5

An Ontological Perspective on Interactive Task Learning

Charles Rich

Abstract

Learning and teaching are best viewed as a collaborative interaction. As participants, both the teacher and learner share the goal of increasing the learner's abilities. Yet what does it mean to *know* how to do something? This chapter analyzes the abstract form, nature, and organization of task knowledge and illustrates these concepts using a shared task of tire rotation. It applies a hierarchical decomposition of knowledge for interactive task learning that involves three levels: domain knowledge, procedural knowledge, and metaknowledge. In addition, the traditional distinction between symbolic versus nonsymbolic task knowledge is noted. Representative examples are given, and open questions and unresolved problems are highlighted as suggested directions for future inquiry.

Introduction

What does it mean to know how to do something? This question concerns the ontology of task knowledge (i.e., the abstract form, nature, and organization of this knowledge) and answering it is a logical prerequisite to understanding how to learn or teach such knowledge. Although our ultimate goal is to identify the frontiers of the current state of the art, I have organized this discussion around the main ideas and trends in current and past work on task knowledge. I will not undertake a comprehensive literature review and will cite only representative examples in each topic area, highlighting open questions, unsolved problems, and controversies where appropriate.

Another underlying premise of this chapter is that learning and teaching are best viewed as kinds of collaboration: the teacher and learner are two participants in a collaborative interaction in which the shared goal is to increase the learner's abilities. Furthermore, learning and teaching are often naturally interleaved with other kinds of collaboration, such as delegation and supervision. Finally, because discussions of ontology have a tendency to become very

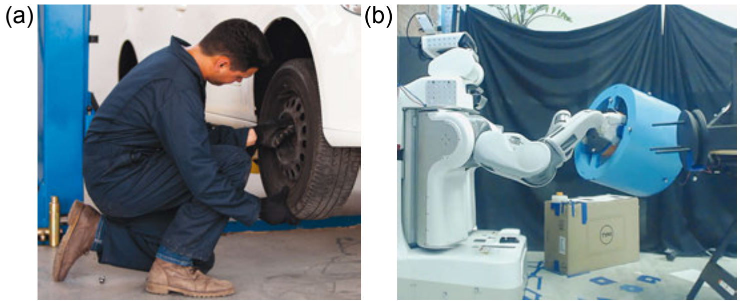

Figure 5.1 Example of tire rotation task performed by a human (a) and a state-of-the-art robot (b).

abstract, I will attempt, whenever possible, to illustrate ideas using the shared task example of tire rotation.

Rotating the tires on a car is a task commonly performed in an automotive service center (Figure 5.1a). It entails unscrewing the four or five lug nuts on each wheel, removing the wheels, and then remounting them on different hubs, according to a rotation pattern, such as back-to-front or crossover (e.g., left rear goes to right front). The overall task involves many individual steps, with repeated subsequences (such as screwing or unscrewing all the nuts on a particular hub), alternatives (the rotation patterns), and the use of a tool (lug wrench or power wrench). Recently, Mohseni-Kabir et al. (2015) taught a PR2 robot (Figure 5.1b) a greatly simplified laboratory version of this task using a combination of demonstration and instructions.

Levels and Types of Knowledge

To start, it is useful to think about the knowledge involved in interactive task learning (ITL) as stratified into three levels: domain knowledge, procedural knowledge, and metaknowledge. Discussion here is organized according to this breakdown, although there are many other ways to conceptualize and organize the knowledge necessary for task learning (see, e.g., Chai et al., Laird et al., and Wray III et al., this volume).

The most fundamental level, *domain knowledge*, provides general knowledge about the world (or parts of the world) and pervades all aspects of the tasks being learned. This knowledge can sometimes be assumed to be already shared between the teacher and student, thus providing “common ground” (see Mitchell et al., this volume). Alternatively, the concepts and relationships that are necessary to achieve common ground during task learning may need to be explicitly taught (see Shah et al., this volume). Physical manipulation tasks such as tire rotation, the example used in this chapter, require a commonsense understanding of everyday physics, such as the fact that an object will fall if you let go of it. This domain knowledge is used in both levels above it.

The middle level, *procedural knowledge*, comprises most of what is usually focused on when teaching tasks: the ordering of steps, preconditions, task hierarchy, and so on. For example, the first step in removing a wheel is to unscrew the lug nuts.

