Simply an easier way of specifying the state transition function
  - Not always supported by **planners**
    - Some **algorithms** are very efficient, but cannot handle disjunctions
    - Some **heuristics** are very informative, but cannot handle disjunctions
    - …
    - **Tradeoff** between convenience and efficiency!

Disjunctive Preconditions (3)

- Workaround 1: **Rewrite** the disjunction using two **distinct operators**



  -
    ```
    (:action drive-on-road
             :parameters (?r - robot ?from ?to - location)
             :precondition (and (road-between ?from ?to) (at ?r ?from))
             :effect (and (at ?r ?to) (not (at ?r ?from)) ))
    ```

  -
    ```
    (:action drive-all-wheel-drive
             :parameters (?r - robot ?from ?to - location)
             :precondition (and        (all-wheel-drive ?r)
                                       (not (road-between ?from ?to)
                                       (at ?r ?from))
             :effect (and (at ?r ?to) (not (at ?r ?from)) ))
    ```

Why should we have this?

- Any problems?

What about the condition $(a \lor b \lor c \lor d) \land (e \lor f \lor g \lor h)$?

Disjunctive Preconditions (4)

- Workaround 2: use a **<u>different domain model</u>**
  - Add a predicate: (can-drive-between ?robot ?from ?to)
    - Specify its value explicitly in the initial state
    - **<u>Redundant</u>** – but planners can use it efficiently!

- Planners could:
  - Directly and efficiently support disjunctions
    - Possible for some algorithms, some heuristics
  - Automatically rewrite into multiple operators
    - Could lead to inefficient planning,
without any indication of **<u>which</u>** constructs are inefficient
  - Disallow disjunctions
    - Encourages writing another domain model – might be more efficient
    - Can still use external rewriting tools

Quantified Preconditions

Quantified Preconditions

- Quantifiers in preconditions can be convenient
  - To drive a car, all doors must be closed

    -
      ```
      (:requirements :universal-preconditions)
      (:action drive
              (:parameters ?car – car ?from ?to – location)
              (:precondition
                      (forall (?door – door)
                              (implies (belongs ?door ?car) (closed ?door)))))
      ```
  - Can be transformed to a conjunction by expanding the quantifier
    - Suppose we have 4 doors: { d1, d2, d3, d4 }

    -
      ```
      (:precondition
              (and (implies (belongs d1 ?car) (closed d1))
                   (implies (belongs d2 ?car) (closed d2))
                   (implies (belongs d3 ?car) (closed d3))
                   (implies (belongs d4 ?car) (closed d4))))
      ```
    - Must know which doors we have (instance-specific!)
    - Suppose we have 100 cars, 400 doors…

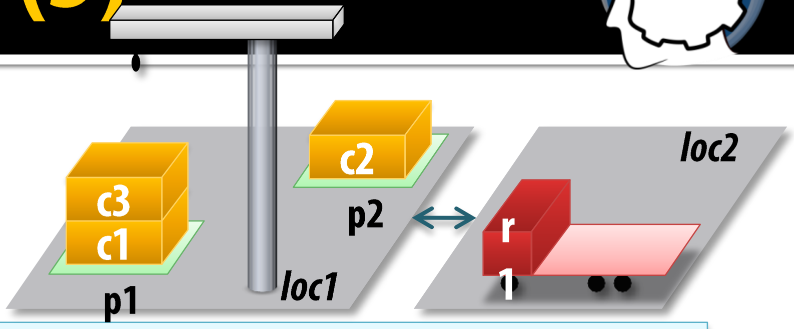
Quantified Preconditions (2)

- **Existential quantifiers** are also convenient
  - To drive a car, I must have **some** matching key

    -
      ```
      (:requirements :existential-preconditions)
      (:action drive
            (:parameters ?c – car ?from ?to – location)
            (:precondition
                  (exists (?k – key)
                        (and (have ?k) (matches ?k ?c)))))
      ```
  - Can be transformed to a **disjunction** by expanding the quantifier
    - Suppose we have 4 keys: { k1, k2, k3, k4 }

    -
      ```
      (:precondition
            (or   (and (have k1) (matches k1 ?c))
                  (and (have k2) (matches k2 ?c))
                  (and (have k3) (matches k3 ?c))
                  (and (have k4) (matches k4 ?c))))
      ```
    - Could then transform this disjunction into multiple operators…
    - Again, the domain can be modeled differently: (have-key-matching ?c)

Quantified Preconditions (3)



- Alternative workarounds exist
  - Introduce **redundant predicates**
    - Dock Worker Robots:

(:**predicates**
(**at** ?r - robot ?l - location) ; **robot ?r** is at location ?l
(**occupied** ?l - location) ; **there is** a robot at location ?l
…)

    - Where (**occupied** ?loc) is the same as (**exists** (?r – robot) (**at** ?r ?loc))!

