Learning Functional Argument Mappings for Hierarchical Tasks from Situation Specific Explanations

Gavin Suddrey, Markus Eich, Frederic Maire, Jonathan Roberts

School of Electrical Engineering and Computer Science,
Science and Engineering Faculty, Queensland University of Technology,
Gardens Point, Brisbane, QLD, 4000, Australia
{g.suddrey,f.maire}@qut.edu.au

Abstract. Hierarchical tasks learnt from situation specific explanations are typically limited in how well they generalise to situations beyond the explanation provided. To address this we present an approach to learning functional argument mappings for enabling task generalisation regardless of explanation specificity. These functional argument mappings allow subtasks within a hierarchical task to utilise both arguments provided to the parent task, as well as domain knowledge, to generalise to novel situations. We validate this approach with a number of scenarios in which the agent learns generalised tasks from situation specific explanations, and show that these tasks provide equal performance when compared to tasks learnt from generalisable explanations.

1 Introduction

In using natural language to explain complex tasks, it is intuitive to describe such tasks as collections of smaller, more manageable tasks, called subtasks. For example, the task of hosting a dinner party might include subtasks such as setting the table, and preparing the meal. Each subtasks can then itself decomposed into its own collection of subtasks. Describing tasks as collections of subtask forms the basis of *Hierarchical tasks*.

A key challenge in learning *hierarchical tasks* from natural language involves determining how parameters for any given subtask should be mapped. Prior approaches have constrained mapping these parameters to either constants [7], or parameters of the parent task [9]. The inability to exploit domain knowledge however limits the generalisability of these tasks. Other work has sought to address this limitation by introducing the ability to map parameters to functions of parameters of the parent task, using what we call *functional argument mappings* [10]. These *functional argument mappings* enable tasks to generalise to a broader range of situations by taking advantage of domain knowledge.

While *functional argument mappings* provide increased generalisability, they currently depend on the user providing generalisable explanations, containing predicate relations, to generate the correct mappings. For instance, the generalisable explanation for clearing a table would be "put away everything on the table", where the objects being put away are a function of the table. Conversely, in the situation specific explanation, "put away the three cups", it is unclear how the cups relate to the table, in which case the agent will forever link clearing the table with putting away the three cups, regardless of where they are located.

To remove the dependence on users giving generalisable task explanations, this paper presents an approach to learning *functional argument mappings* directly from situation specific explanations. Importantly, we show that our approach allows agents to learn *functional argument mappings* to enable task generalisation with only a) a small number of examples, and b) domain knowledge.

2 Related Work

Previous work has demonstrated the use of natural language as means for commanding robots to complete complex real-world tasks including both manipulation tasks [12], and navigation tasks [11]. Research has also shown that agents can make use of natural language in learning everyday tasks from sources including the World Wide Web [13], and situated interactions with users [9].

There exists many different approaches to representing tasks learnt from natural language. This includes representing tasks as sequences of behaviours, in which repeated demonstrations and conditional branching are used to provide generalisability [6, 7, 3]. Other approaches involves representing each task, not as a sequence of behaviours, but rather as a goal state for a planning problem, and generating a new plan each time the task is executed. [8].

Task hierarchies provides a means for combining tasks, creating new and more complex tasks [7, 9]. However, the degree to which these approaches are able to generalise depends on how parameters of subtasks are mapped. This may involve limiting the mapping of parameters within these hierarchies to constant values [7]. These tasks however are then restricted to acting only on the set of objects used during training. Less restrictive approaches add the ability to map parameters of subtasks to the parameters of their respective parent tasks [9]. However, any parameter that cannot be mapped to a parameter of the parent task must still be mapped to a constant, limiting overall generalisability.

More recent work has sought to address this limitation by adding the ability to explain tasks using binary relations, which allow parameters of a subtask to map to functions of parameters of the parent [10]. While this approach allows tasks to generalised to a broader variety of situations however, it depends on the user articulating any relational predicates necessary to ensuring generalisation of the task while avoiding the use of situation specific examples.

