Plánování jako logické fungování

Agent může na základě struktury problému vytvořit složité plány své činnosti. **Jednoduchý plánující agent** je velmi podobný agentovi řešícímu problémy (zmíněnému dříve) v tom, že konstruuje plány k dosažení svých cílů, a po vytvoření plánů je provede. Další, tzv. **částečně uspořádané plánování**, je komplikovanější, prohledává prostor plánů, aby našlo plán zaručující úspěch. Zde je zároveň navíc flexibilita, umožňující zvládnout zcela složité domény.

Jednoduchý plánující agent

Pokud je stav světa přístupný, může agent použít vjemy poskytované okolím (prostředím) k vytvoření kompletního a korektního modelu okamžitého stavu světa. Potom může vzhledem k cíli vyvolat vhodný plánovací algoritmus (`Ideal-Planner`) k vytvoření plánu činností. Plán je vykonáván po krocích, jedna akce v jednom časovém kroku. Agent používá znalostní bázi.

Algoritmus (využívající **bázi znalostí**) pro jednoduchého plánujícího agenta:

```
function Simple-Planning-Agent(percept) returns action

     static:  KB // znalostní báze, obsahuje popis akcí
              p  // plán (na počátku žádný plán)
              t  // čítač časových kroků, na počátku 0

     local variables:  G       // cíl
                       current // popis okamžitého stavu

     Tell(KB,Make-Percept-Sentence(percept,t))
     current ← State-Description(KB,t)
     if p = NoPlan then
          G ← Ask(KB,Make-Goal-Query(t))
          p ← Ideal-Planner(current,G,KB)
     if p = NoPlan or p is empty then action ← NoOp
     else
        action ← First(p)
        p ← Rest(p)
     Tell(KB,Make-Action-Sentence(action,t))
     t ← t+1

     return action
```

Funkce **`Tell`** zde vkládá do znalostní báze `KB` informaci o tom, co se objevilo jako vjem (k čemu po sdělení údajů znalostní bázi pak dojde, je záležitost **inference**). **`Make-Percept-Sentence`** dostane pro daný časový okamžik *t* vjem (*percept*) a vrací representaci vjemu (faktů o světě); individuální representace faktů se nazývají *sentence* nebo *tvrzení*—tato (logická) tvrzení jsou vyjádřena v jazyku, kterému znalostní báze rozumí, tj. *jazyk representace znalosti*.

Funkce **`Ask`** se ptá znalostní báze, jaká akce (po sdělení přes `Tell`, jaká je okamžitá situace) by se pro daný stav měla udělat. Jde tedy o jakýsi dialog typu *poskytnutí informace o skutečnosti* (`Tell`)—*dotaz co za těchto podmínek dělat* (`Ask`).

Funkce **`State-Description`** dostane vjem jako vstup a vrací popis počátečního stavu ve formátu vyžadovaném plánovačem (generátorem plánů).

Funkce **`Make-Goal-Query`** je použita v dotazu vůči znalostní bázi na to, co by mělo být dalším cílem agenta.

`Ideal-Planner` může být jakýkoliv vhodný plánovač (viz dále).

Od řešení problémů k plánování

Plánování a *řešení problémů* považuje moderní umělá inteligence za odlišné záležitosti, protože representace cílů, stavů, akcí a sekvencí akcí jsou odlišné.

Řešení problémů založené na vyhledávání:

- **Representace akcí:** akce jsou popsány programy, které generují popisy následných stavů.
- **Representace stavů:** je dán kompletní popis počátečního stavu; akce jsou representovány programem, který generuje kompletní popisy stavů. Proto jsou všechny representace stavů kompletní. Ve většině problémů je stav jednoduchá datová struktura—permutace čtverečků v hlavolamu s 8 čtverečky, pozice agenta v problému hledání trasy, nebo pozice šesti osob v problému kanibalů a misionářů s loďkou. Representace stavů se používají pouze pro generovaní následníků, pro heuristickou vyhodnocovací funkci a pro testování cíle.

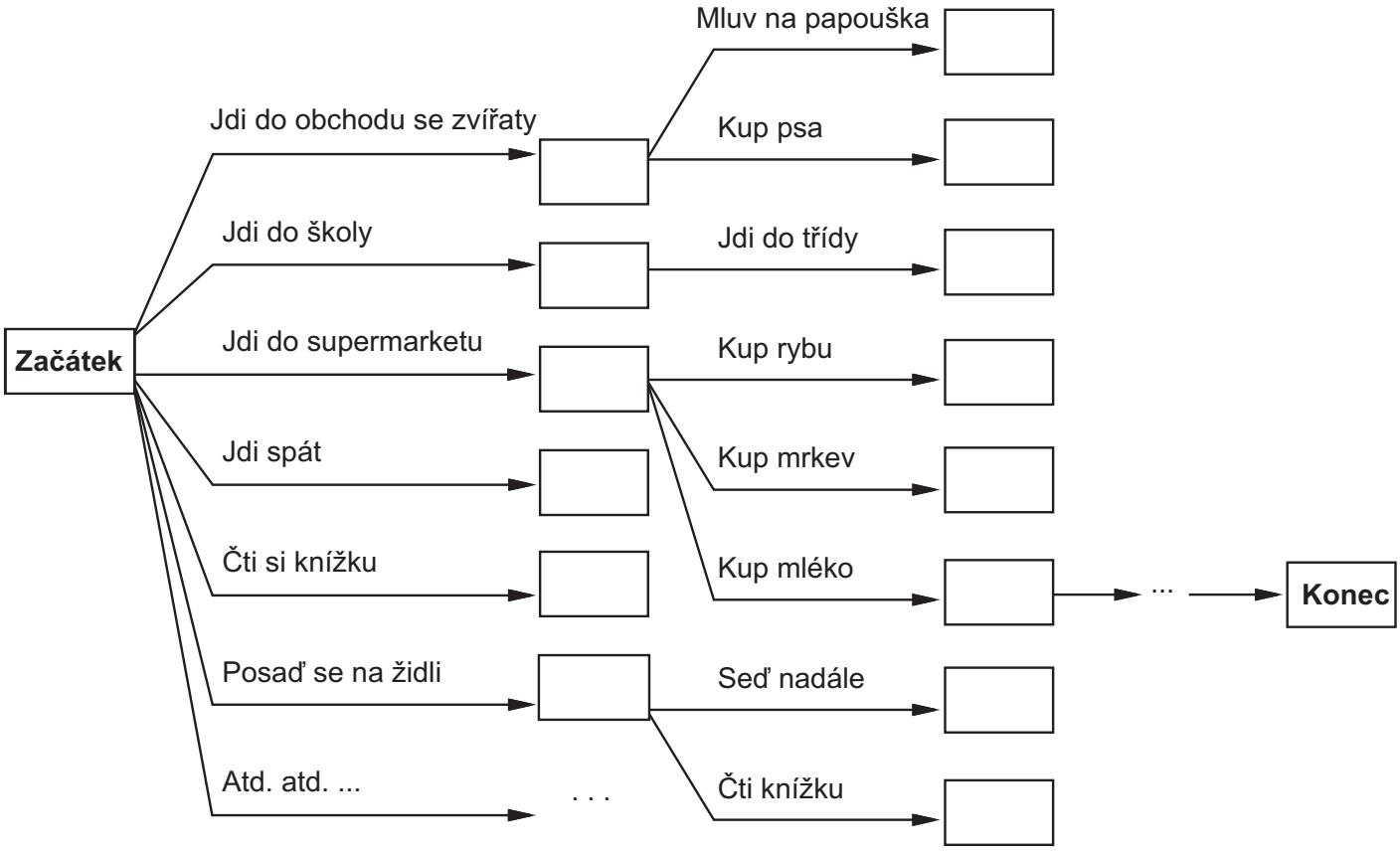
- **Representace cílů:** jediná informace, kterou má řešící agent o svém cíli, je ve formě testu na cíl a heuristické funkci. Oba prostředky mohou být aplikovány na stavy k rozhodnutí, zda jsou stavy vhodné či potřebné, ale zároveň jsou oba prostředky používány ve formě "černé skříňky" (agent řešící problém nemá možnost se podívat dovnitř nástroje při výběru akcí, které mohou být užitečné z hlediska dosažení cíle).
- **Representace plánů:** při řešení problémů jsou výsledkem sekvence akcí typu "jeď z Aradu do Sibiu, dále do Fagarasu a pak do Buchuresti". Během konstrukce řešení uvažují vyhledávací algoritmy pouze nepřerušené sekvence akcí vedoucích z počátečního stavu (při obousměrném hledání akce končící v konečném stavu).

