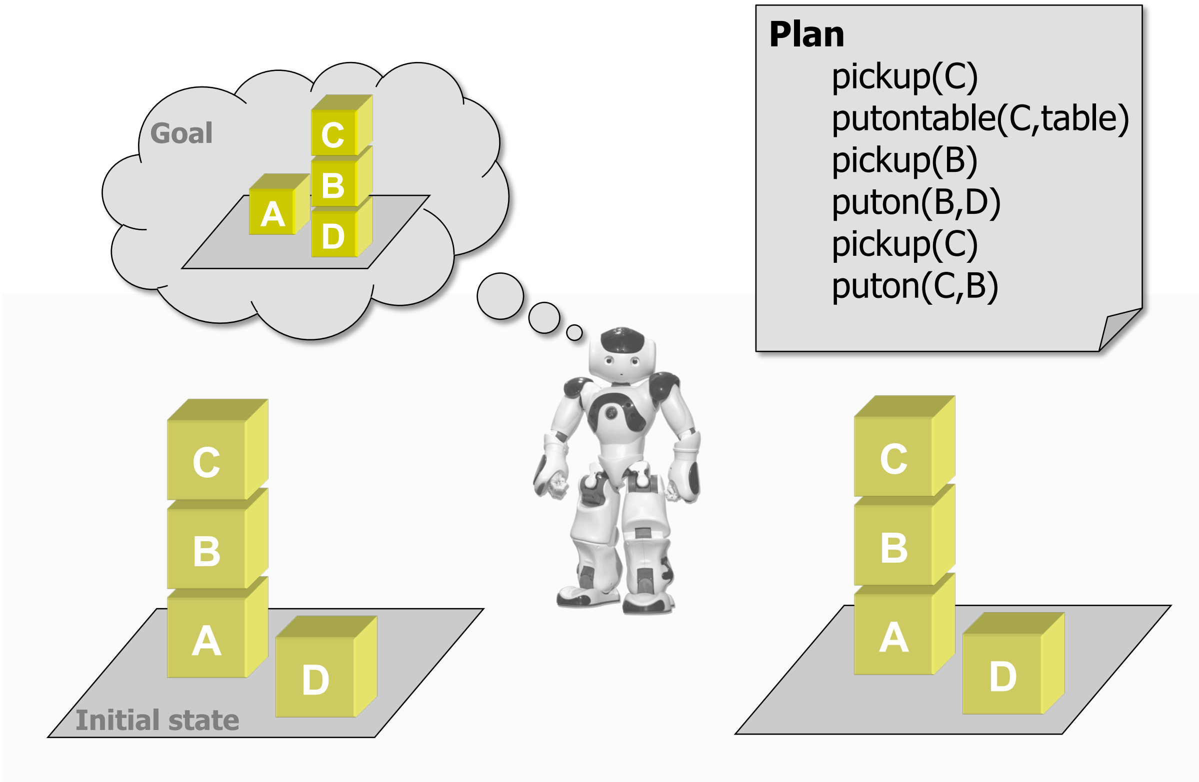
Automated Planning
A Logical Perspective

Roman Barták
Charles University in Prague, Faculty of Mathematics and Physics

Agent capabilities (actions)

Current situation

Goal



Situation calculus

– planning in first-order logic

Classical planning

– ad-hoc planning in simplified first-order logic

Control rules

– help from simple temporal logic

Planning as tabled logic programming

– fast and simple approach to planning

Situation Calculus

Actions and situations

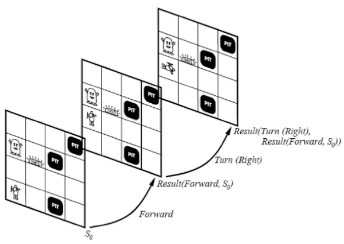
How to reason about actions and their effects in time?

In propositional logic we need a copy of each action for each time (situation):

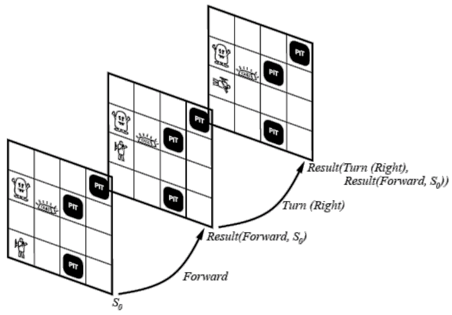
- $L^{t}_{x,y} \wedge FacingRight^{t} \wedge Forward^{t} \Rightarrow L^{t+1}_{x+1,y}$
- We need an upper bound for the number of steps to reach a goal but this will lead to a huge number of formulas.

Can we do it better in **first-order logic**?