3 Hierarchical Tasks

Hierarchical tasks provide a mechanism for decomposing complex everyday tasks, such as serving dinner, into sequences of smaller, more manageable tasks, called subtasks. Formally, a task hierarchy is composed of a collection of *tasks*, *methods* and *operators*. The following definitions define *tasks* and *methods*.

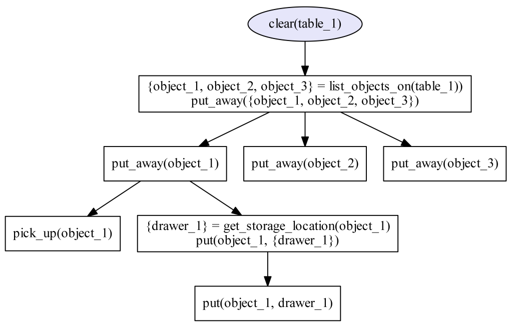
clear(table_1)
{object_1, object_2, object_3} = list_objects_on(table_1))
put_away({object_1, object_2, object_3})
put_away(object_1)
put_away(object_2)
put_away(object_3)
pick_up(object_1)
{drawer_1} = get_storage_location(object_1)
put(object_1, {drawer_1})
put(object_1, drawer_1)

Fig. 1: A decomposition tree for a simple clearing task. This task is initiated with the instruction "*clear the table*" and demonstrates iteration over a set of objects.

Definition 1. *A* ***task*** *is any activity that can be undertaken by the agent, and is comprised of a name and a parameter list [5]. A task is either* ***compound****, and can be decomposed into subtasks through the use of associated methods; or* ***primitive****, and maps to operators that enables the agent to act within its world.*

By definition, a task does not specify how it should be accomplished, rather, this information is deferred to its associated *methods* or *operators*. The following formal definition for *methods* is based on previous work in learning generalisable tasks from natural language [10].

Definition 2. *A* ***method*** *details how to implement a given compound task. We define a method as a 5-tuple, $m = (N, P, E, \Pi, \Sigma)$, where N is the name of the task T; P is the set of preconditions for m; E is the set of positive and negative effects of m; Π is a sequence of subtasks providing a partial plan for completing T; and Σ contains an ordered list of* ***argument mappings*** *for each $\pi_i \in \Pi$.*

Each $\sigma_i \in \Sigma$ corresponds to a subtask $\pi_i \in \Pi$ of T, and specifies to what value each parameter of π_i will be mapped. For any argument to π_i, its corresponding element of σ_i dictates if it will map to a *term*, where a term is a parameter or function of a parameter of T, or a *constant* identifying an object in the domain.

4 Problem Definition

In order to teach an agent a new compound task, the user provides an explanation that describes the steps in completing the task. Table 1 provides three alternative explanations for a table clearing task T.

For each general explanations, binary relations relate the table to the objects that should be put away. This allows us to derive argument mappings that maps from the parameters of the *put away* subtask to terms of T.

Table 1: Alternate explanations for a table clearing task. Among the objects on the table $O = \{o_1, ..., o_n\}$, two objects $E^+ = \{o_1, o_2\}$ do not belong on the table. Note that OG indicates the instruction is over generalised, entirely covering O.

Quality	**Instruction**
General	Put away everything on the table that does not belong on the table.
General (OG)	Put away everything on the table.
Specific	Put away o_1 and o_2.

For the specific explanation, the set of objects to be put away is provided explicitly. This has the effect that argument mappings learnt for the *put away* subtask will map to constants identifying these objects. Future invocations of T will result in the agent not clearing T as intended, but rather tracking down and putting away the objects described in the initial explanation of T.

Given a specific explanation of T, and the *put away* subtask π_0, we therefore wish to induce a function f, that for a parameter β of T, produces $f(\beta) = \Lambda$, where Λ is the set of objects provided for π_0. That is, we wish to find a *functional argument mapping* for the parameter of π_0 describing the objects to be put away.

In order to induce the function f, we reformulate our problem as an Induction Logic Programming (ILP) problem [4], where the set of positive examples E^+ describes the set Λ, while the set of negative examples E^- is initially empty.