  - **Update** redundant predicates when necessary
    - (:**action** move
:**parameters** (?r - robot ?from ?to - location)
:**precondition** (and (adjacent ?from ?to)
(at ?r ?from) (not (occupied ?to)))
:**effect** (and (not (at ?r ?from)) (at ?r ?to)
(not (occupied ?from)) (occupied ?to)))

Corresponds to
(not (exists (?r2 – robot))
(at ?r2 ?to))

Universal and Conditional Effects

Universal and Conditional Effects

- If you drive a truck, all items in the truck should follow it
  - Example:

    -
      ```
      (:requirements :universal-preconditions :conditional-effects ...)
      (:action drive-truck
         :parameters (?truck - truck ?loc-from ?loc-to – location ?city - city)
         :precondition (and    (at ?truck ?loc-from)
                               (in-city ?loc-from ?city)
                               (in-city ?loc-to ?city))
         :effect (and   (at ?truck ?loc-to)
                        (not (at ?truck ?loc-from))
                        (forall (?x - obj)
                              (when   (in ?x ?truck)
                                      (and (not (at ?x ?loc-from)) (at ?x ?loc-to))))))
      ```
  - In this model, if an object is initially at locationA:
    - (at ?obj locationA) **remains true** when the object is loaded into the truck
    - (at ?obj locationA) **becomes false** only when the truck drives away

Universal and Conditional Effects (2)

- If a planner does not support this:
  - **Quantifiers can be expanded** for a specific problem instance, as before

    -
      ```
      (forall (?x – obj) …) →
      (and      (when (in packageA ?truck) (…))
                (when (in packageB ?truck) (…))
                …
                (when (in packageX ?truck) (…))
      ```
  - **Conditional effects** can be expanded into **multiple operators**
    - One with precond (and … (in packageA truck) (in packageB truck) …)
    - One with precond (and … (not (in packageA truck)) (in packageB truck) …), and so on

Works – but can be inefficient!

Universal and Conditional Effects (3)

- Sometimes you can use workarounds
  - Alternative model: A package in a truck is not **at** any location at all!
    - (**at** ?obj ?location) **removed** by load-package action, **before** driving
    - (**in** ?obj ?truck) added **instead**
  - Driving a truck **only moves the truck**
    - Packages are still **in** the same truck,
      at no location at all
    - No need for quantified conditional effects here
  - Unloading a package:
    - (in ?obj ?truck) removed
    - (at ?obj ?new-location) added

Quantified Goals

Quantified Goals

- **Quantified goals**:
  - **Universal** goals (all crates should be at their destinations) are simple
    - Expand into a conjunction
  - **Existential** goals seem more difficult
    - We defined a goal as a **set** of literals, **all** of which must be true

  - *How can we indirectly implement existential goals when only conjunctive goals are explicitly supported?*

  - Through new actions and predicates!
    - Suppose we have a goal: (**or** a b c d)
    - Add a new predicate "goal-achieved", which replaces the goal
    - Make the predicate false in the initial state
    - Add an operator:
      (:**action** finish (:**precondition** (**or** a b c d)) (:**effect** goal-achieved)))

Plan Quality and Action Costs

Plan Quality and Action Costs

- What is **plan quality**?
  - Could aim for *shorter* plans (fewer actions)
    - Reasonable in Towers of Hanoi
    - How to make sure your car is clean?

go to car wash | get supplies | wash car

go to car dealer | buy new car | shortest plan is best?

- Most **current** planners support **action costs**
  - Each action $a \in A$ associated with a cost $c(a)$
  - Plan quality measured in terms of total cost
  - Simple extension to the restricted state transition system!