The upper level, *metaknowledge*, is knowledge about the procedural knowledge. For example, knowing how long it typically takes to rotate the tires on a car is metaknowledge about the tire rotation procedure.

In addition to these three knowledge levels, an independent categorization to keep in mind is *symbolic* versus *nonsymbolic* knowledge. (One might alternatively characterize the distinction as *discrete* versus *continuous*). The crossover tire rotation pattern is an example of symbolic procedural knowledge, whereas the motion trajectory for unscrewing a lug nut exemplifies nonsymbolic knowledge. An example of nonsymbolic domain knowledge would be the maximum weight carrying capacity of a robot. Symbolic and nonsymbolic knowledge can also be intermixed, such as in a parameterized motion trajectory with semantic constraints.

Domain Knowledge

Domain knowledge is the foundation of all task learning, other than the most limited form of mimicry. The type and extent of the required domain knowledge varies, of course, with the task (or more typically, collection of related tasks) being learned. For physical tasks in the everyday world (e.g., tire rotation or household tasks), domain knowledge begins with commonsense physics, which includes knowledge of

- gravity (how objects support other objects, fall, etc.),
- motion (how objects move in free space and slide, roll, or bounce in contact with other objects),
- materials (how solid objects bend or break in response to squeezing or stretching or other manipulations; how liquids flow, pool, and fill containers), and
- causality (how actions on one object result in changes to other objects via connections between them).

Can commonsense physics be built into robots and, if so, will that make it easier to teach them physical tasks?

As one moves into more specialized tasks, the domain knowledge becomes more specialized. For example, the domain knowledge underlying cooking includes specialized knowledge about state changes associated with specific temperatures (e.g., boiling, freezing, and burning). In addition to commonsense physics, tire rotation requires some specialized knowledge about the appropriate forces for screwing and unscrewing lug nuts. A lot of such domain knowledge is nonsymbolic.

Not all tasks are physical. Consider, for example, teaching arithmetic or teaching a virtual agent how to perform online tasks on your behalf. In these cases, the domain knowledge is more abstract and mathematical, such as the application programming interfaces (APIs) for online services.

Formal logic is a useful way to think about symbolic knowledge, even if it is not the form in which the knowledge will actually be used in practice. From this point of view, domain knowledge provides the primitive object types (and perhaps some designated object instances), predicates, and functions that are used for specifying the procedural and metaknowledge, as discussed below. Friedman et al. (1999) have taken this approach for learning probabilistic relational models.

What is the appropriate use of formal logic in research on ITL?

Procedural Knowledge

Procedural knowledge is the knowledge required to execute tasks; this does not include being able to teach, learn, or reason about the tasks. This knowledge has been studied most intensively in the artificial intelligence subfield of planning (Coles et al. 2017) and consists of the following key elements: primitive actions, ordering, conditionals, parameters, constraints, hierarchy, and motion trajectories.

Primitive Actions

The most basic element of procedural knowledge is the *primitive actions*. It is important to note that the notion of "primitive" is contextual: what is primitive in one situation may not be primitive in another. For instance, in teaching a robot how to rotate tires, unscrewing a lug nut and putting it down may be a primitive action, whereas for a different robot or teacher, there are two separate primitive actions: unscrewing and putting down.

In terms of computation, primitives are essentially symbols, although this assertion may be controversial from other points of view (see Wray III et al., this volume). However, "primitive" does not mean predefined or built in. A particular primitive may be directly associated with a precompiled motor (or cognitive) program. It may also be necessary to learn how to execute the primitive, in which case the primitive is ultimately a mixture of symbolic and nonsymbolic knowledge.

How can/does the notion of primitive actions apply to "continuous" activities such as ballroom dancing or dribbling a basketball?

Ordering

The next most basic element of procedural knowledge is *ordering*. The most minimal form of a plan is thus a sequence of primitives. Ordering knowledge also commonly takes the form of a partial order. For instance, in tire rotation, the order in which you unscrew the lug nuts does not matter, but all the unscrew actions must precede unhanging the wheel.

What are the natural kinds of parallel and overlapping execution models appropriate for interactive task learning?

