Planificación clásica: PDDL 1.2

Planificación y PDDL, Parte II (Capítulos 4–7)

Francisco Martín Rico

Intelligent Robotics Lab, URJC



Índice

1. Tipos, constantes e igualdad
2. Condiciones más ricas
3. Efectos más ricos
4. Modelar problemas clásicos: patrones y decisiones
5. Resumen

1 Tipos, constantes e igualdad

2 Condiciones más ricas

3 Efectos más ricos

4 Modelar problemas clásicos: patrones y decisiones

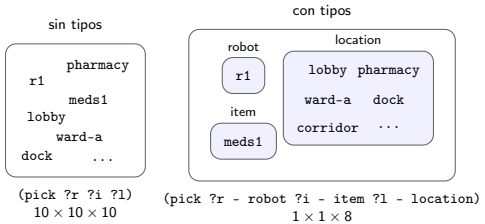
5 Resumen

Lo que cuestan los predicados de tipo

```
(:action pick
  :parameters (?r ?i ?l)
  :precondition (and (robot ?r) (item ?i) (location ?l)
                     (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
  ...
```

Predicados de tipo (Capítulo 2): correctos, cualquier planificador. Tres costes:

- **Ruido**: un tercio de cada precondición, 10 hechos de `courier-p01`
- **No comprueban nada**: `(item-at pharmacy meds1)` pasa `check`; el planificador: «unsolvable»
- **Multiplican las candidatas**: 10^3 por acción, 3000 en total; 30 aplicables



Con **tipos**, cada parámetro solo toma objetos de la caja de su tipo.

Tipos: el dominio del mensajero con tipos

```
(:requirements :strips :typing)

(:types robot location item)

(:predicates
  (connected ?from ?to - location)
  (robot-at ?r - robot ?l - location)
  (item-at ?i - item ?l - location)
  (carrying ?r - robot ?i - item)
  (free ?r - robot))

(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))

(:action pick
  :parameters (?r - robot ?i - item ?l - location)
  :precondition (and (robot-at ?r ?l) (item-at ?i ?l) (free ?r))
  :effect (and (not (item-at ?i ?l)) (not (free ?r))
               (carrying ?r ?i)))
```

Un **tipo** es un nombre para un conjunto de objetos; requisito `:typing`

- Sección `(:types robot location item)`
- **Variables con tipo**: `?r - robot`; un tipo vale para todas las variables sin tipo anteriores: `?from ?to - location`
- Ni predicados ni hechos de tipo

El mismo plan de 7 pasos para `courier-p01`: los tipos cambian lo que puede escribir el *modelador*, no lo que puede hacer el robot.

Lo que detecta el comprobador, lo que gana el planificador

```
$ kairos check courier-domain.pddl courier-p01-swapped.pddl
courier-p01-swapped.pddl:17:14: error: in the initial state: argument 1 of '
    item-at' must be item, 'pharmacy' is location [semantic.type_mismatch]
...
invalid
```

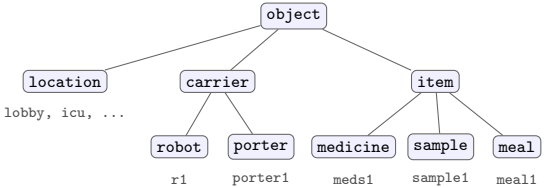
Acción	Sin tipos	Con tipos
`move`	$10^3 = 1000$	$1 \times 8 \times 8 = 64$
`pick`	$10^3 = 1000$	$1 \times 1 \times 8 = 8$
`drop`	$10^3 = 1000$	$1 \times 1 \times 8 = 8$
Total	3000	80

- Fichero, línea, columna y una conjetura: «arguments 1 and 2 seem to be swapped»
- Candidatas: el producto de los tamaños de los tipos, no una potencia de los objetos
- Los dos modelos: las mismas 30 acciones instanciadas. Se gana en la instanciación, no en la búsqueda
- `courier-p02` (2 robots, 3 artículos): 6591 frente a 224

Jerarquías de tipos

```
(:types
  location
  robot porter - carrier
  medicine sample meal - item)
```

- **Subtipo** y **supertipo**, con el guion
- Un objeto de un tipo también es un objeto de *todos sus supertipos*
- `?c - carrier`: robots y celadores; `?p - porter`: solo celadores
- Raíz: el tipo predefinido `object`; sin tipo equivale a `object`



Regla práctica

Declara un predicado o una acción sobre un supertipo cuando vale para toda la familia, y sobre un subtipo cuando vale para una parte de ella.

El equipo: una acción, dos clases de portador

```
;; Any carrier moves, picks and drops any item.
(:action move
  :parameters (?c - carrier ?from ?to - location)
  :precondition (and (carrier-at ?c ?from)
                     (connected ?from ?to))
  :effect (and (not (carrier-at ?c ?from))
               (carrier-at ?c ?to)))
;; Only a porter serves, and only a meal is served.
(:action serve
  :parameters (?p - porter ?m - meal ?l - location)
  :precondition (and (carrier-at ?p ?l) (carrying ?p ?m))
  :effect (and (not (carrying ?p ?m)) (free ?p)
               (served ?m ?l))))
```

- Mejor plan, 13 pasos: los mismos `move` y `pick` con `r1` y con `porter1`
- Un plan en el que sirve el robot: `kairos validate` dice «argument 1 must be porter, 'r1' is robot»

Paso 0: clases antes que propiedades

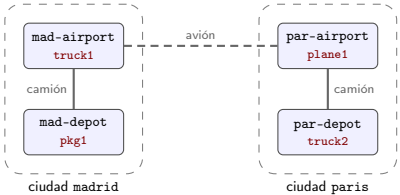
Clase	Es un	Por qué es clase propia
`location`		las cosas están *en* él
`carrier`		se mueve, coge y deja
`robot`, `porter`	`carrier`	solo sirve un celador
`item`		se coge y se deja
`meal`	`item`	solo se sirve una comida

Prueba de un tipo: *algún predicado o acción trata a estos objetos de forma distinta*. Si no, o si cambia mientras se ejecuta el plan, es una propiedad.