5 Approach

The following section describes our approach to learning functional argument mappings from situation-specific explanations.

5.1 Background Knowledge

Knowledge is expressed using OWL, the Web Ontology Language, and provides a complete description of the set of concepts (unary predicates), roles (binary predicates), and individuals (objects) within the domain. To Reason about this knowledge, we use *Racer Knowledge Representation System* and the *new Racer Query Language* (nQRL) [1].

5.2 Finding Pre-Existing Roles

We now describe our induction algorithm (See Algorithm 1), that for a given argument β of the task T, and a set of objects Λ supplied for some parameter of the subtask π_i, will attempt to learn a function f such that $f(\beta) = \Lambda$.

Lines 4 to 11 of Algorithm 1 *search* the knowledge base for any role $P(x, y)$ such that $\forall \lambda \in \Lambda$, $P(\beta, \lambda)$. For each role $P(x, y)$, we take the functional form $f(x) = \{y | P(x, y)\}$. If $f(\beta) = \Lambda$, execution halts and f is returned. Otherwise, we update the set of negative examples $E^- = E^- \cup f(\beta) \backslash \Lambda$, and move to the next role. If no function f is found such that $f(\beta) = \Lambda$, we move to rule induction.

Table 2: A description of the set of roles that comprise the domain in which our agent reasons. For simplicity, we express each role in predicate logic.

Role	**Inverse**	**Example**
is_on(x,y)	has_on(y,x)	is_on(cup, table)
is_in(x,y)	has_in(y,x)	is_in(cup, cupboard)
is_located(x,y)	has_located(y,x)	is_located(table, kitchen)
is_owner(x,y)	has_owner(y,x)	is_owner(agent, gripper)
belongs(x,y)	not used	belongs(vase, table)
is_colored(x,y)	not used	is_colored(ball, red)

5.3 Rule Induction

Lines 12 to 22 of Algorithm 1 begin by inducing a rule $R(x, y)$ from the set of positive E^+ and negative E^- examples. We induce this rule using the *DL-Reasoner* [2] ILP system with the standard CELOE algorithm and a closed-world reasoner. To ensure generalisation of the induced rule, we bias the search such that predicates used in generating the rule are limited to only the concepts associated with β, as well as any concepts for the elements in Λ that encode state (e.g. Dirty). The generated rule $R(x, y)$ with the highest training accuracy, defined as the proportion of true results (both positive and negative) of the total number of examples $|E^+ \cup E^-|$ is then returned. If two or more rules have an equally high training accuracy, the shortest rule is returned.

If DL-Learner is unable to find a rule with a training accuracy of 1, or the generated rule has been induced previously, further iterations will fail to find a valid result. Execution is therefore halted, and null is returned to indicate failure.

If a rule $R(x, y)$ is generated with a training accuracy of 1, we take the functional form $f(x) = \{y|R(x, y)\}$. We then test if f meets the requirement $f(\beta) = \Lambda$. If $f(\beta) = \Lambda$, the function f is returned and execution halted. Otherwise, any new false positives are added to E^-, and the process is repeated.

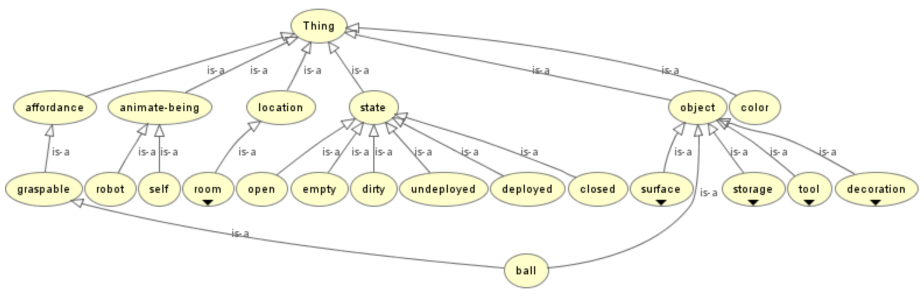

Fig. 2: A partial view of the taxonomy of concepts used during the evaluation described in Section 6. Nodes containing triangles can be expanded further.

