Ability and Knowing How in the Situation Calculus

Yves Lespérance (lesperan@cs.yorku.ca)
Department of Computer Science, York University, Toronto, ON, M3J 1P3 Canada

Hector J. Levesque (hector@cs.toronto.edu)
Department of Computer Science, University of Toronto, Toronto, ON, M5S 1A4 Canada

Fangzhen Lin (flin@cs.ust.hk)
Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

Richard B. Scherl (scherl@homer.njit.edu)
Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102 USA

Abstract. Most agents can acquire information about their environments as they operate. A good plan for such an agent is one that not only achieves the goal, but is also executable, i.e., ensures that the agent has enough information at every step to know what to do next. In this paper, we present a formal account of what it means for an agent to *know how to execute a plan* and to be *able to achieve a goal*. Such a theory is a prerequisite for producing specifications of planners for agents that can acquire information at run time. It is also essential to account for cooperation among agents. Our account is more general than previous proposals, correctly handles programs containing loops, and incorporates a solution to the frame problem. It can also be used to prove programs containing sensing actions correct.

Keywords: reasoning about knowledge and action, knowledge prerequisites of actions.

1. Introduction

Work in the classical planning paradigm has generally made very strong assumptions about the domain in which planning is taking place, in particular, that the planner has complete knowledge of the initial state, and that actions are such that the planner can compute a complete description of any state reachable by doing a sequence of actions in the initial state (for instance, STRIPS [4] operators). Such assumptions cannot be sustained in most real applications (e.g., robotics, information gathering agents); there, agents need to acquire knowledge at execution time by sensing their environment.

Some work, for instance [3, 5, 8, 19, 22], has attempted to generalize classical planning techniques to deal with this. But a key problem is that in such domains, it is not even clear what a plan is and when it

is a solution to a particular planning problem. Plans must at the very least include conditional control structures so that the choice of action can depend on the result of sensing. But then it appears that standard programming language notions of correctness are insufficient. Even if it can be shown that a plan must achieve the goal (and terminate), the agent may not have enough knowledge to execute it. For example, suppose that the agent knows that behind one of two doors there is a treasure and behind the other there is a monster, but does not know which leads to what. Then, even though the plan

if TreasureBehindDoor1 **then** goThrough(Door1)
else goThrough(Door2)

can be shown to achieve the goal of getting the treasure, the agent does not *know how* to execute it because he cannot evaluate the test. Similarly,

goThrough(DoorToTreasure)

achieves the goal, but cannot be executed because the agent does not know which primitive action the program stands for. The nondeterministic plan[1]

[goThrough(Door1) | goThrough(Door2)]; AtTreasure?

also achieves the goal, but cannot be executed since the agent does not know which branch to take. However, if he can look through a window on one of the doors to determine what is behind it, then the following plan is adequate:

EXAMPLE 1.

lookThroughWindow;
if TreasureBehindDoor1 **then** goThrough(Door1)
else goThrough(Door2)

It must achieve the goal and the agent will know how to execute it.

Whether an agent knows how to execute a plan depends on how smart he is — how much he knows and what sort of inferences he can perform. A very smart agent that can do *lookahead* would know how to execute the following nondeterministic plan:

[1] Our use of test actions may be confusing to some; read $\phi?;\delta$ as "action δ occurring when ϕ holds," and for $\delta;\phi?$, read "action δ occurs after which ϕ holds." Thus, the plan in the example involves either going through Door1 or going through Door2, so that one ends up at the treasure.

EXAMPLE 2.

LOOKTHROUGHWINDOW;
pick d : [DOOR(d)?; GOTHROUGH(d)];
ATTREASURE?

The very smart agent could use its knowledge gathered through the sensing action to pick the correct door given its lookahead capability. A dumber executor would not.

All this is really part of our common sense knowledge about agents. We do not delegate a goal to someone unless we believe that he is *able* to achieve it. And even if someone does not know how to achieve a goal on his own, we may still enlist his help by providing instructions he knows how to follow. Such instructions would typically not specify the plan down to the last detail; we assume some intelligence on the part of the executor.

The classical planning paradigm involves a very smart planner and a very dumb executor — it is assumed that the difficult problem solving is performed at planning time, and that execution is relatively direct. But there is no real reason to restrict our attention to this picture. In some cases, planning from scratch may be so hard that it is better to try to build a smart executor that the user can program at a high level — we pursue this in [12]. Others have suggested that the right role for plans is as advice to a relatively smart improvisation module [1]. Also, multi-agent systems are becoming more common and typically involve agents at different levels of smartness. All this suggests studying what knowing how or ability means for agents with varying levels of intelligence.

Before one even starts talking about plans, it is useful to have a formal account of what sort of knowledge is involved in the ability to achieve a goal. This is what we develop in section 3. Plans are partial representations of this kind of knowledge; how complete they must be depends on how smart the intended executor is. In section 4, we develop two accounts of knowing how to execute a plan, one for a very smart agent and another for a much dumber one. In fact, these are merely two points in a space of agents with various kinds of abilities. But as argued in the concluding section, the framework we propose provides a useful foundation for further exploration of this space.

We will discuss related work as it becomes relevant. It is worth singling out, however, the very similar work of Ernest Davis [2]. Like us, Davis develops accounts of knowing how to execute a plan for both smart and dumb executors. However in [2], he fails to show that his account really handles unbounded iteration, a key problem area in earlier work. Nor does he discuss ability to achieve a goal and its relation

to knowing how. Although developed independently, our accounts of knowing how are remarkably similar, and it seems that most of our results could have been obtained using his axiomatization as a starting point. We point out some of the differences as they become pertinent.

2. A Theory of Action

Our theory is based on an extended version of the situation calculus [15], a predicate calculus dialect for representing dynamically changing worlds. In this formalism, the world is taken to be in a certain situation (or state). That situation can only change as a result of an agent doing an action. The term $do(a, s)$ represents the situation that results from the agent's performance of action a in situation s. The initial situation is represented by the constant S_0. Thus for example, the formula $\textsc{On}(\mathrm{A}, \mathrm{B}, do(\textsc{putOn}(\mathrm{A}, \mathrm{B}), S_0))$ could mean that A is on B in the situation that results from the agent's doing $\textsc{putOn}(\mathrm{A}, \mathrm{B})$ in the initial situation. Predicates and function symbols whose value may change from situation to situation (and whose last argument is a situation) are called *fluents*. Note that we write $s < s'$ if and only if s' is the result of doing some sequence of actions in s, where the actions are possible in the situation where they are done[2].

