Planning Using Combinatory Categorial Grammars

Submission: 30

Abstract

This paper presents a new model of planning based on representing domain knowledge using Combinatorial Categorial Grammars taken from natural language processing. This enables the capturing of plans with context-free expressiveness. It uses the same representation that has previously been used for plan recognition and has been shown to be learnable. Thus it represents a solid link between planning, plan recognition, and natural language processing. The paper also compares our open source implementation to two other well known hierarchical planners.

Introduction and Motivation

This paper is motivated by two issues in AI research. First, the idea that actions in AI planning are functions from states to states is foundational to AI (Fikes and Nilsson 1971). Planning based on decomposition is almost as old (Tate 1977), and has been very successful in deployed systems. However, *methods*, used to define decompositions in hierarchical planning, are not defined as functions from states to states (Ghallab, Nau, and Traverso 2004; Dvorak et al. 2014; Bercher, Keen, and Biundo 2014; Shivashankar, Alford, and Aha 2017).

Second, while the close relationship between reasoning about action and natural language processing (NLP) is well known (Carberry 1990), the representations used for them have remained distinct making their integration ad hoc and require distinct learning algorithms. To address these two issues, this paper provides a functional formalization of planning in terms of Combinatory Categorial Grammars (CCG) (Steedman 2000), a formalism taken from NLP, and a planning algorithm that contributes:

- Formulation of hierarchical planning using function application and composition (rather than decomposition).
- A planner with context-free expressiveness (Aho and Ullman 1992) based on an NLP grammar formalism.
- Task level partial order semantics in planning based on that used in NLP rather than current task level interleaving common in AI planning.
- A total order implementation that plans directly with the lifted first order logical representation.
- Runtimes comparable to other hierarchical planners.

However, perhaps more important than the technical contributions of the planner is the linkage that it represents between planning, plan recognition, and NLP. While the relationship between formal grammars and hierarchical planning is well known, it has previously only been used to prove complexity and expressiveness results for planning (Erol, Hendler, and Nau 1994; Geib 2004; Höller et al. 2014; Behnke, Höller, and Biundo-Stephan 2015; Höller et al. 2016). In contrast, it is central to the contribution of this work that the proposed planning domain representation is a grammar that is exactly the same as that used in prior work on probabilistic *plan recognition* (Geib 2009; Geib and Goldman 2011). While the results of this prior work won't be covered here, using the same problem domain description for both planning and plan recognition (P&PR) represents a significant step in unifying these research areas.

In addition, CCGs have been used for both NLP parsing (Collins 1997; Clark and Curran 2004) and generation (White and Rajkumar 2008). Further, work on learning NLP CCGs has been very successful (Kwiatkowski et al. 2012). Perhaps, most importantly, (Geib and Kantharaju 2018) has adapted the NLP CCG grammar induction algorithms to show initial results in learning plan CCGs of the kind used in this paper. Thus, demonstrating that CCGs can be used for planning is a significant step linking P&PR with NLP parsing and generation using a learnable representation. No other representation has yet demonstrated this.

(Geib 2016), has sketched ideas similar to those presented here. This paper moves beyond that discussion in presenting a formulation of CCGs more tightly connected to prior work. Further, it presents an improved planning algorithm and discusses critical implementation details. Finally, it presents the first data showing state-of-the-art runtimes, and presents a more extensive discussion of the relation to other work.

Background Definitions

To bridge terminological differences between research in NLP and P&PR, these definitions differ slightly from those presented in CCG work on NLP and even those in previous plan recognition work (Geib 2009; Geib and Goldman 2011).

Def: 1.1 *A **state**, $s(\vec{x})$, is a first order logical formula using only conjunction and negation over a set of domain predicates, $\mathcal{P}$, where $\vec{x}$ denotes a possibly empty sequence of unbound variables used in s.*

We denote individual states with individual lower case italic letters (possibly subscripted) (e.g. $a, s_1(\vec{x})$ etc.). When necessary, we will also use lower case italic names for predicates in the planning domain (e.g. *in-hand*$(\vec{x})$) or enumerate the variables as needed (e.g. *in-hand*(x_1, x_2)). We denote the set of all states over the domain predicates $\mathcal{P}$ as $\mathcal{S}_\mathcal{P}$.

We assume agents have invokable *motor programs* that drive their effectors and may change the state of the world.

Def: 1.2 *We define a* ***motor program****,* $\mathbf{mp}(\vec{x})$*, as a parameterized function* $\mathcal{S}_\mathcal{P} \rightarrow \mathcal{S}_\mathcal{P}$ *that models the results of an agent executing one of its parameterized control programs for every state of the domain.*

We will use a vertical bar to denote applying a motor program to a state, resulting in another state, (ie. $\mathbf{mp}(\vec{x})|s_0 = s_1$).

Motor programs differ from well known planning *operators* (Fikes and Nilsson 1971), in that motor program execution is defined for every state of the domain. Operators are usually defined as a limited set of precondition and effect rules. In our implementation, motor programs are also implemented as precondition and effect rules. However, each motor program has an exclusive and exhaustive set of them. This has the effect of making traditional forward or backward chaining much more computationally expensive. It also means that motor programs are always applicable and may encode significantly more information than operators. As such they may encode knowledge about and be able to make predictions about the outcome of their execution in states outside achieving any anticipated goals.

For example a traditional operator for grasping might have preconditions that would prevent its use if the object were very hot. A motor program would encode the outcome of possibly burning oneself if such an object is grasped. This is knowledge that might be necessary in an emergency but is not relevant for most problem solving domains. Requiring an encoding for every state helps to more fully model our actual causal knowledge of the domain and prevent the intentional or unintentional encoding of biases about the uses of an agent's lowest level control programs.