Algorithm 1 Using the knowledge base (KB), learns a function f, that for a given argument β, will generate the set of individuals Λ

```
 1: function LEARN_FUNCTION($\beta, \Lambda$, KB)
 2:     $E^+ \leftarrow \Lambda$                        ▷ Positive examples used during induction
 3:     $E^- \leftarrow \emptyset$                      ▷ Initially empty set of negative examples
 4:     roles ← SEARCH($\beta, \Lambda$, KB)            ▷ Search KB for roles between $\beta$ and $\Lambda$
 5:     for $P \in$ roles do
 6:         $f(x) \overset{\text{def}}{=} \{y|P(x,y)\}$      ▷ define function for $P$
 7:         if $f(\beta) = \Lambda$ then
 8:             return $f$
 9:         end if
10:         $E^- \leftarrow E^- \cup f(\beta)\backslash\Lambda$   ▷ Append false positives to negative examples
11:     end for
12:     while true do
13:         accuracy, R ← INDUCE($\beta, E^+, E^-$, KB)
14:         if accuracy < 1 or R generated previously then
15:             return null                        ▷ No solution found
16:         end if
17:         $f(x) \overset{\text{def}}{=} \{y|R(x,y)\}$      ▷ define function associated with $R$
18:         if $f(\beta) = \Lambda$ then
19:             return $f$
20:         end if
21:         $E^- \leftarrow E^- \cup f(\beta)\backslash\Lambda$   ▷ Append false positives to negative examples
22:     end while
23: end function
```

6 Experiments

The following section details our evaluation using a series of task learning scenarios; and provides a comparison of tasks learnt with and without our approach.

6.1 Scenarios

Each scenarios involves learning a task from a situation specific explanation using the task learning approach described in [10], extended with Algorithm 1.

Scenario 1 - Putting Away Objects: This scenario involved learning a *put away* task T, initiated with the command "put away the knife". T contained only a single subtask π_0 explained with: "Move the knife to the kitchen drawer."

As the first argument provided for π_0, the *knife*, matched the argument provided for T, the associated parameter of π_0 was mapped directly to the parameter of T. However, as the argument provided for the second parameter of π_0 did not match any argument to T, the agent initiated a learning problem to explain this parameter, in which $\beta = knife$ and $\Lambda = \{kitchen\ drawer\}$.

Given β and Λ, the Algorithm 1 learnt the function $f(x) = \{y|belongs(x,y)\}$. We then apply $f(knife)$ which correctly generates $\{kitchen\ drawer\}$.

Table 3: Completion rates and number of primitive tasks for each planning problem.

	Put Away		Clear 1		Clear 2	
Task Type	Complete	Primitives	Complete	Primitives	Complete	Primitives
Generalised	Yes	104	Yes	128	Yes	104
Specific (Learning)	Yes	104	Yes	128	Yes	104
Specific (No Learn)	No	104	No	120	No	80

Scenario 2 - Clearing the Table I: For the second scenario the agent learnt a *clear* task T. The explanation provided for T, initiated with the command "clear the table", contained only a single subtask π_0, explained as follows: "Put away the red cup, the blue cup and the green cup."

In the explanation provided, the subtask π_0, was provided with a set of *cups* O. As O was not provided for T, the agent initiated a learning problem in which $\beta = \mathit{table}$, and $\Lambda = O$. For this scenario, O contained all objects on the table.

From these inputs, we learnt the function $f(x) = \{y|\mathit{has\_on}(x, y)\}$. Calling $f(\mathit{table})$ then correctly generates the set O. This function will however include any objects that belong on the table, if present in future.

Scenario 3 - Clearing the Table II: The third scenario involved the agent learning a *clear* task from the explanation provided previously in Scenario 2. However, in this scenario, a *vase* was added to the table prior to learning. This *vase* belongs on the *table* and should therefore not be moved.

For this scenario, O now contains the set of cups and a vase. Given $\beta = \mathit{table}$ and $\Lambda = O\setminus\{\mathit{vase}\}$, the learning algorithm generates the function $f(x) = \{y|\mathit{has\_on}(x, y) \wedge \neg \mathit{belongs}(y, x)\}$. Calling $f(\mathit{table})$ correctly generated Λ.