An action is specified by first stating the conditions under which it can be performed by means of a *precondition axiom*. For example,

$$Poss(\textsc{pickUp}(x), s) \equiv \forall z \neg \textsc{Holding}(z, s) \wedge \textsc{nextTo}(x, s)$$

means that it is possible for the agent to pick up an object x in situation s if and only if he is not holding anything and is standing next to x in s. Then, one specifies how the action affects the world's state with *effect axioms*, for example:

$$Poss(\textsc{drop}(x), s) \wedge \textsc{Fragile}(x) \supset \textsc{Broken}(x, do(\textsc{drop}(x), s)).$$

The above axioms are not sufficient if one wants to reason about change. It is usually necessary to add frame axioms that specify when fluents remain unchanged by actions. The frame problem [15] arises because the number of these frame axioms is of the order of the product of the number of fluents and the number of actions. Our approach incorporates a solution to the frame problem due to Reiter [20] (who extends previous proposals by Pednault [18], Schubert [23] and Haas [7]). The basic idea behind this is to collect all effect axioms about

[2] The relation $<$ on situations is fully axiomatized in the foundational axioms (for example, see [10]).

a given fluent and assume that they specify all the ways the value of the fluent may change. A syntactic transformation can then be used to obtain a *successor state axiom* for the fluent, for example:

$$Poss(a,s) \supset [\textsc{Broken}(x, do(a,s)) \equiv \\ (a = \textsc{drop}(x) \wedge \textsc{Fragile}(x)) \vee (\textsc{Broken}(x,s) \wedge a \neq \textsc{repair}(x))].$$

This says that x is broken after the agent does action a in situation s if and only if either the action was dropping x and x is fragile, or x was already broken in s and the action was not repairing it. This treatment avoids the proliferation of axioms, as it only requires a single successor state axiom per fluent and a single precondition axiom per action.[3]

Scherl and Levesque [21] have generalized this account to handle sensing or knowledge-producing actions. Such actions affect the mental state of the agent rather than the state of the external world. For example, it should be the case that after performing the action senseDown, an agent that is trying to cut down a tree would know whether the tree is down:

$$Poss(\textsc{senseDown}, s) \supset \mathbf{KWhether}(\textsc{Down}, do(\textsc{senseDown}, s)).$$

$\mathbf{KWhether}(\phi, s)$ is an abbreviation for $\mathbf{Know}(\phi, s) \vee \mathbf{Know}(\neg\phi, s)$. Similarly, after doing readCombOfSafe, an agent would know what the combination of the safe he is trying to open is:

$$Poss(\textsc{readCombOfSafe}, s) \supset \\ \exists c\, \mathbf{Know}(\textsc{combOfSafe} = c, do(\textsc{readCombOfSafe}, s)).$$

Knowledge is represented by adapting Kripke's possible world semantics [9] to the situation calculus, as first done by Moore [16]. $K(s', s)$ represents the fact that in situation s, the agent thinks that the world could be in situation s'. $\mathbf{Know}(\phi, s)$ is an abbreviation for the formula $\forall s'(K(s', s) \supset \phi(s'))$. For clarity, we sometimes use the pseudo-variable now to represent the situation bound by the enclosing **Know**; so $\mathbf{Know}(\textsc{Down}(\mathsf{now}), s)$ stands for $\forall s'(K(s', s) \supset \textsc{Down}(s'))$. We require K to be transitive and euclidean, which ensures that the agent always knows whether he knows something (i.e., positive and negative introspection).

For a domain with the two sensing actions described above, the successor state axiom for the knowledge fluent K can be specified as

[3] This discussion ignores the ramification problem; a treatment compatible with our approach has been proposed by Lin and Reiter [14].

follows:

$$\begin{array}{l} Poss(a,s) \supset (K(s^*, do(a,s)) \equiv \\ \quad \exists s'[K(s',s) \wedge s^* = do(a,s') \wedge Poss(a,s') \wedge \\ \qquad (a = \textsc{senseDown} \supset (\textsc{Down}(s') \equiv \textsc{Down}(s))) \wedge \\ \qquad (a = \textsc{readCombOfSafe} \supset \\ \qquad\quad \textsc{CombOfSafe}(s') = \textsc{CombOfSafe}(s))]). \end{array}$$

First note that for non-knowledge-producing actions (e.g. $\textsc{drop}(x)$), the specification ensures that the only change in knowledge that occurs in moving from s to $do(\textsc{drop}(x), s)$ is the knowledge that the action DROP has been successfully performed. For the case of a knowledge-producing action such as SENSEDOWN, the idea is that in moving from s to $do(\textsc{senseDown}, s)$, the agent not only knows that the action has been performed (as above), but also the truth value of the associated predicate DOWN. Since in this case we require that $\textsc{Down}(s') \equiv \textsc{Down}(s)$, DOWN will have the same truth value in all s' such that $K(do(\textsc{senseDown}, s'), do(\textsc{senseDown}, s))$. Observe that for any situation s, DOWN is true at $do(\textsc{senseDown}, s)$ if and only if DOWN is true at s. Therefore, DOWN has the same truth value in all worlds s^* such that $K(s^*, do(\textsc{senseDown}, s))$, and so **KWhether**(DOWN, $do(\textsc{senseDown}, s)$) holds. Similar reasoning explains why we must have $\exists c\, \mathbf{Know}(\textsc{CombOfSafe} = c, do(\textsc{readCombOfSafe}, s))$. This can be extended to an arbitrary number of knowledge-producing actions in a straightforward way.

In general, a particular domain will be specified by the union of the following sets of axioms:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action.
- Successor state axioms, one for each fluent.
- Unique names axioms for the primitive actions.
- Some foundational domain-independent axioms, which are similar to the ones given in [10].

3. Ability

Very roughly, ability to achieve a goal involves knowing what to do when, so as to arrive at a goal state. We make this more precise by appealing to the notion of an *action selection function,* a mapping

from situations to primitive actions. We understand such a function as prescribing which action the agent should perform in a situation. We say that situation s' is on the path prescribed by action selection function σ in situation s if and only if there is a path from s to s' and at every step along the way, the action performed is the one prescribed by σ:

$$\mathbf{OnPath}(\sigma, s, s') \stackrel{\text{def}}{=} s \leq s' \wedge \forall a \forall s^*(s < do(a, s^*) \leq s' \supset \sigma(s^*) = a).$$

Here $s \leq s'$ is shorthand for $s < s' \vee s = s'$. Note that $\mathbf{OnPath}(\sigma, s, s')$ implies that all the actions prescribed by σ between s and s' are possible.

We will say that the agent "can get" to a situation where a goal ϕ holds by following action selection function σ in situation s if and only if there is a situation s' on the path prescribed by σ in s where the agent knows that the goal holds, and at every step between s and s', the agent knows what the next action prescribed by σ is:

$$\begin{aligned}\mathbf{CanGet}(\phi, \sigma, s) \stackrel{\text{def}}{=} \exists s'(&\mathbf{OnPath}(\sigma, s, s') \wedge \mathbf{Know}(\phi, s') \wedge \\ &\forall s^*[s \leq s^* < s' \supset \exists a\, \mathbf{Know}(\sigma(\mathsf{now}) = a, s^*)]).\end{aligned}$$

Finally, we say that the agent can achieve a goal ϕ in situation s if and only if there exists an action selection function σ such that he knows in s that he can get to a situation where the goal holds by following σ:

$$\mathbf{Can}(\phi, s) \stackrel{\text{def}}{=} \exists \sigma\, \mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), s).$$

For the example sketched in the introduction, where an agent wants to get to a treasure but does not know which of two doors leads to it, it is straightforward to verify that our definition yields the right results, i.e., that the agent can achieve the goal if and only if it is possible for him to sense whether the treasure is behind a given door. Our account also gives the right results for more challenging examples involving unbounded iteration, such as the following:

EXAMPLE 3. Consider a situation where an agent wants to cut down a tree. This yields the following definition and axioms:

$$\textsc{Down}(s) \stackrel{\text{def}}{=} \textsc{remainingChops}(s) = 0,$$

$$\begin{aligned}Poss(a, s) \supset [&\textsc{remainingChops}(do(a, s)) = n \equiv \\ &a = \textsc{chop} \wedge \textsc{remainingChops}(s) = n + 1 \vee \\ &a \neq \textsc{chop} \wedge \textsc{remainingChops}(s) = n],\end{aligned}$$

$$Poss(\textsc{chop}, s) \equiv \textsc{remainingChops}(s) > 0.$$

We assume that the tree will fall down after some number (unknown to the agent) of primitive chopping actions (in other words, there is a natural number n such that REMAININGCHOPS$(S_0) = n$). We also assume that the agent can always find out whether the tree is down by sensing. This yields the following successor state axiom for K and precondition axiom for SENSEDOWN:

$$Poss(a,s) \supset (K(s^*, do(a,s)) \equiv$$
$$\exists s'[K(s',s) \wedge s^* = do(a,s') \wedge Poss(a,s') \wedge$$
$$(a = \text{SENSEDOWN} \supset (\text{DOWN}(s') \equiv \text{DOWN}(s)))]),$$

$$Poss(\text{SENSEDOWN}, s) \equiv True.$$

Notice however that we do not assume that the agent knows how many chop actions are necessary to get the tree down. Even then, it seems that the agent should be able to achieve the goal of cutting the tree down; all he needs to do is to keep sensing and chopping until the tree is down. Indeed, it is straightforward to verify that the above axioms imply that **Can**(DOWN, S_0). Consider the action selection function such that $\sigma(s)$ is CHOP whenever $\exists s^*\, s = do(\text{SENSEDOWN}, s^*)$, and SENSEDOWN otherwise. It is easy to show that the agent must always know what action is prescribed by σ. And since in any belief alternative REMAININGCHOPS chops are sufficient to get the tree down, it follows that the agent can get to a goal state by following σ.

EXAMPLE 4. Now, suppose that the agent has no way of sensing whether the tree is down. Then, we get the following successor state axiom for K:

$$Poss(a,s) \supset (K(s^*, do(a,s)) \equiv$$
$$\exists s'[K(s',s) \wedge s^* = do(a,s') \wedge Poss(a,s')]).$$

Suppose also that $\neg$**Know**(DOWN, S_0). Then, we would expect the agent to be unable to get the tree down. Indeed, it can be verified that $\neg$**Can**(DOWN, S_0): the assumptions imply that **Know**($\forall s^*$(now $\leq$ $s^* \supset \neg$**Know**(DOWN, s^*)), S_0); by the definition of **Can**, the result follows.

To our knowledge, this is the first time an account has been shown to handle both ability and inability in cases involving unbounded iteration. The earlier accounts of Moore [16] and Morgenstern [17] have problems with such cases; we explain their inadequacies in the next section. Van der Hoek, van Linder and Meyer [25] have also developed a logic of ability that handles unbounded iteration properly, but in a more restrictive propositional modal framework.

Let us now examine some properties of our definition of ability and see how some alternative definitions fail to handle important cases. To simplify the discussion, for the remainder of this section we will be assuming that all actions are possible, i.e., $\forall a \forall s Poss(a, s)$. Our results could easily be generalized. If one were to try to give an inductive definition of **Can**, one would likely start from the observations that:

- if a goal is known to hold already, then it can be achieved, and
- if there is an action such that the agent knows that he can achieve the goal after the action is performed, then he can achieve the goal from the beginning.

In fact, we have shown that given our definition, **Can** holds if and only if one of the above conditions hold:

PROPOSITION 5.

$$\mathbf{Can}(\phi, s) \equiv (\mathbf{Know}(\phi, s) \vee \exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s)).$$

Note that establishing this result (in either direction) requires the assumption that agents have negative introspection (i.e., that K is euclidean). This is one point over which our account differs from Davis's [2], so the proposition would not hold in his system.

The above result might suggest a simpler way of defining ability: use the above equivalence as an axiom to somehow define **Can**. Unfortunately, this approach does not seem to work. By itself, the axiom is too weak; for instance, it is consistent with it that **Can** (for any given goal) is always true. If on the other hand, we try to define ability as the least fixed-point of the above equivalence, the resulting version of ability ends up being too strong. Let

$$\begin{aligned} \mathbf{Can}_{\perp}(\phi, s) \stackrel{\text{def}}{=} \forall C(& \\ \forall s'[C(s') \equiv \mathbf{Know}(\phi, s') \vee \exists a\, \mathbf{Know}(C(do(a, \mathsf{now})), s')] & \\ \supset C(s)). & \end{aligned} \tag{1}$$

Now using proposition 5, it is easy to show that $\mathbf{Can}_{\perp}$ is stronger than **Can**, i.e. $\forall s(\mathbf{Can}_{\perp}(\phi, s) \supset \mathbf{Can}(\phi, s))$. However, $\mathbf{Can}_{\perp}$ is not implied by **Can**. In fact, $\mathbf{Can}_{\perp}$ fails to handle our tree chopping example — we get that $\neg\mathbf{Can}_{\perp}(\textsc{Down}, S_0)$ despite the fact that intuitively, the agent can get the tree down by repeatedly sensing and chopping. To see this, take C to be true of a situation if and only if the tree is known to be down in that situation. Then C clearly satisfies the equivalence in (1). But this means that $\mathbf{Can}_{\perp}$ will be true in no additional situations, as it is a least fixed point. Since the tree is not down in the initial situation S_0, this means that $\mathbf{Can}_{\perp}$ is false in S_0.

Historically, our definition of **Can** was motivated by $\mathbf{Can}_\perp$, and its failure on the tree example. It remains an open question whether there is a natural fixed-point equation like the equivalence inside (1) for which **Can** is the least fixed-point solution. We also considered an iterative analogue to $\mathbf{Can}_\perp$, which is discussed in Appendix B; it too failed to handle the tree example properly.

4. Knowing How

To get help from other agents in achieving our goals, we often need to give them explicit instructions, some sort of program to execute. Whether an agent knows how to execute a program depends on how smart the agent is. We will now formalize some notions of knowing how that appear significant; towards the end, we also relate knowing how to ability to achieve a goal.

4.1. Programs in the Extended Situation Calculus

Our programs will include the following nondeterministic forms:

$\delta_1 \vert \delta_2$	nondeterministic choice of branch
$\pi x\, \delta(x)$	nondeterministic choice of argument
$\pi a\, \delta(a)$	nondeterministic choice of primitive action

To be able to talk about the different deterministic execution paths through a nondeterministic program, we will extend our earlier notion of action selection function. Let a *path selection function* σ be a mapping from situations into pairs of objects and actions.[4] To simplify our notation, for any path selection function σ, and any situation s, we denote the left member of $\sigma(s)$ as $\sigma_l(s)$, and the right member as $\sigma_r(s)$, i.e. $\sigma(s) = (\sigma_l(s), \sigma_r(s))$. We will use σ_l to pick an object in interpreting $\pi x\, \delta(x)$ and similarly for σ_r and $\pi a\, \delta(a)$. To handle $\delta_1 | \delta_2$, we introduce a reserved action constant symbol $null$; we will take the left branch if and only if $\sigma_r(s) = null$. Semantically, $null$ behaves like a no-op, and has no effects.