Un segundo dominio: logistics

```
(:types
  city
  airport depot - place
  truck airplane - vehicle
  package vehicle - physobj)
;; A truck drives between any two places of one city.
(:action drive
  :parameters (?t - truck ?from ?to - place ?c - city)
  :precondition (and (at ?t ?from)
                     (in-city ?from ?c) (in-city ?to ?c))
  :effect (and (not (at ?t ?from)) (at ?t ?to)))

;; An airplane flies only between airports.
(:action fly
  :parameters (?a - airplane ?from ?to - airport)
  :precondition (at ?a ?from)
  :effect (and (not (at ?a ?from)) (at ?a ?to))))
```



- `truck` → `vehicle` → `physobj` → `object`
- Un solo `load` para camiones y aviones
- «Los aviones solo aterrizan en aeropuertos»: en la lista de parámetros, sin precondición
- 12 pasos; `validate` rechaza un plan que vuela a `par-depot`

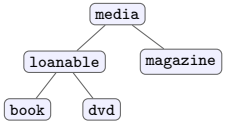
Un parámetro de varios tipos: `either`

```
(:types book dvd magazine member)

(:predicates
  (on-shelf ?x - (either book dvd magazine))
  (borrowed ?x - (either book dvd) ?m - member)
  (reading ?g - magazine ?m - member)
  (can-borrow ?m - member))      ; one loan at a time

(:action borrow
  :parameters (?m - member ?x - (either book dvd))
  :precondition (and (on-shelf ?x) (can-borrow ?m))
  :effect (and (not (on-shelf ?x)) (not (can-borrow ?m))
               (borrowed ?x ?m)))
```

- `(either t1 t2 ...)`: la unión de varios tipos; parte de `:typing`
- Una revista prestada: `check` dice «must be (either book dvd), 'mag1' is magazine»



la unión tiene nombre

- Una unión repetida es un concepto que aún no tiene nombre
- La jerarquía se lee mejor y es más portable (LAMA rechaza `either` en los parámetros; ENHSP, en todas partes)
- Guarda `either` para una unión que *atraviesa* la jerarquía

Constantes

```
(:constants pharmacy - location)
;; The pharmacist hands a medicine to a robot. The action
;; names the pharmacy: it has no location parameter.
(:action dispense
  :parameters (?r - robot ?m - medicine)
  :precondition (and (robot-at ?r pharmacy) (in-stock ?m)
                     (free ?r))
  :effect (and (not (in-stock ?m)) (not (free ?r))
               (carrying ?r ?m))))
```

```
$ kairos courier-pharmacy-domain.pddl
    courier-pharmacy-p01.pddl
; cost = 21
(move r1 dock lobby)
(move r1 lobby pharmacy)
(dispense r1 meds2)
...
```

- **Constante**: objeto declarado en el dominio, en todos los problemas
- `:constants`, sin requisito; se usa como una variable
- Los problemas la usan pero *no la declaran*
- Los pasos del plan solo llevan lo que se eligió

La prueba

¿Lo nombra alguna regla del dominio? El vestíbulo está en todos los problemas, pero ninguna regla lo nombra: no es una constante.

Igualdad: un robot no es su propio testigo

```
$ kairos
    courier-check-noeq-domain.
    pddl courier-check-p01.pddl
; cost = 9
...
(pick r1 meds1 pharmacy)
(verify r1 r1 meds1 pharmacy)
...
```

- Dos parámetros siempre pueden tomar el mismo objeto si su tipo vale para los dos
- `(= ?a ?b)`: requisito `:equality`; casi siempre negada

```
;; Robot ?r carries the medicine, robot ?w witnesses.
(:action verify
  :parameters (?r ?w - robot ?m - medicine ?l - location)
  :precondition (and (robot-at ?r ?l) (robot-at ?w ?l)
                     (carrying ?r ?m)
                     (not (= ?r ?w)))
  :effect (verified ?m)))
```

Ahora p01 (un robot) es irresoluble; en p02 el mejor plan tiene `(verify r1 r2 meds1 corridor)`

Paso E: la prueba de la trampa

Para dos parámetros de tipos que se solapan: un objeto de ese tipo y un objetivo que necesite dos. Si sale un plan, lee el paso en el que el mismo nombre aparece dos veces.

El método MODEL: clases de cosas

Paso	Preguntas nuevas
O	**¿Qué clases de cosas hay?** Un tipo si algún predicado o acción las trata de forma distinta y ningún objeto cambia de clase
O	**¿Una jerarquía?** Predicados compartidos: supertipo; acciones para una parte: subtipo
O	**¿En todos los problemas?** Si lo nombra una regla: una constante
D	**¿Pueden dos parámetros ser el mismo objeto?** Si no, y nada lo impide: una desigualdad

Paso E: `kairos check` sobre cada fichero; un plan erróneo por regla de subtipo, rechazado por `kairos validate`

Predicados de tipo o tipos (`courier-p02`)

	Predicados de tipo	Tipos
Fidelidad	admite intercambios	intercambio: error
Tamaño	6591 candidatas	224 candidatas
	124 acc. instanciadas	124 acc. instanciadas
Guía	la misma búsqueda	la misma búsqueda
Portabilidad	cualquier planificador	cualquier clásico

Veredicto: usa tipos; deja los predicados de tipo para una clase que cambia o que atraviesa el árbol.

Una constante o un parámetro (la regla de la farmacia) Constante: el objeto forma parte de las *reglas*. Parámetro y predicado estático `dispensary`: parte de la *situación*. Los mismos planes de 21 pasos, 64 acciones instanciadas. En caso de duda, el parámetro.

Errores comunes con los tipos

Un tipo que se traga la variable anterior

`(?r ?i - item ?l - location)`: `?r` es un artículo. `check` informa de «argument 1 of 'robot-at' must be robot, '?r' is item». Mira los guiones de la declaración.

Tipos sin `:typing`

Kairos avisa (con `check` o `-v`); otros se detienen en el primer guion. Lo mismo con `:equality`.

Un objeto declarado con el supertipo

`meds1 meds2 - item`: se rechaza donde se espera un `medicine`, o en silencio nunca se elige. Usa el tipo más específico.

Una constante declarada otra vez en el problema

«'pharmacy' is already a domain constant»: un aviso en Kairos; un planificador más estricto rechaza el problema.

¿Variables distintas, objetos distintos?

PDDL legal, ningún mensaje: solo la prueba de la trampa lo encuentra. Arreglo: `(not (= ?r ?w))`.

1 Tipos, constantes e igualdad

2 Condiciones más ricas

3 Efectos más ricos

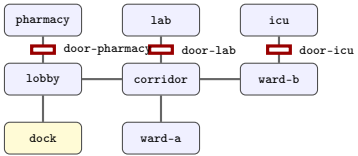
4 Modelar problemas clásicos: patrones y decisiones

5 Resumen

Condiciones sobre la ausencia: precondiciones negativas

```
;; Through a door: only if it is open.
(:action move-through
  :parameters (?r - robot ?d - door ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (joins ?d ?from ?to)
                     (open ?d))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))

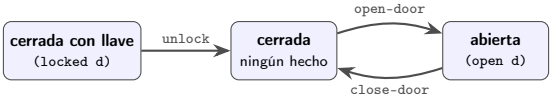
;; A closed door: one that is not open and not locked.
(:action open-door
  :parameters (?r - robot ?d - door ?a ?b - location)
  :precondition (and (robot-at ?r ?a) (joins ?d ?a ?b)
                     (not (open ?d)) (not (locked ?d)))
  :effect (open ?d))
```



Pasos sin puerta: `connected`, se cruzan con `move`.
Con puerta: `joins`, se cruzan con `move-through`.