6.2 Quantitative Evaluation

Table 3 compares the problem solving ability of tasks learnt in Section 6.1 (Specific (Learning)) to those without the ability to learn functional mappings (Specific (No Learn)); and tasks learnt with generalised explanations (Generalised).

The generalised *put away* task was explained using the instruction "move the knife to where it belongs". The first generalised *clear* task was explained with: "put away everything on the table". The final clear task was explained with: "put away everything on the table that does not belong on the table".

Each problem required the agent to affect a number of object. This included putting away 13 items in the first problem, and clearing 5 tables in the remaining two. The number of *primitive* tasks was recorded for each problem.

7 Conclusion

In this paper we have presented for approach for learning functional argument mappings from situation specific task explanations. We show that an agent was able to learn generalisable tasks from a situation specific task explanations in a number of scenarios. We also provide a comparison of these tasks to tasks learnt from both generalisable explanations, and situation specific explanations in which the agent was unable to learn functional mappings. We show that tasks learnt using our approach were capable of equalling the performance of tasks learnt from generalisable explanations. At present our approach relies on two assumptions. First, that the agent has complete knowledge of its domain, and second, that generated functions will generalise in a way that matches the intention of the user. We plan to address these assumptions in future work.

References

1. Haarslev, V., Möller, R.: Description of the RACER System and its Applications. In: International Workshop on Description Logics. vol. 1, pp. 132–142 (2001)
2. Lehmann, J.: DL-Learner: Learning Concepts in Description Logics. Journal of Machine Learning Research 10, 2639–2642 (2009)
3. Meriçli, C., Klee, S., Paparian, J., Veloso, M.: An Interactive Approach for Situated Task Teaching through Verbal Instructions. AAAI (2013)
4. Muggleton, S.: Inductive Logic Programming. New Generation Computing 8(4), 295–318 (feb 1991)
5. Nau, D., Au, T.C., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: SHOP2: an HTN planning system. Journal of Artificial Intelligence Research 20(1), 379–404 (dec 2003)
6. Nicolescu, M.N., Mataric, M.J.: Natural Methods for Robot Task Learning: Instructive Demonstrations, Generalization and Practice. Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent systems (AAMAS) p. 241 (2003)
7. Rybski, P., Yoon, K., Stolarz, J., Veloso, M.: Interactive Robot Task Training through Dialog and Demonstration. In: 2nd ACM/IEEE International Conference on Human-Robot Interaction (HRI). pp. 49–56. IEEE (2007)
8. She, L., Yang, S., Cheng, Y., Jia, Y., Chai, J.Y., Xi, N.: Back to the Blocks World: Learning New Actions through Situated Human-Robot Dialogue. In: 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue. vol. 89 (2014)
9. Shiwali, M., Laird, J.: Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. In: Proceedings of the Twenty Eighth AAAI Conference on Artificial Intelligence. Québec (2014)
10. Suddrey, G., Lehnert, C., Eich, M., Maire, F., Roberts, J.: Teaching Robots Generalisable Hierarchical Tasks Through Natural Language Instruction. IEEE Robotics and Automation Letters pp. 1–1 (2016)
11. Talbot, B., Schulz, R., Upcroft, B., Wyeth, G.: Reasoning about natural language phrases for semantic goal driven exploration. In: Proceedings of the Australasian Conference on Robotics and Automation (2015)
12. Tellex, S., Kollar, T., Dickerson, S., Walter, M.R., Banerjee, A.G., Teller, S.J., Roy, N.: Understanding Natural Language Commands for Robotic Navigation and Mobile Manipulation. In: AAAI (ed.) Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 1507–1514. San Francisco (2011)
13. Tenorth, M., Nyga, D., Beetz, M.: Understanding and executing instructions for everyday manipulation tasks from the World Wide Web. In: 2010 IEEE International Conference on Robotics and Automation. pp. 1486–1491. IEEE (may 2010)