We introduce programs into the formalism as abbreviations (macros), in the style of [12]. The abbreviation $Do(\delta, \sigma, s, s')$, where δ is a program and σ is a path selection function, means that the execution of δ according to σ starting in situation s terminates in the situation s'. It

[4] It will generally be clear from context whether σ refers to a path selection function or an action selection function; we shall be explicit when confusion could arise.

is defined inductively as follows:

$Do(\theta, \sigma, s, s') \stackrel{\text{def}}{=} Poss(\theta, s) \wedge s' = do(\theta, s)$, for any primitive action θ.

$Do(\phi?, \sigma, s, s') \stackrel{\text{def}}{=} \phi(s) \wedge s' = s.$

$Do(\delta_1; \delta_2, \sigma, s, s') \stackrel{\text{def}}{=} \exists s''(Do(\delta_1, \sigma, s, s'') \wedge Do(\delta_2, \sigma, s'', s')).$

$Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, \sigma, s, s') \stackrel{\text{def}}{=}$
$(\phi(s) \supset Do(\delta_1, \sigma, s, s')) \wedge (\neg\phi(s) \supset Do(\delta_2, \sigma, s, s'))$

$Do(\delta_1|\delta_2, \sigma, s, s') \stackrel{\text{def}}{=} (\sigma_r(s) = null \supset Do(\delta_1, \sigma^+, s, s')) \wedge$
$(\sigma_r(s) \neq null \supset Do(\delta_2, \sigma^+, s, s')).$

$Do(\pi x\, \delta(x), \sigma, s, s') \stackrel{\text{def}}{=} Do(\delta(\sigma_l(s)), \sigma^+, s, s').$

$Do(\pi a\, \delta(a), \sigma, s, s') \stackrel{\text{def}}{=} Do(\delta(\sigma_r(s)), \sigma^+, s, s').$

$Do(\textbf{while } \phi \textbf{ do } \delta, \sigma, s, s') \stackrel{\text{def}}{=}$
$\forall P\{\forall s_1(\neg\phi(s_1) \supset P(s_1, s_1)) \wedge$
$\forall s_1, s_2, s_3(\phi(s_1) \wedge Do(\delta, \sigma, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3))\}$
$\supset P(s, s').$

Here σ^+ is defined by the following axiom:

$$\forall s\, \sigma^+(s) = \sigma(do(null, s)).$$

This is needed in order to properly handle cases like $(A|B)|C$ and $\pi x(\pi y\, A(x, y))$. So the *null* action plays two roles: it handles the nesting of $|$ and π operators by advancing the path selection function after each selection, and as a possible value of a path selection function, it is used to select which branch of $\delta_1|\delta_2$ one should take.

Given a program δ and a path selection function σ, there is at most one terminating situation:

PROPOSITION 6. $Do(\delta, \sigma, s, s_1) \wedge Do(\delta, \sigma, s, s_2) \supset s_1 = s_2.$

If $|$ and π do not occur in a program δ, we say that it is *determinate*. It is clear from the definition that path selection functions play no role in the interpretation of determinate programs:

PROPOSITION 7.

If δ is determinate, then $\forall \sigma, \sigma', s, s'(Do(\delta, \sigma, s, s') \equiv Do(\delta, \sigma', s, s'))$.

Let us define

$$Do(\delta, s, s') \stackrel{\text{def}}{=} \exists \sigma\, Do(\delta, \sigma, s, s').$$

Thus, $Do(\delta, s, s')$ means that there is an execution of δ (determined by some path selection function) in s that terminates in s'. Then, from the above proposition, we have that if δ is determinate, then $\forall \sigma, s, s' (Do(\delta, s, s') \equiv Do(\delta, \sigma, s, s'))$.

In formalizing knowing how, we must consider not just terminating situations, but also all intermediate situations. We shall use the abbreviation $During(\delta, \sigma, s, s')$ to mean that situation s' occurs during the execution of δ starting in s according to σ. If there is a situation s^* such that $Do(\delta, \sigma, s, s^*)$ holds, then $During(\delta, \sigma, s, s')$ holds if and only if $s \leq s' \leq s^*$. However, we also want $During$ to hold for the situations encountered in executions that do not successfully terminate. For non-terminating executions, all situations encountered are $During$; so for example, $During(\textbf{while}\ True\ \textbf{do}\ null, \sigma, s, s')$ holds if and only if s' is a successor of s where only $null$ actions happen between s and s'. For executions that terminate unsuccessfully, all situations between the starting situation and the one where the program fails are $During$; for example, $During(\text{STACKONTO(A, B)}; False?, \sigma, s, s')$ holds if and only if s' is s or $do(\text{STACKONTO(A, B)}, s)$.

We define $During(\delta, \sigma, s, s')$ in a way similar to Do:[5]

$During(\theta, \sigma, s, s') \stackrel{\text{def}}{=} s = s' \vee (Poss(\theta, s) \wedge s' = do(\theta, s))$,
for any primitive action θ.

$During(\phi?, \sigma, s, s') \stackrel{\text{def}}{=} \phi(s) \wedge s' = s.$

$During(\delta_1; \delta_2, \sigma, s, s') \stackrel{\text{def}}{=} During(\delta_1, \sigma, s, s') \vee$
$\exists s'' (Do(\delta_1, \sigma, s, s'') \wedge During(\delta_2, \sigma, s'', s')).$

$During(\textbf{if}\ \phi\ \textbf{then}\ \delta_1\ \textbf{else}\ \delta_2, \sigma, s, s') \stackrel{\text{def}}{=}$
$(\phi(s) \supset During(\delta_1, \sigma, s, s')) \wedge (\neg\phi(s) \supset During(\delta_2, \sigma, s, s'))$

$During(\delta_1 | \delta_2, \sigma, s, s') \stackrel{\text{def}}{=}$
$(\sigma_r(s) = null \supset During(\delta_1, \sigma^+, s, s')) \wedge$
$(\sigma_r(s) \neq null \supset During(\delta_2, \sigma^+, s, s')).$

$During(\pi x\, \delta(x), \sigma, s, s') \stackrel{\text{def}}{=} During(\delta(\sigma_l(s)), \sigma^+, s, s').$

[5] Although $During$ cannot be defined in terms of Do, there is a way that Do can be defined in terms of $During$. However doing this requires that we introduce some special mechanism to distinguish failing states and terminating states, and we shall not pursue this further here.

