Classical Planning Problems: Representation Languages

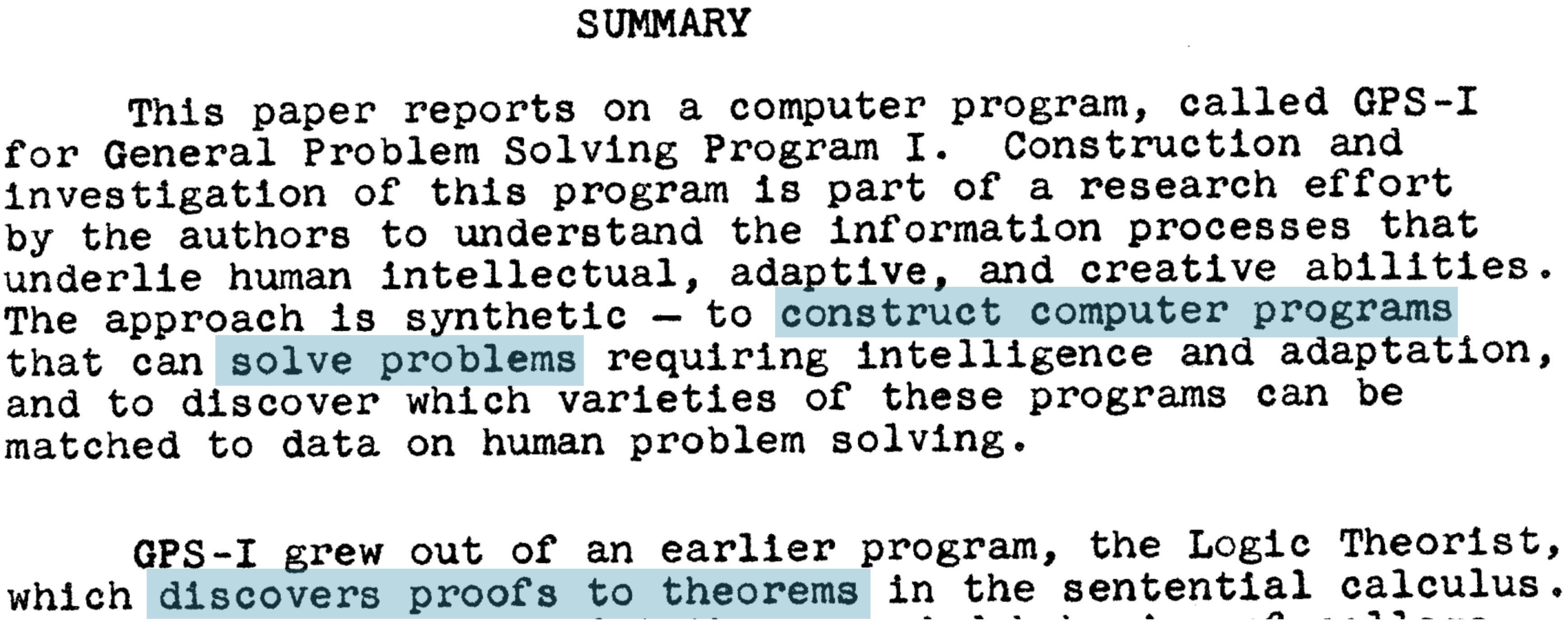
jonas.kvarnstrom@liu.se – 2018

Classical Representation

History: 1959

- The *language of Artificial Intelligence* was/is **logic**
  - **First-order**, second-order, modal, …

- 1959: **General Problem Solver** (Newell, Shaw, Simon)



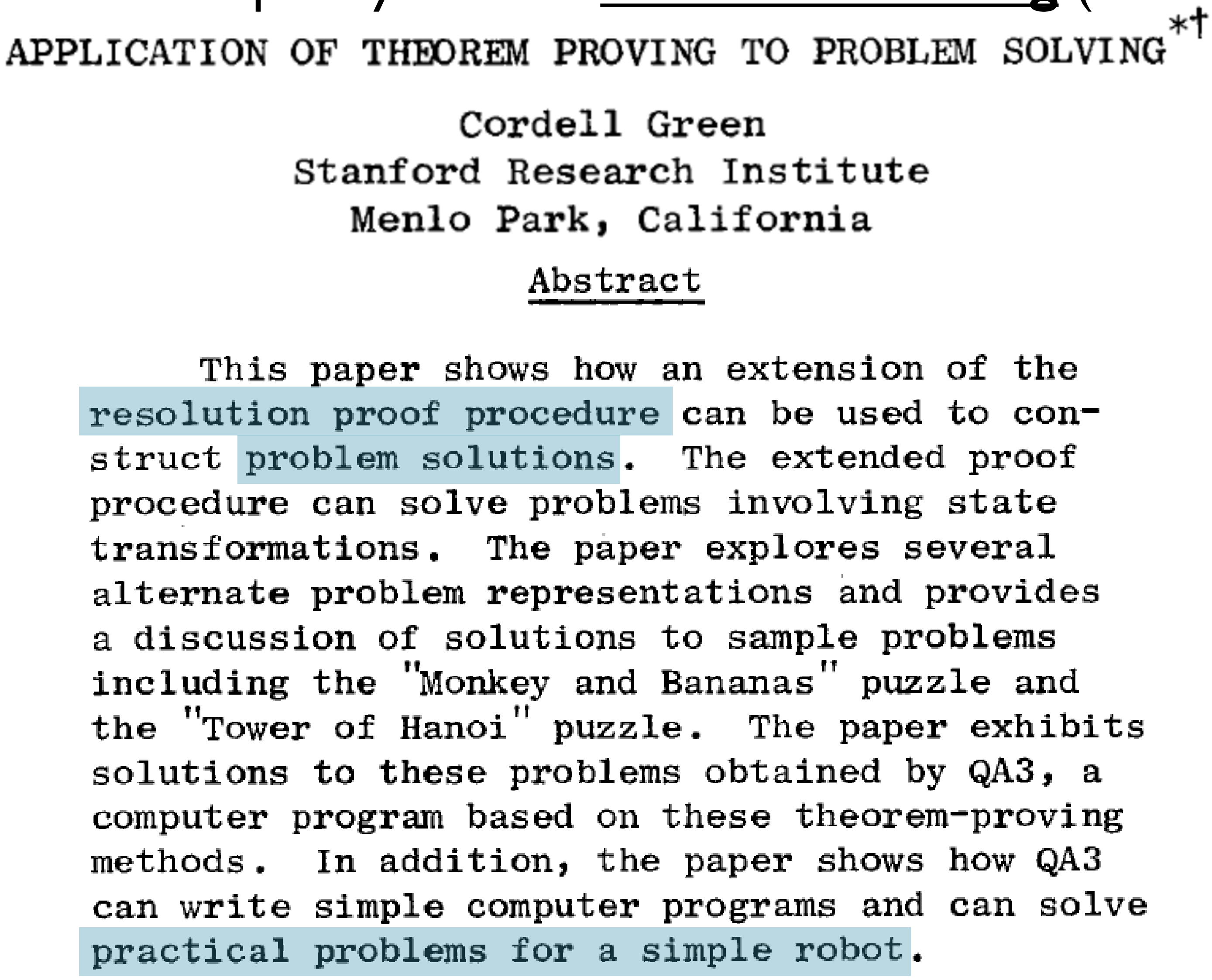
SUMMARY

This paper reports on a computer program, called GPS-I for General Problem Solving Program I. Construction and investigation of this program is part of a research effort by the authors to understand the information processes that underlie human intellectual, adaptive, and creative abilities. The approach is synthetic — to construct computer programs that can solve problems requiring intelligence and adaptation, and to discover which varieties of these programs can be matched to data on human problem solving.

GPS-I grew out of an earlier program, the Logic Theorist, which discovers proofs to theorems in the sentential calculus.

History: 1969

- 1969: Planner explicitly built on **Theorem Proving** (Green)



APPLICATION OF THEOREM PROVING TO PROBLEM SOLVING[*†]

Cordell Green
Stanford Research Institute
Menlo Park, California

Abstract

This paper shows how an extension of the resolution proof procedure can be used to construct problem solutions. The extended proof procedure can solve problems involving state transformations. The paper explores several alternate problem representations and provides a discussion of solutions to sample problems including the "Monkey and Bananas" puzzle and the "Tower of Hanoi" puzzle. The paper exhibits solutions to these problems obtained by QA3, a computer program based on these theorem-proving methods. In addition, the paper shows how QA3 can write simple computer programs and can solve practical problems for a simple robot.

Basis in Logic

- **Full theorem proving** generally proved impractical for planning
  - Different *techniques* were found
  - **Foundations in logical languages** remained!
    - Languages use *predicates, atoms, literals, formulas*
    - We define *states, actions,* … relative to these
    - ➔ Allows us to specify an STS at a higher level!

Formal representation using a first-order language:
"Classical Representation" (from the book)

"The *simplest* representation that is (more or less) reasonable to use for modeling"

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

Objects and Object Types

Objects 1: Intro

- We are interested in **objects** in the world
  - Buildings, cards, aircraft, people, trucks, pieces of sheet metal, …
  - Classical → must be a **finite** set!

Modeling: Which objects **exist** and **are relevant** for the **problem** and **objective**?