Jako příklad lze uvést návrh řešení, ovlivňujících agentovu schopnost vyřešit jednoduchý problém "získej litr mléka, trs banánů a bezdrátovou vrtačku s měnitelnou rychlostí otáček".

Pro řešení je nutno specifikovat *počáteční stav*: agent je doma bez požadovaných předmětů, a *soubor operátorů*: vše, co agent může dělat. Lze případně použít i nějakou *heuristiku*, např. počet předmětů, které dosud nebyly získány.

Na následujícím obrázku je velmi malá a symbolicky omezená část prvních dvou úrovní prohledávaného prostoru pro řešení zadaného problému a náznak cesty vedoucí k cíli. Skutečný faktor větvení závisí na tom, jak jsou akce specifikovány (*b* mohou být tisíce nebo miliony), a délka řešení může být v desítkách kroků—existuje příliš mnoho akcí a stavů.

Heuristická vyhodnocovací funkce může pouze vybrat mezi mnoha stavy k rozhodnutí, který stav je blíže k cíli; neumí z úvah vyloučit akce. I když se agent dostane do supermarketu, nezbývá mu, než hádat. Agent při pokusu o uhodnutí uvažuje akce (koupit pomeranče, rybu, brambory, mléko) a jejich ohodnocení evaluační funkcí (chybné, chybné, chybné, správné). Agent pak ví, že koupit mléko je správná činnost, *ale nemá možnost zjistit, co udělat pak a musí proces pokusů zahájit znovu*. K potížím agenta tedy přispívá také fakt, že musí vždy uvažovat sekvenci akcí z počátečního stavu—musí tedy se napřed rozhodnout, co udělat v počátečním stavu, kde relevantní výběr mezi činnostmi je jít na nějaké z mnoha možných míst. Pokud agent neví, *jak* získat různé položky (zakoupení, zapůjčení, pronajatí, vypěstování, vyrobení, ukradení...), pak se nemůže fakticky *rozhodnout*, kam jít.



Z uvedených problémů plyne, že agent potřebuje mnohem flexibilnější způsob ke strukturalizaci svého uvažování tak, aby mohl zpracovat kteroukoliv část problému, která je—za předpokladu okamžité informace—nejpravděpodobnější k vyřešení.

Základní a klíčová idea pro *plánování* je “otevření černé skříňky”, tj. representací stavů, cílů a akcí. Plánovací algoritmy používají popisy ve formálních jazycích, např. pomocí logiky prvního řádu (viz příslušnou kapitolu matematické logiky: výrazy, termy, konstanty, proměnné, predikáty, funkce, spojky $\Rightarrow$, $\wedge$, $\vee$, $\Leftrightarrow$, kvantifikátory $\forall$, $\exists$) nebo její podmnožiny.

Stavy a cíle jsou representovány soubory výrazů, akce jsou representovány logickým popisem předpokladů a důsledků—to umožňuje agentovi přímé propojení mezi stavy a akcemi:

> Ví-li agent, že cílem je konjunkce zahrnující konjunkt *Mít(Mléko)* a že *Koupit(x)* docílí *Mít(x)*, pak také ví, že stojí za úvahu plán, v němž je *Koupit(Mléko)*. Nemusí uvažovat irelevantní akce jako jsou např. *Koupit(Šlehačku)* nebo *JítSpát*.

Další ideou plánování je skutečnost, že plánovač má volnost v přidávání akcí k plánu, kdykoliv jsou ty akce zapotřebí—na rozdíl od postupu, kde se používá inkrementální sekvence začínající v počátečním stavu. Např. agent se může rozhodnout, že použije *Koupit(Mléko)*, i když ještě se nerozhodl kde, jak se tam dostat, nebo co dělat pak.

Není nutná spojitost mezi pořadím plánování a pořadím uskutečňování. Prvně se stanoví rozhodnutí, která jsou "zřejmá" nebo "důležitá", a pak může plánovač redukovat faktor větvení pro budoucí výběry a dále redukovat nutnost sledování všech libovolných rozhodnutí.

Representace stavů jako souborů logických výrazů je podstatná, protože umožňuje tuto volnost výběru: přidání akce *Koupit(Mléko)* má representaci stavu, kde se akce vykoná, jako *Být_v(Supermarketu)*, což ve skutečnosti representuje celou třídu stavů: s banány a bez banánů, s vrtačkou a bez vrtačky, apod. Prohledávací algoritmy, vyžadující kompletní popis stavů, tuto možnost nemají.

Poslední podstatnou ideou pro plánování je skutečnost, že *většina částí* ***reálného*** *světa je nezávislá na většině ostatních částí*. Proto je uskutečnitelné vzít konjunktivní cíl typu "získej litr mléka $\wedge$ trs banánů $\wedge$ vrtačku..." a řešit ho pomocí strategie "rozděl a panuj" (viz níže potíže s hlavolamy).