- `:negative-preconditions`: `(not (p ...))` se cumple cuando el hecho *no está*
- `not` en un efecto borra; en una precondición, comprueba

Tres estados con dos predicados; objetivos negativos



- No hay predicado `closed`: cerrada es ni abierta ni cerrada con llave
- **Hipótesis del mundo cerrado**: lo que no está en la lista es falso
- Todas las puertas cerradas: ningún hecho de puertas en el estado inicial

Objetivo negativo (`courier-doors-p02`):

```
(:goal (and (item-at meds1 ward-a)
            (not (open door-pharmacy))))
```

- Un paso más: `close-door` al salir de la farmacia
- A menudo cierto desde el principio: objetivos solo negativos pueden dar el plan vacío
- POPF y OPTIC no aceptan condiciones negativas

Ensayar las puertas: la prueba de la trampa se gana su sitio

El paso E con las puertas: cuatro problemas

Trivial: p01, una puerta cerrada. *Típica*: p03, puerta de la UCI cerrada con llave, robot con tarjeta (12 pasos). *Sin plan*: p04, sin la tarjeta. *Trampa*: p04 con cada versión anterior del dominio.

v0: `open-door` solo con `(not (open ?d))`

```
$ kairos courier-doors-v0-domain.pddl
    courier-doors-p04.pddl
; cost = 11
...
(open-door r1 door-icu ward-b icu)
...
```

- Una puerta cerrada con llave no está abierta, así que el robot la abrió
- Dos predicados, cuatro combinaciones: *abierta y cerrada con llave* se volvió alcanzable
- Si una ausencia define un estado, mira qué otros estados tienen la misma ausencia

Alternativas: or e imply

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to)
                     (or (not (restricted ?to))
                         (has-keycard ?r)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

O, tal como se lee la regla en la historia:

```
                     (imply (restricted ?to)
                            (has-keycard ?r))
```

- `:disjunctive-preconditions`
- `(or a b ...)`: se cumple alguna parte
- `(imply a b)`: lo mismo que `(or (not a) b)`
- `courier-access-p01`: solo `r2` tiene tarjeta y hace todo el viaje de 9 pasos
- Nadie con tarjeta (`p02`): la nota tiene dos líneas sobre `move`, una por alternativa

Algún objeto: exists

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition
    (and (robot-at ?r ?from) (connected ?from ?to)
         (or (not (restricted ?to))
             (has-keycard ?r)
             (exists (?e - robot)          ; an escort
               (and (has-keycard ?e) (robot-at ?e ?from)))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

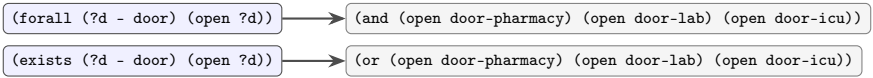
```
$ kairos courier-escort-domain.pddl
    courier-escort-p01.pddl
; cost = 8
(move r2 ward-a corridor)
...
(move r1 ward-b icu)
```

- `(exists (?x - t) c)`: c se cumple para algún objeto del tipo `t`;
  `:existential-preconditions`
- El plan no nombra la escolta

Llaves y cofres

«Tienes alguna llave que encaja» es un `exists`. Si una llave se rompe al usarla, el efecto debe borrarla: la llave pasa a ser un parámetro, `(open oak iron)`.

Todos los objetos: `forall`



- `:universal-preconditions`; los dos: `:quantified-preconditions`
- Abrevia sobre los objetos del problema
- Tipo vacío: `forall` cierto, `exists` falso
- `not exists` = `forall not`

Mundo de bloques: `clear` es la ausencia de `on`

```
;; ?y must be another block, with nothing on it.
(:action stack
  :parameters (?x ?y - block)
  :precondition (and (holding ?x) (not (= ?x ?y))
                     (not (exists (?z - block) (on ?z ?y))))
  :effect (and (not (holding ?x)) (hand-empty)
               (on ?x ?y)))
```

El mismo plan de 6 pasos que el dominio clásico. Allí la desigualdad estaba oculta: un bloque sujeto no estaba `clear`.

Un solo move para todos los pasos; objetivos cuantificados

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition
    (and (robot-at ?r ?from) (connected ?from ?to)
         (forall (?d - door)
           (imply (joins ?d ?from ?to) (open ?d))))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

`(forall (?x - t) (imply a b))`: todo `x` que cumple `a` cumple también `b`. Sin puerta en el paso: todas las implicaciones son ciertas.

```
(:goal (and (forall (?m - medicine) (item-at ?m icu))
            (forall (?d - door) (not (open ?d))))))
```

Independiente de los objetos; mejor plan de 22 pasos

Prueba de la trampa p02:

```
(forall (?i - item) (not (item-at ?i corridor)))
```

```
$ kairos courier-ward-domain.pddl
      courier-ward-p02.pddl
; cost = 1
(pick r1 meal1 corridor)
```

- Un artículo en la pinza no está en ninguna sala
- p03 añade `(forall (?r - robot) (free ?r))`: coger, mover, dejar
- Un objetivo sobre una ausencia debe decir dónde pueden acabar las cosas

Qué hace un planificador con las condiciones ricas

- **Los cuantificadores se expanden** sobre los objetos; anidados, se multiplican
- **Las partes estáticas se evalúan**: `(joins ?d ?from ?to)` deja una sola condición sobre puertas
- **Las disyunciones parten las acciones**, una por alternativa; un `exists` se parte en una por objeto
- **La negación**: se trata directamente, o con un segundo hecho opuesto

Codificación	Instanciadas	Estados
`or` (acceso)	64	352
dos acciones (partida)	52	352
`exists` (escolta)	98	
escolta como parámetro	74	

Las 12 de más son movimientos de `r2` en los que se cumplen las dos partes del `or`; los duplicados llevan al mismo sucesor.

Señal de alarma

Un `forall` cuyo cuerpo tiene un `or`, `imply` o `exists` sobre hechos que cambian: 2^{10} para diez condiciones así.

El método MODEL: ausencia, alguno, todos

Paso	Preguntas nuevas
O	**¿La propiedad es de sí o no, o tiene varios valores?** Guarda un lado, el que es cierto con menos frecuencia; con tres valores o más, comprueba que las combinaciones sin sentido no pueden alcanzarse
D	**¿La condición trata de una ausencia?** `not`
D	**¿Trata de algún objeto?** `exists` si la acción no le hace nada; un parámetro si se lo hace, o si el plan debe nombrarlo
D	**¿Trata de todos los objetos?** `forall`, con `imply` para seleccionar
D	**¿Hay varias maneras de satisfacerla?** `or`, o una acción por manera

Paso E: dos pruebas de la trampa más

- Para cada estado definido por una ausencia, busca otro con la misma ausencia (la puerta cerrada con llave que «no estaba abierta»)
- Para cada objetivo sobre una ausencia o sobre «todos», pregúntate cómo podría cumplirse por una razón tonta (el artículo en la pinza; un tipo sin objetos)

Alternativas comparadas

Elección	Medido	Veredicto
`not` o un `closed` complementario	Los mismos planes; 21 frente a 24 hechos, 42 acciones instanciadas y 102 estados cada uno	`not` para una propiedad de sí o no: nada que acompasar. `closed` para POPF y OPTIC. Un hecho olvidado: una puerta cerrada, o que no es nada
`or` o dos acciones	64 frente a 52 acciones instanciadas (12 duplicadas); 352 estados cada uno	`or` si las alternativas son un detalle de una regla con los mismos efectos; dos acciones si van a diferir o el plan debe distinguirlas
Objetivo `forall` o una lista	46 frente a 45 hechos; 23 679 estados cada uno	`forall` para una regla («todo entregado»), problemas a mano; la lista, para problemas generados o planificadores sin cuantificadores (LPG-td, Metric-FF)

- Los mismos planes en cada par: se elige por Fidelidad, Claridad y Portabilidad

Predicados complementarios

Se desacompasan si un efecto olvida uno: el Cap. 7 lo llama *gemelos desacompasados*.