$$During(\pi a\, \delta(a), \sigma, s, s') \stackrel{\text{def}}{=} During(\delta(\sigma_r(s)), \sigma^+, s, s').$$

$$\begin{aligned} &During(\textbf{while } \phi \textbf{ do } \delta, \sigma, s, s') \stackrel{\text{def}}{=} \\ &\quad \forall P\{\forall s_1(\neg\phi(s_1) \supset P(s_1, s_1)) \wedge \forall s_1, s_2[\phi(s_1) \wedge \\ &\qquad (During(\delta, \sigma, s_1, s_2) \vee \\ &\qquad\ \exists s_3(Do(\delta, \sigma, s_1, s_3) \wedge P(s_3, s_2)) \supset P(s_1, s_2))]\} \supset P(s, s'). \end{aligned}$$

4.2. Executability under a Strategy

A path selection function specifies a kind of execution strategy. We say that an agent *can execute* a program when he follows *a given strategy* if and only if the program terminates when executed according to the strategy and at every point during the execution, either the agent knows that the program has terminated or knows which action to perform next. We define this formally as follows:[6]

$$\begin{aligned} &\textbf{CanExec}(\delta, \sigma, s) \stackrel{\text{def}}{=} \exists s^*\, Do(\delta, \sigma, s, s^*) \wedge \\ &\ \forall s_i(During(\delta, \sigma, s, s_i) \supset \\ &\quad \{\forall s', s_i'[K(s', s) \wedge K(s_i', s_i) \wedge s' \leq s_i' \supset Do(\delta, \sigma, s', s_i')] \vee \\ &\qquad \exists a \forall s', s_i'[K(s', s) \wedge K(s_i', s_i) \wedge s' \leq s_i' \supset During(\delta, \sigma, s', do(a, s_i'))]\}). \end{aligned}$$

Note that an agent may be able to execute a program according to a strategy without knowing in advance that the program will terminate:

$$\textbf{CanExec}(\delta, \sigma, s) \not\supset \textbf{Know}(\exists s'\, Do(\delta, \sigma, \mathsf{now}, s'), s).$$

For example, consider the program SENSE_P; **while** $\neg P$ **do** $null$. Assume that P holds initially but the agent is not aware of that, i.e., $P(S_0) \wedge \neg\textbf{Know}(P, S_0)$. Then the agent can execute the program in S_0 because after doing SENSE_P, he will know that P holds, and will not enter the infinite while loop. But initially, the agent does not know that the program will terminate, because as far as he is concerned, it may well be the case that $\neg P$. This implies that an agent may be able to execute a program according to a strategy without realizing that this is the case:

$$\textbf{CanExec}(\delta, \sigma, s) \not\supset \textbf{Know}(\textbf{CanExec}(\delta, \sigma, \mathsf{now}), s).$$

It is also worth noting that since the execution of determinate programs does not depend on the execution strategy, we have:

[6] Another way of understanding this is the following: the combination of a nondeterministic program and an execution strategy stands for the deterministic specialization of the program obtained by executing it with the strategy; then $\textbf{CanExec}(\delta, \sigma, s)$ stands for ability to execute the deterministic program referred to by $\langle\delta, \sigma\rangle$.

PROPOSITION 8.

For all determinate programs δ,
$\exists\sigma \mathbf{CanExec}(\delta, \sigma, s) \supset \forall\sigma \mathbf{CanExec}(\delta, \sigma, s)$.

4.3. Dumb Knowing How

One way an agent may execute a possibly nondeterministic program is by arbitrarily picking an alternative at every choice point. Since we cannot rule out any execution strategy, we must require that he be able to execute the program according to *all* strategies to ensure he will succeed. This ability to blindly execute a program is what we call *dumb knowing how*. We define the notion formally as follows:

$$\mathbf{DKH}(\delta, s) \stackrel{\text{def}}{=} \forall\sigma[\forall s'\exists x\, \mathbf{Know}(\sigma(\mathsf{now}) = x, s') \supset \mathbf{CanExec}(\delta, \sigma, s)].$$

Note that we only consider path selection functions whose value is always known to the agent, that is, strategies that the agent knows how to follow. With respect to the situation described earlier where someone is seeking a treasure, a dumb agent knows how to execute the program in example 1, but not the one in example 2.

One can show that if an agent can blindly execute a program, then the program must terminate no matter what execution strategy is used:

PROPOSITION 9. $\mathbf{DKH}(\delta, s) \supset \forall\sigma\exists s'\, Do(\delta, \sigma, s, s')$.

The **DKH** notion is particularly useful for cases where an agent wants to delegate a task to another agent. For instance, in a cooperative environment, agent A may come up with a plan to achieve one of his goals, make sure that agent B knows how to dumbly execute this plan, and then ask B to execute it. If B collaborates and tries to execute the program, he will be able to do so. The execution will eventually terminate, A's goal will be achieved, and B will be able to go on to other business. (B, having faith in agent A, need not know that he knows how to execute the program; he can simply trust agent A on this.) A special case is when A and B are the same agent (e.g., one that does off-line planning and later dumb execution). Then the agent knows that he knows how to dumbly execute the program, i.e., $\mathbf{Know}(\mathbf{DKH}(\delta, \mathsf{now}), s)$.

4.4. Smart Knowing How

Another way an agent may execute a possibly nondeterministic program is by considering ahead of time whether there are alternatives

at every choice point whose choice guarantees that he will be able to complete the execution of the program. Such an ideal agent is looking ahead before committing to any execution strategy. It seems that if such an agent knows of some strategy that he can execute the program under this strategy, then we can be confident that he will pick that strategy (or some equally good one) and succeed in executing the program. We call this ability to smartly execute a program *smart knowing how*. It is defined formally as follows:

$$\mathbf{SKH}(\delta, s) \stackrel{\text{def}}{=} \exists \sigma\, \mathbf{Know}(\mathbf{CanExec}(\delta, \sigma, \mathsf{now}), s).$$

For instance, a smart agent does know how to execute the program in example 2 (as well as that in example 1). However, no agent will ever know how to execute $\mathit{False}?|\mathbf{while}\ \mathit{True}\ \mathbf{do}\ \mathit{null}$, because neither of its branches can be executed; the first one fails and the second one loops forever.

An immediate consequence of the definition is that if an agent knows how to smartly execute δ, then he knows that δ has a terminating execution path:

PROPOSITION 10. $\mathbf{SKH}(\delta, s) \supset \mathbf{Know}(\exists s'\, \mathit{Do}(\delta, \mathsf{now}, s'), s)$.

We mentioned earlier that the accounts proposed by Moore [16] and Morgenstern [17] are inadequate for dealing with unbounded iteration. The problem arises with non-terminating programs such as $\mathbf{while}\ \mathit{True}\ \mathbf{do}\ a$. Intuitively, we would want to say that no agent knows how to execute such a program, as it is impossible to bring it to termination. Our account conforms to this and yields $\neg\mathbf{SKH}(\mathbf{while}\ \mathit{True}\ \mathbf{do}\ a, s)$ as well as $\neg\mathbf{DKH}(\mathbf{while}\ \mathit{True}\ \mathbf{do}\ a, s)$. The axioms provided by Moore and Morgenstern however, do not rule out an agent's knowing how to execute such a program. Davis [2] does not discuss the issue of knowing how for programs involving unbounded iteration. His account appears to handle such cases properly, but no examples are provided. Singh [24] has also developed an account of knowing how in a more restrictive propositional modal setting.