- We do not need copies of axioms describing state changes; this can be implemented using a universal quantifier for time (situation)
- $\forall t$ P is the result of action A in time t+1



Situation calculus



- **actions** are represented by terms
  - **Go(x,y)**
  - **Grab(g)**
  - **Release(g)**
- **situation** is also a term
  - initial situation: S_0
  - situation after applying action *a* to state *s*: **Result(a,s)**
- **fluent** is a predicates changing with time
  - the situation is in the last argument of that term
  - **Holding(G, S_0)**
- **rigid** (eternal) **predicates**
  - **Gold(G)**
  - **Adjacent(x,y)**

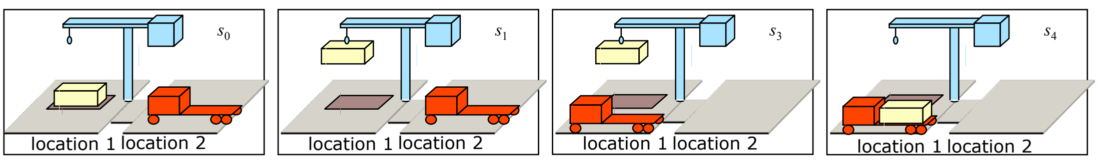
Situation calculus: plans

We need to reason about sequences of actions – about **plans**.

- Result([],s) = s
- Result([a|seq],s) = Result(seq, Result(a,s))

What are typical tasks related to plans?

- **projection task** – what is the state/situation after applying a given sequence of actions?
  - At(Agent, [1,1] , S_0) ∧ At(G, [1,2], S_0) ∧ ¬Holding(o, S_0)
  - At(G, [1,1], Result([Go([1,1],[1,2]),Grab(G),Go([1,2],[1,1])], S_0))
- **planning task** – which sequence of actions reaches a given state/situation?
  - ∃seq At(G, [1,1], Result(seq, S_0))



Situation calculus: actions

Each **action** can be described using two axioms:

- **possibility axiom:** Preconditions ⇒ Poss(a,s)
    - At(Agent,x,s) ∧ Adjacent(x,y) ⇒ Poss(Go(x,y),s)
    - Gold(g) ∧ At(Agent,x,s) ∧ At(g,x,s) ⇒ Poss(Grab(g),s)
    - Holding(g,s) ⇒ Poss(Release(g),s)
- **effect axiom:** Poss(a,s) ⇒ Changes
    - Poss(Go(x,y),s) ⇒ At(Agent,y,Result(Go(x,y),s))
    - Poss(Grab(g),s) ⇒ Holding(g,Result(Grab(g),s))
    - Poss(Release(g),s) ⇒ ¬Holding(g,Result(Release(g),s))

Beware! This is not enough to deduce that a plan reaches a given goal.

We can deduce At(Agent, [1,2], Result(Go([1,1],[1,2]), S_0))
but we **cannot deduce** At(G, [1,2], Result(Go([1,1],[1,2]), S_0))

Effect axioms describe what has been changed in the world but they say nothing about the property that everything else is not changed!

This is a so called **frame problem.**

Frame problem

We need to represent properties that are not changed by actions.

A simple **frame axiom** says what is not changed:

At(o,x,s) ∧ o≠Agent ∧ ¬Holding(o,s) ⇒
At(o,x,Result(Go(y,z),s))

- for F fluents and A actions we need O(FA) frame axioms
- This is a lot especially taking in account that most predicates are not changed.

Frame problem: better axioms

Can we use less axioms to model the frame problem?

- **successor-state axiom**

  $Poss(a,s) \Rightarrow$
  (fluent holds in $Result(a,s) \Leftrightarrow$
  fluent is effect of a $\lor$ (fluent holds in s $\land$ a does not change fluent))

  We get F axioms (F is the number of fluents) with O(AE) literals in total (A is the number of actions, E is the number of effects).

  Examples:

  $Poss(a,s) \Rightarrow$
  $(At(Agent,y,Result(a,s)) \Leftrightarrow a{=}Go(x,y) \lor (At(Agent,y,s) \land a \neq Go(y,z)))$

  $Poss(a,s) \Rightarrow$
  $(Holding(g,Result(a,s)) \Leftrightarrow a{=}Grab(g) \lor (Holding(g,s) \land a \neq Release(g)))$