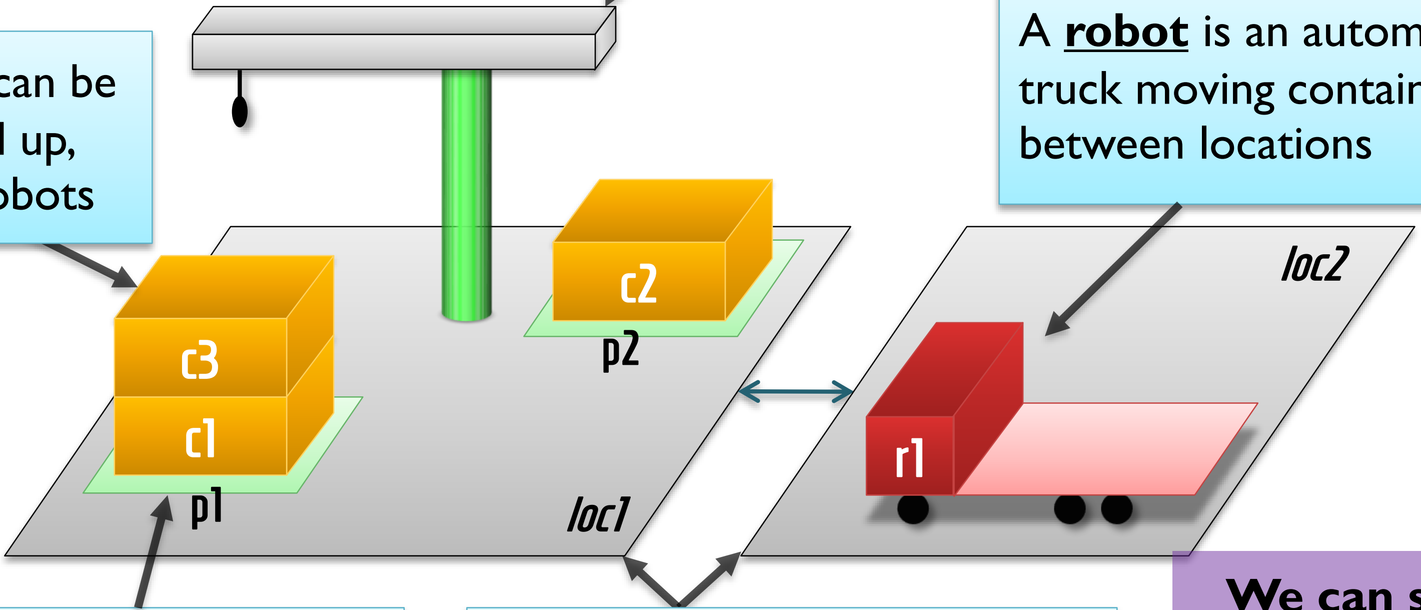
Objects 2: Dock Worker Robots

- **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



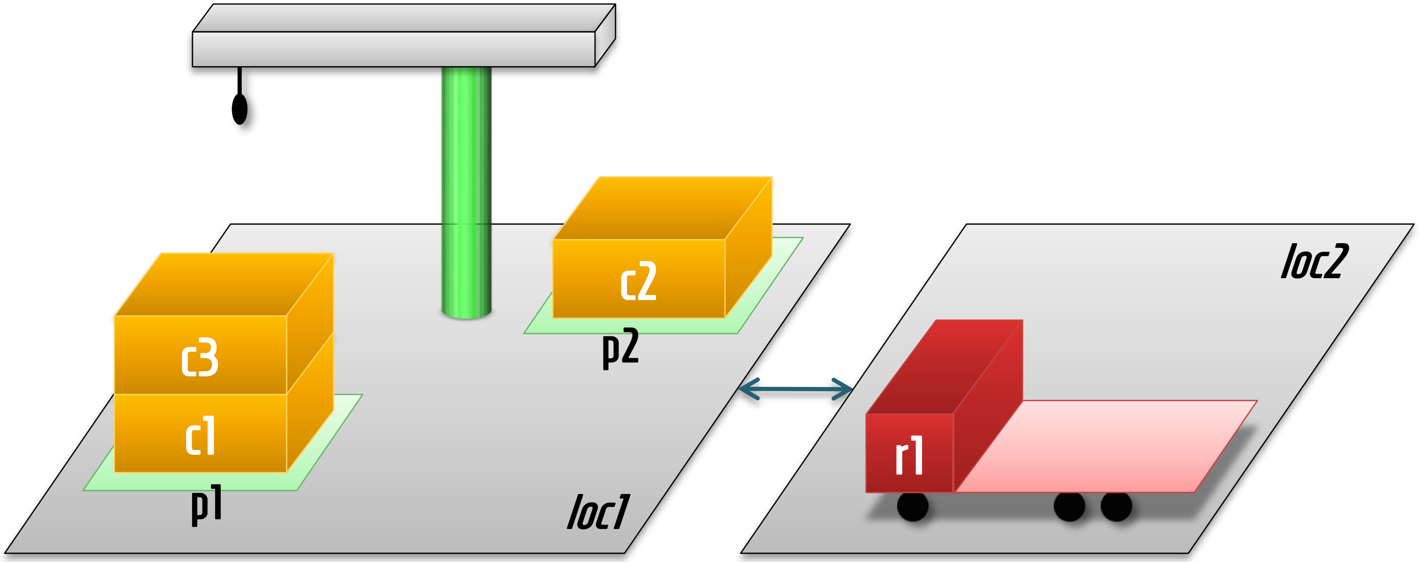
A **pile** is a stack of containers – at the bottom, there is a **pallet**

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

We can skip:
Hooks
Wheels
Rotation angles
Drivers
– not relevant for *this* problem!

Objects 3: Classical Representation

- **Classical representation**:
  - We are constructing a **first-order language** L (as in logic)
  - Every object is modeled as a **constant**
  - Add a **constant symbol** ("object name") for each object:
    L contains { **c1,c2,c3, p1,p2, loc1,loc2, r1,…** }

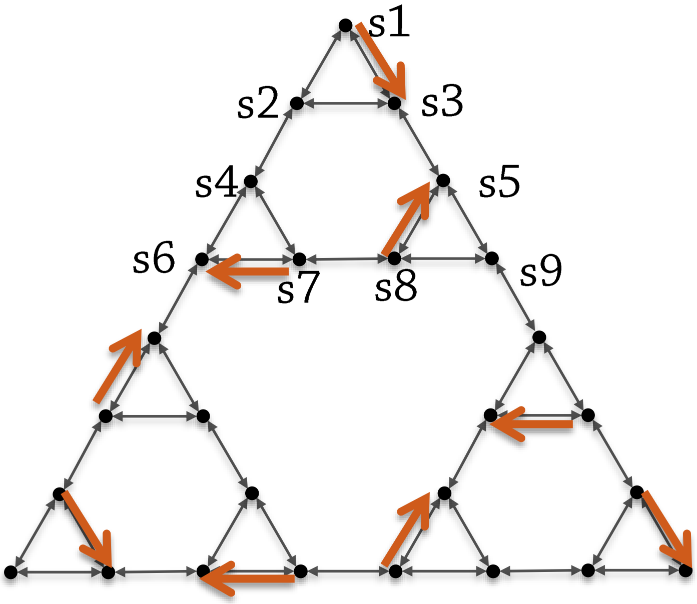


Information about the World: Predicates, Atoms, States

Internal Structure?

- An STS only assumes there are **states**
  - What **is** a state? The STS doesn't care!
  - Its definitions don't depend on what s "represents" or "means"
    - Can execute a in s if $\gamma(s, a) = \{s'\}$

- We (and planners) **need more structure**!
  - "state $s_{23862497124985}$" ➔ "the state where all disks are on peg 1, in ascending order"



Predicates

- First-order language: Start with a set of **predicates**
  - **Properties** of the **world**
    - **raining** – it is raining [not part of the DWR domain!]
  - **Properties** of single **objects**
    - **occupied**(*robot*) – the robot has a container
  - **Relations** between objects
    - **attached**(*pile, location*) – the pile is in the given location
  - **Relations** between >2 objects
    - **can-move**(*robot, loc, loc*) – the robot can move between two locations
  - **Non-boolean properties** are "relations between constants"
    - **has-color**(*robot, color*) – the robot has the given color

Modeling:
Color values must be **constants** (**red, green, blue**)
-- so that they can be handled the same way as *real* objects

Essential: Determine what is **relevant** for the **problem** and **objective**!

Predicates for DWR

- All predicates for DWR, and their *intended* meaning:

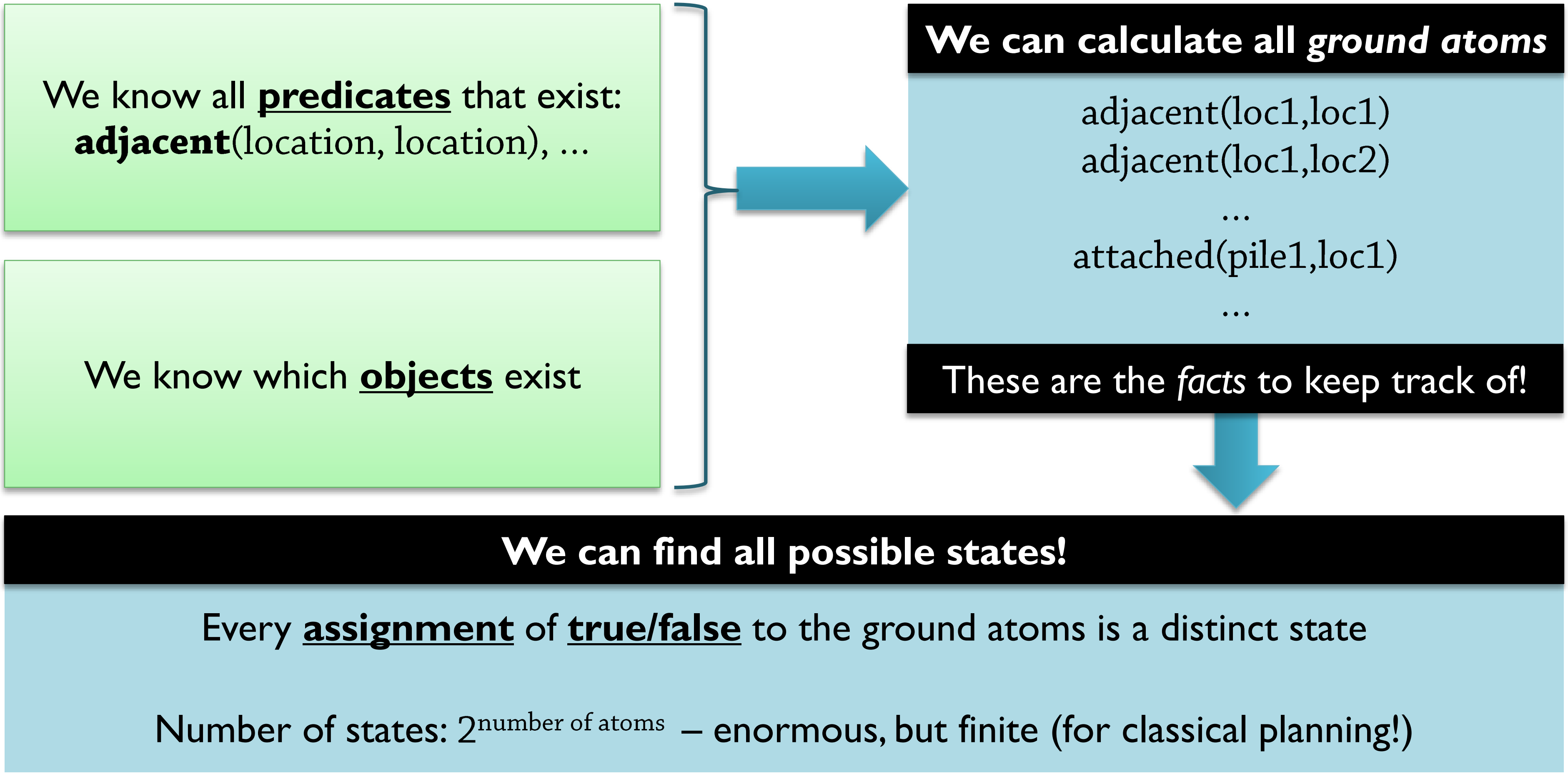
"Fixed/Rigid" (can't change)	**adjacent**	(*loc1*, *loc2*)	; can move from *loc1* directly to *loc2*
	attached	(*p*, *loc*)	; pile *p* attached to *loc*
	belong	(*k*, *loc*)	; crane *k* belongs to *loc*
"Dynamic" (modified by actions)	**at**	(*r*, *loc*)	; robot *r* is at *loc*
	occupied	(*loc*)	; there is a robot at *loc*
	loaded	(*r*, *c*)	; robot *r* is loaded with container *c*
	unloaded	(*r*)	; robot *r* is empty
	holding	(*k*, *c*)	; crane *k* is holding container *c*
	empty	(*k*)	; crane *k* is not holding anything
	in	(*c*, *p*)	; container *c* is somewhere in pile *p*
	top	(*c*, *p*)	; container *c* is on top of pile *p*
	on	(*c1*, *c2*)	; container *c1* is on container *c2*

Predicates, Terms, Atoms, Ground Atoms

- Terminology:
  - **Term**: Constant symbol or variable
    - **loc2** -- **constant**
    - ***location*** -- ***variable***
  - **Atom**: Predicate symbol applied to the intended number of terms
    - **raining**
    - **occupied(*location*)**
    - **at(r1, loc1)**
  - **Ground atom:** Atom without variables (only constants) – a *fact*
    - **occupied(loc2)**
- Plain first-order logic has no distinct **types for objects**!
  - ➔ Some "strange" atoms are perfectly valid:
    - at(loc1,loc2)
    - holding(loc1, c1)
    - …

States 1: Internally Structured

- A **state (of the world)** should specify exactly *which facts (**ground atoms**) are true/false in the world* at a given time



We know all **predicates** that exist:
adjacent(location, location), …

We know which **objects** exist

We can calculate all *ground atoms*

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

These are the *facts* to keep track of!

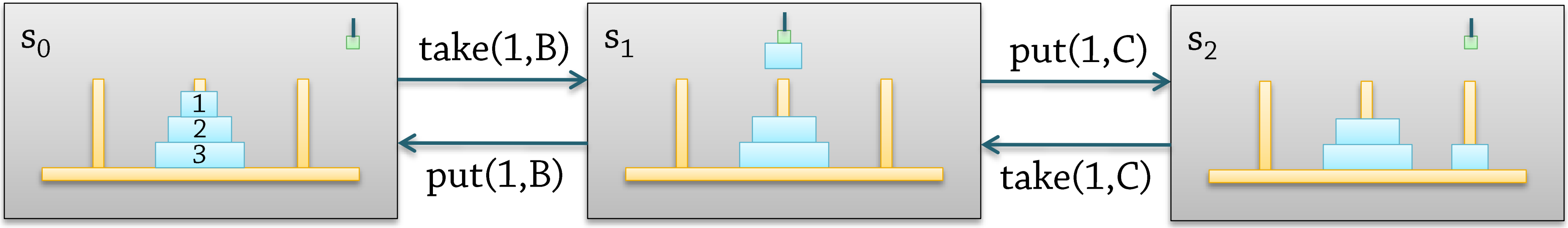
We can find all possible states!

Every **assignment** of **true/false** to the ground atoms is a distinct state

Number of states: $2^{\text{number of atoms}}$ – enormous, but finite (for classical planning!)

States 2: Structure, DifferencesStructured

- Then we can compute **differences** between 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
…

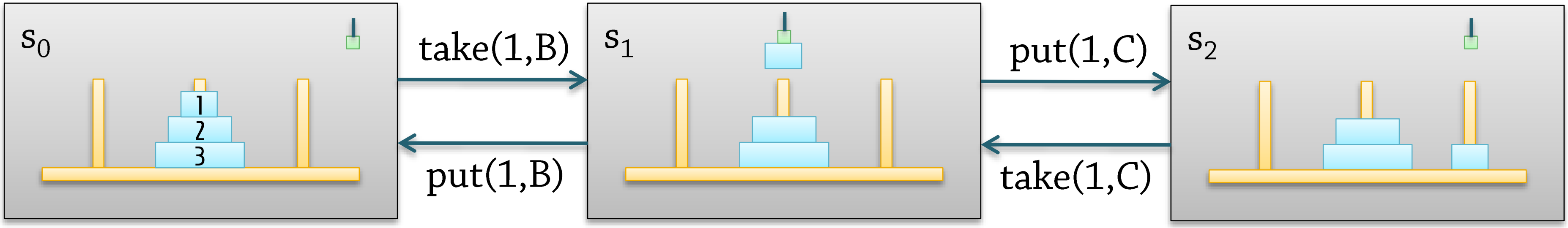
s_1
top(1) is **false**
top(2) is **true**
top(3) is false
on(1,1) is false
on(1,2) is **false**
on(1,3) is false
on(2,1) is false
…

Structure is essential!

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

States 3: First-order Representation

- **Efficient specification / storage** of a **single state**:
  - Specify **which facts are true**
    - All other facts have to be false – what else would they be?
  - ➔ A classical state **is** a **set** of all **ground atoms** 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

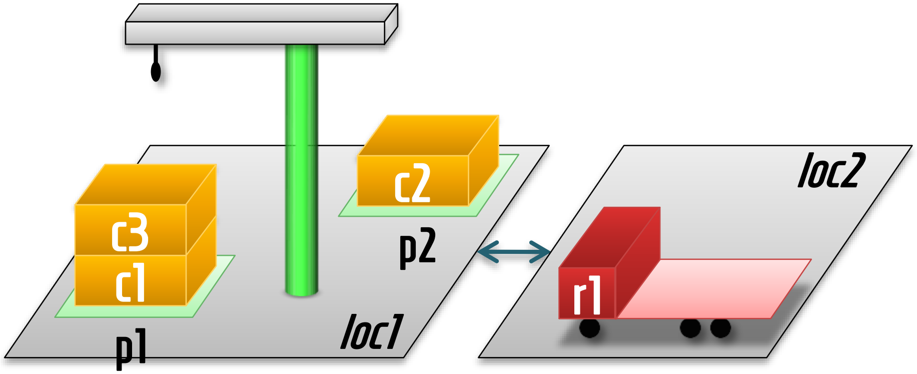
Why not store all ground atoms
that are **false** instead?

States 4: Initial State

- The STS assumes a single *initial state* s_0
  - *Complete information* about the current state of the world

Complete ***relative to the model***:
We must know everything
about those predicates and objects
we have specified…

- State = set of true facts…
  - $s_0 = \{\text{attached(p1,loc1), in(c1,p1), on(c1,pallet), on(c3,c1), ...}\}$



States 5: Goal States, Positive Goals

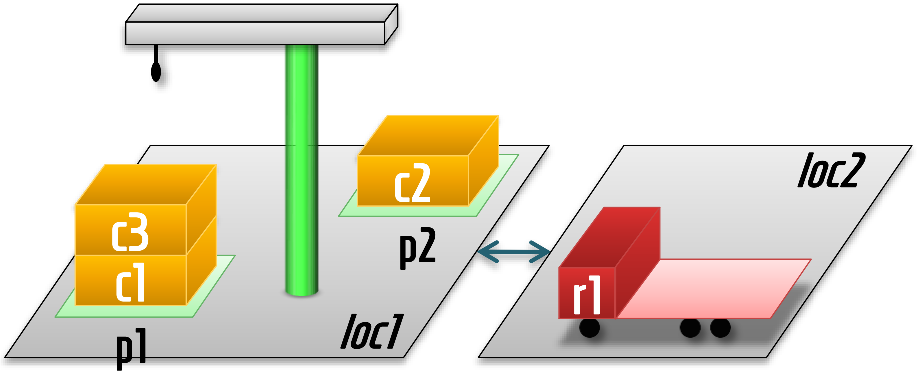
- One way of **efficiently** defining a **set** of goal states:
  - A **goal** g is a **set** of ground **atoms**
    - Example: g={in(c1,p2), in(c3,p2)}
    - *In the final state, containers I and 3 should be in pile 2, and we don't care about any other facts*
  - Then $S_g = \{s \in S \mid g \subseteq s\}$
    - S_g = {