Errores comunes con las condiciones

`and` bajo un `forall` donde tocaba `imply`

```
(forall (?d - door)
  (and (joins ?d ?from ?to) (open ?d)))
```

Todas las puertas deben unir esas salas: ningún robot se mueve nunca, y Kairos solo dice «every state reachable ... was searched (1 states)». Bajo `forall`, se selecciona con `imply`; bajo `exists`, con `and`.

Un `forall` que se cumple sin objetos

Medicamentos declarados - `item`: el objetivo `(forall (?m - medicine) ...)` se cumple al principio, ; `cost = 0`. Cuenta los objetos del tipo.

Una variable cuantificada sin tipo

`(forall (?m) (item-at ?m icu))` pide que salas y robots estén en la UCI: sin plan. Dale siempre un tipo.

Requisitos olvidados; `not` en el estado inicial

Cada construcción, su indicador. Un hecho inicial negativo no añade nada: lo que no está en la lista es falso.

1 Tipos, constantes e igualdad

2 Condiciones más ricas

3 Efectos más ricos

4 Modelar problemas clásicos: patrones y decisiones

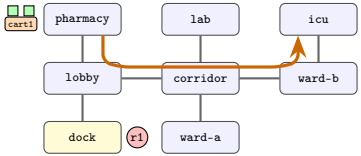
5 Resumen

Efectos que dependen de la situación: el carro

```
; Exactly ?i and ?j on the cart.
(:action push-cart-2
  :parameters (?r - robot ?c - cart ?from ?to - location
               ?i ?j - item)
  :precondition (and (pushing ?r ?c) (robot-at ?r ?from)
                     (connected ?from ?to)
                     (on-cart ?i ?c) (on-cart ?j ?c)
                     (not (= ?i ?j))
                     (forall (?k - item)
                       (imply (on-cart ?k ?c)
                              (or (= ?k ?i) (= ?k ?j)))))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (cart-at ?c ?from)) (cart-at ?c ?to)
               (not (item-at ?i ?from)) (item-at ?i ?to)
               (not (item-at ?j ?from)) (item-at ?j ?to)))
```

Una acción por número de artículos, de `push-cart-0` a `push-cart-3`:

- **Un límite que nadie pidió**: con cuatro artículos, el carro no se mueve
- **Grande**: $14\,n^k$ candidatas; cada par se cuenta dos veces
- **Repetitivo**: una regla nueva, cuatro sitios



- Un artículo en el carro conserva su `item-at`: el empujón debe actualizarlo
- Lo que queremos: *el robot y el carro cambian de sala, y también todo artículo que esté en el carro*

Efectos condicionales: `when`

```
(define (domain lights)
  (:requirements :strips :typing :negative-preconditions
                 :conditional-effects)

  (:types lamp)

  (:predicates
    (lit ?l - lamp))          ; the lamp is on

  ; One action for both cases: on becomes off, off becomes on.
  (:action toggle
    :parameters (?l - lamp)
    :effect (and (when (lit ?l) (not (lit ?l)))
                 (when (not (lit ?l)) (lit ?l))))

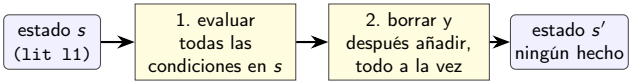
  ; The master switch: every lamp goes off, whatever it was.
  (:action all-off
    :parameters ()
    :effect (forall (?l - lamp) (not (lit ?l)))))
```

- `toggle` no tiene precondición: siempre es aplicable
- `lights-p01`: `(toggle l1)`, `(toggle l2)`: la misma acción enciende una lámpara y apaga la otra. Lee un plan con el estado presente

`(when <condition> <effect>)`: si la condición se cumple, el efecto ocurre también; requisito `:conditional-effects`

1. Las condiciones **no deciden** si la acción se aplica: eso solo lo hace la precondición
2. Toda condición se evalúa en el **estado anterior** a la acción
3. Los efectos que se disparan se aplican **a la vez**: borrados y después adiciones

Una condición en el efecto o en la precondición



	En la precondición	En un `when`
Decide	si la acción se aplica	qué hechos cambian
Si es falsa	la acción no se puede usar	se usa, sin ese efecto
Se evalúa en	el estado anterior	el estado anterior

- Pregunta al mundo: cuando es falsa, ¿puede el agente hacerlo igualmente?
- Lámpara encendida: se puede pulsar (efecto). Puerta cerrada: no se atraviesa (precondición)
- Un botón con tres lámparas: 2^3 acciones STRIPS, una con `when`

Efectos universales: `forall`

```
; The master switch: every lamp goes off, whatever it was.
(:action all-off
  :parameters ()
  :effect (forall (?l - lamp) (not (lit ?l)))))
```

- `(forall (<vars>) <effect>)`: una vez por cada objeto
- El reflejo de la precondición universal: aquella *pregunta*, este *hace*
- Ningún requisito nuevo: parte de `:conditional-effects`
- Se expande al instanciar; borrar un hecho ausente no cambia nada

```
$ kairos lights-domain.pddl
    lights-p02.pddl
; cost = 6
...
; better plan 2
; cost = 2
(all-off)
(toggle l2)
```

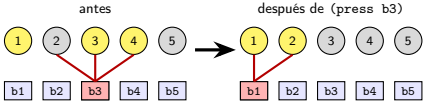
Una herramienta tosca: *todas* las lámparas. Casi siempre el efecto es para *algunos* objetos, y el `forall` lleva un `when` dentro.