Thus, motor programs capture all the causal knowledge the system has about the domain. We will use a CCG to encode information about how to build plans to achieve objectives, but first we need to define *planning domains*, *planning problems*, and *solutions*.

Def: 1.3 *A* ***planning domain****,* $\mathcal{D}$*, is a four-tuple* $\langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$ *where:*

- $\mathcal{O}$ *is a finite set of objects in the domain,*
- $\mathcal{P}$ *is a finite set of first order domain predicates,*
- $\mathcal{S}_\mathcal{P}$*, is a finite set of states defined by* $\mathcal{P}$ *and* $\mathcal{O}$*, and*
- $\mathcal{M}$ *is a finite set of motor programs defined on* $\mathcal{S}_\mathcal{P}$

Def: 1.4 *A* ***planning problem*** *is a triple* $\langle \mathcal{D}, s_0, s_G \rangle$ *where:*

- $\mathcal{D}$ *is planning domain,*
- s_0 *is an initial state in* $\mathcal{S}_\mathcal{P}$ *defined in* $\mathcal{D}$*, and*
- s_G *is a goal state in* $\mathcal{S}_\mathcal{P}$ *defined in* $\mathcal{D}$*.*

We will use $\sigma_{\vec{x}}$ to denote a set of bindings of domain objects to the variables, $\vec{x}$, and their application to a state or motor program (e.g. $s(\sigma_{\vec{x}})$ or $\mathbf{mp}(\sigma_{\vec{x}})$) to denote their application producing a *ground instance*.

Def: 1.5 *Given a domain,* $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$*, and problem,* $P = \langle \mathcal{D}, s_0, s_G \rangle$*, defined on* $\mathcal{D}$*, a* ***solution*** *or* ***plan*** *for* P *is a sequence of motor program instances from* $\mathcal{M}$*:* $[(\mathbf{mp}_1(\sigma(\vec{x_1})), ...\mathbf{mp}_n(\sigma(\vec{x_n}))]$ *, such that:*

$$\mathbf{mp}_n(\sigma(\vec{x_n})) \,|...|\, \mathbf{mp}_1(\sigma(\vec{x_1}))|s_0 = s_G.$$

Thus, a plan is just a sequence of ground motor programs that when executed in the initial state results in the goal state.

Representing Planning Knowledge in CCGs

We will assign *categories* to motor programs to capture declarative knowledge of the, possibly multiple, effects the motor program may be used to accomplish. We will denote categories in capitals and their parameters will be treated, like those in states or motor programs, in a parenthesized list or vector (e.g. A, B($\vec{x}$), H-FULL(x_1, x_2)). We will define categories recursively based on two kinds of categories: atomic and complex.

Def: 1.6 *We define an* ***atomic category****,* $\mathrm{A}(\vec{x})$*, as a parameterized function from every state in the domain to a state unique to the category.* $\mathrm{A}(\vec{x}) : \mathcal{S}_\mathcal{P} \rightarrow \{s_\mathrm{A}(\vec{x})\}$.

An atomic category defines a constant function achieving a particular state. Thus executing a motor program described by such category always achieves the defined state. For example, consider the following specification of the category H-FULL with a single parameter for a simple object-moving domain that we will use for our examples. The first line defines its associated state, and the second assigns the motor program **grasp** to it indicating that the motor program can be used to achieve it.

H-FULL(x_1) := [*in-hand*$(x_1) \wedge$!*on-table*(x_1)].
grasp$(x_1) \rightarrow$ [H-FULL(x_1)].

Thus, **grasp**(cup2), would be a function that always results in states where *in-hand*(cup2) is true and *on-table*(cup2) is not. Where informality is possible, we may use atomic categories as identifiers for the states they achieve.

Following the use of categories in natural language CCGs (Steedman 2000), we will define complex categories using two category construction operators, "/" and "\".

Def: 1.7 *Given a set of categories* $\mathcal{C}$*, where* $Z \in \mathcal{C}$ *and* $\{W, X, Y, ...\} \neq \emptyset$ *and* $\{W, X, Y, ...\} \subset \mathcal{C}$*, we define* $Z/\{W, X, Y, ...\}$ *and* $Z\backslash\{W, X, Y...\}$ *as* ***complex categories****. The category on the left of the slash is called the category* ***result*** *and the categories on the right are the* ***arguments****.*

The slash operators define new, categories that combine a set of argument categories to produce a result category. They also define the direction in which the category looks for its arguments, either before or after as determined by the slash. For example, consider extending our example:

H-FULL(x_1) := [*in-hand*$(x_1) \wedge$!*on-table*(x_1)].
H-ARND(x_1) := [*hand-around*(x_1)].
grasp$(x_1) \rightarrow$ [H-FULL$(x_1) \setminus$ { H-ARND(x_1) }].

This specifies the motor program **grasp** is a function that can be used to achieve the state associated with the atomic

category H-FULL, but to do this, immediately before its execution, another function must be executed that results in the state associated with the atomic category H-ARND. Likewise, the forward slash operator requires its argument categories occur after it to produce the state associated with its result category.

Note that the definition of complex categories does not require the use of atomic categories for arguments or results. Thus, complex categories can be built recursively. For example, consider extending our domain fragment for grasping:

release $\rightarrow$ [H-EMTY],
reach4gr(x_1) $\rightarrow$ [H-ARND(x_1)],
grasp(x_1) $\rightarrow$
[((PICK(x_1)/{H-AT-S})\{H-EMTY})\{H-ARND(x_1)}],
unreach $\rightarrow$ [H-AT-S].