Classical Planning

natural language processing

planning

machine learning

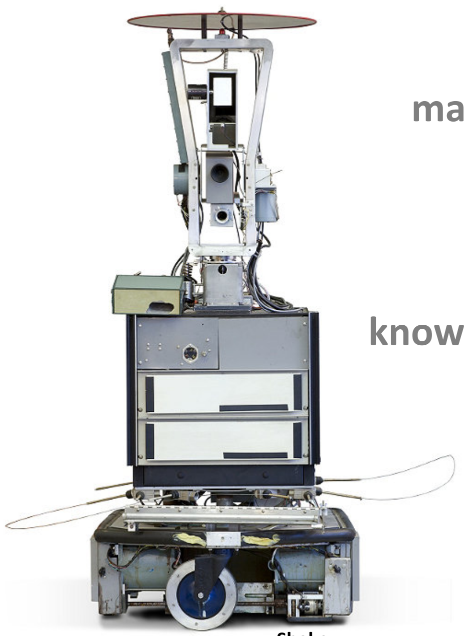
STRIPS

knowledge representation

computer vision

algorithm A*

robotic

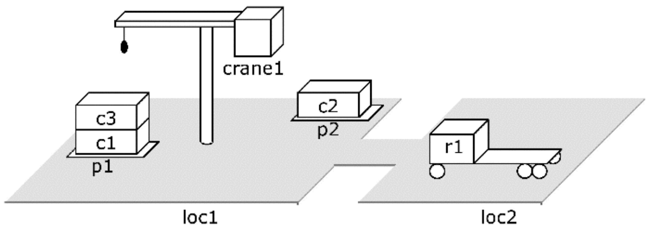


Shakey

Classical representation: states

We can simplify the full FOL model into a so called **classical representation** of planning problems.

State is a set of instantiated atoms (no variables). There is a finite number of states!



{attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

- The truth value of some atoms is changing in states:
  - **fluents**
  - *example: at(r1,loc2)*
- The truth value of some state is the same in all states
  - **rigid atoms**
  - *example: adjacent(loc1,loc2)*

We will use a classical **closed world assumption**.
An atom that is not included in the state does not hold at that state!

Classical representation: operators

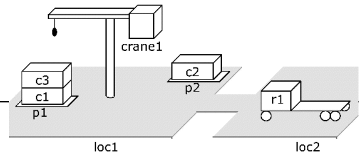
operator o is a triple (name(o), precond(o), effects(o))

- **name(o): name of the operator** in the form n(x_1,…,x_k)
  - n: a symbol of the operator (a unique name for each operator)
  - x_1,…,x_k: symbols for variables (operator parameters)
    - Must contain all variables appearing in the operator definition!
- **precond(o):**
  - literals that must hold in the state so the operator is applicable on it
- **effects(o):**
  - literals that will become true after operator application (only fluents can be there!)

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

An action is a fully instantiated operator

– substitute constants to variables



operator

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

action

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

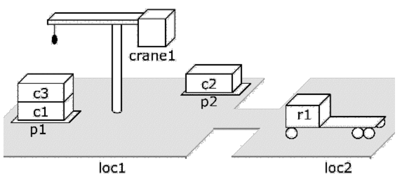
Notation:

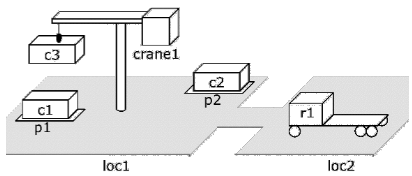
– S^+ = {positive atoms in S}
– S^- = {atoms, whose negation is in S}

Action **a** is **applicable** to state **s** if and only if
precond$^+$(**a**) $\subseteq$ **s** $\wedge$ precond$^-$(**a**) $\cap$ **s** = $\varnothing$

The result of application of action a to **s** is
γ(**s**,**a**) = (**s** – effects$^-$(**a**)) $\cup$ effects$^+$(**a**)

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





Classical representation: planning problem

Planning problem P is a triple (Σ, s_0, g):

- $\Sigma = (S, A, \gamma)$ is a **planning domain** (states, actions, transition)
- s_0 is an initial state, $s_0 \in S$
- g is a set of instantiated literals
  - state **s** satisfies the goal condition **g** if and only if $\mathbf{g}^{+} \subseteq \mathbf{s} \wedge \mathbf{g}^{-} \cap \mathbf{s} = \emptyset$
  - $S_g = \{\mathbf{s} \in S \mid \mathbf{s} \text{ satisfies } \mathbf{g}\}$ – a set of goal states