Los dos juntos: Lights Out

```
(:action press
  :parameters (?b - button)
  :effect (forall (?l - lamp)
            (and (when (and (wired ?b ?l) (lit ?l))
                   (not (lit ?l)))
                 (when (and (wired ?b ?l) (not (lit ?l)))
                   (lit ?l))))))
```

Para todo `?x` **tal que**: el `forall` trae todos los objetos, el `when` se queda con los afectados. Las lámparas no conectadas no se tocan (la regla del marco).

```
$ kairos lightsout-domain.pddl
    lightsout-p01.pddl
...
; cost = 2
(press b3)
(press b1)
```



- `press`: un parámetro, cualquier número de lámparas
- p02: irresoluble tras 16 estados de 32; la respuesta es una demostración

El carro, en una frase

```
; The robot, the cart and every item on the cart change room.
(:action push-cart
  :parameters (?r - robot ?c - cart ?from ?to - location)
  :precondition (and (pushing ?r ?c) (robot-at ?r ?from)
                     (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from)) (robot-at ?r ?to)
               (not (cart-at ?c ?from)) (cart-at ?c ?to)
               (forall (?i - item)
                 (when (on-cart ?i ?c)
                   (and (not (item-at ?i ?from))
                        (item-at ?i ?to)))))))
```

- El `when` contiene un borrado *y* una adición
- `pick`:
  `(forall (?c - cart) (not (on-cart ?i ?c)))`
- `move`:
  `(not (exists (?c - cart) (pushing ?r ?c)))`

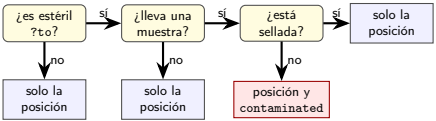
```
$ kairos courier-cart-domain.
    pddl courier-cart-p01.pddl
...
; better plan 3
; cost = 11
...
(push-cart r1 cart1 ward-b icu)
```

- El objetivo se cumple con un empujón cuyos parámetros no nombran ninguna caja
- p02: el robot descarga antes de que vuelva el carro

Efectos que hay que evitar: el laboratorio estéril

```
; The move is always possible. What it changes, besides the
; position, depends on the room entered and on what is carried.
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)
               (forall (?i - sample)
                 (when (and (sterile ?to) (carrying ?r ?i)
                            (not (sealed ?i)))
                   (contaminated ?i)))))
; Put the carried sample in a sealed bag.
(:action seal
  :parameters (?r - robot ?i - sample)
  :precondition (and (carrying ?r ?i) (not (sealed ?i)))
  :effect (sealed ?i)))
```

- La esclusa no detiene a nadie: nada en la precondición
- `forall`: `move` no tiene parámetro de artículo; el robot puede no llevar nada



- Tres condiciones, todas leídas antes del movimiento
- Una esclusa que *impide* pasar: una precondición, otro mundo; una pregunta para el paso M

Ensayar un efecto condicional

Paso E: una batería de pruebas para un efecto condicional

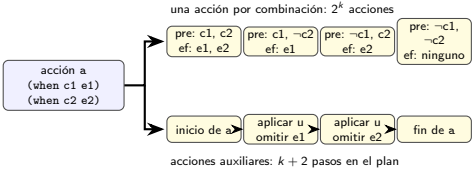
Trivial	medicamentos al laboratorio: la condición nunca se cumple (7 pasos)
Típica	muestra al laboratorio, objetivo `(not (contaminated sample1))`: primero sellar (8 pasos)
Sin plan	abierta *y* utilizable: irresoluble tras 71 estados
Trampa	solo la muestra en el laboratorio

```
$ kairos courier-sterile-domain.pddl
    courier-sterile-p04.pddl
; cost = 7
...
(move r1 ward-a corridor)
(move r1 corridor lab)
(drop r1 sample1 lab)
```

Un paso más corto: la muestra llega contaminada. `kairos validate` acepta este plan para p04 y lo rechaza para p02: «(contaminated sample1) is true».

Un efecto condicional puede ser un peligro: el planificador solo lo evita si un objetivo o una precondición posterior necesita lo contrario. Todo daño que importe debe estar en el objetivo.

Qué hacen los planificadores con los efectos condicionales



Ninguna tercera vía evita los dos costes (Nebel, 2000): los efectos condicionales son una ganancia real de poder expresivo.

- **Los efectos universales se expanden**; las condiciones estáticas desaparecen
- **Efectos condicionales conservados, o eliminados por compilación**: crece el modelo, o crecen los planes
- **Heurística**: cada efecto condicional como una pequeña acción (tradición de FF)
- Kairos los conserva: 3 lámparas, 4 acciones instanciadas; 6 en el modelo STRIPS
- Planes válidos solo de Kairos, ENHSP y LAMA; no de OPTIC, POPF, TFD, LPG-td ni Metric-FF

El método MODEL: efectos que dependen de la situación

Paso D: dos preguntas nuevas

1. **¿Depende de la situación lo que cambia?** Un «si» que no impide actuar al agente: `when`
2. **¿Muchos objetos a la vez?** «Cada», «lo que esté encima»: `forall`

Tabla de acciones: *siempre cambia* y *también cambia, si*. Paso E: la condición se cumple, no se cumple, y una prueba de la trampa.

p02	A. Efecto condicional	B. Acción por caso	C. Acción auxiliar
Fidelidad	exacta	4 artículos: no se mueve	exige (`not (unsettled)`) en los objetivos
Tamaño	109 acc. instanciadas	343, crece como n^k	189
Guía	25 pasos, con el carro	25 pasos	el mejor plan (28) evita el carro

D. No almacenar lo que se puede deducir: `load` borra `item-at`; un empujón STRIPS simple; 133 acciones instanciadas, 13 variables de estado (37 en A). Pero `p01` necesita 16 pasos: hay que dejar las cajas.

Veredicto: A cuando el resultado depende del estado; B para pocos casos fijos; C solo si no hay soporte; antes, pregúntate si hay que guardar el hecho (D).

Errores comunes con los efectos

Leer los efectos como un programa

```
:effect (and (when (lit ?l) (not (lit ?l)))
             (lit ?l))))
```

Borrado y añadido: gana la adición, la lámpara sigue encendida (irresoluble, 1 estado).

El `when` olvidado

Un empujón sin `(on-cart ?i ?c)`: 7 pasos, nadie carga nada y todos los artículos del hospital siguen al carro.

Un `when` en una precondición

Se lee como un predicado no declarado, «never true». Usa `imply`.

El requisito que falta; `when` dentro de `when`

Declara `:conditional-effects`. `(when a (when b e))` es `(when (and a b) e)`.

Un daño que el objetivo no vigila

La muestra contaminada, las cajas olvidadas: modelos correctos, objetivos incompletos.

1 Tipos, constantes e igualdad

2 Condiciones más ricas

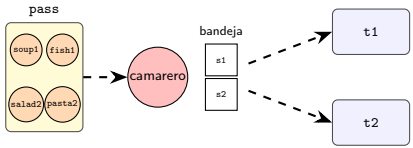
3 Efectos más ricos

4 Modelar problemas clásicos: patrones y decisiones

5 Resumen

Un problema a partir de una historia: el camarero

El cocinero deja los platos terminados en el pase. El camarero los lleva a las mesas en una bandeja con dos sitios. En cada mesa los platos llegan en orden. La noche ha ido bien cuando cada plato que se ha pedido está en su mesa.