      {**in(c1,p2), in(c3,p2)**}, -- one acceptable final state

      {**in(c1,p2), in(c3,p2), on(c1,c3)**}, -- another acceptable final state

      …

      }



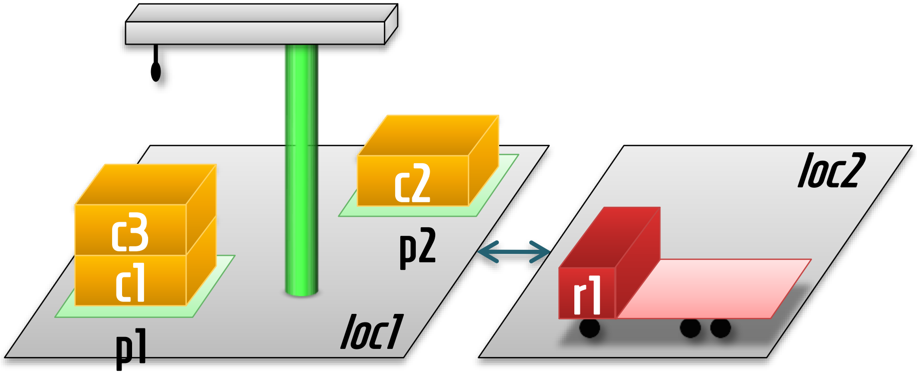
States 6: Goal States, Literal Goals

- To increase **expressivity**:
  - A **goal** g is a set of ground **literals**
    - A **literal** is an atom or a *negated* atom: in(c1,p2), ¬in(c2,p3)
    - in(c1,p2) ➔ **Container 1 should be in pile 2**
    - **¬in(c2,p3)** ➔ **Container** 2 should *not* be in pile 3
  - Then $S_g = \{s \in S \mid s \text{ satisfies } g\}$
    - Positive atoms in g are also in s
    - Negated atoms in g are *not* in s

More expressive than positive goals

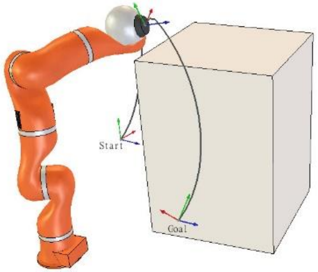
Still not as expressive as the STS:
”arbitrary set of states”

Many classical planners use one
of these two alternatives (atoms/lits);
some are more expressive

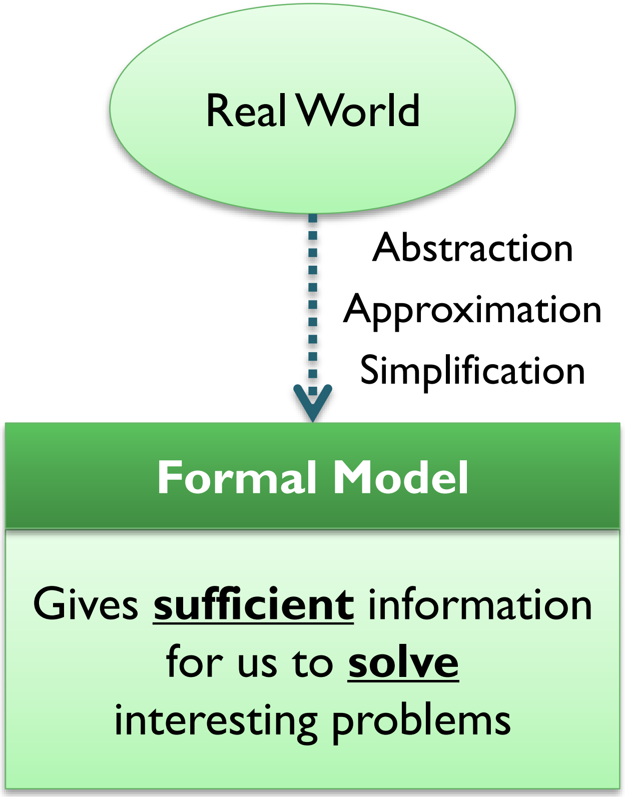


Abstraction

- We have **abstracted** the **real world**!
  - Motion is really continuous in 3D space
    - Uncountably infinite number of positions for a crane



  - But for the purpose of **planning**:
    - We model a finite number of *interesting* positions
      - On a specific robot
      - In a specific pile
      - Held by a specific crane



Real World

Abstraction
Approximation
Simplification

Formal Model

Gives **sufficient** information for us to **solve** interesting problems

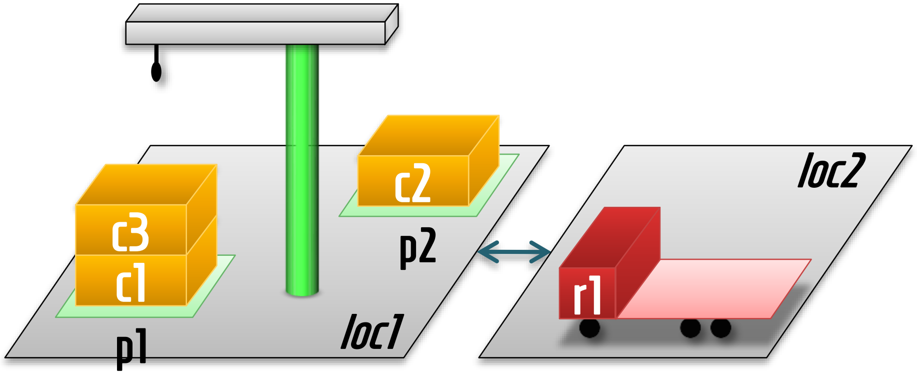
Operators and Actions

Actions with Structure

- If **states** have internal structure:
  - Makes sense for **actions** to have internal structure
    - ”$\gamma(s_{291823}, a_{120938}) = \emptyset$” ➔
      “action **move**(diskA, peg1, peg3) **requires** a state where on(diskA,peg1)”
    - ”$\gamma(s_{975712397}, a_{120938}) = \{s_{12578942}\}$” ➔
      “action **move**(diskA, peg1, peg3) **makes** on(diskA,peg3) true, and …”

Operators

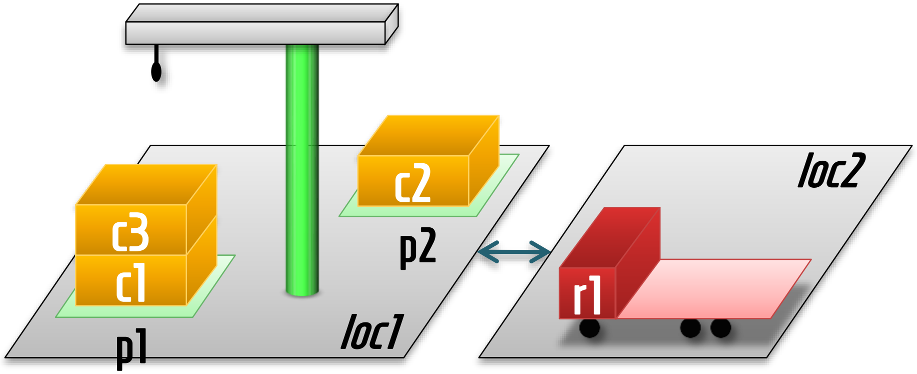
- In the classical representation: Don't define actions directly
  - Define a set O of operators
  - Each ***operator*** is parameterized, defines many actions
    - ;; crane *k* at location *l* takes container *c* off container *d* in pile *p*
      take(*k*, *l*, *c*, *d*, *p*)
  - Has a **precondition**
    - precond(o): **set** of **literals** that must hold before execution
    - precond(take) = { belong(*k*,*l*), empty(*k*), attached(*p*,*l*), top(*c*,*p*), on(*c*,*d*) }
  - Has **effects**
    - effects(o): **set** of **literals** that will be made to hold after execution
    - effects(take) = { holding(*k*,*c*), ¬empty(*k*), ¬in(*c*,*p*), ¬top(*c*,*p*), ¬on(*c*,*d*), top(*d*,*p*) }



Actions

- In the classical representation:
  - Every **ground instantiation** of an operator is an **action**
    - a_1 = take(crane1, loc2, c3, c1, p1)
  - Also has (instantiated) precondition, effects
    - precond(a_1) = { belong(crane1,loc2), empty(crane1), attached(p1,loc2), top(c3,p1), on(c3,c1) }
    - effects(a_1) = { holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1) }

$$A = \left\{ a \,\middle|\, \begin{array}{l} a \text{ is an instantiation} \\ \text{of an operator in } O \\ \text{using constants in } L \end{array} \right\}$$



Untyped Actions and Applicability

- If every **ground instantiation** of an operator is an **action**…
  - …then so is this:
    - **take**(c3, crane1, r1, crane2, r2)
      ;; Container **c3** at location **crane1** takes **robot1** off **crane2** in pile **robot2**
  - But when will this action be ***applicable***?
    - **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*)
    - **take**(c3, crane1, r1, crane2, r2):
      precond: belong(c3,crane1), empty(c3), attached(r2,crane1),
      top(r1,r2), on(r1,crane2)

For *these* preconditions to be true,
something must already have gone wrong!