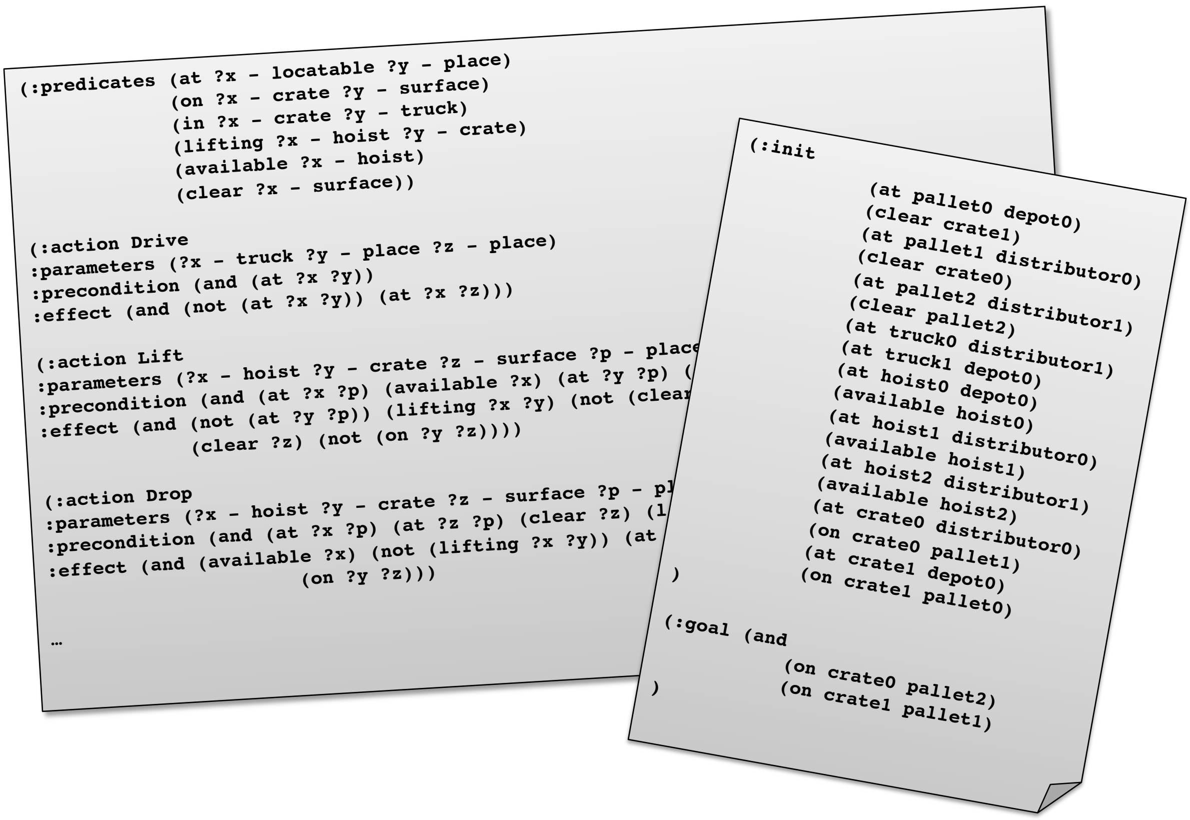
Plan is a sequence of actions $\langle a_1, a_2, \ldots, a_k \rangle$.

Plan $\langle a_1, a_2, \ldots, a_k \rangle$ is a **solution plan** for problem P iff $\gamma^*(s_0, \pi)$ satisfies the goal condition g.

Usually the planning problem is given by a triple (O, s_0, g).

- O defines the the operators and predicates used
- s_0 provides the particular constants (objects)

Planning Domain Description Language (PDDL)



```
(:predicates (at ?x - locatable ?y - place)
             (on ?x - crate ?y - surface)
             (in ?x - crate ?y - truck)
             (lifting ?x - hoist ?y - crate)
             (available ?x - hoist)
             (clear ?x - surface))

(:action Drive
:parameters (?x - truck ?y - place ?z - place)
:precondition (and (at ?x ?y))
:effect (and (not (at ?x ?y)) (at ?x ?z)))

(:action Lift
:parameters (?x - hoist ?y - crate ?z - surface ?p - place
:precondition (and (at ?x ?p) (available ?x) (at ?y ?p) (
:effect (and (not (at ?y ?p)) (lifting ?x ?y) (not (clear
             (clear ?z) (not (on ?y ?z))))

(:action Drop
:parameters (?x - hoist ?y - crate ?z - surface ?p - pl
:precondition (and (at ?x ?p) (at ?z ?p) (clear ?z) (l
:effect (and (available ?x) (not (lifting ?x ?y)) (at
                  (on ?y ?z)))

…
```

```
(:init
        (at pallet0 depot0)
        (clear crate1)
        (at pallet1 distributor0)
        (clear crate0)
        (at pallet2 distributor1)
        (clear pallet2)
        (at truck0 distributor1)
        (at truck1 depot0)
        (at hoist0 depot0)
        (available hoist0)
        (at hoist1 distributor0)
        (available hoist1)
        (at hoist2 distributor1)
        (available hoist2)
        (at crate0 distributor0)
        (on crate0 pallet1)
        (at crate1 depot0)
        (on crate1 pallet0)
)

(:goal (and
        (on crate0 pallet2)
        (on crate1 pallet1)
)
```