Paso M: la ficha del modelo

Agentes	un camarero; el cocinero y los clientes no son agentes
Éxito	cada plato pedido en su mesa
Alcance	*dentro*: posición, bandeja, servido, orden de los platos. *Fuera*: cocinar, la ruta, el tiempo, comer, el pago
Supuestos	un plato servido sigue servido; un plato no vuelve al pase

- Sin ruta: una **abstracción**, acertada si la navegación la resuelve
- «No vuelve» parece inofensivo: creará callejones sin salida

Pasos O y D: tabla de estado y dominio

Propiedad	Clase	Patrón
`(waiter-at ?p)`	dinámico	a lo sumo uno
`(ready ?d)`	dinámico	
`(on-tray ?d ?s)`	dinámico	huecos
`(empty ?s)`	dinámico	huecos
`(served ?d)`	dinámico	
`(ordered ?d ?t)`	estático	tabla estática
`(before ?a ?b)`	estático	tabla estática

- Un solo camarero: sin objeto, un predicado de un argumento
- pass: una constante
- El orden de los platos es un *dato*: un postre no exige cambiar el dominio

```
; The dining room is open: any place is one move away.
(:action move
  :parameters (?from ?to - place)
  :precondition (and (waiter-at ?from) (not (= ?from ?to)))
  :effect (and (not (waiter-at ?from)) (waiter-at ?to)))

(:action load
  :parameters (?d - dish ?s - slot)
  :precondition (and (waiter-at pass) (ready ?d) (empty ?s))
  :effect (and (not (ready ?d)) (not (empty ?s))
               (on-tray ?d ?s)))

; A dish is served at its table, after the dishes before it.
(:action serve
  :parameters (?d - dish ?s - slot ?t - place)
  :precondition (and (waiter-at ?t) (on-tray ?d ?s)
                     (ordered ?d ?t)
                     (forall (?e - dish)
                       (imply (before ?e ?d) (served ?e))))
  :effect (and (not (on-tray ?d ?s)) (empty ?s)
               (served ?d))))
```

Una constante y una desigualdad (Cap. 4); `forall` con `imply` (Cap. 5)

Paso E: ensayar el modelo

Trivial un plato: cargar, mover, servir

Típica cuatro platos: 16, 13 y luego 11 pasos, óptimo

Sin plan sopa no `ready`, objetivo con el pescado: irresoluble

Trampa solo los segundos: 11 pasos, los cuatro servidos

Dos mitades de una prueba de la trampa

El planificador no toma el atajo, y el atajo escrito a mano se rechaza, por la razón que esperamos.

```
$ kairos waiter-domain.pddl
    waiter-p02.pddl
...
; better plan 3
; cost = 11
(load fish1 s1)
(load soup1 s2)
(move pass t1)
(serve soup1 s2 t1)
...
```

El plan tramposo, validado: «step 4 (serve fish1 s1 t1) . . . precondition false: for ?e = soup1: (imply (before ?e ?d) (served ?e))»

Once patrones: el catálogo

Un **patrón**: una respuesta con nombre a una pregunta del paso O o D, con un **rival** y un **coste**.

	Patrón	Úsalo cuando	Lo que cuesta
Estado	Tabla estática	una regla varía entre problemas	problemas más largos
	Hechos espejo	una relación vale en ambos sentidos	una tabla el doble de larga
	Invariante de a lo sumo uno	una propiedad tiene un solo valor	nada; un borrado olvidado lo rompe
	Cadena de sucesores	cantidad pequeña; solo importa cuánto	dos parámetros por acción
	Huecos	cada unidad de capacidad es un sitio	simetría entre los huecos
Control	Testigo de fase	las etapas no deben mezclarse	callejones sin salida; acción por etapa
	Testigo de turno	alternan agentes, o agente y mundo	planes el doble de largos
	Cerrojo de recurso	un solo usuario a la vez	hay que pedir que se libere
Acciones	Par inverso	el mundo puede deshacerlo	un espacio de estados mayor
	Acción sin vuelta atrás	el mundo no puede deshacerlo	callejones sin salida
	Acción fusionada	ninguna decisión entre los pasos	lo de dentro no se puede optimizar

Patrones para el estado: tabla estática, hechos espejo

Tabla estática: una relación que ninguna acción cambia

```
; The farmer crosses alone: nothing may be eaten behind him.
(:action cross-alone
  :parameters (?from ?to - bank)
  :precondition (and (boat-at ?from) (not (= ?from ?to))
    (not (exists (?a ?b - thing)
           (and (eats ?a ?b) (at ?a ?from) (at ?b ?from)))))
  :effect (and (not (boat-at ?from)) (boat-at ?to)))
```

- Lobo, cabra, col: quién se come a quién es `(eats ?a ?b)`, un dato
- 7 travesías; un hecho más, `(eats wolf cabbage)`: sin plan
- Si otra instancia obliga a editar una acción, el conocimiento está mal colocado
- Gratis al buscar; un hecho que falta es otro mundo

Hechos espejo: una relación simétrica, en los dos sentidos

```
(connected lobby dock) (connected dock lobby)
```

Rival: leerla en los dos sentidos

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from)
                     (or (connected ?from ?to)
                         (connected ?to ?from)))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

- 7 hechos en vez de 14; las mismas 30 acciones instanciadas
- El `or` pierde las puertas de un solo sentido. Los hechos espejo son STRIPS puro: la opción por defecto

Patrones para el estado: invariante de a lo sumo uno

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (and (not (robot-at ?r ?from))
               (robot-at ?r ?to)))
```

- Un **invariante** se cumple en todo estado alcanzable; PDDL no puede declararlo
- **A lo sumo uno**: toda acción que añade un valor borra el anterior; exactamente uno en el estado inicial

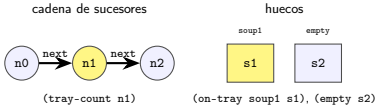
- Para «*la* posición», «*su* estado», «*el* que lo tiene»
- Un **grupo mutex**: a lo sumo uno de sus hechos es cierto; el planificador los encuentra
- Ocho hechos `robot-at` se convierten en una variable: 18 hechos, 3 variables
- Gratis, y frágil: un borrado olvidado lo rompe

```
$ kairos courier-domain.pddl courier-p03.pddl -v 2>&1 | grep sas
[   0.001s] [sas] SAS+ in 0.1 ms: 3 variables (2 from 3 mutex groups, 1 binary
    ), 3 invariants from 13 candidates, 0 axiom layers
```

Contar sin números: cadena de sucesores o huecos

```
; One more dish: the count goes one step up the chain. With
; the tray full there is no next count, and no load.
(:action load
  :parameters (?d - dish ?n ?m - count)
  :precondition (and (waiter-at pass) (ready ?d)
                     (tray-count ?n) (next ?n ?m))
  :effect (and (not (ready ?d)) (on-tray ?d)
               (not (tray-count ?n)) (tray-count ?m)))
```

- **Cadena de sucesores**: números como objetos, `(next ?n ?m)` estático, un hecho guarda la cuenta
- La capacidad es la longitud de la cadena en el problema: el límite es la *ausencia* de datos
- Solo $+1$ y -1 gratis; más allá de unas decenas, números (Cap. 8)



- **Huecos**: un objeto por unidad de capacidad, y un hecho que dice que está libre
- Para sitios distintos: un estante, un aparcamiento, dos manos
- Coste: simetría. Camarero p02: 220 estados con huecos, 123 con la cadena

Cadena o huecos

Cuenta cuando solo importa cuánto; huecos cuando los sitios difieren.

Patrones para el control: testigo de fase