Untyped Actions and Applicability (2)

- More common solution: Separate **type predicates**
  - Ordinary predicates that happen to represent types:
    - crane(x), location(x), container(x), pile(x)
  - Used as part of preconditions:
    - **take**(*k*, *l*, *c*, *d*, *p*): ;; crane *k* at location *l* takes container *c* off container *d* in pile *p*
      precond: **crane(k), location(l), container(c), container(d), pile(p),**
      belong(*k*,*l*), empty(*k*), attached(*p*,*l*), top(*c*,*p*), on(*c*,*d*)
  - DWR example was "optimized" somewhat
    - belong(k,l) is only true for crane+location, replaces two type predicates
  - So:
    - **take**(c3, crane1, r1, crane2, r2) **is** an action
    - Its preconditions can never be satisfied in reachable states!
    - Type predicates are *fixed, rigid, never modified*
      ➔ such actions can be filtered out before planning even starts

Useful Properties

- **Some useful properties:**
  - If a is an operator or action…
    - precond+(a) = { *atoms* that appear **positively** in a's preconditions }
    - precond–(a) = { *atoms* that appear **negated** in a's preconditions }
    - effects+(a) = { *atoms* that appear **positively** in a's effects }
    - effects–(a) = { *atoms* that appear **negated** 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 (Executable) Actions

- An action *a* is **applicable** in a state *s*…
  - … if precond+(a) ⊆ s and precond–(a) ∩ s = ∅
- Example:
  - take(crane1, loc1, c3, c1, p1):

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

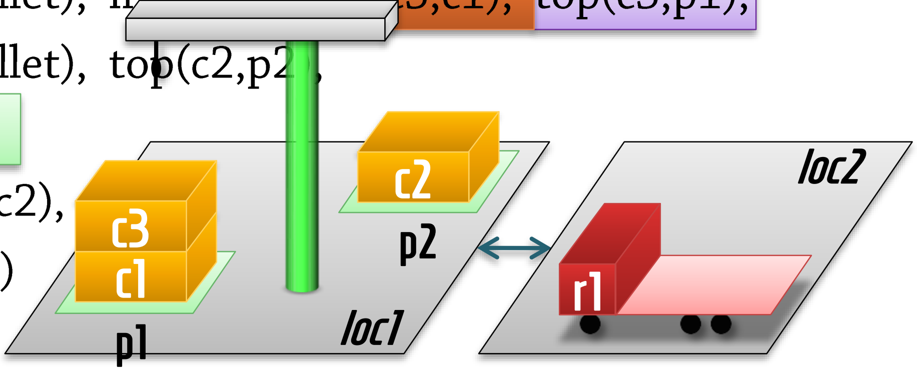
    precond: { belong(crane1,loc1), empty(crane1),
    attached(p1,loc1), top(c3,p1), on(c3,c1) }

    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)
    }

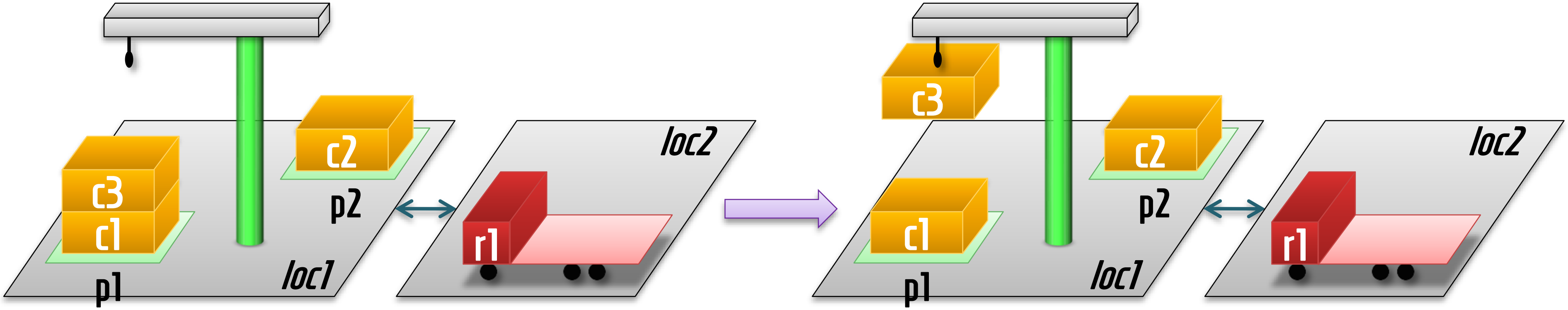
Action → ground
→ preconds are
ground atoms

Simple representation (sets)
→ simple definitions!



Result of Performing an Action

- **Applying** will **add** positive effects, **delete** negative effects
  - If a is applicable in s, then
    the new state is (s – effects–(a)) ∪ effects+(a)
  - **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)



Defining γ

- From actions to γ:

Positive preconditions missing from state

Negated preconditions present in state

  - $\gamma(s,a) =$

$$\begin{cases} \emptyset & \text{if } \mathrm{precond}^{+}(a) \not\subseteq s \text{ or } \mathrm{precond}^{-}(a) \cap s \neq \emptyset \\ \{s - \mathrm{effects}^{-}(a) \cup \mathrm{effects}^{+}(a)\} & \text{otherwise} \end{cases}$$

From the classical representation language,
we know how to define $\Sigma = (S, A, \gamma)$
and a problem (Σ, s_0, S_g)