Conditionals

To achieve a formally complete computational system, all that needs to be added to primitives and ordering is *conditionals*. Conditionals can take many forms and be used in different ways. Basically, however, they are all about changing which actions are executed depending on the state of the world.

The most common form of conditional is a simple if-then-else, which specifies two alternative primitives (or sequences of primitives), depending on the value of some Boolean condition. In tire rotation, you can use either a lug wrench or power tool to tighten the nuts, depending on whether the power tool is available.

Also commonly used are *preconditions*, which, when false, block the execution of a specified primitive or sequence. This is particularly useful when actions are partially ordered, because there may be something else that can be done while waiting for the precondition to become true. For example, after squirting oil on a rusty nut, the unscrew action on that nut is blocked until five minutes have passed; meanwhile, other nuts can be unscrewed.

A less commonly used form of conditional is a maintenance condition, which is a condition that is expected to hold throughout the execution of a long-running action. If the condition ever becomes false, the action is supposed to be terminated. In the tire rotation example, the pneumatic power supply is expected to be maintained to the power lug nut tool throughout execution of the screw or unscrew action.

Conditions need not make only binary choices. For example, imagine a sorting task in which the color of the object is supposed to match the color of the sorting bin, where there are more than two colors.

Conditionals are a key place where domain knowledge is used in procedural knowledge, because the domain knowledge specifies the relevant and perceivable properties of the world that are tested in the conditions. In tire rotation, it is domain knowledge that specifies that the position of the wheels on the hubs is relevant to the tire rotation task, and not the color of the tires. To frame this in formal logic terms, the domain knowledge specifies the predicates that are used in the conditions.

What are the natural kinds of uncertain and probabilistic conditional models appropriate for interactive task learning?

Parameters

Using only what has been discussed above yields a very limited form of procedural knowledge, because the actions can only be applied to specific objects in specific settings—think of programs without inputs. To achieve generality and reusability, most procedural knowledge also includes *parameters*. For example, the procedural knowledge of how to rotate the tires on a car can be applied to any car. Thus the car (and implicitly all of its parts, such as the hubs, wheels, etc.) is an input parameter of the task. Input parameters also often have restrictions on what values can be provided. These are very much like preconditions. For example, the input to the tire rotation task must be a four-wheeled vehicle.

In addition to input parameters, it is also common for procedural knowledge to include output parameters. Generally speaking, the outputs of a task are considered to be the objects whose properties are changed or that are created during the execution of the task. For example, the output of the unscrew action is the nut, because its location has changed.

How can procedural knowledge be parameterized automatically?

Constraints

Knowledge of parameters is important because it enables the specification of *constraints* between the input and output values of different actions. The simplest form of constraint is equality. For instance, if you are painting a matching set of chairs, the color input parameters of all the paint actions would be constrained to be the same.

An equality constraint between an output parameter of an action and an input parameter of a subsequent action[1] is called *data flow.* Figure 5.2 demonstrates the data flow in our tire rotation example: the arrow specifies that the output of the unscrew action (the nut) becomes the input to the putdown action. Data flow is an aspect of the causal structure of a procedure. Causal structure is important knowledge for, among other things, recovering from errors.

How is causal structure used in interactive task learning?

Constraints other than equality can also be part of procedural knowledge. For instance, in a robotics procedure, for stability, the weight of an object placed on top of another object may be constrained to be less than the weight of the first object.

Hierarchy

Hierarchy is important because humans approach complex tasks by breaking them down into subtasks. Figure 5.3 illustrates one way to break down tire rotation into five levels of hierarchy. The tree notation in this figure, called a

[1] It is logically possible, but unusual, to constrain two output parameters to be the same.

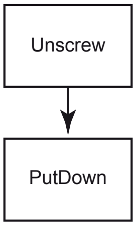

Figure 5.2 Schematic data flow constraint for the tire rotation example.

hierarchical task network, has been widely used in artificial intelligence—a version has even been formalized as an ANSI standard (see Rich 2009). The fringe of the tree specifies the sequence[2] of 64 primitive actions required to rotate the tires on a car (some repeated structure is elided in the figure to save space). Notice that there are multiple instances of the same type of primitive (e.g., Unscrew) with different parameters (i.e., different nuts).