První dva konjunkty vyřeší sub-plán obsahující cestu do supermarketu, další sub-plán (jít do železářství nebo půjčit si u souseda) vyřeší splnění třetího konjunktu. Sub-plán zahrnující supermarket lze dále rozdělit na sub-plán koupě mléka a sub-plán koupě banánů. Sjednocením všech sub-plánů dohromady je řešen celý problém. Funguje to právě z důvodu velmi malé interakce mezi dvěma sub-plány: cesta do supermarketu nepřekáží půjčce od souseda, koupě mléka nepřekáží koupi banánů (pokud samozřejmě agent nevyčerpá nějaké zdroje k provádění akcí, např. po koupi mléka namá na banány).

Strategie *rozděl a panuj* je účinná proto, že je téměř vždy snadnější řešit problém rozdělením na menší podproblémy. Může selhat v případech, kdy cena kombinací řešení všech podproblémů je příliš vysoká. Tuto vlastnost má mnoho hlavolamů, např. 8-hlavolam má cílový stav, který je konjunktivním cílem: **přesunout čtvereček 1 na pozici *A* $\wedge$ čtvereček 2 na pozici *B* $\wedge$** ... až do čtverečku 8. Z hlediska plánování lze použít rozdělení na nezávislé sub-problémy, ale *potíž s hlavolamy* je právě v obtížnosti dát sub-plány dohromady k dosažení cíle (1 lze přesunout na *A*, ale 2 na *B* může vyžadovat odsunutí 1 z *A*).

Plánování v situačním kalkulu

Problém plánování je v tomto případě představován logickými výrazy (používá se logika prvního řádu), které popisují tři hlavní části problému:

- **Počáteční stav:** libovolný logický výraz o situaci S_0. Pro problém nákupu to může např. být:

$Být\_v(Doma,S_0) \wedge \neg Mít(Mléko,S_0) \wedge \neg Mít(Banány,S_0) \wedge \neg Mít(Vrtačka,S_0)$

- **Cílový stav:** logický dotaz na vhodné situace *s*, např.:

$\exists\ s\ Být\_v(Doma,s) \wedge Mít(Mléko,s) \wedge Mít(Banány,s) \wedge Mít(Vrtačka,s)$

- **Operátory:** soubor popisů akcí *a* pomocí logiky prvního řádu, např.:

$$\forall\ a,s\ Mít(Mléko,Result(a,s)) \quad \Leftrightarrow \quad [(a = Koupit(Mléko) \wedge Být\_v(Supermarket,s)) \vee (Mít(Mléko,s) \wedge a \neq Vylít(Mléko))]$$

Pozn.: Situační kalkul je založen na myšlence, že akce mění stavy: *Result(a,s)* pojmenovává situaci vzniklou po provedení akce *a* v situaci *s*. Pro účely plánování může být užitečné manipulovat se sekvencemi akcí stejně jako s jednotlivými akcemi. *Result'(l,s)* znamená situaci vzniklou provedením sekvence akcí *l* počínaje situací *s*. *Result'* je stanoven tím, že prázdná sekvence akcí nemá vliv na situaci, a výsledkem neprázdné sekvence akcí je totéž jako aplikace první akce a pak zbytku akcí z počáteční situace.

Problém s dobrými teoretickými řešeními a formálními zápisy je často v tom, že nezaručují dobrá praktická řešení (k nimž umělá inteligence ovšem směřuje—to byl i důvod vzniku oboru). Důvodem je např. exponenciální časová závislost na délce řešení (v nejhorším případě), nebo existující problémy s logickou inferencí (např. *modus ponens*) pro řadu reálných problémů, např. z důvodu odlišnosti způsobu uvažování lidí od formálně logického přístupu (problém typický pro mnoho aplikací expertních systémů). Další potíž pro umělou inteligenci způsobuje problém tzv. *semi-rozhodnutelnosti*, vztahující se k výsledkům inference v logice prvního řádu: lze ukázat, že výrazy vyplývají z premis, pokud z nich vyplynuly—ale nemůžeme vždy ukázat, že vyplynou, pokud z nich nevyplynuly (detaily viz v příslušných kapitolách matematiky).

Základní representace pro plánování

Aby plánování agentů bylo použitelné v praktických situacích, umělá inteligence používá dva speciální přístupy:

(1) Restrikce jazyka pro definici problémů—výsledkem je méně možných řešení, mezi nimiž je nutno hledat.
(2) Použití specialisovaného algoritmu **plánovač** místo obecně zaměřeného algoritmu dokazování teorémů pro nalezení řešení problému.

Oba přístupy jsou spojeny: nový definiční jazyk vyžaduje nový plánovací algoritmus pro zpracování popisu problému daného jazykem.

V současnosti se pro velkou většinu plánovačů využívá jazyk zvaný `Strips` (STanford Research Institute Problem Solver).

Stavy ve Strips jsou representovány konjunkcemi funkčně nezávislými základními literály, tj. predikáty aplikovanými na konstatní symboly, s případnou negací. Např. počáteční stav problému s mlékem apod. může být popsán jako

Být_v(Doma) ∧ ¬*Mít(Mléko)* ∧ ¬*Mít(Banány)* ∧ ¬*Mít(Vrtačka)* ∧ ...

Popis stavů nemusí být úplný—neúplný popis (získaný např. agentem v ne zcela přístupném prostředí) odpovídá množině možných kompletních stavů, pro něž by agent chtěl získat úspěšný plán. Mnoho plánovacích systémů používá konvenci, která v případě, že popis stavu nezmiňuje daný positivní literál, považuje literál za negativní (v logickém programování je to “negace jako neúspěch”).

Cíle jsou rovněž popisovány jako konjunkce literálů:

Být_v(Doma) ∧ *Mít(Mléko)* ∧ *Mít(Banány)* ∧ *Mít(Vrtačka).*

Cíle mohou také obsahovat proměnné. Např. cíl *být v obchodě, který prodává mléko* může být representován jako

Být_v(x) ∧ *Prodávat(x,Mléko).*

Proměnné mohou být existenčně kvantifikované. Zde je ale rozdíl mezi cílem pro plánovač a dotazem pro dokazovač teorémů. *Plánovač vyžaduje sekvenci akcí, která vede ke splněnému cíli, pokud je provedena*. Dokazovač vyžaduje údaj o tom, zda dotazovací sekvence *je pravdivá* za předpokladu pravdivosti výroků ve znalostní bázi.