Using categories, and following NLP terminology, we next define a *Plan Lexicon*.

Def: 1.8 *Given a domain, $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$, we define a **plan lexicon**, $\mathcal{L}$, as a triple $\langle \mathcal{D}, \mathcal{C}^*, \Lambda \rangle$ where,*

- $\mathcal{C}^* = \mathcal{C}_A \cup \mathcal{C}_C$,
- $\mathcal{C}_A$ *= a finite set of atomic categories for states in* $\mathcal{S}_\mathcal{P}$,
- $\mathcal{C}_C$ *= a finite set of complex categories built up recursively starting from* $\mathcal{C}_A$, *and*
- Λ *is a function that maps each motor program in* $\mathcal{M}$ *to a set of categories in* $\mathcal{C}^*$.

We will also refer to such plan lexicons as *plan grammars*. A lexicon augments a planning domain by associating with each motor program a set of categories capturing domain-specific knowledge about how it can be used to achieve specific states. To aid our discussion, we define.

Def: 1.9 *A category, R, is the **root result** of a complex category, C_c, if it is the leftmost, atomic result of C_c.*

For example, PICK is the root result of the category:

((PICK(x_1)/{H-AT-S})\{H-EMTY})\{H-ARND(x_1)}.

Def: 1.10 *A motor program, $\mathbf{mp}_i$, is a possible **anchor** of a plan for an (atomic) category, C_a, if the lexicon's Λ maps $\mathbf{mp}_i$ to at least one category whose root result is C_a.*

In our example, **grasp** is an anchor for PICK. Further, we define a *lexical planning problem*.

Def: 1.11 *Given a domain, $\mathcal{D} = \langle \mathcal{O}, \mathcal{P}, \mathcal{S}_\mathcal{P}, \mathcal{M} \rangle$, and a lexicon, $\mathcal{L}$, defined over $\mathcal{D}$ we define a **lexical planning problem** as a triple $\langle \mathcal{L}, s_0, s_G \rangle$*

- s_0 *is an initial state in $\mathcal{S}_\mathcal{P}$ defined in $\mathcal{D}$,*
- s_G *is a goal state in $\mathcal{S}_\mathcal{P}$ defined in $\mathcal{D}$.*

Note that our definition of a lexical planning problem allows for encoding domain knowledge in the lexicon using categories and the Λ function, but solutions are not defined in terms of this knowledge (See Definition 1.5). This distinguishes this work from most prior work on hierarchical planning. We will discuss this in detail later. This said, the new algorithm described in the next section, does make use of such lexically encoded knowledge.

```
1 Procedure LEX_gen(L, s_0, G) {
2   C_G ← { c ∈ C* such that G = root(c)};
3   CHOOSE c_i(x⃗) ∈ C_G such that c_i(x⃗) ∈ Λ(mp_j(x⃗));
4   CHOOSE σ_x⃗;
5   Plan ← [mp_j(σ_x⃗)]; C ← c_i;
6   WHILE (C(σ_x⃗) ≠ G) {
7     IF ((C = X\α ∪ {c_k}) {
8       Plan ← APPEND(BuildPlan(c_k, L), Plan);
9       IF (α = ∅){C ← X; } ELSE {C ← X\α; }}
10    ELSE-IF ( C = X/α ∪ {c_k}) {
11      Plan ← APPEND(Plan, BuildPlan(c_k, L));
12      IF (α = ∅){C ← X; } ELSE {C ← X/α; }}
13  }
14  IF (Plan|s_0 = G){ RETURN Plan; }}
```

Figure 1: Nondeterministic plan generation pseudocode.

Planning Using CCGs

In addition to viewing a category as declarative knowledge of a motor program's functional role in achieving a goal, we can use its structure to guide the generation of a plan to achieve its root result. Thus, CCG-based planning can be viewed as a recursive structure-following algorithm similar to those used in NLP sentence generation (White and Rajkumar 2008).

Given an atomic category that is a function to a state we wish to achieve, choose a motor program that is one of the category's anchors and add it to the plan, binding any parameters associated with the category. Then recursively build plans to achieve its argument categories, in order, appending the resulting sub-plans either to the left or right of the existing plan as determined by the category's slashes. Note, this builds plans from the anchor motor program outward.

Figure 1 gives pseudocode for this process in a procedure called LEX$_{gen}$ that takes a lexical planning problem as input and returns a plan. To make the category-directed search as clear as possible, we use nondeterministic CHOOSE operators to avoid explicit code for backtracking over category and action choice and parameter bindings. X is a variable over category structures and α is a possibly empty set of categories.

Note that each iteration of the while loop builds a plan for one of the arguments to the category. When all of the arguments in a given set have been processed (see lines 9 and 12) the associated slash is removed allowing the algorithm to access the next set of argument categories (or the root result). Note the resulting plan is tested to verify its success (line 14) before being returned; if not, the algorithm backtracks. Next we will discuss a short example including plan verification.