4.5. Relationships among these Notions

It is interesting to examine the relationships among these notions. First, if an agent knows that he knows how to execute a program acting as a dumb executor, then he also knows how to execute it acting smart:

PROPOSITION 11.

For all complex actions δ, $\mathbf{Know}(\mathbf{DKH}(\delta, \mathsf{now}), s) \supset \mathbf{SKH}(\delta, s)$.

The converse does not hold in general because there may be strategies under which the program cannot be executed and a smart executor will be able to avoid these, while a dumb one will not. However, since the execution of determinate programs is independent of any strategy, we have:

PROPOSITION 12.

For all determinate complex actions δ,
$\mathbf{Know}(\mathbf{DKH}(\delta, \mathsf{now}), s) \equiv \mathbf{SKH}(\delta, s)$.

The notion of ability to achieve a goal defined earlier can be related to that of smart knowing how in a very natural way. Let us define

$$Achieve(\phi) \stackrel{\text{def}}{=} \mathbf{while}\ \neg\mathbf{Know}(\phi)\ \mathbf{do}\ \pi a\, a.$$

$Achieve(\phi)$ is a kind of universal program for achieving the goal ϕ. Then, we can show that being able to achieve a goal is equivalent to knowing how to achieve it by executing the universal program:

PROPOSITION 13. $\mathbf{Can}(\phi, s) \equiv \mathbf{SKH}(Achieve(\phi), s)$.

This is an appealing property. We could take this as a definition for **Can**, but we find our earlier definition simpler and easier to work with.

One could also consider defining smart knowing how in terms of ability, that is, taking $\mathbf{SKH}(\delta, s)$ as standing for something like $\mathbf{Can}(\exists s'\, Do(\delta, s', \mathsf{now}), s)$ [7] — a smart agent knows how to execute δ if and only if it can achieve the goal of having done δ. We plan to explore this approach and determine how it relates to our current definition.

5. Planning Reconsidered

In this paper, we presented a definition of ability and two definitions of knowing how as macro abbreviations in the situation calculus, and showed that they had reasonable formal properties and generalized a number of other accounts. These definitions, we claimed, were a necessary first step to any theory of planning in a context involving incomplete knowledge of the initial state, knowledge-producing actions, and actions with context-dependent effects.

What our account does not provide, however, is a theory of planning itself. What exactly is a plan? If we simply say that it is any

[7] This isn't quite right because it does not require the execution of δ to start in situation s; but this can be fixed.

program that achieves a goal and that the agent knows how to execute, then for smart agents, the planning problem is absolutely trivial: when $\neg\mathbf{Can}(\phi, s)$, there can be no plan for ϕ; but when $\mathbf{Can}(\phi, s)$, the agent also knows how to execute the universal program defined above: $\mathbf{SKH}(\mathit{Achieve}(\phi), s)$.

For dumber agents, however, the case is not so clear. Even if $\mathbf{Can}(\phi, s)$, must there exist a program δ that will bring about ϕ and such that $\mathbf{DKH}(\delta, s)$ holds? What would be ideal in this case would be a way of synthesizing a suitable program from a proof of $\mathbf{Can}(\phi, s)$, that is, from a proof of $\exists\sigma\, \mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), s)$. This is can be thought of as a generalization of planning by deduction and answer extraction [6] that would somehow convert an action selection function into a program of the appropriate sort.

We can also imagine a variety of types of programs for agents of varying power. For a very dumb agent, we might require that all tests in all if-then-elses and while-loops in the program consist of comparisons among known sensor values. This would decouple the agent from any background theory of the world. Another alternative might be to allow tests that refer to values of fluents, and assume that the agent can use successor state axioms at run time. Yet another possibility is to allow tests and actions that incorporate limited versions of planning. For instance, we might let the agent decide at run time whether or not it needs to perform a knowledge-producing action before executing a test. There is clearly a tradeoff here: the more we assume of our agent at execution time (with whatever effects on performance this might have), the less work will be necessary at planning time.

An answer to the question of what it means to solve the planning problem for a dumb executor that can perform sensing at run time is provided in [11]. The account assumes that the executor is very dumb and neither performs reasoning about fluents, nor memorizes sensor values. A plan language suitable for such an executor is presented. The language only allows branching based on the result of an immediately preceding sensing action, and has the property that agents always know how to execute any plan that can be expressed. In [13], this language is shown to be universal in that any effectively achievable goal can be achieved by getting an agent to execute a program in this language.

It would be interesting to develop accounts of planning for executors at different points in the dumb-smart spectrum. Another interesting area for future research is group ability, i.e., when can a group of agents (that can perform sensing) jointly achieve a goal.

Appendix

A. Proof of Proposition 5

The proof uses three lemmas. Remember that for simplicity here, we are assuming that primitive actions are always physically possible. First, we show that whenever a goal is known to hold already, it can be achieved:

LEMMA 14. $\mathbf{Know}(\phi, s) \supset \mathbf{Can}(\phi, s)$.

Proof. Take arbitrary σ and s' such that $K(s', s)$. Since K is transitive and we are given that $\mathbf{Know}(\phi, s)$, we have that $\mathbf{Know}(\phi, s')$. Thus, $\mathbf{CanGet}(\phi, \sigma, s')$ and $\mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), s)$.

Then, we show that if there is an action such that the agent knows that he can achieve his goal after the action is performed, then he can achieve the goal from the beginning:

LEMMA 15. $\exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s) \supset \mathbf{Can}(\phi, s)$.

Proof. Suppose that there is an action a such that $\mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s)$. This means that

$$\forall s'(K(s', s) \supset \mathbf{Can}(\phi, do(a, s'))),$$

and thus that

$$\forall s'(K(s', s) \supset \exists \sigma_{s'}\, \mathbf{Know}(\mathbf{CanGet}(\phi, \sigma_{s'}, \mathsf{now}), do(a, s'))) \quad (*)$$

i.e., for every K-accessible situation s', there is a action selection function $\sigma_{s'}$ that the agent knows will get him to the goal. We will show that $\mathbf{Can}(\phi, s)$, by constructing a single action selection function that works for every K-accessible situation.

First, notice that we can partition the accessible situations into equivalence classes according to whether they remain mutually accessible after the performance of action a. Given s_1 and s_2 such that $K(s_1, s)$ and $K(s_2, s)$, let $s_1 \approx s_2$ iff $K(do(a, s_2), do(a, s_1))$. It is easy to show that $\approx$ must be an equivalence relation given the successor state axiom for K and the requirement that K be transitive and euclidean. We must select a single action selection function for all situations in a given equivalence class in order to construct a global action selection function for which the agent can get to the goal. Let f be some arbitrary function that maps an equivalence class into the action selection function associated with one of its member, i.e., such that $f([s_1]) = \sigma_{s_2}$ where $s_1 \approx s_2$. We claim that $\forall s'(K(s', s) \supset \mathbf{CanGet}(\phi, f([s']), do(a, s')))$,

i.e., in every accessible situation the agent can get to the goal by following the action selection function selected by f after doing a. To see this, suppose that $f([s']) = \sigma_{s^*}$; then $K(s^*, s)$ and $K(do(a, s'), do(a, s^*))$; so by $(*)$ $\mathbf{CanGet}(\phi, \sigma_{s^*}, do(a, s'))$.

