How an agent can start with a KB and a goal, and come up with a plan for how to achieve that goal by means of a sequence of actions that, one by one, change the state of the world into one where the goal has been achieved?

At one level, this is a search problem: various available actions allow various changes in the world, and one wants to find a suitable sequence that results in that goal. In effect, the given information provides a conceptual "search tree" that one must navigate to find a path to a goal. This sort of observation leads directly to the large and well-studied area of AI Search.

But if the problem can be specified in terms of logic, then there is a chance that the search can be carried out by a specialized logic engine that then lets us focus on the careful specification of the KB rather than the details of the search. This is the basis for the situation-calculus, due to McCarthy and Hayes (and implemented by Cordell Green) in 1969.

SITUATION CALCULUS

In the "sit-calc", an action is represented by means of a function f that takes a state of the world, s, as input and returns another state s' as output: s' is the state the world will be in after that action has been performed when the world was in state s. Since the action of f may involve various specific things in the world, it will also have other arguments in addition to s. Here is an example, with the action "push":

At(x,y,s) -> At(x,y,push(x,y,loc,s))

Intuitively, this says that if an entity x happens to be where entity y is, in world state s, and if while in that state s, x performs a push action on y (to some location loc), this will change the world to a new state -- written as push(x,y,loc,s) -- in which x is still where y is. This is an example of a so-called "frame axiom", stating that a certain "fluent" (a predicate that can change truth-value from one state to another – in this case At(x,y,)) is *not* changed when the state changes due to x pushing y.

The *if* part of this -- At(x,y,s) -- is a *precondition* for push: in order to push something, one must be where it is. And the postcondition of this axiom is that x is still where y is.

Of course, when pushing something, the more interesting point is that the object being pushed moves to the location it is pushed to:

At(x,y,s) -> At(y,loc,push(x,y,loc,s))

So this is a *change axiom*: it tell us how a push changes things.

One might think that what a robot's KB really needs is info about all the changes that occur when an action is done. But of course it is just as important to know what does NOT change; whatever the resulting world state is, one needs to know enough about it to be able to decide on appropriate actions. For instance, if a potted plant is on a table in the center of the room, and if we want it to get more sunlight, we might push the table close to a window. However, this will only work if the plant moves along with the table as we push it. We need to know that the fluent OnTable(plant,_) remains true when the state changes as the result of the push.

Why not just assume that a fluent does not change unless we know it will change? That is a good idea, but not so easy to make it work out in practice. It requires either:

1. knowing all changes that actions can cause, or

2. knowing "enough" about typical or important changes so that the ones one doesn't know won't cause difficulties, or

3. having really good error-detection and correction algorithms to clean up whatever messes arise from such difficulties.

Here are further examples of frame axioms: If we push a table toward a window, the walls do not fall down; the air does not leave the room; the table does not change color or weight; we do not die. These may sound silly. But when we flip a switch on the wall, the ceiling lights go on -- so things not in direct contact can affect each other, and in ways that on the face of it are not related: what does an up-down motion of a one-inch piece of plastic on the wall have to do with the presence or absence of light coming from the ceiling? -- and why not light from the floor? So, there would seem to be a lot of special knowledge that we have, about what does and does not happen.

In fact, the frame axioms that might be needed for a given planning problem can vastly outnumber all other axioms. Worse, in some cases it appears to be impossible to completely write them all down, even in principle. One speaks of the "frame problem" as the difficulty in dealing with the need for such axioms. We will say a little more about this later.

EXAMPLE: the Monkey and Bananas Problem

As we have seen, this problem involves a room, fruit f suspended from the ceiling, a movable box b, and a monkey m (you can pretend it is a robot). Actions are: push, goto, climb, grasp. Fluents are: On, Has, At. The initial situation is s_0. The location on the floor just under the fruit is L. The monkey cannot reach the fruit except by standing on the box (while the box is

positioned at L). The goal is a state s in which Has(m,f,s) is true; or at least to prove there is such a state: (Exists s) Has(m,f,s).