The nonprimitive nodes in the tree are called *abstract* tasks. The top-level abstract task in this example is "Rotate tires." Other abstract tasks at intermediate levels of the hierarchy include "UnscrewHubs" (unscrewing all the lug nuts on one wheel) and "UnhangHubs" (removing the wheel from the car and putting it down). Although knowing these abstract tasks is not fundamentally necessary to execute tire rotation, it is important for two reasons: reuse and communication.

Regarding reuse, not only are there multiple occurrences of the same type of primitive in this task hierarchy, there are also multiple instances of the same type of abstract task. For example, UnscrewHub, UnhangHub, HangTire, and ScrewHub each appear four times (once for each wheel on the car). Furthermore, it is common for the same abstract tasks to be reused in other top-level tasks in the same domain. For instance, these four abstract tasks are also steps in changing a flat tire.

Abstract tasks also provide a richer communication vocabulary than just the primitives for collaborative interactions, including teaching and delegation. For example, if there is a breakdown, rather than saying "pickup nut step 49 failed," one can say "pickup nut failed while unscrewing the first wheel." Similarly, one can delegate an entire abstract task, rather than specifying the sequence of primitives (e.g., "now, please unhang all the hubs").

Conditionality in hierarchical task networks is usually conceptualized in terms of decomposition choices, called *recipes*. Notice that Figure 5.3 is a tree with two kinds of nodes. The primitive and abstract tasks discussed above are rendered as rectangles. Each abstract task is decomposed into subtasks, each of which may be abstract or primitive. The oval recipe nodes in the tree

[2] Technically, in this case, it is not a sequence but rather a partial order, since some actions (e.g., unscrewing lug nuts on a single wheel) are unordered.

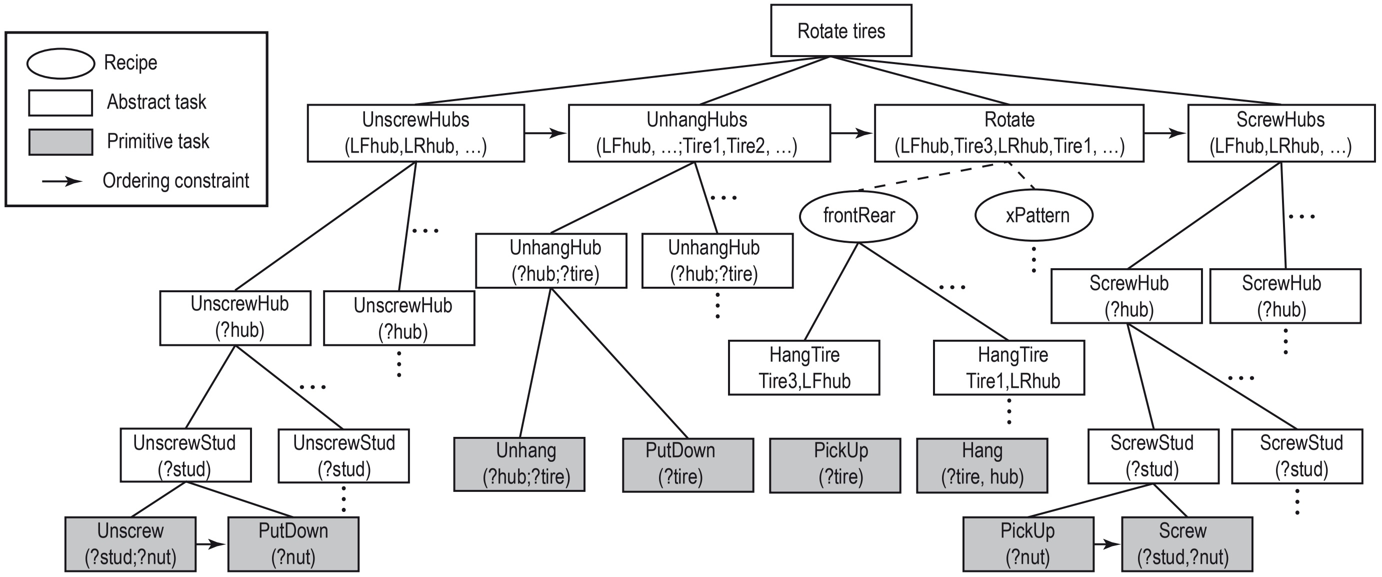

Figure 5.3 Hierarchical procedural knowledge required for tire rotation.

indicate alternative decompositions.[3] For example, in Figure 5.3 there are two alternative recipes for the rotate task, frontRear and xPattern, corresponding to the two standard patterns for rotating tires on a car: front wheels to rear wheels on the same side, or left-front to right-rear, etc. (The figure does not show the constraints that specify the details of these two patterns.)