Modeling: 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):
        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) }
```

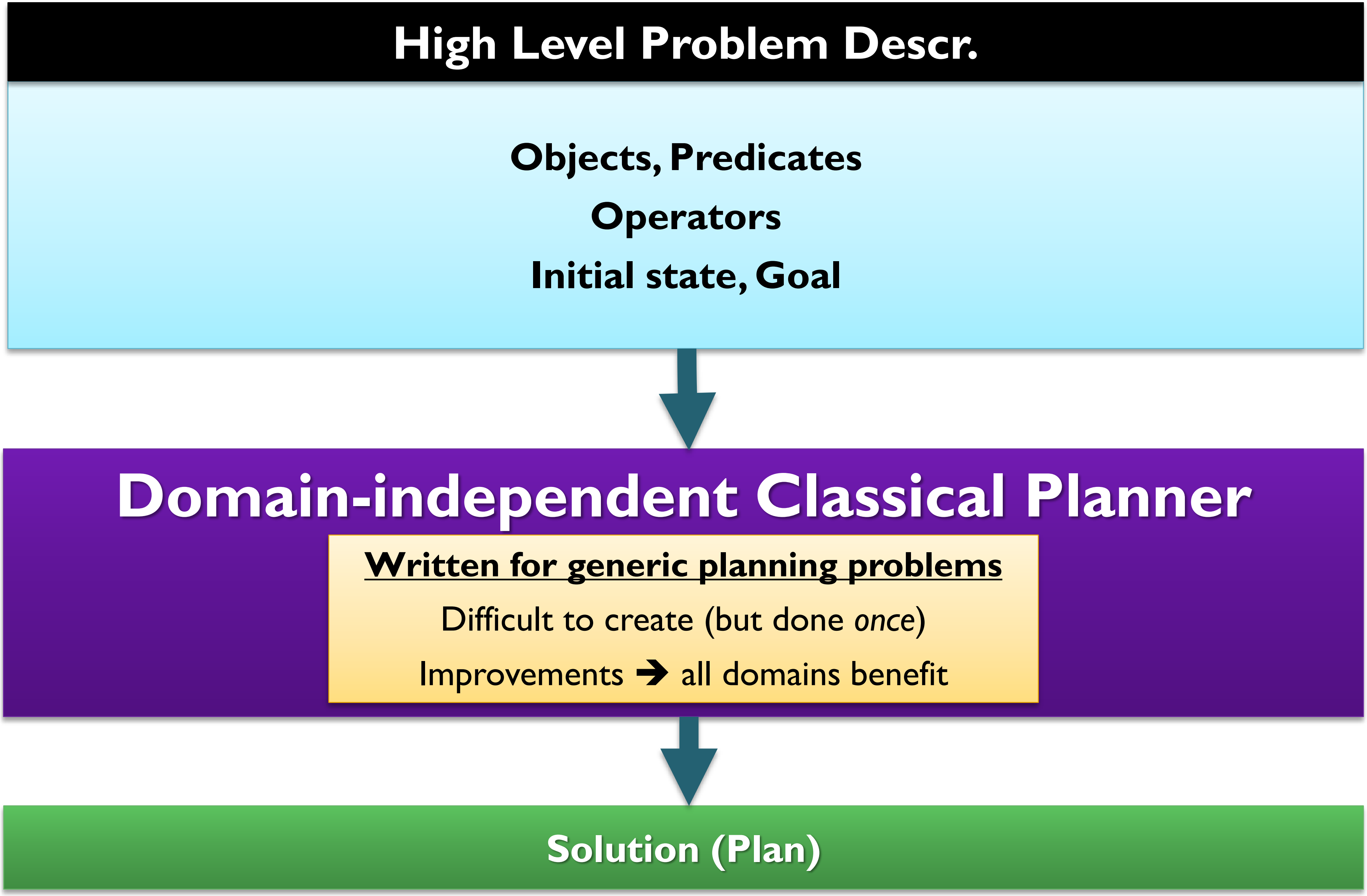
These are *physical* requirements for taking a container!

- The *planner* chooses which actions are *suitable*, using heuristics (etc.)

- Add explicit "suitability preconditions" ➔ *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"

Domains and Problem Instances

Domain-Independent Planning



High Level Problem Descr.

Objects, Predicates

Operators

Initial state, Goal

Domain-independent Classical Planner

Written for generic planning problems

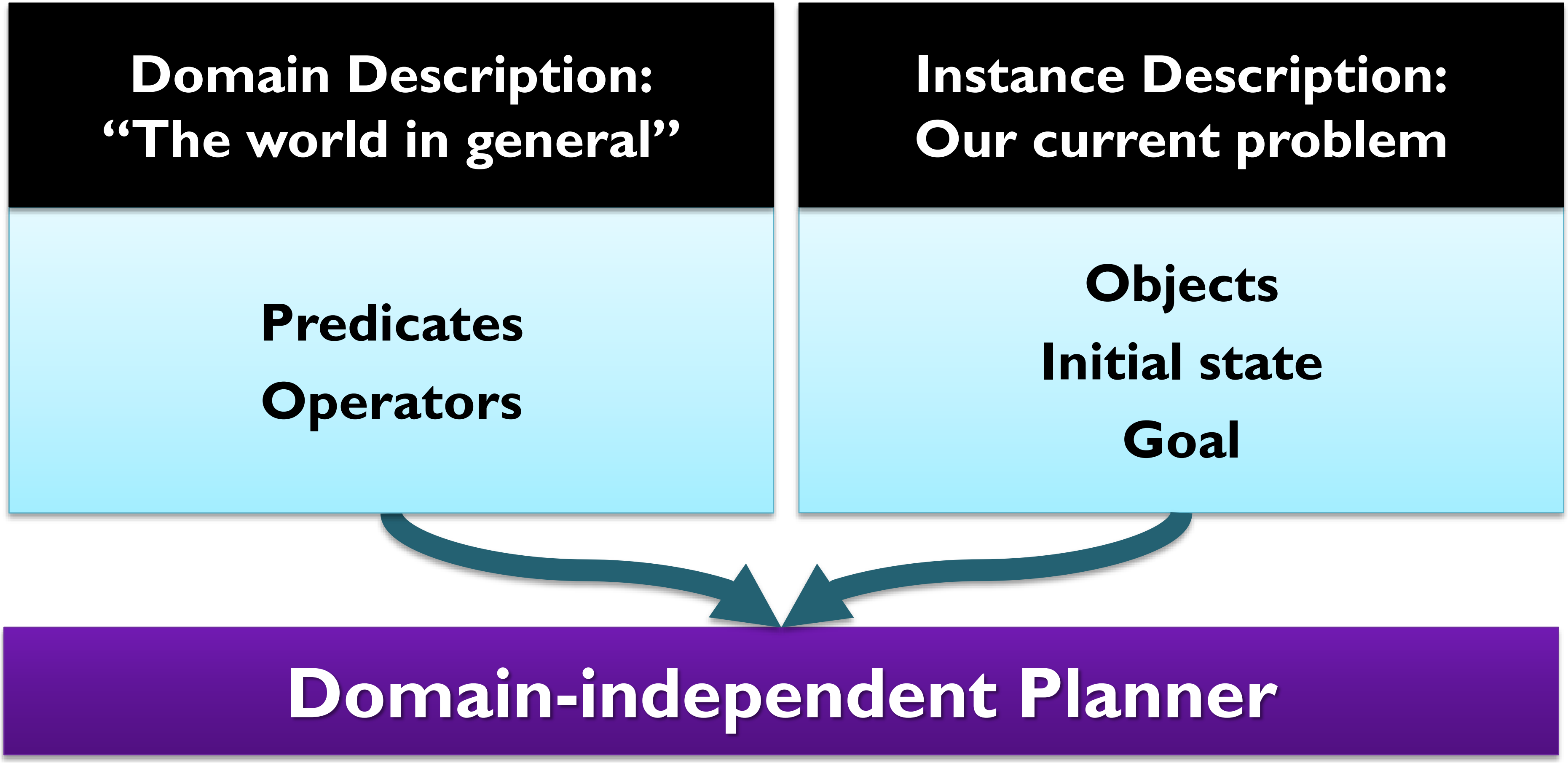
Difficult to create (but done *once*)

Improvements ➔ all domains benefit

Solution (Plan)

Domain vs Instance

- Makes sense to split the information



Domain Description:
“The world in general”

Predicates

Operators

Instance Description:
Our current problem

Objects

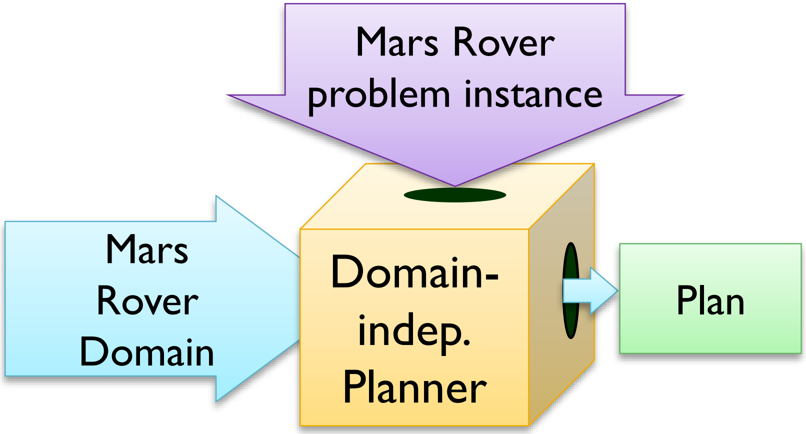
Initial state

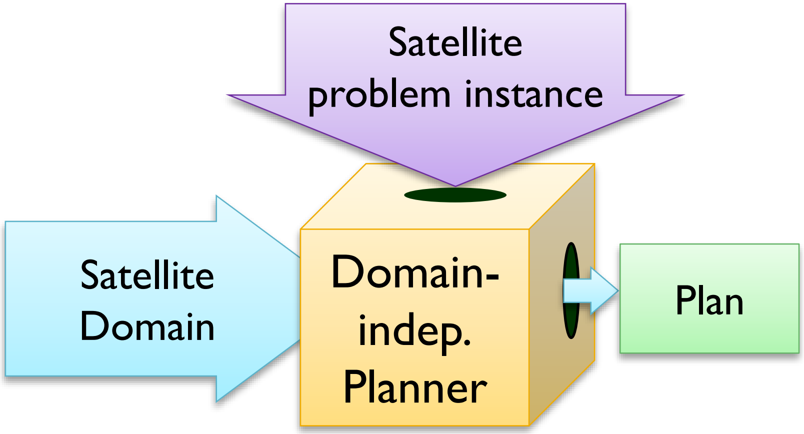
Goal

Domain-independent Planner

Domain-Independent Planning

- To solve problems in other domains:
  - Keep the **planning algorithm**
  - Write a new **high-level description** of the problem domain





PDDL:
Planning Domain Definition Language

Now: Extensible representation language

Classical Representation is simple, but not easily extended with complex preconditions, effects, timing, action costs, concurrency, …

Misc.	Separation: Domain / instance
Misc.	PDDL object *types*
Preconditions	Formulas: Disjunctions, …
Effects	Conditional effects, …
Extensions	Timing, action costs, …

Formal representation language

Closer to how we think

Provides more *structural information*, very useful for planning algorithms

Objects	{ car1, car2, car3, loc1, loc2 }
Fact atoms	{ at(car1,loc1),at(car1,loc2),… }
State	Set of true atoms
Operators	drive(loc1, loc2) – with params
Preconditions	{ at(car1,loc1), ¬broken(car1) }
Effects	{ ¬at(car1,loc1), at(car1,loc2) }
	This *indirectly* defines $\gamma(s,a)$!

Underlying formal model

Concepts as *simple* as possible: States, actions, transition function

Good for *analysis*, *correctness* proofs, understanding what planning *is*

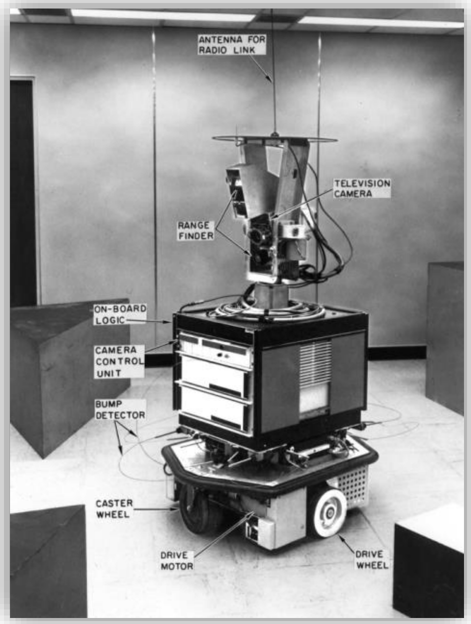
States	s1 … s1000000000000,
Actions	a1 … a10000 – no structure!
Transition function	defining the result of an action, $\gamma(currentstate, action)=newstate$
Goals	{s1,s3,s282} – set of end states

Covered in the book

PDDL

- PDDL: **Planning Domain Definition Language**
  - Origins: *First International Planning Competition*, 1998
  - Most used language today
  - General; many expressivity levels

- Lowest level of expressivity: Called **STRIPS**
  - After the planner used by Shakey,
    STRIPS: Stanford Research Institute *Problem Solver*
  - One *specific* predicate-based ("logic-based")
    syntax/semantics for classical planning domains/instances



PDDL: Domain and Problem Definition

- **PDDL** separates **domains** and **problem instances**

Domain file

Named

```
(define (domain dock-worker-robots)
        …
)
```

Problem instance file

Named

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

Associated with a domain

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

PDDL: Domain and Problem Definition

- Domains declare their **expressivity requirements**

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

We will see some other levels as well…

Warning:
Many planners' parsers *ignore* expressivity specifications

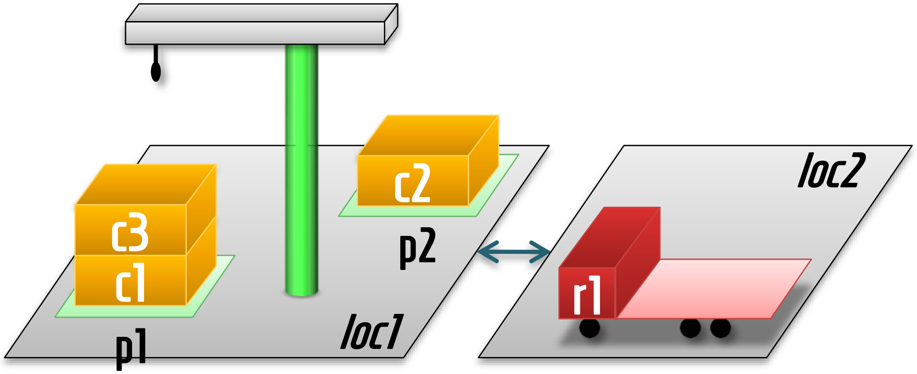
Objects and Object Types

PDDL Objects 1: Types

- In PDDL and most planners:
  - Constants have **types**, defined in the domain

    -
      ```
      (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)
      )
      ```

Tell the planner which features you need…

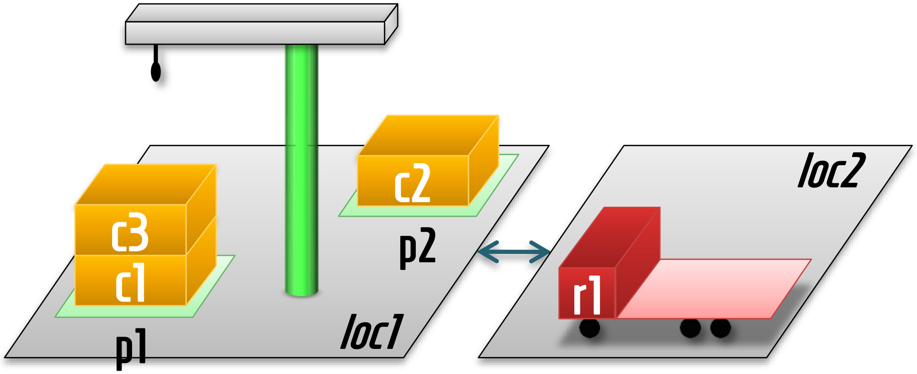


PDDL Objects 2: Type Hierarchies

- Many planners support **type hierarchies**
  - Convenient, but often not used in domain examples
    - (:**types**