Consider using the lexicon fragment CCG 1 to build a plan to achieve *in-hand*(cup2). First the algorithm must find an atomic category that includes the desired state. PICK satisfies this requirement. PICK's result state and the goal state are unified to find bindings for the category's parameters resulting in PICK(cup2). Figure 3 shows how this instantiated category directs the rest of the plan search. Given a category, the system selects one of its anchors and uses it to bind the motor program's parameters. Line 2 of Figure 3

shows the selection of motor program **grasp** as the anchor for PICK. Parameter binding produces **grasp**(cup2) and a ground instance of **grasp**'s category to direct the rest of the plan search:

((PICK(cup2)/{H-AT-S})\{H-EMTY})\{H-ARND(cup2)}

The system adds the bound motor program to the plan, and then looks at the next argument of the current category, in this case H-ARND(cup2) (see line 3 of Figure 3) and repeats the process. In line 4, the system has selected an motor program that is an anchor for H-ARND and bound it, resulting in:

P = [**reach4gr**(cup2)]

Since H-ARND only requires this single motor program, planning for it is complete. In line 5, because H-ARND was a leftward argument of the category directing the search, the motor program is added to the front of the plan, resulting in:

P = [**reach4gr**(cup2), **grasp**(cup2)]

In lines 6 to 8, the same process is repeated on the H-EMTY category, giving the plan fragment:

P = [**release**, **reach4gr**(cup2), **grasp**(cup2)]

Lines 9 to 11 in the figure repeat the same process for H-AT-S, with the difference that a rightward-looking argument to the category directs the search. As a result, on line 11 the **unreach** motor program is added to the end of the current plan:

P = [**release**, **reach4gr**(cup2), **grasp**(cup2), **unreach**]

Since **grasp**'s category has no more arguments, the plan is done. Again note, each argument's sub-plan is added either to the beginning or end of the plan as dictated by the category, building the plan recursively from the middle outward.

Verifying the Plan Categories here are intended to encode functional knowledge that is *likely*, not guaranteed, to hold in all world states. Motor programs are defined in all states of the world, and are executable even where they do not achieve the states specified in their categories.

Consider the first example CCG that mapped **grasp** to H-FULL. While this is a likely outcome of grasping, the **grasp** motor program may predict that H-FULL will not result for objects that are slippery. As we have said, the causal knowledge encoded in the motor programs can be more complete than the knowledge of how to go about building plans to achieve goals encoded in the categories. As such, LEX$_{gen}$'s algorithm is not guaranteed to produce a successful plan by construction and the plan must be verified using the motor programs. If this test fails, the algorithm backtracks across chosen categories and motor programs. Thus, it uses the **mp**'s verify if the constructed plans achieve the goal.

Some might argue against building possible plans and testing them for validity afterwards. However, these objections rest on an empirical question; is this approach actually less efficient than building plans that are valid by construction? The answer to this question is dependent on the speed of the algorithm and the CCG-encoded domain knowledge.

CCG: 1

PICK(x_1) := [*in-hand*(x_1)],
H-FULL(x_1) := [*in-hand*(x_1) $\wedge$!*on-table*(x_1)],
H-EMTY := [!*in-hand*(x_1)],
...
release $\rightarrow$
[H-EMTY, (PLACE(x_1)/{H-AT-S})\{H-ABV(x_1)}],
reach4gr(x_1) $\rightarrow$ [H-ARND(x_1)],
reach4pl(x_1) $\rightarrow$ [H-ABV(x_1)],
unreach $\rightarrow$ [H-AT-S],
grasp(x_1) $\rightarrow$
[((PICK(x_1)/{H-AT-S})\{H-EMTY})\{H-ARND(x_1)}],
orient(x_1) $\rightarrow$ [FACE(x_1)],
move(x_1, x_2) $\rightarrow$
[(((MOVE-OBJ(x_1, x_2)/{PLACE(x_2)})/{FACE(x_1)})...
\{PICK(x_2)})\{FACE(x_1)}].

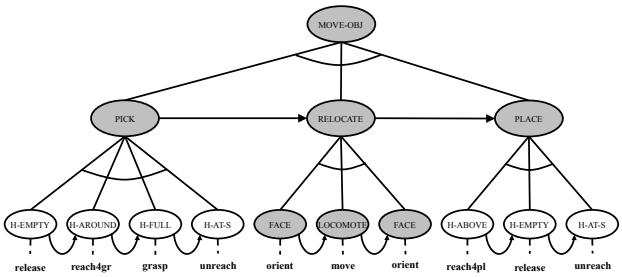

Figure 2: The hierarchical plan to move an object given in CCG 1. Shaded nodes indicate the steps covered by the the **move** motor program's complex category (sixth plan step).

Formal Properties of LEX$_{gen}$

Theorem 1.1 *(Soundness) Let PP= $\langle \mathcal{L}, s_0, s_G \rangle$ be a planning problem. If a trace of LEX$_{gen}$returns a solution π, then π is a solution to PP.*

Proof Sketch: The planner algorithm works by generating a sequence of motor program instances using a category with the goal as its root result that is within the yield of the grammar. It tests if it is a solution before returning it. (See the last line of the pseudo code.) As such, any plan returned by the algorithm must be a solution to the problem. □

Although the algorithm is sound, it is not is not complete. Since solutions are not restricted by the information encoded in Λ, the system's completeness is contingent on the completeness of the lexicon relative to the planning problem. That is, if the lexicon encodes all of the possible solutions to the problem within its yield then the algorithm is complete. However, we can imagine lexicons that simply don't have a plan within their yield to address a goal. In such cases, the algorithm would fail to produce a plan even if a plan could be assembled from the available motor programs. Thus we can claim only a qualified completeness for the algorithm.