Action Costs in PDDL

- PDDL: Specify requirements
  - (:**requirements** :**action-costs**)
- Specify a *numeric state variable* for the *total* plan cost
  - And possibly numeric state variables to *calculate* action costs

  -
    ```
    (:functions (total-cost)                              - number
                (travel-slow-cost  ?f1 - count ?f2 - count) - number
                (travel-fast-cost  ?f1 - count ?f2 - count) - number)
    ```

Built-in type supported by cost-based planners

- Specify the **initial state**

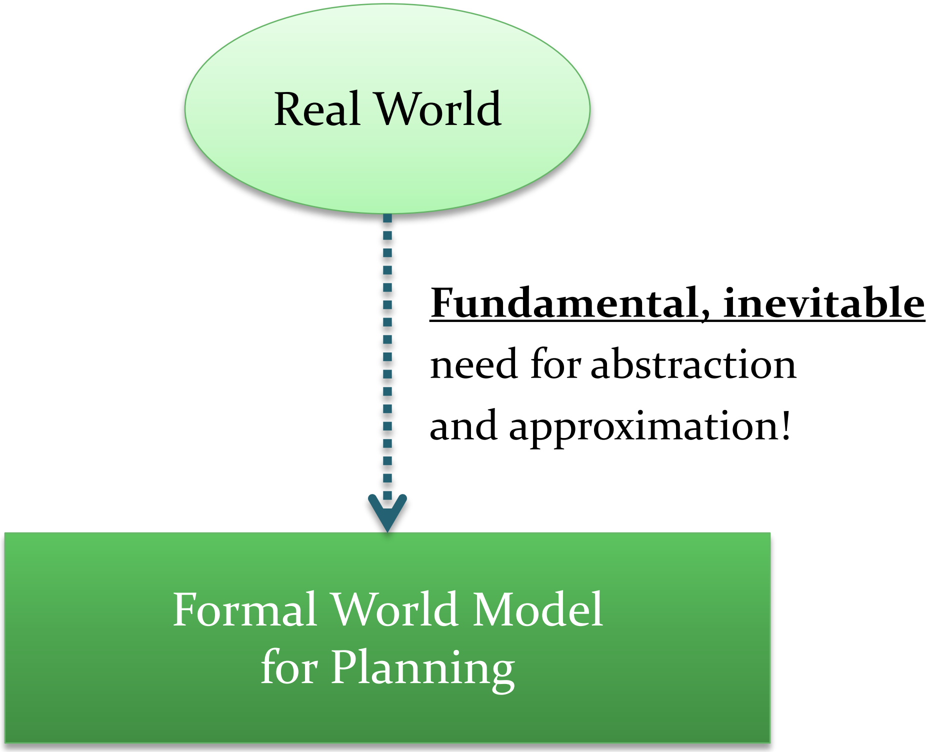
  -
    ```
    (:init   (= (total-cost) 0)
             (= (travel-slow-cost n0 n1) 6) (= (travel-slow-cost n0 n2) 7)
             (= (travel-slow-cost n0 n3) 8) (= (travel-slow-cost n0 n4) 9)
             …)
    ```
- Use special **increase effects** to increase total cost

  -
    ```
    (:action move-up-slow
       :parameters (?lift - slow-elevator ?f1 - count ?f2 - count )
       :precondition (and (lift-at ?lift ?f1) (above ?f1 ?f2 ) (reachable-floor ?lift ?f2))
       :effect (and (lift-at ?lift ?f2) (not (lift-at ?lift ?f1))
                (increase (total-cost) (travel-slow-cost ?f1 ?f2))))
    ```

Modeling: Abstraction vs. Precision

Real World and Formal Models

- Fundamental fact:
  We cannot provide **<u>all</u>** information about the world!
  - Define planning domains in terms of physical laws, quantum mechanics, …?



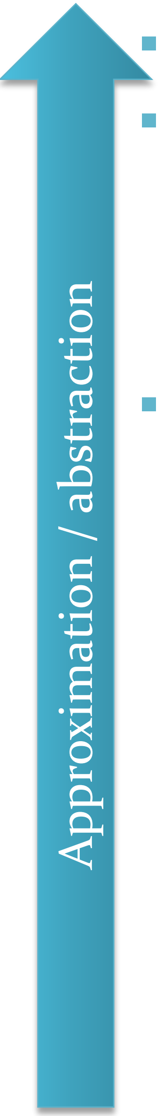
Real World

<u>Fundamental, inevitable</u>
need for abstraction
and approximation!

Formal World Model
for Planning

Approximations 1

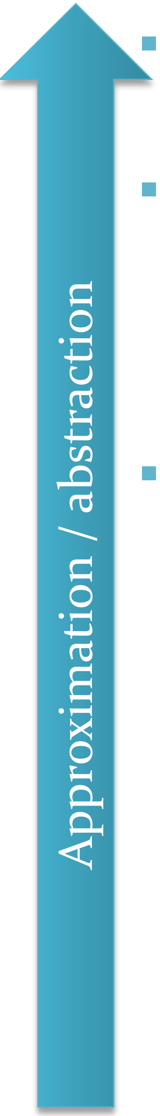
- So **<u>how much</u>** must the planner know?
  - That a helicopter can take off?
  - That a helicopter can take off by:
    - Turning on the engine,
    - Lifting,
    - Going to stable hover mode?
  - Or maybe that it must:
    - Open main fuel valve
    - Turn on fuel pump
    - Open throttle 25%
    - Activate ignition
    - Signal the starter motor,
      waiting for confirmation
      that the engine has successfully started
    - …