Now let us define a global action selection function as follows:

$$\sigma_g(s^*) = \begin{cases} f([s'])(s^*) & \text{if } \exists s'(K(s', s) \wedge s' < s^*) \\ a & \text{otherwise} \end{cases}$$

It follows that $\forall s'(K(s', s) \supset \mathbf{CanGet}(\phi, \sigma_g, do(a, s')))$. Since $\forall s'(K(s', s) \supset \sigma_g(s') = a)$, we must also have that $\forall s'(K(s', s) \supset \mathbf{CanGet}(\phi, \sigma_g, s'))$, and thus that $\mathbf{Can}(\phi, s)$.

Finally, we show the converse of the above two results:

LEMMA 16.

$$\mathbf{Can}(\phi, s) \supset \mathbf{Know}(\phi, s) \vee \exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s)$$

Proof. We assume that $\mathbf{Can}(\phi, s)$ and $\neg\mathbf{Know}(\phi, s)$ and show that

$$\exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s).$$

From the first assumption, we have that

$$\exists \sigma\, \mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), s). \quad (*)$$

Take an arbitrary s' such that $K(s', s)$. Since $\neg\mathbf{Know}(\phi, s)$ and K is euclidean, it follows that $\neg\mathbf{Know}(\phi, s')$. Now by (*), we have that $\mathbf{CanGet}(\phi, \sigma, s')$. By the definition of $\mathbf{CanGet}$, this together with $\neg\mathbf{Know}(\phi, s')$ implies that $\mathbf{CanGet}(\phi, \sigma, do(\sigma(s'), s'))$ and $\exists a\, \mathbf{Know}(\sigma(\mathsf{now}) = a, s')$. Since $K(s', s)$ and K is transitive, we also have that $\mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, do(\sigma(\mathsf{now}), \mathsf{now})), s')$, and thus also that there is an a such that $\mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, do(a, \mathsf{now})), s')$. By the successor state axiom for K, this implies that $\mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), do(a, s'))$. Therefore $\exists \sigma\, \mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), do(a, s'))$, and $\mathbf{Can}(\phi, do(a, s'))$. Thus, $\exists a\, \mathbf{Know}(\mathbf{Can}(\phi, do(a, \mathsf{now})), s)$.

B. An Iterative Definition of Ability

Let us write $\mathbf{Can}^k_I(\phi, s)$ if the agent is able to achieve ϕ in at most k steps, for any natural number k. Inductively, we define:

$$\mathbf{Can}^k_I(\phi, s) = \begin{cases} \mathbf{Know}(\phi, s) & \text{if } k = 0 \\ \mathbf{Can}^{k-1}_I(\phi, s) \vee & \text{if } k > 0 \\ \exists a\, \mathbf{Know}(\mathbf{Can}^{k-1}_I(\phi, do(a, \mathsf{now})), s) & \end{cases}$$

An easy consequence of lemmas 14 and 15 is the following:

PROPOSITION 17.

$$\text{For any } k \geq 0,\ \mathbf{Can}_I^k(\phi, s) \supset \mathbf{Can}(\phi, s).$$

In many cases where an agent is able to achieve a goal, the agent knows that it can achieve the goal in at most k steps, i.e. $\mathbf{Can}_I^k(\phi, s)$. For instance, this applies to the treasure example in the introduction (provided that the agent can perform the sensing action). But this fails to apply for the tree example. Roughly speaking, we have that the agent knows that there is a k such that k chops are sufficient, but that there is no k such that the agent knows that k chops are sufficient. Models of the theory are such that in each situation accessible from S_0, there is fixed finite number of chops that will fell the tree, but that there are accessible situations for every natural number.

C. Proof of Proposition 13

The proof uses the following lemma:

LEMMA 18.

$$\begin{aligned}&During(Achieve(\phi), \sigma, s, s') \equiv \\ &\qquad \mathbf{OnPath}(\sigma_r, s, s') \wedge \forall s^* (s \leq s^* < s' \supset \neg \mathbf{Know}(\phi, s^*)),\end{aligned}$$

$$Do(Achieve(\phi), \sigma, s, s') \equiv During(Achieve(\phi), \sigma, s, s') \wedge \mathbf{Know}(\phi, s'),$$

where σ ranges over path selection functions.

We prove each direction of the theorem as follows:

LEMMA 19. $\mathbf{Can}(\phi, s) \supset \mathbf{SKH}(Achieve(\phi), s)$

Proof. Suppose that the antecedent holds, i.e., that there exists an action selection function σ such that $\mathbf{Know}(\mathbf{CanGet}(\phi, \sigma, \mathsf{now}), s)$. Take an arbitrary s_s such that $K(s_s, s)$. By the assumption and the definition of **CanGet**, we have that

$$\begin{aligned}&\exists s_e (\mathbf{OnPath}(\sigma, s_s, s_e) \wedge \mathbf{Know}(\phi, s_e) \wedge \\ &\qquad \forall s_i [s_s \leq s_i < s_e \supset \exists a\, \mathbf{Know}(\sigma(\mathsf{now}) = a, s_i))]).\end{aligned}$$

Since situations are well founded, we must also have

$$\begin{aligned}&\exists s_e (\mathbf{OnPath}(\sigma, s_s, s_e) \wedge \mathbf{Know}(\phi, s_e) \wedge \\ &\qquad \forall s_i [s_s \leq s_i < s_e \supset \neg \mathbf{Know}(\phi, s_i) \wedge \exists a\, \mathbf{Know}(\sigma(\mathsf{now}) = a, s_i))]).\end{aligned}$$

Let σ' be an arbitrary path selection function such that $\sigma'_r = \sigma$. By the above and lemma 18, we have that $\exists s_e\, Do(Achieve(\phi), \sigma', s_s, s_e)$. Take arbitrary s_i, s'_s and s'_i such that $s_s \leq s_i \leq s_e$, $K(s'_s, s_s)$, $K(s'_i, s_i)$, and $s'_s \leq s'_i$. By the above and the fact that K is transitive, it follows that if $s_i = s_e$ then $\mathbf{Know}(\phi, s'_i)$. By the above and the fact that K is euclidean, it follows that if $s_i \neq s_e$ then $\neg\mathbf{Know}(\phi, s'_i)$. As well, by the above, we must have $\mathbf{OnPath}(\sigma, s'_s, s'_i)$. Thus by lemma 18, we must have that $During(Achieve(\phi), \sigma', s'_s, s'_i)$, and $Do(Achieve(\phi), \sigma', s'_s, s'_i)$ for $s_i = s_e$. Therefore, $\mathbf{CanExec}(Achieve(\phi), \sigma', s_s)$.