The condition that determines whether a particular recipe can be chosen is called an *applicability condition*. Like the other conditions discussed above, applicability conditions test the state of the world and are built on domain knowledge. At execution time, if the applicability condition of more than one recipe for a given abstract task is true, then the procedural knowledge allows any of the applicable recipes to be chosen.

What is the best way of teaching recipe applicability conditions? Like abstract tasks, recipes also serve an important communication function. For example, it is efficient to simply say, "let's use the x pattern."

Finally, a cautionary note: the hierarchy in Figure 5.3 is not the only way to break down tire rotation into subtasks. Some people might insert an extra abstract task (say, RemoveHubs) and group together UnscrewHubs and UnhangHubs. Someone else might not group Unscrew and PutDown into UnscrewStud. Even the choice of primitives cannot be assumed to be universal. For some people, Pickup and Screw may be a single primitive, rather than a sequence of two primitives. This adds a lot of challenges to communication and collaboration, beyond even the problem of different people using different words for the same task concept (discussed further below).

The Policy View

In reinforcement learning, which is increasingly being applied to robotics, procedural knowledge is viewed somewhat differently than above. Instead of embedding the predicates that check the state of the world in conditional structures on sequences of actions, a single *policy* function, π, is used that maps from (all possible) states of the world, S, to the possible actions, A:

$$\pi : S \rightarrow A.$$

Torrey and Taylor (2013) have explored combining this approach with interactive teaching.

The policy view and the conditional plans view are mathematically equivalent in the sense that each can be mechanically translated into the other. However, they are very different in terms of the affordances they provide for human interaction. The policy view offers only a single monolithic function, π, to discuss, teach, correct, and so on, whereas the conditional plans view

[3] Such trees are also called *and-or* trees, where the abstract tasks are *and* nodes, because all of their children are executed, and the recipes are *or* nodes, because only one of their children is chosen.

exposes much more structure for interaction. There has also been work on learning parameterized policies and hierarchical policies.

Can reinforcement learning algorithms be applied directly to conditional plans?

Motion Trajectories

All of the kinds of procedural knowledge discussed above are essentially symbolic. However, there is clearly also a kinesthetic dimension to "knowing how to do" a task. For example, as Figure 5.4 suggests, there are only certain motion paths through space that work to unhang a tire. Unlike the action sequences and hierarchies discussed above, this knowledge is in the form of continuous functions. At its simplest, such knowledge is a single motion path through three-dimensional space (e.g., implementing a primitive action such as Unhang). More complicated versions of such knowledge may be in the form of regions in three-dimensional space that constrain the motion, or trajectories/regions in the higher-order joint configuration space of the manipulator. Sometimes it is also important to know the appropriate forces to be exerted at various points in the trajectory (e.g., how much to tighten the lug nuts).

What is the relationship between symbolic and continuous procedural knowledge, and how can they be learned together? Mohseni-Kabir et al. (2018) have recently begun to explore this question.

Metaknowledge

Metaknowledge is knowledge *about* other knowledge. Here I review some key categories of knowledge about procedural knowledge that are generally

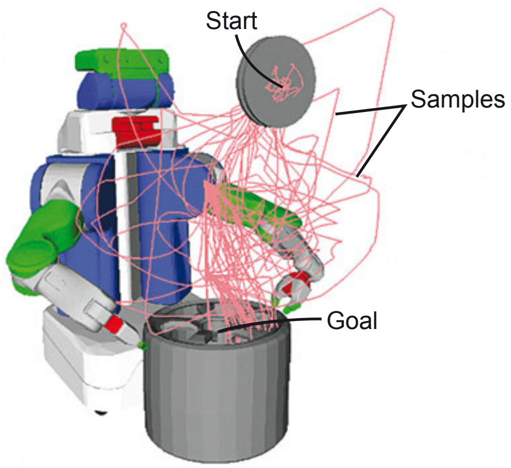

Figure 5.4 Motion trajectories for unhanging a tire.

important for learning/teaching and collaboration: capabilities/tools required, postconditions, failure recovery, duration, difficulty, uncertainty, and pedagogical knowledge.