Approximation / abstraction

Different **<u>granularity</u>**
in **<u>actions</u>**

Approximations 2

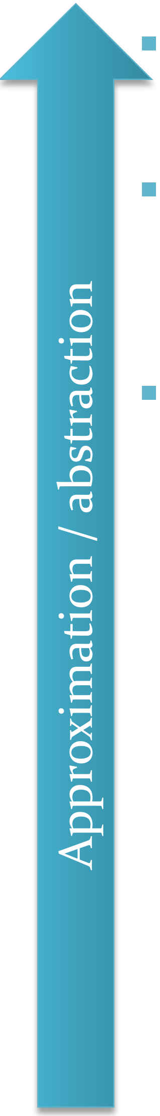


- So **<u>how much</u>** must the planner know?
  - The helicopter's…
    - Altitude and position?
  - The helicopter's…
    - Altitude and a position,
    - Velocity, and
    - Current camera angle?
  - Or maybe also…
    - Its engine speed in RPM
    - Its battery voltage
    - Its fuel level
    - The pressure in the fuel line
    - The time since it was last serviced
    - Its color
    - …

Different **<u>aspects of the world</u>**
modeled in the problem
(initial state, action effects, goals)

Approximations 3

- So **<u>how much</u>** must the planner know?

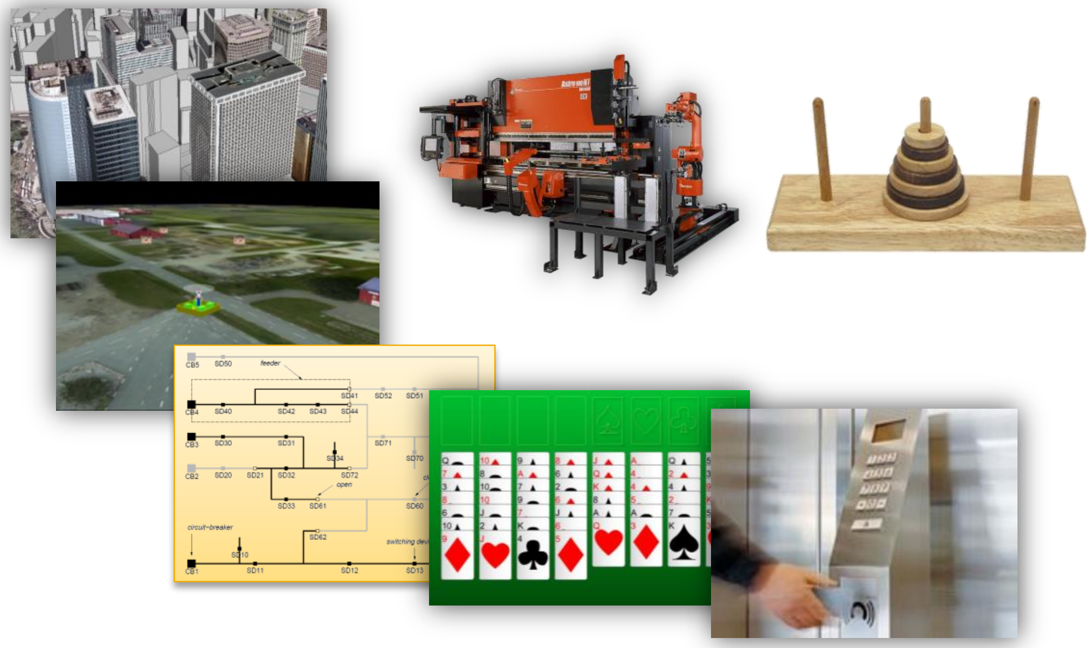


  - When we turn on the fuel pump:
    - Eventually, the engine will be running
  - Or maybe:
    - Within 4 to 10 seconds, the engine will be running
  - Or maybe:
    - Over the first 1.5 seconds, there will be a linear increase in fuel pressure
    - In the **<u>next step</u>** in the same action, fuel will have been injected
    - …

Different **<u>temporal information</u>**, different model of **<u>changes in the world</u>**

Different Requirements

Where is the “right” level of abstraction?



Different requirements for different domains!

A trade-off…

Different Requirements 2

- **<u>Precision</u>** may be required:
  - To ensure correctness
    - Without modeling fuel usage, you may create infeasible plans
  - To determine which plan is better / worse
    - A model of time is required to determine which plan takes less time to execute
  - ...and so on

- **<u>Decreasing</u>** precision has advantages as well!
  - Less information to specify ➔ easier to create a model
  - More restrictions ➔ faster but less general algorithms can be used
    - Discrete change instead of continuous ➔ less information to keep track of, simpler calculations ➔ faster

- The proper trade-off **<u>depends on the application</u>**!
  - Model those aspects of the world that are important for planning and plan quality **<u>for your current purposes!</u>**