LEMMA 20. $\mathbf{SKH}(Achieve(\phi), s) \supset \mathbf{Can}(\phi, s)$

Proof. Suppose that the antecedent holds, i.e., that there exists a path selection function σ such that $\mathbf{Know}(\mathbf{CanExec}(Achieve(\phi), \sigma, \mathsf{now}), s)$. Take an arbitrary s_s such that $K(s_s, s)$. The assumption implies that $\exists s_e\, Do(Achieve(\phi), \sigma, s_s, s_e)$. Thus by lemma 18, we have that

$$\exists s_e(\mathbf{OnPath}(\sigma, s_s, s_e) \wedge \mathbf{Know}(\phi, s_e) \wedge \forall s_i[s_s \leq s_i < s_e \supset \neg\mathbf{Know}(\phi, s_i)]).$$

Take an arbitrary s_i such that $s_s \leq s_i < s_e$. Clearly, it must be the case that $\neg Do(Achieve(\phi), \sigma, s_s, s_i)$. Since $K(s_s, s)$, by transitivity of K and the successor state axiom for K, we must also have that $K(s_s, s_s) \wedge K(s_i, s_i)$. This implies that

$$\neg\forall s'_s, s'_i[K(s'_s, s_s) \wedge K(s'_i, s_i) \wedge s'_s \leq s'_i \supset Do(Achieve(\phi), \sigma, s'_s, s'_i)].$$

Thus, by the assumption and the definition of **CanExec**, we have that

$$\exists a \forall s'_s, s'_i[K(s'_s, s_s) \wedge K(s'_i, s_i) \wedge s'_s \leq s'_i \supset During(Achieve(\phi), \sigma, s'_s, do(a, s'_i))].$$

Clearly $\sigma_r(s_i) = a$; so we have $\exists a\mathbf{Know}(\sigma_r(\mathsf{now}) = a, s_i)$. Therefore, we have that $\mathbf{Know}(\mathbf{CanGet}(\phi, \sigma_r, \mathsf{now}), s)$.

Acknowledgements

This research received financial support from Communications and Information Technology Ontario (and its earlier incarnation ITRC), the Natural Science and Engineering Research Council of Canada, and the Institute for Robotics and Intelligent Systems (Canada).

References

1. Agre, P. E. and D. Chapman: 1990, 'What Are Plans for?'. *Robotics and Autonomous Systems* **6**, 17–34.

2. Davis, E.: 1994, 'Knowledge Preconditions for Plans'. *Journal of Logic and Computation* **4**(5), 721–766.
3. Etzioni, O., S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson: 1992, 'An Approach to Planning with Incomplete Information'. In: B. Nebel, C. Rich, and W. Swartout (eds.): *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*. Cambridge, MA, pp. 115–125.
4. Fikes, R. and N. Nilsson: 1971, 'STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving'. *Artificial Intelligence* **2**, 189–208.
5. Golden, K. and D. Weld: 1996, 'Representing Sensing Actions: The Middle Ground Revisited'. In: L. C. Aiello, J. Doyle, and S. C. Shapiro (eds.): *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference*. Cambridge, MA, pp. 174–185.
6. Green, C.: 1969, 'Theorem Proving by Resolution as a Basis for Question-Answering Systems'. In: B. Meltzer and D. Michie (eds.): *Machine Intelligence*, Vol. 4. New York: American Elsevier, pp. 183–205.
7. Haas, A. R.: 1987, 'The Case for Domain-Specific Frame Axioms'. In: F. Brown (ed.): *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*. Lawrence, KA, pp. 343–348.
8. Krebsbach, K., D. Olawsky, and M. Gini: 1992, 'An Empirical Study of Sensing and Defaulting in Planning'. In: *Proceedings of the First Conference on AI Planning Systems*. San Mateo, CA, pp. 136–144.
9. Kripke, S. A.: 1963, 'Semantical Considerations on Modal Logic'. *Acta Philosophica Fennica* **16**, 83–94.
10. Lakemeyer, G. and H. J. Levesque: 1998, 'AOL: A Logic of Acting, Sensing, Knowing, and Only-Knowing'. In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR-98)*. pp. 316–327.
11. Levesque, H. J.: 1996, 'What is Planning in the Presence of Sensing?'. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*. Portland, OR, pp. 1139–1146.
12. Levesque, H. J., R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl: 1997, 'GOLOG: A Logic Programming Language for Dynamic Domains'. *Journal of Logic Programming* **31**(59–84).
13. Lin, F. and H. J. Levesque: 1998, 'What Robots Can Do: Robot Programs and Effective Achievability'. *Artificial Intelligence* **101**(1–2), 201–226.
14. Lin, F. and R. Reiter: 1994, 'State Constraints Revisited'. *Journal of Logic and Computation* **4**(5), 655–678.
15. McCarthy, J. and P. Hayes: 1979, 'Some Philosophical Problems from the Standpoint of Artificial Intelligence'. In: B. Meltzer and D. Michie (eds.): *Machine Intelligence*, Vol. 4. Edinburgh, UK: Edinburgh University Press, pp. 463–502.
16. Moore, R. C.: 1985, 'A Formal Theory of Knowledge and Action'. In: J. R. Hobbs and R. C. Moore (eds.): *Formal Theories of the Common Sense World*. Norwood, NJ: Ablex Publishing, pp. 319–358.
17. Morgenstern, L.: 1987, 'Knowledge Preconditions for Actions and Plans'. In: *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*. Milan, Italy, pp. 867–874.
18. Pednault, E. P. D.: 1989, 'ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus'. In: R. Brachman, H. Levesque, and R.

Reiter (eds.): *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*. Toronto, ON, pp. 324–332.
19. Peot, M. and D. Smith: 1992, 'Conditional Nonlinear Planning'. In: *Proceedings of the First Conference on AI Planning Systems*. San Mateo, CA, pp. 189–197.
20. Reiter, R.: 1991, 'The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression'. In: V. Lifschitz (ed.): *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. San Diego, CA: Academic Press, pp. 359–380.
21. Scherl, R. B. and H. J. Levesque: 1993, 'The Frame Problem and Knowledge-Producing Actions'. In: *Proceedings of the Eleventh National Conference on Artificial Intelligence*. Washington, DC, pp. 689–695.
22. Schoppers, M. J.: 1992, 'Building Plans to Monitor and Exploit Open-Loop and Closed-Loop Dynamics'. In: *Proceedings of the First Conference on AI Planning Systems*. San Mateo, CA, pp. 204–213.
23. Schubert, L.: 1990, 'Monotonic Solution to the Frame Problem in the Situation Calculus: An Efficient Method for Worlds with Fully Specified Actions'. In: H. Kyberg, R. Loui, and G. Carlson (eds.): *Knowledge Representation and Defeasible Reasoning*. Boston, MA: Kluwer Academic Press, pp. 23–67.
24. Singh, M. P.: 1994, *Multiagent Systems*. Berlin: LNAI 799, Springer-Verlag.
25. van der Hoek, W., B. van Linder, and J.-J. C. Meyer: 1994, 'A Logic of Capabilities'. In: A. Nerode and Y. V. Matiyasevich (eds.): *Proceedings of the Third International Symposium on the Logical Foundations of Computer Science (LFCS'94)*.