Representace akcí

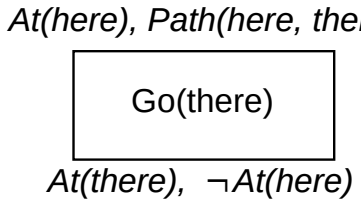
Operátory Strips se skládají ze tří částí:

- **Popis akce** agent vrací prostředí (aby něco udělal). Plánovač používá popis jenom jako název možné akce.
- **Předpoklad** je konjunkce atomů (positivních literálů), která říká, co musí být pravdivé před aplikací operátoru.
- **Účinek (efekt)** operátoru je konjunkce literálů (positivních nebo negativních), která popisuje, jak se situace mění při aplikaci operátoru.

Příklad syntaxe použité pro vytvoření *operátoru* Strips pro přesun z jednoho místa na druhé:

Op(`Action`: *Go(there)*, `Precond`: (*At(there)* ∧ *Path(here, there)*,
`Effect`: *At(there)* ∧ ¬*At(here)*)

Grafická notace pro *Go(there)*. Předpoklad je nad akcí, efekt pod akcí:



Operátor s proměnnými se nazývá **operátorové schéma**, protože neodpovídá jediné proveditelné akci, nýbrž skupině akcí, kde každá je různou instancí (specifickým výskytem) proměnných. Obvykle lze provádět pouze operátory s úplnou možností specifikace (plánovací algoritmy musí zajistit, aby každá proměnná měla hodnotu v okamžiku činnosti plánovače).

Předpoklady musí být konjunkcí positivních literálů.

Efekt musí být konjunkcí positivních a/nebo negativních literálů.

O všech proměnných se předpokládá, že jsou universálně kvantifikovatelné a že nejsou žádné přídavné kvantifikátory (tyto požadavky mohou být za určitých okolností zmírněny).

Operátor *o* je aplikovatelný ve stavu *s* pokud existuje nějaký způsob specifikace proměnných v operátoru *o* tak, že každý předpoklad *o* je pravdivý v *s* (tj. když *Precond(o)* $\subset$ *s).*

Ve výsledném stavu platí všechny positivní literály v *Effect(o)* tak, jako platí všechny literály platné v *s*, kromě těch, které jsou negativními literály v *Effect(o).* Např. pokud počáteční situace zahrnuje literály

At(Doma), Path(Doma, Supermarket), ...

pak akce *Go(Supermarket)* je aplikovatelná a výsledná situace obsahuje literály

¬*At(Doma), At(Supermarket), Path(Doma, Supermarket), ...*

Prostor situace a prostor plánu

Na obrázku stromu větvení činností při požadavku na obstarání mléka apod. je ilustrován prohledávací prostor situací ve světě (zde svět nakupování). Cesta tímto prostorem z počátečního do cílového stavu vytváří plán pro problém nákupu.

Plánovač, který daný prostor prohledává z hlediska možných situací, je **plánovač situačního prostoru**. Zároveň jde o **progresívní plánovač,** protože hledá směrem vpřed z výchozího stavu do cílového. Hlavním problémem je vysoký faktor větvení a tím velký rozměr prohledávaného prostoru.

Jednou možností pokusit se snížit faktor větvení je hledání zpětným směrem (z cíle k počátku)—tzv. **regresní** plánování. Takový přístup je možný, protože operátory obsahují dosti informace k regresi z částečného popisu výsledného stavu k částečnému popisu stavu před aplikací operátoru. Obecně není možné tímto způsobem získat kompletní popis stavů, ale to není zapotřebí.

Uvedený regresní přístup je žádoucí, protože v typických problémech má cílový stav pouze několik konjunktů, kde každý z nich má jenom několik vhodných operátorů, zatímco počáteční stav obvykle má mnoho aplikovatelných operátorů (*vhodným operátorem* rozumíme takový, který je vhodný pro cíl pokud cíl je efektem toho operátoru).

Bohužel zpětné hledání je komplikováno faktem, že často je nutno dosáhnout *konjunkce cílů*, nikoliv jen jednoho. To může snadno vést k nekompletnosti, jejíž odstranění však zase vede k vysoké neúčinnosti algoritmu.

Alternativou je hledání přes *prostor plánů* místo přes prostor situací. Znamená to, že se začne s jednoduchým, nekompletním plánem—tzv. **částečný plán**. Dále jsou zvažovány cesty k rozšíření plánu, dokud není nalezen kompletní plán řešící problém. Operátory v tomto hledání jsou operátory nad plány: přidání kroku; zavedení takového pořadí, které vloží nějaký krok před jiný; specifikace dosud neomezené proměnné; apod. Řešením je finální plán a cesta, která vede k dosažení cíle, je irelevantní.

Operace nad plány jsou ze dvou kategorií: **Zdokonalovací operátory** přidávají omezení k danému plánu—jednou z možností pohledu na částečný plán je jako na representaci souboru kompletních, plně omezených plánů. Operátory zdokonalování eliminují některé plány z tohoto souboru, ale žádné nepřidávají.

Jakýkoliv jiný, než zdokonalovací operátor, je **operátor modifikační**. Některé plánovače pracují tak, že konstruují potenciálně nekorektní plány a pak je "odlaďují" pomocí modifikačních operátorů. V dalším jsou popisovány jen operátory zdokonalovací.

Representace plánů

Pro prohledávání prostoru plánů je zapotřebí jejich representace. Jako příklad poslouží jednoduchý problém: obutí si páru bot. Cílem je konjunkce složená z

$$PraváBotaObuta \wedge LeváBotaObuta,$$

počáteční stav nemá žádné literály.

K dispozici jsou čtyři operátory:

Op(`Action:` *PraváBota*, `Precond:` *PraváPonožkaOblečena*,
`Effect:` *PraváBotaObuta*)

Op(`Action:` *PraváPonožka*, `Effect:` *PraváPonožkaOblečena*)

Op(`Action:` *LeváBota*, `Precond:` *LeváPonožkaOblečena*,
`Effect:` *LeváBotaObuta*)

Op(`Action:` *LeváPonožka*, `Effect:` *LeváPonožkaOblečena*)

Částečný plán pro daný problém má dva kroky: *PraváBota* a *LeváBota*. Otázka je, který krok se má udělat jako první? Mnoho plánovačů používá přístup zvaný **nejmenší závazek**, podle nějž se má vybrat k činnosti jen to, co je právě na starosti, a ostatní věci nechat na později. To je pro hledání vhodná metoda, protože při okamžitém výběru něčeho, co není právě aktuální, může dojít k chybě, která si později vynutí návrat. Plánovač s nejmenším závazkem by mohl pořadí obou činností (*PraváBota* a *LeváBota*) nechat nespecifikované.

Třetím krokem je *PraváPonožka*, a při přidání k plánu je třeba zajistit, aby tento cíl byl splněn před zahájením plnění cíle *PraváBota*. Vzhledem k levé botě to ale nehraje roli.