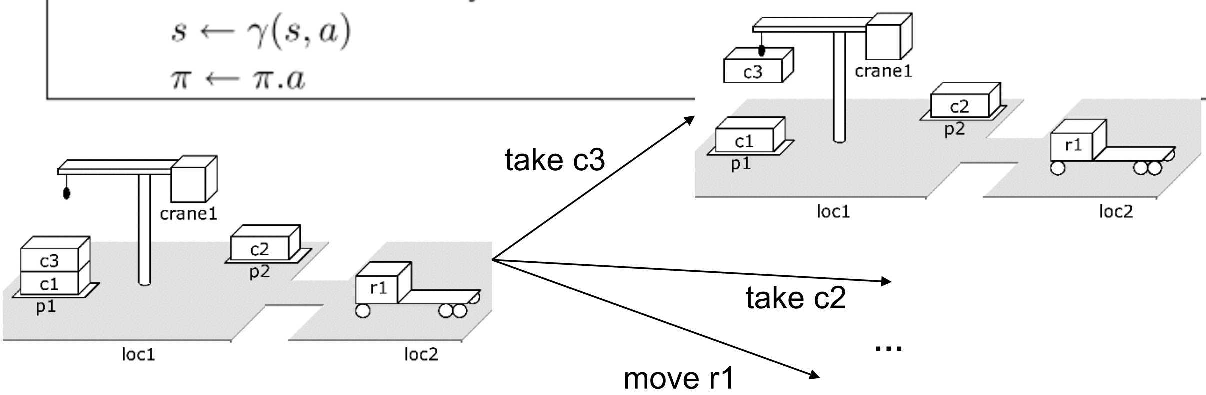
The search space corresponds to the state space of the planning problem.

- search nodes correspond to world states
- arcs correspond to state transitions by means of actions
- the task is to find a path from the initial state to some goal state

Basic approaches

- forward search (progression)
  - start in the initial state and apply actions until reaching a goal state
- backward search (regression)
  - start with the goal and apply actions in the reverse order until a subgoal satisfying the initial state is reached
  - lifting (actions are only partially instantiated)

Forward-search(O, s_0, g)
    $s \leftarrow s_0$
    $\pi \leftarrow$ the empty plan
    loop
        if s satisfies g then return π
        $E \leftarrow \{a | a$ is a ground instance of an operator in *O*,
                and $\mathrm{precond}(a)$ is true in $s\}$
        if $E = \emptyset$ then return failure
        nondeterministically choose an action $a \in E$
        $s \leftarrow \gamma(s, a)$
        $\pi \leftarrow \pi.a$



Control Rules

Pruning

Heuristics guide the planner towards a goal state by ordering alternative plans. They do not solve the problem with the **large number of alternatives**.

Can we **detect and prune bad alternatives**?

Example (blockworld)

- If a block is placed correctly (consistent with the goal) then any action that moves that block just enlarges the plan.
- If a block is on a wrong place and there is an action that moves it to the correct place then any action that moves the block elsewhere just enlarges the plan.

Domain dependent information can prune the search space, but the open question is how to express such information for a general planning algorithm.

- **control rules**

We need a formalism to express relations between the current world state and future states.

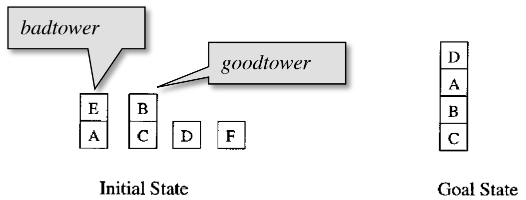
Simple temporal logic

– extension of first-order logic by **modal operators**

- $\phi_1 \cup \phi_2$ (until) — ϕ_1 is true in all states until the first state (if any) in which ϕ_2 is true
- $\Box\, \phi$ (always) — ϕ is true now and in all future states
- $\Diamond\, \phi$ (eventually) — ϕ is true now or in any future state
- $\bigcirc\, \phi$ (next) — ϕ is true in the next state
- GOAL(ϕ) — ϕ (no modal operators) is true in the goal state

– ϕ is a logical formula expressing relations between the objects of the world (it can include modal operators)

- *Goodtower* is a tower such that no block needs to be moved. *Badtower* is a tower that is not good.