```
(:action cut
  :parameters (?x - part)
  :precondition (and (now cutting) (not (cut ?x)))
  :effect (cut ?x))
; The token moves forward and never comes back.
(:action advance
  :parameters (?a ?b - phase)
  :precondition (and (now ?a) (follows ?b ?a))
  :effect (and (not (now ?a)) (now ?b))))
```

```
$ kairos workshop-domain.pddl
    workshop-p01.pddl
; cost = 11
...
(advance cutting joining)
(join seat)
...
```

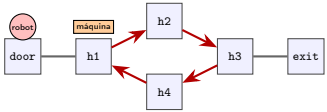
- Un **testigo**: un hecho que dice quién o qué tiene permiso ahora
- **Testigo de fase**: cada acción exige su fase; advance hace avanzar el testigo, nunca retroceder
- Para etapas que no deben mezclarse: todo el corte antes de cualquier encolado
- Las precondiciones ordenan el trabajo *de una pieza*; el testigo, *el del taller*
- Coste: una acción por cambio de fase; callejones sin salida. Ganancia: una búsqueda menor

Patrones para el control: testigo de turno

```
; The robot's turn: one step, never into the machine.
(:action move
  :parameters (?from ?to - cell)
  :precondition (and (turn robot) (robot-at ?from)
                     (connected ?from ?to)
                     (not (machine-at ?to)))
  :effect (and (not (robot-at ?from)) (robot-at ?to)
               (not (turn robot)) (turn machine)))

; The robot may also let its turn pass.
(:action wait
  :parameters ()
  :precondition (turn robot)
  :effect (and (not (turn robot)) (turn machine)))

; The machine's turn: the only thing that can happen. If the
; robot stands in its way, nothing can, and the plan dies.
(:action machine-step
  :parameters (?from ?to - cell)
  :precondition (and (turn machine) (machine-at ?from)
                     (route ?from ?to) (not (robot-at ?to)))
  :effect (and (not (machine-at ?from)) (machine-at ?to)
               (not (turn machine)) (turn robot))))
```



- **Testigo de turno**: las acciones necesitan su turno y lo pasan
- 9 pasos: `wait` y después seguir a la máquina una celda por detrás
- Un paso del mundo que no puede ocurrir significa «prohibido»
- Coste: planes el doble de largos

Cerrojo de recurso y acción fusionada

Cerrojo de recurso: un hecho dice que el recurso está disponible

```
(:action take-keycard
  :parameters (?r - robot ?l - location)
  :precondition (and (robot-at ?r ?l) (keycard-at ?l))
  :effect (and (not (keycard-at ?l)) (has-keycard ?r)))
(:action leave-keycard
  :parameters (?r - robot ?l - location)
  :precondition (and (robot-at ?r ?l) (has-keycard ?r))
  :effect (and (not (has-keycard ?r)) (keycard-at ?l))))
```

- Prueba de la trampa, la tarjeta en dos manos: `(has-keycard r1)` y `(has-keycard r2)`: irresoluble
- Nunca se libera si no hace falta: ponlo en el objetivo

Acción fusionada: los pasos sin ninguna decisión entre ellos pasan a ser uno

```
(:action deliver
  :parameters (?d - dish ?t - place)
  :precondition (and (ready ?d) (ordered ?d ?t)
                     (forall (?e - dish)
                       (imply (before ?e ?d) (served ?e))))
  :effect (and (not (ready ?d)) (served ?d))))
```

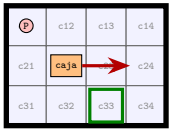
- El controlador agrupa la bandeja: 4 acciones instanciadas en vez de 22, un plan de 4 pasos
- **Granularidad**: modela al nivel de las decisiones que debe tomar el planificador

Patrones para las acciones: par inverso, acción sin vuelta atrás

- **Par inverso**: cada una restaura lo que cambió la otra (coger y dejar)
- **Acción sin vuelta atrás**: permanente (`seal`, `serve`, `advance`)
- Sigue al mundo: un inverso que falta crea **callejones sin salida** que el mundo no tiene; uno inventado deja escapar al planificador de errores definitivos

Camarero p05, bandeja llena de segundos: sin plan. El inverso de `load`:

```
(:action unload
  :parameters (?d - dish ?s - slot)
  :precondition (and (waiter-at pass) (on-tray ?d ?s))
  :effect (and (ready ?d) (empty ?s)
               (not (on-tray ?d ?s))))
```



- Sokoban: `walk` es su propio inverso, `push` no tiene. Dos empujones al este: la caja queda contra la pared para siempre
- Inversos: sin falsos callejones sin salida, un espacio mayor. Las heurísticas de relajación de borrados no ven la mayoría de los callejones

Antipatrones: ficheros válidos, modelos erróneos

Antipatrón: un error tan común que merece un nombre. `kairos check` no protesta: solo lo delata el síntoma.

Antipatrón	Síntoma
Borrado olvidado	un plan demasiado corto
Objetivo en el estado inicial	un plan vacío, coste 0
Gemelos desacompasados	pasa una prueba pequeña, una mayor no tiene plan
Parámetro ocioso	demasiadas acciones instanciadas

Objetivo en el estado inicial: `waiter-done-p01` da `; cost = 0`. Peor: un objetivo *en parte* cierto al principio.

Borrado olvidado

```
(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (robot-at ?r ?from) (connected ?from ?to))
  :effect (robot-at ?r ?to))
```

```
$ kairos courier-nodelete-domain.pddl
    courier-nodelete-p03.pddl
; cost = 6
...
(pick r1 meds1 pharmacy)
(move r1 lobby corridor)
...
```

6 pasos, no 10: sale *desde el vestíbulo*. La línea `[sas]`: 10 variables, no 3.

Antipatrones: gemelos desacompasados, parámetro ocioso

Gemelos desacompasados

```
:effect (and (not (on-tray ?d ?s))
             (served ?d))))
```

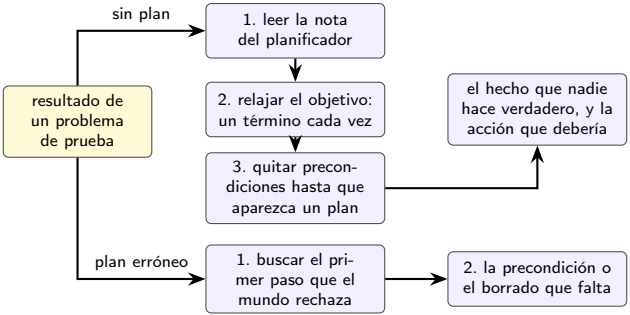
- Gemelos: `empty` y `on-tray`, `free` y `carrying`; legítimos
- `serve` olvida `(empty ?s)`: tras dos platos la bandeja no lleva nada y no tiene sitio libre
- Síntoma: pasa la prueba trivial, la típica no tiene plan
- Prevención: escribe el invariante de cada par; revisa cada acción

Parámetro ocioso