Plánovač, který representuje plány, v nichž některé kroky jsou dávány do pořadí (*před* a *po*) vzhledem ke vzájemnému vztahu, přičemž jiné kroky do pořadí dávány nejsou, se nazývá **plánovač částečného pořadí**.

Alternativou je **plánovač úplného pořadí**, v jehož plánech je jednoduchý seznam kroků. Plán s úplným pořadím, který je odvozen z plánu *P* přidáním omezení na pořadí, se nazývá **linearisací *P***.

Plánovače musí také brát ohled na vazby proměnných v operátorech. To v problému bot a ponožek není ukázáno, ale např. jedním z cílů může být *Mít(Mléko)*, a k dispozici nechť je akce *Koupit(položka, obchod)*. Rozumným závazkem k výběru této akce je *položku* svázat s *Mlékem*. Zde však není dán dobrý důvod k vazbě *obchodu*, takže princip nejmenšího závazku ponechává výběr na později. Tato strategie pomáhá zbavit se špatných plánů a vynechat celé větve stromu. Pokud je každá proměnná vázána na konstantu, pak plán se nazývá **plně konkretizovaný plán**.

Plán je formálně definován jako datová struktura skládající se z těchto složek:

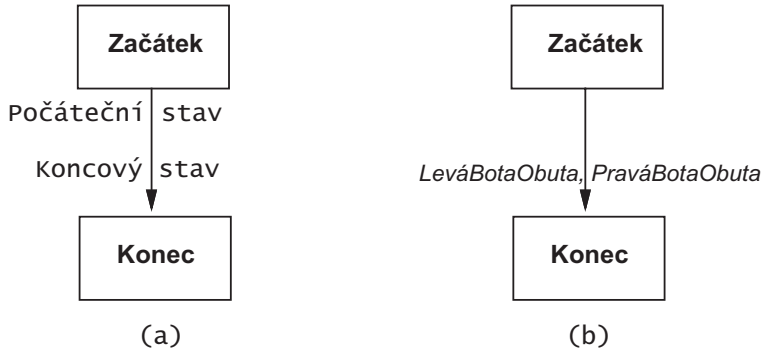
- Soubor kroků plánu. Každý krok je jeden z operátorů pro daný problém.
- Soubor omezení pro uspořádání kroků. Každé omezení, určující pořadí, má formu $S_i \prec S_j$, což znamená "S_i je před S_j", neboli krok S_i musí *někdy* předcházet krok S_j (ale *nikoliv bezprostředně*).
  Pozn.: notace $A \prec B \prec C$ znamená $(A \prec B) \wedge (B \prec C)$.
- Soubor omezení pro vazby proměnných. Každé omezení proměnné má formu $v = x$, kde v je proměnná nějakého kroku a x je buď konstanta nebo jiná proměnná.
- Soubor příčinných propojení. Příčinné propojení $S_i \xrightarrow{c} S_j$ znamená "S_i dosáhne c pro S_j". Příčinná propojení slouží k záznamu účelu/účelů kroků v plánu: zde je účelem S_i dosáhnout předpoklad pro S_j.

Před zahájením jakéhokoliv zlepšování popisuje jednoduše počáteční plán nevyřešený problém. Počáteční plán se skládá ze dvou kroků *Začátek* (nebo *Start*) a *Konec* (nebo *Finish*), s omezením na pořadí *Začátek* $\prec$ *Konec*. Oba tyto kroky mají asociované prázdné akce, takže když dojde na provádění plánu, jsou ignorovány. Krok *Start* nemá žádné předpoklady a jeho efektem je přidání všech proposic pravdivých v počátečním stavu. Krok *Konec* má jako předpoklad cílový stav a žádné efekty.

Plánovače tedy mohou začít s počátečním plánem a manipulovat s ním, dokud nedojdou k plánu, jenž je řešením.Problém boty-a-ponožky je definován dříve uvedenými čtyřmi operátory a počátečním plánem, který lze zapsat takto:

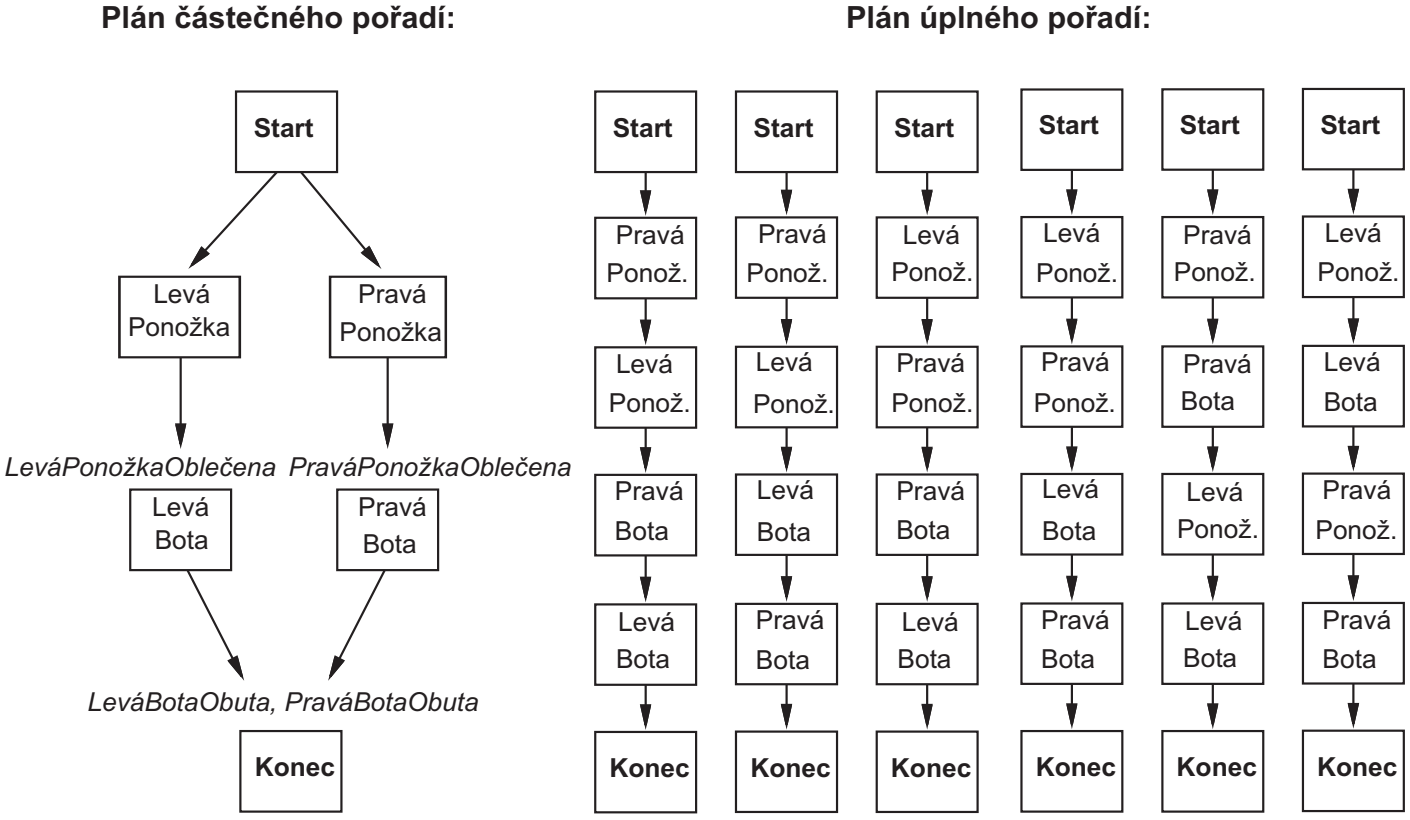
Plan (`Steps:` { S_1: *Op*(`Action:` *Start*),
S_2: *Op*(`Action:` *Finish*,
`Precond:` *PraváBotaObuta* $\wedge$ *LeváBotaObuta*)},
`Orderings:` { $S_1 \prec S_2$ },
`Bindings:` {},
`Links:` {})