$$goodtower(x) \triangleq clear(x) \land \neg \mathrm{GOAL}(holding(x)) \land goodtowerbelow(x)$$

$$\begin{aligned} goodtowerbelow(x) \triangleq\ & (ontable(x) \land \neg\exists[y{:}\mathrm{GOAL}(on(x,y))]\,)) \\ & \lor \exists[y{:}on(x,y)]\ \neg\mathrm{GOAL}(ontable(x)) \land \neg\mathrm{GOAL}(holding(y)) \land \neg\mathrm{GOAL}(clear(y)) \\ & \quad \land \forall[z{:}\mathrm{GOAL}(on(x,z))]\ z = y \land \forall[z{:}\mathrm{GOAL}(on(z,y))]\ z = x \\ & \quad \land goodtowerbelow(y) \end{aligned}$$

$$badtower(x) \triangleq clear(x) \land \neg goodtower(x)$$

Control rule:

goodtower remains goodtower

$$\begin{aligned} \Box \big(& \forall[x{:}clear(x)]\ goodtower(x) \Rightarrow \bigcirc(clear(x) \lor \exists[y{:}on(y,x)]\ goodtower(y)) \\ & \land badtower(x) \Rightarrow \bigcirc(\neg\exists[y{:}on(y,x)]\,) \\ & \land (ontable(x) \land \exists[y{:}\mathrm{GOAL}(on(x,y))]\ \neg goodtower(y)) \\ & \qquad \Rightarrow \bigcirc(\neg holding(x))\big) \end{aligned}$$

do not put anything on badtower

do not take a block from a table until you can put it on a goodtower

Progression

To use control rules in planning we need to express how the formula changes when we **go from state s_i to state s_{i+1}**.

- We look for a formula progr(ϕ, s_i) that is true in s_{i+1}, if ϕ is true in state s_i

- ϕ does not contain any modal operator
  - progr(ϕ, s_i) = true if $s_i \vdash \phi$
    = false if $s_i \vdash \phi$ does not hold
- ϕ with logical connectives
  - progr($\phi_1 \wedge \phi_2$, s_i) = progr(ϕ_1, s_i) $\wedge$ progr(ϕ_2, s_i)
  - progr($\neg\phi$, s_i) = $\neg$progr(ϕ, s_i)
- ϕ with quantifiers (no function symbols, just k constants c_j)
  - progr($\forall x\ \phi$, s_i) = progr($\phi\{x/c_1\}$, s_i) $\wedge$ … $\wedge$ progr($\phi\{x/c_k\}$, s_i)
  - progr($\exists\ x\ \phi$, s_i) = progr($\phi\{x/c_1\}$, s_i) $\vee$ … $\vee$ progr($\phi\{x/c_k\}$, s_i)
- ϕ with modal operators
  - progr($\phi_1 \cup \phi_2$, s_i) = (($\phi_1 \cup \phi_2$) $\wedge$ progr (ϕ_1, s_i)) $\vee$ progr (ϕ_2, s_i)
  - progr($\Box\ \phi$, s_i) = ($\Box\ \phi$) $\wedge$ progr(ϕ, s_i)
  - progr($\Diamond\ \phi$, s_i) = ($\Diamond\ \phi$) $\vee$ progr(ϕ, s_i)
  - progr($\bigcirc\ \phi$, s_i) = ϕ

Technical notes:

- progress(ϕ, s_i) is obtained from progr(ϕ, s_i) by cleaning (true $\wedge$ d $\rightarrow$ d, $\neg$true $\rightarrow$ false, …)
- Can be extended to a sequence of states $\langle s_0, \dots, s_n\rangle$
  progress(ϕ, $\langle s_0, \dots, s_n\rangle$) = ϕ if n = 0
  = progress(progress(ϕ, $\langle s_0, \dots, s_{n-1}\rangle$), s_n) otherwise

Planning with control rules

Forward state-space planning guided by control rules.

- If a partial plan S_π violates the control rule progress(ϕ, S_π), then the plan is not expanded.

```
STL-plan(O, s0, g, φ)
    s ← s0
    π ← the empty plan
    loop
        if φ = false then return failure
        if s satisfies g then return π
        A ← {a | a is a ground instance of an operator in O
                  and precond(a) is true in s}
        if A = ∅ then return failure
        nondeterministically choose an action a ∈ A
        s ← γ(s, a)
        π ← π.a
        φ ← progress(φ, s)
```

a partial plan violates the control rule ϕ

a complete plan found

actions applicable to state s

a new state after the action

control rule progression ϕ

Control rules in practice - IPC 2002

Planners with control rules

Forward planning

Domain	# insts	TLPlan	TALPlanner	FF
Depots	22	**22**	**22**	**22**
DriverLog	20	**20**	**20**	15
Zenotravel	20	**20**	**20**	**20**
Rovers	20	**20**	**20**	**20**
Satellite	20	**20**	**20**	**20**
Total	-	**894 (100%)**	**610 (100%)**	237 (83%)