```
; containers and robots are movable objects
container robot – movable
…)
```

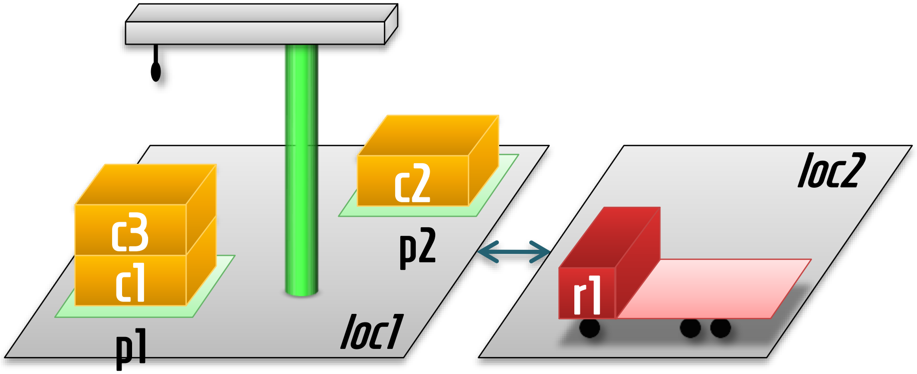
  - Predefined ”topmost supertype”: **object**



PDDL Objects 3: Object Definitions

- Instance-specific constants are called **objects**

  -
    ```
    (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)
    ```



PDDL Objects 4: PDDL Constants

- Some constants should exist in **all instances**

```
(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

➔ Can use in the domain definition as well!

Properties of the World

Predicates in PDDL

- In PDDL: Lisp-like *syntax* for predicates, atoms, …

Variables are prefixed with “?”

```
(define (domain dock-worker-robots)
  (:requirements ...)
  (:predicates
    (adjacent  ?l1  ?l2 - location)         ; can move from ?l1 directly 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 not holding anything

    (in        ?c - container ?p - pile)    ; container ?c is somewhere in 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
  )
```

Modeling: Different predicates per type?

- Modeling Issues: Single or multiple predicates?

  -
    ```
    (define (domain dock-worker-robots)
      (:requirements …)
      (:predicates
        (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
    ```

3 predicates with similar meaning

- Could use **type hierarchies** instead – in *most* planners

  -
    ```
    (define (domain dock-worker-robots)
      (:requirements …)
      (:types     robot crane container pile – thing
                  location
      (:predicates
        (at       ?t – thing ?l - location)       ; thing ?t is at location ?l
      )
    )
    ```

Modeling: Duplicate information

- Models often provide *duplicate information*
  - A location is occupied ⇔ there is *some* robot at the location

    -
      ```
      (define (domain dock-worker-robots)
        (:requirements …)
        (:predicates
          (at        ?r - robot ?l - location)       ; robot ?r is at location ?l
          (occupied  ?l - location)                  ; there is a robot at location ?l
      ```
  - Strictly speaking, **occupied** is *redundant*
    - Still necessary in many planners
    - No support for quantification: (exists ?r (at ?r ?l))
    - Have to write (occupied ?l) instead
    - Have to provide this information + update it in actions!

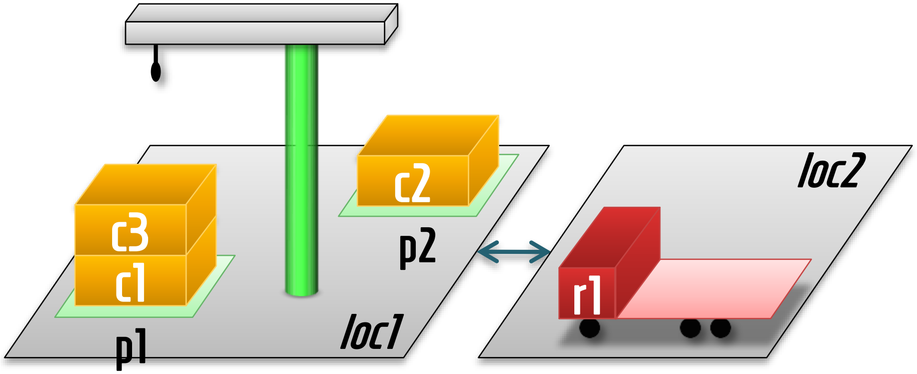
States in PDDL

States 1: Initial State in PDDL

- Initial states in PDDL:
  - Set (list) of true atoms

    -
      ```
      (define (problem dwr-problem-1)
          (:domain dock-worker-robot)
          (:objects …)
          (:init
              (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)
          )
      )
      ```

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

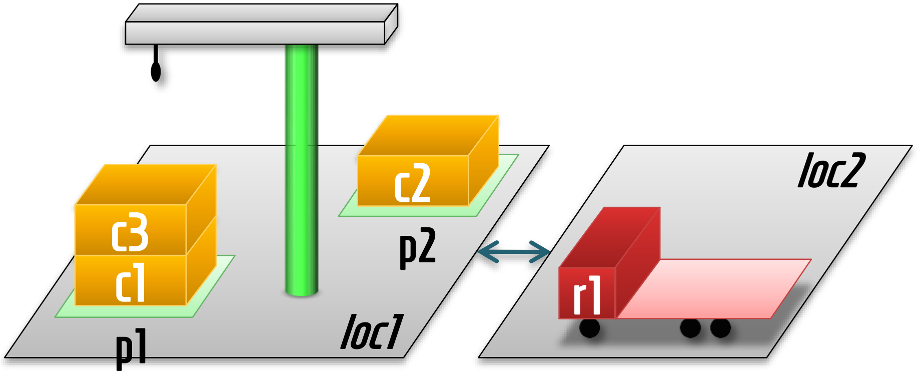


States 2: Goal States

- The **:strips** level supports **positive conjunctive goals**
  - Example: **Containers 1 and 3 should be in pile 2**
    - We don't care about their order, or any other fact

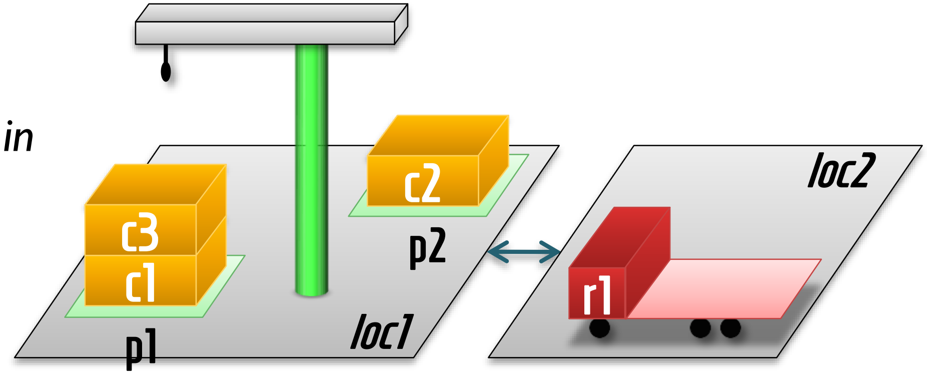
    -
      ```
      (define (problem dwr-problem-1)
              (:domain dock-worker-robot)
              (:objects …)
              (:goal (and (in c1 p2) (in c3 p2))))
      ```

Write as a **formula** (and …), not a **set**:
Other levels support "or", "forall", "exists", …



States 3: Goal States

- **Some** planners: **Conjunctions** of **positive / negative literals**
  - Example:
    - Containers 1 and 3 should be in pile 2
    - Container 2 should *not* be in pile 4
    - (:**requirements** :**negative-preconditions** …)
    - (**define** (**problem** dwr-problem-2)
      (:**domain** dock-worker-robot)
      (:**objects** …)
      (:**goal** (**and** (in c1 p2) (in c3 p2) (**not** (in c2 p4)))
  - **Buggy support** in some planners
    - Can be worked around
    - Define *outside* predicate = inverse of *in*
    - Make sure actions update this
    - (:**goal** (**and** (in c1 p2) (in c3 p2)
      (outside c2 p4))



Operators and Actions

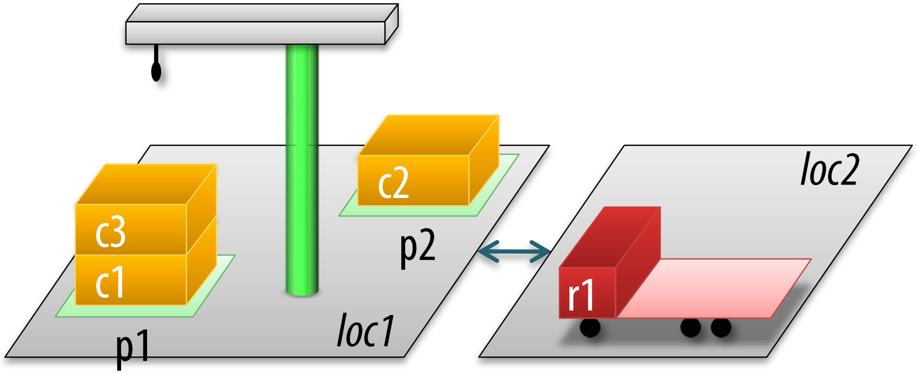
Operators in PDDL

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

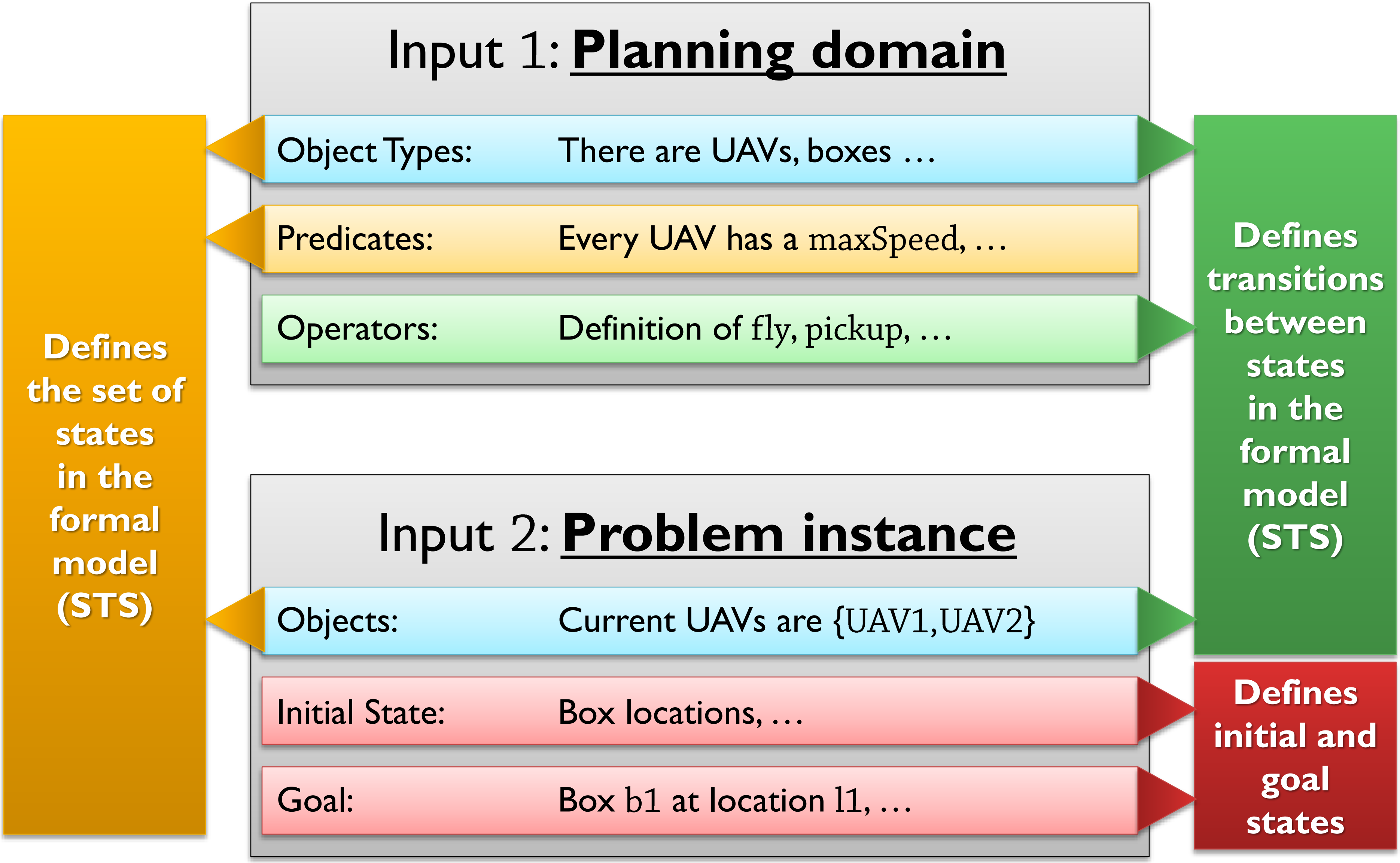
```
(define (domain dock-worker-robots) …
        (:action move
                :parameters     (?r – robot
                                 ?from ?to - location)