Následující obrázek (a) ukazuje grafickou notaci popisující plány, podobně jako u individuálních operátorů. Další obrázek (b) ilustruje počáteční plán pro problém boty-a-ponožky:



V části (a) jsou problémy definovány částečnými plány obsahujícími pouze kroky *Začátek* a *Konec*. Do počátečního stavu se vstupuje vlivem efektu kroku *Začátek*, cílový stav je předpokladem kroku *Konec*. Omezení na pořadí je ukázáno šipkami mezi bloky. V části (b) je počáteční plán pro problém boty-a-ponožky.

Následující obrázek ukazuje plán s částečným uspořádáním, který je řešením daného problému, a dále šest linearisací plánu:



Zde je vidět, že plán částečného uspořádání je silný, protože umožňuje plánovači ignorovat výběry, které nemají žádný vliv na korektnost plánu. S narůstajícím počtem kroků roste počet možných výběrů uspořádání exponenciálně.

Řešení

Řešení je plán, který může agent provést a který zaručuje dosažení cíle. Pro kontrolu, zda plán je řešením, by bylo nutno trvat na tom, že pouze plně specifikované a zcela uspořádané plány mohou být řešeními. To je však nevyhovující ze tří důvodů:

(1) Pro problémy typu z posledního obrázku je plánovačům přirozenější vracet plán s částečným uspořádáním než libovolně vybírat jednu z mnoha linearisací plánu.
(2) Někteří agenti jsou schopni provádět akce paralelně, takže dává smysl umožnit řešení s paralelními akcemi.
(3) Při vytváření plánů, které mohou později být kombinovány s jinými plány při řešení rozsáhlejších problémů, se vyplácí uchovávat flexibilitu poskytovanou částečným uspořádáním akcí.

Proto se umožňují částečně uspořádané plány jako řešení při použití jednoduché definice: **řešení** je **kompletní** a **konsistentní** plán:

> **Kompletní plán** je ten, v němž každý předpoklad každého kroku je **dosažen** nějakým jiným krokem. Krok dosáhne předpokladu, pokud předpoklad je jedním z efektů kroku a pokud žádný jiný krok nemůže zrušit platnost předpokladu. Formálněji vzato, krok S_i dosáhne předpokladu c kroku S_j pokud:
>
> 1. $S_i \prec S_j$ a $c \in \texttt{Effects}(S_i)$;
> 2. neexistuje žádný krok S_k takový, že $(\neg c) \in \texttt{Effects}(S_k)$, přičemž platí $S_i \prec S_k \prec S_j$ v některé linearisaci plánu. □
>
> **Konsistentní plán** je ten, v němž nejsou žádné kontradikce v uspořádání nebo vázání proměnných. Kontradikce vznikne, když platí jak $S_i \prec S_j$ tak $S_j \prec S_i$ nebo platí jak $v = A$ a $v = B$ (pro dvě různé konstanty A a B). Zároveň platí, že jak $\prec$ tak = jsou transitivní, takže např. plány s $S_1 \prec S_2$, $S_2 \prec S_3$ a $S_3 \prec S_1$ jsou nekonsistentní. □

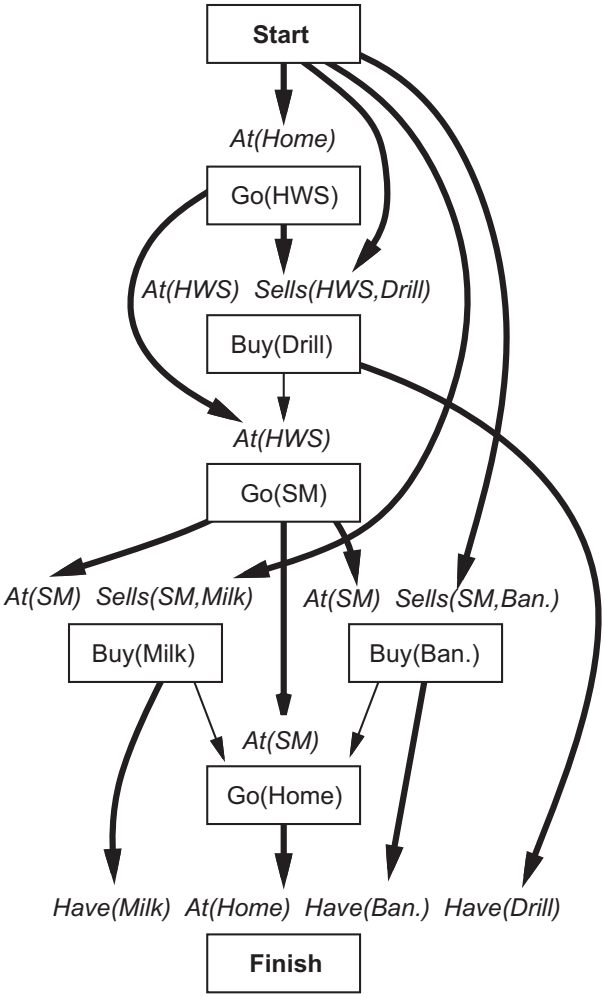
Částečný plán z posledního obrázku je řešením, protože všechny předpoklady jsou dosaženy. Z uvedených definic je zřejmé, že libovolná z linearisací řešení je také řešením. Agent může provádět kroky v libovolném pořadí s omezeními a vždy má zaručeno dosažení cíle.