Def: 1.12 *Given a lexical planning problem PP = $\langle \mathcal{L}, s_0, s_G, \rangle$, $\mathcal{L}$ is defined to be **complete with respect to** PP if it holds that: if π is a solution to PP, then π is in the yield of the CCG defined by $\mathcal{L}$.*

Theorem 1.2 *(Contingent Completeness) Let PP = $\langle \mathcal{L}, s_0, s_G \rangle$ be a planning problem where $\mathcal{L}$ is complete with respect to PP. If π is a solution for PP, then there is a nondeterministic trace of LEX$_{gen}$that returns π.*

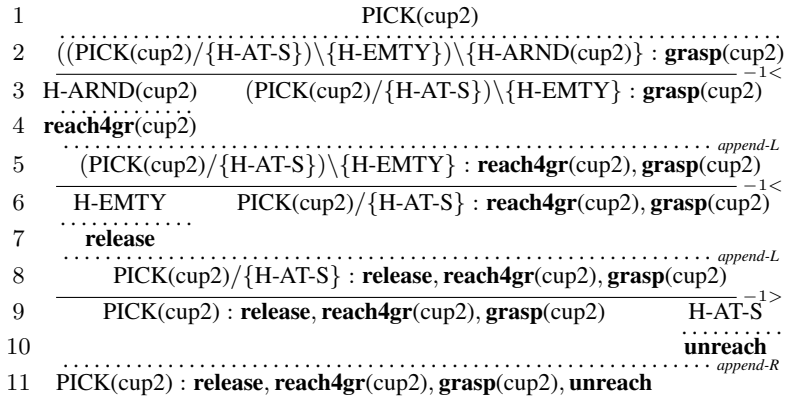

Figure 3: Building a plan to achieve the goal category PICK(cup2). Solid lines denote deconstructing a complex category (-1 and a direction indicator). Dotted lines separate two subtasks of the planning process: choosing a motor program to achieve a particular category (no annotation) and adding the chosen motor program to the plan ("append-L" or "append-R") .

Proof Sketch: The yield of any CCG is determined by the categories chosen (and hence the motor programs chosen) and the binding of its arguments to produce instances. Since LEX$_{gen}$nondeterministically searches over all such possible choice points, there is at least one trace of LEX$_{gen}$that results in each element of Λ's yield, and since Λ is complete with respect to *PP*, π must be the result one such trace. □

A short discussion of the expressiveness of a CCG plan lexicon is also worthwhile. The expressiveness of CCG grammars is well studied in NLP and is known to depend on the set of *combinators* (Curry 1977) used to combine categories for recognition. Following prior work on plan recognition using CCGs (Geib 2009) our system uses only three combinators:

$$\begin{array}{ll} \textit{right application:} & X/\alpha \cup \{Y\},\ Y \ \Rightarrow\ X/\alpha, \\ \textit{left application:} & Y,\ X\backslash\alpha \cup \{Y\} \ \Rightarrow\ X\backslash\alpha, \text{and} \\ \textit{right composition:} & X/\alpha \cup \{Y\},\ Y/\beta \ \Rightarrow\ X/\alpha \cup \beta. \end{array}$$

X and Y are categories, and α and β are possibly empty sets of categories. Intuitively they capture the named functional operations respecting the directionality of the category's slash operator. Limiting CCGs to these three operators results in context-free languages (Kuhlmann, Koller, and Satta 2015) making LEX$_{gen}$'s representation less expressive than some other planners like SHOP2 (Nau et al. 2003).

Implementation Details

A short discussion of some details of our first-order, C++ implementation of these ideas is also valuable. First, the CCG-directed search is implemented using iterative deepening allowing it to build recursive plans. Given a recursive category (e.g. $A\backslash\{A\}$) the algorithm searches for plans of increasing length preventing infinite regress or finding suboptimal plans. This represents an improvement over planners that require a fixed bound on search depth or preclude recursive hierarchical plans (Dvorak et al. 2014).

Second, we found it helpful to require LEX$_{gen}$ models be formulated so as to not require a single object be bound to two different parameters in a predicate, category, or motor program. This allows LEX$_{gen}$'s algorithm to remain complete while reducing the binding search space by eliminating from consideration bindings of motor programs such as **drive**(truck1, loc1, loc1), that would drive truck1 from a location to the same location. In cases where this is required, the model and lexicon can be extended with a specialized instance (e.g. **drive-cycle**(truck1, loc1)). Thus, this does not effect the system's expressiveness or completeness and reduced runtimes by a factor of ten. We have found no domains in which this caused an increase in the runtime of the system, however a complete theoretical investigation of this is an area for future work.

Third, the mapping that defines each **mp** is deterministic. One could imagine nondeterministic models that predict a distribution over resulting states. However, this is an area for future work. Finally, our implementation has an implicit **observe** motor program for all atomic categories that are true in the world, this prevents the building of plans to achieve already existing states. Thus given a category like $A\backslash\{B\}$, if B is already true in the current world state, the planner does not attempt to build a plan for it.

Finally, like other hierarchical planners, for goal states that are a conjunction of multiple predicates, we found it helpful to construct specialized complex categories to achieve conjunctive goals. For example, consider a typical goal state in the blocks world (Gupta and Nau 1992) that looks like *on*(block1, block2), on(block3, block4) and assuming that the category STACKED-ON(x_1, x_2) has the state *on*(x_1, x_2), we could add to the planner the complex category: $G\backslash\{\text{STACKED-ON}(b_1, b_2), \text{STACKED-ON}(b_3, b_4)\}$ to simplify the search for a plan.

Relation to Previous Work

We will focus this discussion on specific technical issues.