```
(:action load
  :parameters (?d - dish ?s - slot ?t - place)
  :precondition (and (waiter-at pass) (ready ?d) (empty ?s))
```

- `?t`: ninguna precondición lo restringe, ningún efecto lo usa
- `(load fish1 s2 pass)`, `(load fish1 s2 t1)`: lo mismo
- 38 acciones instanciadas en vez de 22
- Revisa cada parámetro: ¿restringido, y usado en un efecto? (el `?t` de `serve` no es ocioso)

Depurar un modelo: el procedimiento



Sin plan: el objetivo más pequeño que falla, luego la precondición cuya eliminación devuelve un plan, y luego: ¿qué acción debería hacer cierto ese hecho?

Plan erróneo: una **precondición que falta**, o una **acción anterior dejó un hecho erróneo**. Después, una prueba de regresión: el atajo como fichero de plan que `validate` rechaza.

Depurar los gemelos desacompasados

2. Relajar el objetivo: p06, solo (served soup1): 3 pasos. p07, los dos primeros: 6 pasos. Después:

```
$ kairos waiter-bug-domain.pddl
    waiter-bug-p08.pddl      # two
    starters and the fish
note: no plan: every state reachable
    from the initial one was searched
    (63 states) and none satisfies the
     goal. [orchestrator.why]
kairos: unsolvable (exhausted)
```

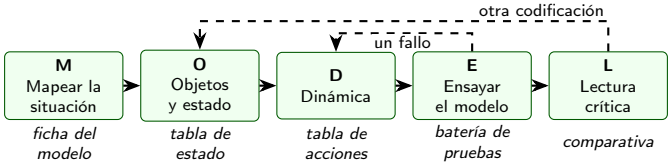
Nunca *el tercer* plato: mira la bandeja.

3. Quitar precondiciones: load sin (empty ?s)

```
$ kairos
    waiter-bug-relaxed-domain.
    pddl waiter-bug-relaxed-p02
    .pddl
; cost = 10
(load soup1 s1)
(load salad2 s1)
...
```

- *¿Quién añade (empty ?s)?* Nadie: serve perdió una línea

La lista de comprobación clásica de MODEL



M ¿Quién actúa? ¿Qué cuenta como resuelto? ¿Qué se deja fuera y quién se ocupa de ello? ¿Qué supuestos lo hacen clásico?

O Clases, jerarquía, constantes; estático o dinámico; ausencia, alguno, todos; *¿un solo valor a la vez? ¿una cantidad? ¿se deduce de otra?*

D Cuándo se aplica, qué cambia; el mismo objeto dos veces; depende de la situación; *¿se puede deshacer? ¿una decisión o varias? ¿hace falta cada parámetro?*

E Estima el tamaño; trivial, típica, sin plan, trampa; valida; si algo falla, el procedimiento

L Todo patrón tiene un rival: compara según Fidelidad, Tamaño, Guía, Claridad y Portabilidad

Paso L para el camarero

p02	Huecos	Cadena de sucesores	Acción fusionada
Fidelidad	exacta; sabe dónde está cada plato	exacta para la historia	no representa capacidad ni viajes
Tamaño	22 acciones instanciadas	22 acciones instanciadas	4 acciones instanciadas
Guía	220 estados; cada plan, un gemelo	123 estados; sin simetría	9 estados
Claridad	`(load soup1 s2)`	`(load soup1 n0 n1)`	dominio y planes más cortos
Portabilidad	precondición universal	lo mismo	lo mismo

Veredicto: una pregunta del paso M, *¿quién decide los viajes?*

- El planificador: **cadena de sucesores**; sitios intercambiables, mitad de estados
- Sitios distintos: **huecos**
- El controlador del robot agrupa platos: **acción fusionada**

El método es un bucle

El modelo que escribimos primero no era el mejor para la historia tal como se contó: el resultado normal del paso L.

Errores comunes: un patrón al que le falta una pieza

La mitad de los hechos espejo

Solo (`connected lobby dock`): la nota retrocede desde el objetivo hasta un `move` que «is never applicable». Un sentido que falta: un robot que entra en una sala y no puede salir.

Dos testigos

(`now cutting`), (`now joining`), (`now painting`): 9 pasos, ningún `advance`; las fases no ordenan nada. Exactamente un testigo en `:init`.

Una cadena con un eslabón roto

Solo (`next n0 n1`): la bandeja lleva un plato, el plan pasa de 11 a 15 pasos. Prueba un problema que necesite toda la capacidad.

Un cerrojo que nadie devuelve

`r1` termina con la tarjeta encima. Si el turno siguiente la espera en su gancho: (`keycard-at lobby`) en el objetivo.

1 Tipos, constantes e igualdad

2 Condiciones más ricas

3 Efectos más ricos

4 Modelar problemas clásicos: patrones y decisiones

5 Resumen

Qué llevarse

- Los **tipos** convierten errores silenciosos en mensajes; una jerarquía enuncia una regla una vez; **constantes** para los objetos que nombran las reglas; **igualdad**: el planificador nunca supone dos parámetros distintos
- **Condiciones**: `not`, `or`, `imply`, `exists`, `forall`; desconfía de los objetivos que se cumplen porque falta algo
- **Efectos**: `when` y `forall`; todo daño que importe va en el objetivo
- **Patrones** y **antipatrones**: nombres para las decisiones de todo modelo clásico; depurar es un procedimiento

`:adl` abrevia	Capítulo
`:strips`	2
`:typing`, `:equality`	4
`:negative-preconditions`	5
`:disjunctive-preconditions`	5
`:quantified-preconditions`	5
`:conditional-effects`	6

Enumera lo que el fichero usa de verdad, no `:adl`.

Aquí termina la planificación clásica

ADL es un lenguaje mejor para los mismos problemas. Los números, el tiempo y las preferencias cambian el propio problema.

Ejercicios para practicar

- **4.3.2**: el zoo; dibuja la jerarquía y valida un plan en el que un cuidador da de comer al león
- **4.6.1**: `swap` con y sin desigualdad; cuenta las acciones instanciadas
- **5.1.3**: `lock` una puerta; ¿qué precondiciones negativas?
- **5.5.1**: `close-floor` sobre el dominio de las salas, todos los robots en la base de carga
- **6.4.3**: dos robots y un carro; encuentra el fallo y repáralo
- **6.5.2**: la esclusa que impide pasar; compara las baterías de pruebas
- **7.5.1**: tres antipatrones en `gripper-buggy-domain.pddl`
- **7.7.1**: el método completo con un transbordador para dos coches, cinco artefactos