Při *částečném regresívním plánování* plánovač prohledává prostor plánů. Začíná s počátečním plánem, kde jsou representovány kroky začátku a konce, a pak iteračně přidává jednotlivé kroky. Pokud je dosaženo nekonsistentního plánu, plánovač se vrátí a zkouší jinou větev v prohledávaném prostoru.

Aby bylo hledání směrováno k požadovanému cíli, plánovač pouze uvažuje přidání kroků, které slouží k dosažení předpokladu, který doposud nebyl splněn.

Pozn.: Je zřejmé, že plánovač musí udržovat záznam o cestě, aby se mohl v případě nutnosti vracet.

Příklad vyřešení problému s mlékem, banány a vrtačkou (cílem je mít všechno doma):



(*At(home)* je *doma*, *Go(HWS)* je *Jdi_do(Železářství)*, tj. *HardWare Shop*, *At(x)* je *být v x*, *SM* je *SuperMarket*, *Sells(SM,Milk)* je *Prodává(Supermarket,Mléko)* apod., *Buy(Drill)* je *Kup(Vrtačku)* apod.). Jediná nejednoznačnost v plánu je možnost zvolit jakékoliv pořadí *Buy(Milk)* a *Buy(Bananas)*, protože na tom pořadí nezáleží vzhledem k místě nákupu.

Znalostní inženýrství pro plánování

Metodologii řešení problémů s plánováním lze vyjádřit takto:

1. Definice problému.
2. Definice podmínek, operátorů a objektů.
3. Zakódování operátorů do domény.
4. Zakódování popisu výskytu specifického problému.
5. Předložení problému plánovači a získání plánů.

Příklad pro tzv. problém bloků:

1. **Definice problému:** doména obsahuje soubor bloků (kostek) na stole. Kostky mohou být kladeny na sebe, ale pouze jedna určitá kostka může být umístěna na jinou. Ruka robota může zvednout nějakou kostku a přesunout ji do jiné polohy (na stůl nebo na jinou kostku). V jednom časovém kroku může být manipulováno s jednou kostkou, nelze manipulovat s kostkou, která má další kostku/kostky na sobě. Cílem je vždy vybudovat jeden nebo více sloupků z kostek, přičemž sloupky jsou popsány pomocí toho, jaké kostky jsou na jiných kostkách—např. cílem může být vytvořit dva sloupky, jeden s kostkou *A* na kostce *B* a druhý s kostkou *C* na *D*.
2. **Definice podmínek, operátorů a objektů:** Objekty v této doméně jsou kostky a stůl. Representovány jsou konstantami. K indikaci, že kostka *b* je na *x* (*x* je buď jiná kostka nebo stůl) se použije *On(b,x)*. Operátor pro přesun kostky *b* z pozice na *x* na *y* je *Move(b,x,y)*. Jednou z podmínek přesunu *b* je, že na ní není žádná jiná kostka, tedy v logice prvního řádu: $\neg\exists x\ On(x,b)$ nebo alternativně $\forall x\ \neg On(x,b)$. Vzhledem k *representaci plánů* (viz výše) je zapotřebí vytvořit predikát representující fakt, že žádná kostka není na *b* a pak zajistit, že operátory korektně tento predikát zpracují—např. zde *Clear(x)* bude znamenat, že na *x* není nic.
3. **Operátory:** operátor *Move* přesune jednu kostku *b* z *x* na *y* pokud na *b* a na *y* není nic; jakmile je přesun hotový, *x* je volné (nic na něm není), ale *y* již nadále volné není (je na něm kostka). Formálnější zápis operátoru *Move*:

Op(`Action`: *Move(b,x,y)*,
`Precond`: *On(b,x)* ∧ *Clear(b)* ∧ *Clear(y)*,
`Effect`: *On(b,y)* ∧ *Clear(x)* ∧ ¬*On(b,x)* ∧ ¬*Clear(y)*)

Zde ale *operátor* nezpracovává *Clear* správně, pokud *x* nebo *y* je na stole. Pokud *x* = *Table*, výsledkem je *Clear(Table)*, ale stůl nemůže být zbaven kostek. Pokud *y* = *Table*, pak má podmínku *Clear(Table)*, ale stůl nemusí být bez ničeho, aby se na něj dala umístit kostka.

K vyřešení uvedených potíží se zavede další operátor *MoveToTable* pro přesun kostky *b* z *x* na *Table*:

Op(`Action`: *MoveToTable(b,x)*,
`Precond`: *On(b,x)* ∧ *Clear(b)*,
`Effect`: *On(b,Table)* ∧ *Clear(x)* ∧ ¬*On(b,x)*)

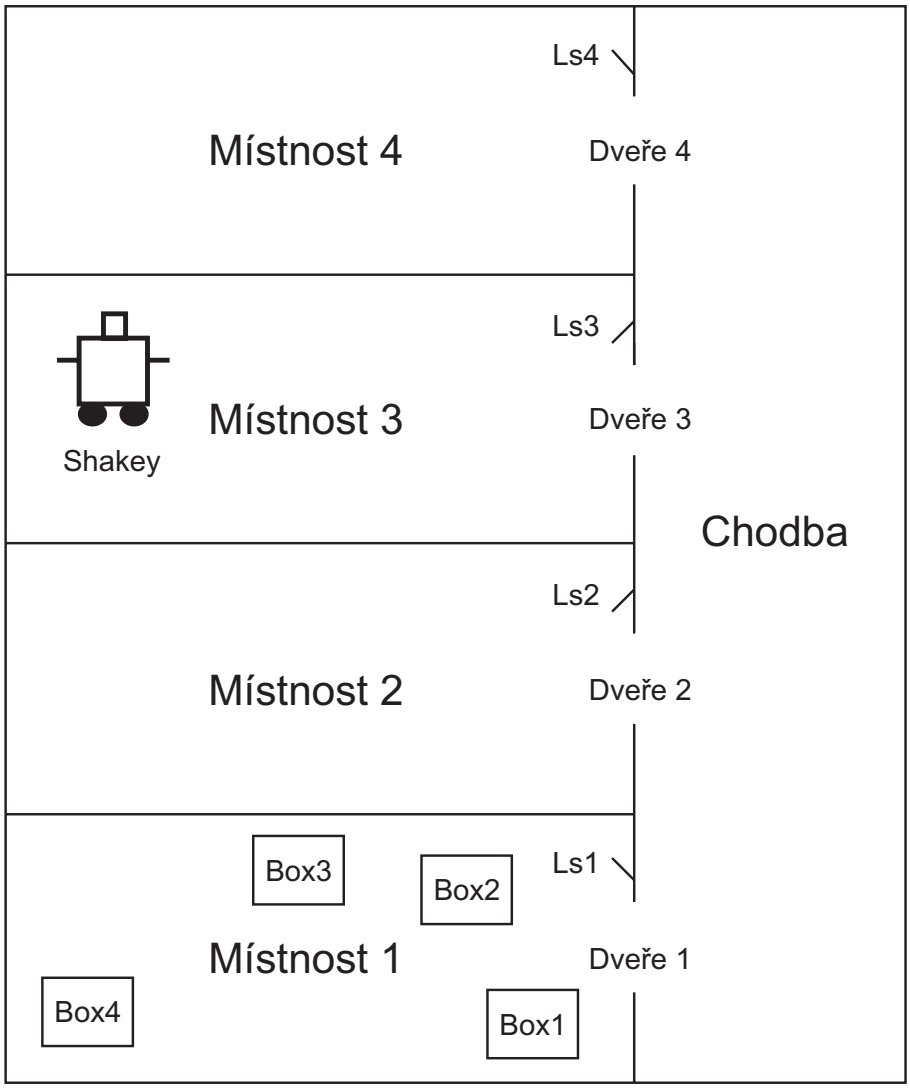
K tomu je nutno zavést ještě interpretaci *Clear(x)* ve smyslu "na *x* je volno a lze tam uložit kostku". *Clear(Table)* pak vždy bude součástí počáteční situace a je korektní, že *Move(b,Table,y)* má jako výsledek *Clear(Table)*.