Methods vs. Categories: Almost all prior work on hierarchical planning, uses *methods* to define how tasks are decomposed into a series of less abstract subtasks and eventually ground in executable operators. Given a set of task names, $\mathcal{T}$, a method is often defined as a four-tuple $\langle t_h, Pre, T_n, \prec\rangle$ where $t_h \in \mathcal{T}$ is the name of the task this method expands,

Pre is a precondition defining when the method is applicable, $T_n \in \mathcal{T}$ is a set of tasks that will replace t_h in the plan, and $\prec$ is a set of ordering constraints on the tasks in T_n for the resulting plan to be a valid. The work on Hierarchical Goal Networks (HGNs) (Shivashankar et al. 2012), also use methods to describe plan knowledge, however the names in $\mathcal{T}$ refer to goal states. CCGs as formalized here are most similar to such HGNs since atomic categories represent functions to states. This said, setting aside method preconditions that will be discussed later, we can think of HTN/HGN methods as encoding a set of context-free grammar (CFG) production rules where the t_h is the left side of the rules and each of the possible orderings of T_n is the right side of a rule.

Given this, and CCG's context free expressiveness, it should not be a surprise that each assignment of a motor program to a CCG category can be rewritten as an equivalent, precondition free HTN/HGN method. The method will have exactly one executable motor program in their right hand side. The root result of the original category defines the thing to be expanded and the argument categories the expansion arrayed around the assigned motor program. For example,

$$\textbf{move}(x_1, x_2) \rightarrow$$
$$[\ (((\text{MOVE-OBJ}(x_1, x_2)/\{\text{PLACE}(x_2)\})/\{\text{FACE}(x_1)\})...$$
$$\backslash\{\text{PICK}(x_2)\})\backslash\{\text{FACE}(x_1)\}\ ].$$

(with some abuse of notation) could be seen as the precondition free HTN/HGN method, M_1,

$$\text{M}_1 =$$
$$\langle t_h = \text{MOVE-OBJ}(x_1, x_2), Pre = [], T_n = \{\text{PICK}(x_2),$$
$$\text{FACE}(x_1), \textbf{move}(x_1, x_2), \text{FACE}(x_1), \text{PLACE}(x_2)\},$$
$$\prec\ = \{(1,2),(2,3),(3,4),(4,5)\}\rangle.$$

We note, while this instance is totally ordered, CCGs can represent task level partial order plans.

Since every HTN/HGN method produced by converting CCG categories will contain an executable motor program, CCGs can be seen as an alternative syntax for a generalized version of Greibach Normal Form Grammar (GNFG) (Greibach 1965). In GNFs a single terminal begins the right hand side of every production rule. CCGs are more general in that the terminal can occur anywhere in the rules right hand side. Similar to a GNFG, each motor program category pair can be thought of as storing the results of precompiling search in the decomposition space. Each such pair defines a tree spine from a root to a leaf node. This highlights another difference between CCGs and HTN/HGN methods. A CCG category's can be thought of a slicing the plan tree vertically while HTN/HGN methods slice horizontally. Thus a category's argument categories will most often be at multiple levels of abstraction. This contrasts with HTN/HGN methods that usually capture only one level of plan decomposition and do not require terminals in their expansion.

Further, while a motor program, category pair can be uniquely converted to a HTN/HGN method and thus a CCG plan lexicon could be converted for use by an HTN/HGN planner, a given set of HTN/HGN methods is not uniquely convertible to a CCG plan lexicon. The compiled plan-space search captured in the motor program, category pairs can be done in multiple ways while still producing a complete grammar with the same yield. For example, **release** could be the anchor for MOVE rather than **move**. This would result in a very different category (with only rightward argument categories), that is equivalent to a very different HTN/HGN method. Further it would require a very different rest of the lexicon in order to keep the yield the same. (Geib 2009) has shown that the choice of which motor programs anchor which categories can have a profound effect on the efficiency of plan recognition. We believe a similar effect holds in planning resulting in the earlier pruning of plans that cannot be ground and enabling the early termination of search by producing a complete plan prefix. This is an area for future work.

Preconditions: Many, if not most, other decompositional planners support preconditions restricting a method's application. Such preconditions present theoretical difficulties since they may play at least three different roles: 1) defining causal enablement conditions on the method, 2) preventing a method's use when it is unlikely to result in a successful or desirable plan, and 3) variable binding and search control.

In fact, true causal preconditions (case 1) are very rare. Far more often preconditions are used to control search preventing the application of a method where the domain designer knows it will lead to excessive search or to bind method parameters to reduce search. While LEX_{gen} does not support search control preconditions, true causal preconditions for a CCG are easily encoded in a category by adding leftward looking argument categories.

Total vs. Partial Order: LEX_{gen} is a total order planner similar to (Shivashankar et al. 2012). That said, LEX_{gen}'s CCGs do capture task level partial ordering. That is, two argument categories can be specified as partially ordered but their plans cannot be interleaved. First one must be done and then the other. This keeps it in line with CCG use in NLP.

Task Insertion: Unlike several of the latest decompositional planners (Höller et al. 2014; Alford et al. 2016), LEX_{gen}does not support task insertion which allows these planners to add actions outside of a known decomposition method. Enabling this in $\text{LEX}_{gen}$is an area for future work.

Heuristic Search: (Shivashankar et al. 2012; Shivashankar, Alford, and Aha 2017) provide heuristics for choice points in their decompositional planners. We believe these or similar heuristics can be used to improve LEX_{gen}'s search, and will discuss this further in the context of our experimental results. However, this is still an area for future work.