problems solved

Control rules in practice

```
…
(forall (?x ?y) (on ?x ?y)
          (and
          (print ?stream "(on ~A ~A) --" ?x ?y)
          (implies (good-tower ?x)
                   (print ?stream "  (good-tower ~A) " ?x))
          (implies (bad-tower ?x)
                   (print ?stream "  (bad-tower ~A) " ?x))
          (implies (good-tower ?y)
                   (print ?stream "  (good-tower ~A)~%" ?y))
          (implies (bad-tower ?y)
                   (print ?stream "  (bad-tower ~A)~%" ?y))))

   (forall (?x ?y) (in ?x ?y)
          (and
          (print ?stream "(in ~A ~A) " ?x ?y)
          (exists (?l) (at ?y ?l)
                   (print ?stream "(at ~A ~A) " ?y ?l))
          (implies (has-goal-loc ?x)
                   (print ?stream "(crate-goal-location ~A) = ~A (crate-goal-surface ~A)= ~A"
                            ?x (crate-goal-location ?x) ?x (crate-goal-surface ?x)))

          (print ?stream "~%")))
…
```

933 lines of code!

Planning as Tabled Logic Programming

Logic programming

Logic programming (Prolog) represents knowledge in the form of Horn clauses and uses backward chaining as a method to answer queries (with unification and backtracking to explore alternatives).

rule head   rule body

```
criminal(X) :-
    american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
owns(nono,m1).
missile(m1).
sells(west,X,nono) :-
    missile(X), owns(nono,X).
weapon(X) :-
    missile(X).
hostile(X) :-
    enemy(X,america).
american(west).
enemy(nono,america).

?- criminal(west).
```

```
?- criminal(west).
?- american(west), weapon(Y),
   sells(west,Y,Z), hostile(Z).
?- weapon(Y), sells(west,Y,Z),
   hostile(Z).
?- missile(Y), sells(west,Y,Z),
   hostile(Z).
?- sells(west,m1,Z), hostile(Z).
?- missile(m1), owns(nono,m1),
   hostile(nono).
?- owns(nono,m1), hostile(nono).
?- hostile(nono).
?- enemy(nono,america).
?- true.
```

The idea:

Tabling memorizes calls and their **answers** in order to prevent infinite loops and to limit redundancy.

An example (in Picat):

```
table
fib(0) = 1.
fib(1) = 1.
fib(N) = fib(N-1) + fib(N-2).
```

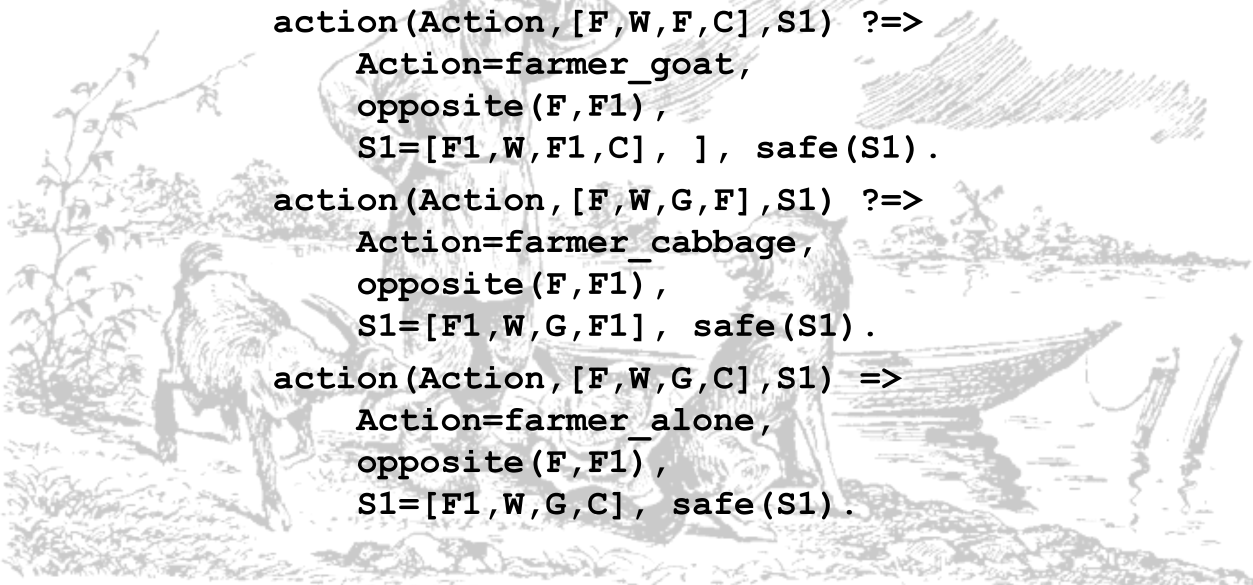
Without tabling, **`fib(N)`** takes exponential time in `N`.