Here are axioms describing the initial state, and the effect of actions on fluents (loc is a variable representing possible floor locations):

INIT: -On(m,b,s_0)

Change axioms:
1. -On(m,b,s) -> At(m,b,goto(m,b,s)) [wherever you go, there you are]
2. On(m,b,s) & At(b,l,s) -> Has(m,f,grasp(m,f,s))
3. At(m,b,s) -> At(b,loc,push(m,b,loc,s)) [m is wherever it is pushed]
4. At(m,b,s) -> On(m,b,climb(m,b,s)) [you are on what you climb]

Frame axioms:
5. At(m,b,s) -> At(m,b,push(m,b,loc,s)) [pusher stays with pushed object]
6. At(b,loc,s) -> At(b,loc,climb(m,b,s)) [climbing does not move b]

NG (negated goal):
(Forall s)-Has(m,f,s)

Now we need to put all the axioms and the negated goal into CNF:

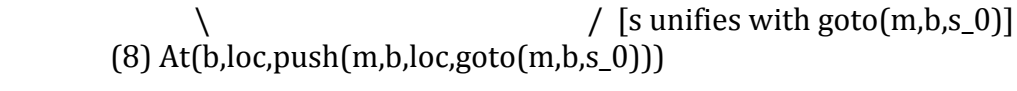
INIT: -On(m,b,s_0)

1. On(m,b,s) v At(m,b,goto(m,b,s))
2. -On(m,b,s) v -At(b,l,s) v Has(m,f,grasp(m,f,s))
3. -At(m,b,s) v At(b,loc,push(m,b,loc,s))
4. -At(m,b,s) v On(m,b,climb(m,b,s))
5. -At(m,b,s) v At(m,b,push(m,b,loc,s))
6. -At(b,loc,s) v At(b,loc,climb(m,b,s))

NG: -Has(m,f,s)

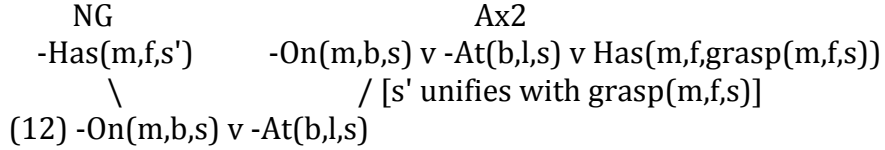
Then we try to prove the null clause from the above. Below is a start, where numbers in parens indicate results of resolution steps. Note that when the same variable (such as s) occurs in two different wffs being resolved, one has to be renamed since they need not have the same meaning.

```
INIT                          Ax1
-On(m,b,s_0)          On(m,b,s) v At(m,b,goto(m,b,s))
      \                / [s is unified with s_0]              Ax3
 (7) At(m,b,goto(m,b,s_0))              -At(m,b,s) v At(b,loc,push(m,b,loc,s))
```

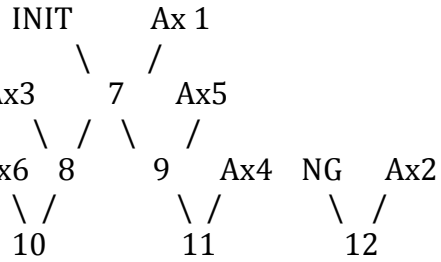


...

and so on; eventually we also need this step:



The overall pattern that results is this:



and then two more resolutions are needed (11 with 12, and that result with 10) to get the null clause.

OK, fine. This proves the monkey can get the fruit: there is a state s where Has(Mm,f,s) holds.

But how does the monkey do it? What is the sequence of actions -- i.e., the "plan" -- that gets there? That after all is what we want. There is a “trick” known as *answer extraction* that solves this. The idea is quite simple: we adjoin to the CNF version of NG a brand-new disjunct – for example ANSWER(s) – with the same situation variable s as NG has; and then whenever s is unified with something, it happens inside ANSWER as well. This keeps track of all the actions that occur in the proof of a contradiction, except that instead of a contradiction, ANSWER(…) is left standing alone, and whatever is inside (…) expresses the actions nested together in the right order.