Status of Domain Knowledge vs. Solutions: It should be clear by now that CCGs in this formulation represent general knowledge about how plans are to be built rather than inviolate knowledge about the causal domain relations. In this, it is more similar to planners that allow for task insertion that see methods as advice for plan construction. This is in sharp contrast to most prior work that defines the correctness of plans in terms of the methods. However, in a deep sense, to define plan correctness this way doesn't actually solve the problem, instead it pushes it into the grammar to which it is more explicitly linked in our formulation. Given a set of motor programs, it is possible that the yield of a grammar does not include plans for all reachable states in the domain. Thus it must be that either we claim that 1) some states are not acceptable goals, 2) the grammar is incomplete or 3)

force the grammar to have a yield that reaches every possible state and thereby minimize the aid it might provide.

We feel our formulation makes this issue clearer and is more accurate. We recognize there may be plans an agent has the motor programs to achieve, but hasn't yet leaned the CCG representation for its construction. To claim such a planner is complete before this learning is done seems, to us, counter intuitive. Further, this approach preserves the NLP distinction between syntax (CCG categories) and semantics (motor programs) that is critical to the alignment of these area. If a plan is defined as being correct solely because it is in the yield of the grammar it would be equivalent to saying that only syntactically correct sentences could have any meaning which daily experience refutes.

Experiments

We have compared this approach to planning to two other state of the art planners. Note that given the use of CCGs in plan recognition and NLP research, our objective in doing this is not to show that our system always performs better but that it has comparable performance to state of the art planners and therefore is attractive as an integrative framework. We have compared our implementation of LEX$_{gen}$ with the hierarchical SHOP2 planner (Nau et al. 2003) and the ICARUS system (Langley and Choi 2006). Both our code and problem domains are available at www.XXXXXXXX. We have compared the three systems using twenty two different problems that fall across four domains: blocks world (Gupta and Nau 1992), the satellites domain from the international planning competition (IPC), the logistics domain (Veloso 1992), and a robot-based kitchen domain.

We have chosen not to compared our system to non-hierarchical planners like FF (Hoffmann and Nebel 2001) for several reasons. First, it is generally agreed in the community, that with sufficient domain knowledge hierarchical planners will outperform non-hierarchical planners on large problems. This has a tendency to reduce such comparisons to knowledge and domain engineering competitions. Second, given the universal applicability of motor programs as opposed to operators, FF-style planners might be at a significant disadvantage given the breadth of their search space. Allowing the FF encoding of the domain to reduce the motor-programs back to being operators again opens the question of domain engineering and a level playing field. Third, most FF-style planners uses a propositional representation. In order to be consistent with prior work in NLP, LEX$_{gen}$ works on a first order representation opening questions about counting the cost of grounding. Fourth and finally, while CCGs are well established in the NLP community, we know of no work that suggests the use of non-hierarchical representations for either parsing or generation of language. Thus, even exceptional performance against non-hierarchical planners, while possibly of theoretical interest, would in no way strengthen our argument that this representation represents an important link between work on P&PR and NLP.

Note both SHOP2 and ICARUS use of preconditions to direct method search make them more powerful than LEX$_{gen}$. Further SHOP2 supports the use of "assert" and "retract" operators on its model of the state. This can enable SHOP2 to entirely reformulate a problem before solving even adding new predicates to the domain. In fact, this capability is used to great effect in prior work to produce runtimes that show almost no increase as problems significantly increase in size. However, it raises the question of what limits are placed on this capability. To compare like with like, none of the SHOP2 domains we tested used assert and retract.

We have encoded domains for all three systems with the same level of causal domain knowledge (converting causal preconditions to argument categories in the CCG). We have not used the assert and retract actions in SHOP2, but we have included the results of SHOP2 and ICARUS domains making use of non-causal precondition-directed method decomposition search even though LEX$_{gen}$ does not support such preconditions. For all three systems, the motor programs ("operators" in SHOP2 and "basic skills" in ICARUS) are identical, including the parameter lists, and the same logical propositions describe world states. The same plan structures were encoded in the SHOP2 methods, ICARUS complex skills, and LEX$_{gen}$ complex categories, however as we have discussed the assignment of motor programs to LEX$_{gen}$ categories effectively stores precompiled search in a way not easily replicated in the other systems. We believe this encoding aided LEX$_{gen}$'s performance. A complete study of this is an area for future work.

In Figure 4, "SHOP2–" refers to the execution time of the planner without "assert" and "retract" actions and method preconditions and SHOP2 refers to domains with method preconditions. The execution times reported under ICARUS are for domains with the basic ICARUS skills, the runtimes under ICARUS$'$ contains the same complex skills without preconditions, and runtimes reported under ICARUS$''$ are for domains with complex skills with preconditions.

The satellites domain involve plans for taking one, two and three images. We conducted two types of blocks world tests. The single goal tests involve domains having one to six blocks in the domain with a goal of having a specified block on top of another (the problems were designed such that the target blocks were at the bottom of two stacks). The multiple goal tests included three to five blocks with three to five conjunctive goals. The logistics domain problems included transporting one to three packages within the same city and across two different cities. The four problems in the robotic kitchen domain involve setting the table and mixing ingredients to make a cake. All times are real/wall clock times in seconds. Dashes (—) are runtimes over ten minutes.

Our hypothesis was that LEX$_{gen}$ would perform at about the level of SHOP2 and ICARUS without search control method preconditions, however, it easily surpassed this benchmark. Our results show that given the same domain knowledge, LEX$_{gen}$ has the best performance for ten of the twenty two problems and is beaten only by SHOP2 using search directing preconditions in an additional five domains.