With tabling, **`fib(N)`** takes linear time.

Forward planning in Picat language (using tabling):

```
table (+,-,min)
plan(S,Plan,Cost),final(S) =>
    Plan=[],Cost=0.
plan(S,Plan,Cost) =>
    action(Action,S,S1,ActionCost),
    plan(S1,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = Cost1+ActionCost.
```

Example: The farmer's problem



Locations of
Farmer, Wolf, Goat, and Cabbage

```
action(Action,[F,F,G,C],S1) ?=>
    Action=farmer_wolf,
    opposite(F,F1),
    S1=[F1,F1,G,C], safe(S1).
action(Action,[F,W,F,C],S1) ?=>
    Action=farmer_goat,
    opposite(F,F1),
    S1=[F1,W,F1,C], ], safe(S1).
action(Action,[F,W,G,F],S1) ?=>
    Action=farmer_cabbage,
    opposite(F,F1),
    S1=[F1,W,G,F1], safe(S1).
action(Action,[F,W,G,C],S1) =>
    Action=farmer_alone,
    opposite(F,F1),
    S1=[F1,W,G,C], safe(S1).
```

NoMystery problem

A truck moves between locations to pickup and deliver packages while consuming fuel during moves.

- setting:
  - initial locations of packages and truck
  - goal locations of packages
  - initial fuel level, fuel cost for moving between locations
- possible actions: **load**, **unload**, **drive**
- assumption: track can carry any number of packages

NoMystery model

State representation:

```
s(Loc,Fuel,LoadedCGs,Cargoes)
LoadedCGs = [CargoGoal]
Cargoes = [[CargoLoc|CargoGoal]]
```

Actions

- **Unload** package only at its destination
- **Load** all not-delivered packages at current location
- **Move** somewhere

Post-processing

- Returning back the names of cargoes

NoMystery actions

```
action(Action, s(Loc,Fuel,LoadedCGs,Cargoes), NextState),
    select(Loc,LoadedCGs,LoadedCGs1)
=>
    Action = unload(Loc,Loc),
    NextState = s(Loc,Fuel,LoadedCGs1,Cargoes).

action(Action, s(Loc,Fuel,LoadedCGs,Cargoes), NextState),
    select([Loc|CargoGoal],Cargoes,Cargoes1)
=>
    insert_ordered(CargoGoal,LoadedCGs,LoadedCGs1),
    Action = load(Loc,CargoGoal),
    NextState = s(Loc,Fuel,LoadedCGs1,Cargoes1).

action(Action, s(Loc,Fuel,LoadedCGs,Cargoes), NextState)
?=>
    Action = drive(Loc,Loc1),
    NextState = s(Loc1,Fuel1,LoadedCGs,Cargoes),
    fuelcost(Cost,Loc,Loc1),
    Fuel1 is Fuel-Cost,
    Fuel1 >= 0.
```

Comparison to PDDL planners

no tabling used

no heuristics used

IPC 2014 winner

Domain	# insts	Picat	Picat-nt	Picat-nh	Symba
Barman	14	**14**	0	**14**	6
Cave	20	**20**	0	**20**	3
Childsnack	20	**20**	**20**	**20**	3
Citycar	20	**20**	17	18	17
Floortile	20	**20**	0	**20**	**20**
GED	20	**20**	19	13	19
Parking	20	**11**	4	0	1
Tetris	17	**13**	**13**	9	10
Transport	20	**10**	0	4	8

number of optimally solved problems

Comparison to control rules

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**22**	**22**	**22**	**22**
Zenotravel	20	**20**	**20**	**20**	**20**
Satellite	20	**20**	**20**	**20**	**20**

problems solved

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**21.90**	19.93	20.53	18.63
Zenotravel	20	**19.13**	18.56	18.96	17.30
Satellite	20	**19.95**	18.90	17.10	17.68

quality score

Domain	PDDL	Picat	TLPlan
Depots	42	147	933
Zenotravel	61	111	308
Satellite	75	122	186

encoding size

- using **structured representation** of states instead of factored representation
  - symmetry breaking
- **deterministic** vs. non-deterministic actions
  - smaller branching factor during search
- using **domain knowledge**
  - smaller branching factor during search
- **no prior grounding** of actions
  - smaller memory consumption

Roman Barták
Charles University in Prague, Faculty of Mathematics and Physics
bartak@ktiml.mff.cuni.cz