                :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))
```

Typed params
→ can only instantiate
with the intended objects

Again, written as logical conjunctions, instead of sets!



Transformation: PDDL/strips → STS



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 the set of states in the formal model (STS)

Defines transitions between states in the formal model (STS)

Defines initial and goal states

Finding the value of a property

Properties of Objects

- Let's model a **"drive" operator** for a truck
  - "**Natural**" parameters: The *truck* and the *destination*
    - (:**action** **drive** :**parameters** (**?t** – *truck* **?dest** – *location*)
      :**precondition** …
      :**effect** …
)
  - "**Natural**" precondition:
    - There must exist a path between the *current location* and the *destination*
    - Should use the predicate (**path-between ?loc1 ?loc2** – *location*)
  - How?
    - (:**precondition** (**path-between** …something… ?dest)) **???**
    - In a first-order predicate representation,
      we can only **test whether** a truck is at some **specific** location:
      (**at** ?t ?location)

Alternative Representations

***Three wide classes* of logic-based representations**
(general classes, containing many languages!)

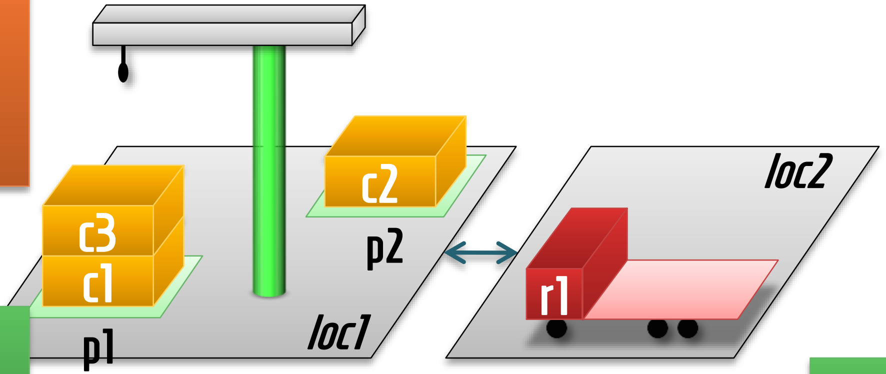
Propositional	**First-order**	**State-variable-based**
(boolean *propositions*)	(boolean *predicates*)	(non-boolean *functions*)
atHome, atWork	at(*truck*, *location*)	loc(*truck*) = *location*
PDDL :strips (if you avoid objects)	PDDL :strips, …	

Read chapter 2 of the book for another perspective on representations…

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?
    - loaded(robot, container) – is the robot loaded with the given container?

May be *wasteful*:
Can represent a container being on *many* robots, which never happens



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

Seems more powerful, but is equivalent!

- **Alternative**: **Classical** with **state-variable representation**
  - A state defines the values of **arbitrary state variables**
    - boolean adjacent(location, location) ;; still boolean!
    - container carriedby(robot) ;; *which* container is on the robot?

Property Values 1

- Back to the **"drive" operator**…
  - "Natural" parameters: The *truck* and the *destination*

    -
      ```
      (:action drive :parameters (?t – truck ?dest – location)
                :precondition …
                :effect …
      )
      ```
  - "Natural" precondition:
    - There must exist a path between the *current location* and the *destination*
    - Should use the predicate (**path-between ?loc1 ?loc2** – *location*)
    - State variable representation → can express **the location** of the truck:
      (:**precondition** (**path-between** (location-of ?t) ?dest))
    - No STS extensions are required!

Property Values 2

- If the planner only supports boolean predicates:
  - **Add a parameter** to the operator

    -
      ```
      (:action drive :parameters (?t – truck ?from – location ?dest – location)
                 :precondition …
                 :effect …
      )
      ```
  - **Constrain** that variable in the precondition
    - `:precondition (and (at ?t ?from) (path-between ?from ?dest))`
    - Can only apply those instances of the operator where ?from *is* the current location of the truck

Property Values 3

These parameters are "**extraneous**"
in the sense that they *do not add choice*:
We can choose *truck* and *dest* (given some constraints);
from is uniquely determined by state + other params!

- Example:
  - Initially:
    - (**at** truck5 home)
  - Action:
    - (:**action** **drive** :**parameters** (**?t** – *truck* **?from** – *location* **?dest** – *location*)
      :**precondition** (**and** (**at** ?t ?from) (**path-between** ?from ?dest))
      :**effect** …
)
  - Which actions are executable?
    - **(drive truck5 work home)** – no, precond false: not (at truck5 work)
    - **(drive truck5 work work)** – no, precond false
    - **(drive truck5 work store)** – no, precond false
    - **(drive truck5 home store)** – precond true, can be applied!

With quantification, we could have changed the precondition:
(**exists** (?from – location) (and (**at** ?t ?from) (**path-between** ?from ?dest))
No need for a new parameter – in *this* case…

Property Values 4

- What about *effects*?
  - Same ”natural” parameters: The *truck* and the *destination*

    -
      ```
      (:action drive :parameters (?t – truck ?dest – location)
                 :precondition …
                 :effect …
      )
      ```
  - ”Natural” effects:
    - The truck *ends up* at the destination:
    - The truck *is no longer* where it started:

      ```
      (at ?t ?dest)
      (not (at ?t …???…))
      ```
  - How do you find out where the truck was **before** the action?
    - Using an additional parameter still works:
      ```
      (not (at ?t ?from))
      ```
    - The value of ?from is constrained in the precondition – before
    - The value is *used* in the effect state