Jedinou nedokonalostí je nyní fakt, že nic nemůže zabránit plánovači v použití *Move(b,x,Table)* místo *MoveToTable(b,x)*. V případě uskutečnění to vede k většímu prostoru prohledávání, než je nutno, ale nevede to k nekorektním odpovědím. Lze to odstranit zavedením dalšího predikátu *Block* a přidat *Block(b)* ∧ *Block(y)* k podmínkám v *Move* (zajistit, že jak *b* tak *y* jsou kostky, nikoliv připustit, že by se pro *y* mohlo jednat taky o stůl zpracovávaný jako kostka, přestože má určité odlišnosti—např. může na něm být více kostek, na kostce ale nikoliv).

Dalším problémem by mohla být vcelku neopodstatněná operace *Move(B,C,C)*, která by měla kontradiktorické výsledky. Obecně se tyto problémy ignorují, protože nemají žádný vliv na vytvořené plány (přesun kostky *B* z *C* na *C* je v podstatě prázdná operace). Bylo-li by nutno zajistit, aby k tomu nedocházelo, pak by se do podmínek musely zavést nerovnosti: $b \neq x \neq y$, tj. všechny objekty jsou různé.

Příklad pro tzv. Shakeyho problém:

Původní program v `Strips` byl navržen k řízení robota jménem Shakey (jméno pochází z faktu, že jeho motory ho při pohybu činily poněkud nestabilním—*to shake = kymácet se, třást se*). Robot Shakey byl vyvinut jako součást robotického projektu ve *SRI* (Stanford Research Institute) v Kalifornii počátkem 70. let a potuloval se v institutu po místnostech a chodbách. Většina Shakeyho činností byly simulace, ale občas se skutečně pohyboval v prostorách institutu s cílem vzít něco nebo přemístit něco. Obrázek ukazuje Shakeyho svět, který se skládal ze čtyř místností, podél nichž vedla chodba spojená s místnostmi dveřmi. Shakeyho úkolem bylo se pohybovat z místnosti do místnosti, posunovat pohyblivými objekty (krabicemi), vylézt na pevný objekt a slézt dolů (opět krabice), a zapnout/vypnout světlo v místnosti:



Pozn. 1: `Ls` je zapínač/vypínač osvětlení místnosti (*light switch*).
Pozn. 2: Shakey nebyl mechanicky dost obratný ke skutečnému lezení po krabicích a ovládání vypínače světla, ale `Strips` byl schopen vytvořit plány.

Pro Shakeyho lze určit následující literály a operátory:

1. ***Jdi*** ze současné pozice na jinou pozici *y*: *Go(y)*.
   Podmínka *At(Shakey,x)* stanovuje okamžitou pozici a je vhodné, aby *x* i *y* bylo v téže místnosti *r*: *In(x,r)* ∧ *In(y,r)*. Aby bylo možné naplánovat cestu z jedné místnosti do druhé, považují se tytéž dveře za součást obou místností z hlediska *In*.
2. ***Posuň*** objekt *b* z místa *x* na místo *y*: *Push(b,x,y)*.
   Zde mohou opět být *x* i *y* v téže místnosti. Zavést je nutno predikát *Pushable(b)*, tj. přemístitelnosti, ale jinak je situace obdobná *Go*.
3. ***Vylez*** na box *b*: *Climb(b)*.
   Je nutno zavést predikát *On* (kde robot je) a konstantu *Floor* (podlaha), a zajistit, aby podmínka pro *Go* byla *On(Shakey,Floor)*, tj. aby robot lezl z podlahy na krabici (a resp. zpět), nikoliv např. skákal z krabice na krabici. Pro *Climb(b)* musí být podmíka, že Shakey je *At* na tomže místě jako je *b*, a *b* musí být tzv. *Climbable* (tj. *b* je předmět, na nějž lze vylézt, tedy *vylezitelný*).
4. ***Slez*** dolů z boxu *b*: *Down(b)*.
   V principu zrušení efektu vylezení *Climb*.
5. ***Zapni*** světlo: *TurnOn(ls)*.
   Vzhledem k tomu, že Shakey na vypínač ze země nedosáhl, musel pro rozsvícení světla v místnosti vylézt na krabici *b* u vypínače *ls*.
6. ***Vypni*** světlo: *TurnOff(ls)*.
   Obdoba *Zapni*. Jazyk plánování by musel mít k dispozici podmínku pro určení stavu světla, aby se dalo určit, zda světlo je zapnuto nebo vypnuto kvůli schopnosti přepínání z jednoho stavu do druhého.

V jazyce `Strips` bylo nutno pro každý box přidat individuální literál do počátečního stavu, zatímco v situačním kalkulu se napíše axiom, který říká, že každý box lze posunout a na každý lze vylézt. Rovněž by bylo zapotřebí začlenit kompletní mapu světa v počátečním stavu—pomocí termů, které objekty jsou v které místnosti, tj. pomocí *In*. Stejně by se popsalo pomocí *At*, kde který objekt přesně je.

Jazyk Strips (a jemu obdobné) umožňují representovat jednoduché domény pomocí operátorů Strips, ale návrh množiny správných a potřebných operátorů a predikátů není vůbec snadný.