The preceding discussion of procedural knowledge assumes that the agent performing the task is *capable* of executing all of the primitives. This, however, is not always true. In fact, a common motivation for collaboration is that one particular agent alone does not have all necessary capabilities. Knowledge of which primitives a particular agent is capable of performing (or restrictions on the parameters on the primitives) is thus an important category of metaknowledge. For instance, if a human and a robot are collaborating on tire rotation, the robot may be capable of performing all of the screw/unscrew primitives but any action that involves physically removing the tire must be done by the human, because the tire is too heavy for the robot to lift. The availability of a particular tool, such as a lug nut or power wrench for tire rotation, is a closely related kind of capability metaknowledge.

Should metaknowledge about capabilities include preferences (for different types of tasks)?

Actions do not always succeed. Knowing how to decide whether an action has succeeded or failed is a type of metaknowledge about the action: the *postcondition* of the action.[4] Both primitives and (in the case of hierarchical tasks) abstract tasks can have postconditions. Like the other conditions discussed above, postconditions test the state of the world and are built on domain knowledge. For example, the postcondition of the Pickup primitive is that the input object is in the agent's hand or manipulator. This condition could fail, for example, if the agent has not properly grasped the object before it moves its arm.

In which kinds of procedures is it most important to know postconditions?

Action failure introduces the issue of *failure recovery*. One of the differences between a novice and an expert in a particular task is that the expert knows more about how to recover from specific failures. For example, suppose Unscrew fails because the lug nut is frozen. An expert knows to squirt some penetrating oil on the nut, let it sit for five minutes, and then try again. Failure recovery knowledge is obviously of great practical importance and, in some cases, can be much larger than the basic knowledge of how to do the task when everything goes well.

What is the role of generalized failure recovery knowledge?

When planning for tasks in the future, especially in the context of teaching or delegation, it would be very useful to have estimates of the *duration* and *difficulty* of the tasks. For hierarchical tasks, one might have independent estimates for abstract tasks or derive them by combining estimates for the primitives.

[4] Unlike preconditions, which are needed to control execution (as discussed in the section on Conditionals), postconditions are not strictly needed to execute a procedure if everything goes well. This is why postconditions are being categorized as metaknowledge here.

What is a natural metric with which to estimate task difficulty?

All of the types of metaknowledge discussed above are subject to *uncertainty*. As a simple example, estimated task duration may be plus or minus 10%. More significantly, even the success of a task may be uncertain, for instance, due to sensing difficulties. To return to the Pickup example, what if the robot's hand does not incorporate a sensor to detect whether it is holding something? Here, in order to proceed intelligently with the task, it would be helpful to know the *a priori* probability of Pickup succeeding. What if the sensor is unreliable? It would then be helpful to know the probability it actually succeeded, based on the sensed information.

Is probability the best approach to expressing uncertainty for interactive task learning?

Knowing how to do something is not the same knowing how to teach it. A good teacher has extra *pedagogical* knowledge about a task, such as appropriate fading and scaffolding strategies, the mistakes students typically make, and so on (for more on pedagogy in the context of ITL, see Shah et al. and VanLehn, this volume). Like error recovery knowledge, this pedagogical knowledge can be larger than the basic knowledge of how to do the task.

How do teachers learn pedagogical knowledge?

Conclusion

In this chapter, I have reviewed the ontological aspects of task knowledge and have excluded other important issues in ITL, most notably the use of natural language. I have delineated the levels and types of knowledge needed for ITL and provided examples of each using a tire rotation example.

In hierarchical tasks, although the recipes and abstract tasks (as well as the primitives) provide an important communication "vocabulary," the reference here is to concepts, not the specific words or phrases being used. We would not expect, for instance, anyone to spontaneously say "UnhangHubs" as the second top-level step of tire rotation. Understanding what people actually say is a major research challenge. Similar natural language challenges apply for all the types of knowledge discussed here.

Finally, I have, as much as possible, skirted the issue of knowledge representation in the discussion above. You have to know what the knowledge is before you think about how to best represent it for particular computational purposes.