As we have already stated, LEX$_{gen}$ does not support preconditions to control search. However, when there are multiple categories with a desired root result, ordering the search among these categories using a heuristic is an obvious area for future work that we have discussed in the previous section. Conditioning this order on the search will provide a similar

Domain	LEX$_{gen}$	SHOP2–	SHOP2	ICARUS	ICARUS′	ICARUS″
Satellites 1 image	0.0342	0.0365	0.0355	4.6037	10.6581	4.0650
Satellites 2 images	0.2408	0.0734	0.0362	6.0789	16.1588	6.3339
Satellites 3 images	53.7502	10.266	0.051	7.6421	25.6157	8.4330
Blocks 3 single goal	0.0003	0.0959	0.0353	6.3337	13.6586	2.8048
Blocks 4 single goal	0.0011	1.4492	0.0352	9.7077	12.1976	3.4093
Blocks 5 single goal	0.0390	—	0.0355	19.0961	36.4703	4.3555
Blocks 6 single goal	0.0717	—	0.0363	26.8817	89.5109	5.3468
Blocks 7 single goal	133.614	—	0.0364	25.7804	89.5109	6.4030
Blocks 3 multi goal	0.0098	—	0.0362	4.0956	49.4053	4.8493
Blocks 4 multi goal	0.0016	—	0.0371	21.5772	11.2691	10.1875
Blocks 5 multi goal	0.0110	—	0.039	37.0607	—	24.3125
Blocks 6 multi goal	0.0645	—	0.0382	26.9061	18.6062	5.3530
Blocks 7 multi goal	143.685	—	0.0385	25.7329	89.4028	6.4028
Logist 1 pack, 1 city	0.0003	0.0345	0.0344	2.2329	2.3463	2.5730
Logist 2 pack, 1 city	0.0017	0.0353	0.0348	4.2632	8.0660	4.2977
Logist 3 pack, 1 city	0.0008	0.0353	0.0353	11.0704	9.6281	9.910
Logist 1 pack, multi city	12.5573	0.0793	0.0371	11.9139	49.2183	48.9495
Logist 2 pack, multi city	568.6783	—	113.037	—	143.136	66.1528
Robot Table Setting 1	0.0019	0.0359	0.036	0.1504	0.1944	0.1748
Robot Table Setting 2	0.1265	0.0386	0.039	0.3932	0.3048	0.326
Robot Table Setting 3	6.4188	0.1701	0.0538	0.232	0.3576	0.3408
Robot Kitchen mixing	0.4657	0.0401	0.0359	—	6.3292	0.5552

Figure 4: Runtimes in seconds for twenty-two problems across four domains for LEX$_{gen}$, SHOP2, and ICARUS.

capability to that shown in SHOP2 and ICARUS. We anticipate conditioning this search on the state of the world and previous successful uses of the category in building plans. That said, these results clearly show LEX$_{gen}$ has comparable performance to these state of the art planners. That said, with search control knowledge provided by method preconditions, SHOP2's runtime is less affected than LEX$_{gen}$ as problem size increases. While this gives us a good reason to consider encoding heuristic category-selection search, it does not suggest an issue with scaling to larger domains.

Further exploration of where SHOP2 and ICARUS out performed LEX$_{gen}$ has revealed cases attributable specifically to specialized reasoning about the binding of objects to action parameters. For example, we might want to prevent that application of method or skill that resulted in the exploration of plans for the moving of certain specific blocks or blocks that are not of a particular shape. Capturing such restrictions in preconditions on methods is very common in hierarchical planning systems. However, to us this seems more amenable to simple typing of the action and category parameters rather than full preconditions. As a result, we are working on extending LEX$_{gen}$ with typed parameters for predicates, motor programs, and categories.

Finally, we note that ICARUS′ (ICARUS with complex skills but without preconditions) sometimes performs worse than ICARUS with just basic skills. We believe this is because ICARUS was designed to have complex skills with preconditions, and without the preconditions, the complex skill definitions add extra overhead to the search process.

Conclusions

This paper has presented a reformulation of planning and a planning algorithm, LEX$_{gen}$, in terms of CCGs, a state of the art learnable grammar formalism taken from NLP (Geib and Kantharaju 2018). This representation is also exactly the same as that used to perform plan recognition in (Geib 2009; Geib and Goldman 2011). It uses these CCG categories to direct the search for plans, organizing all planning knowledge around executable actions making it a very attractive framework for unifying reasoning about action and language.

References

Aho, A. V., and Ullman, J. D. 1992. *Foundations of Computer Science*. New York, NY: W.H. Freeman Press.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *IJCAI*, 3022–3029. IJCAI/AAAI Press.

Behnke, G.; Höller, D.; and Biundo-Stephan, S. 2015. On the complexity of htn plan verification and its implications for plan recognition. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 25–33.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS*. AAAI Press.

Carberry, S. 1990. *Plan Recognition in Natural Language Dialogue*. ACL-MIT Press Series in Natural Language Processing. MIT Press.

Clark, S., and Curran, J. 2004. Parsing the wsj using ccg and log-linear models. In *ACL '04: Proceedings of the 42th Annual Meeting of the Association for Computational Linguistics*, 104–111.

Collins, M. 1997. Three generative, lexicalised models for statistical parsing. In *ACL '97: Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*.

Curry, H. 1977. *Foundations of Mathematical Logic*. Dover Publications Inc.

Dvorak, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. Planning and acting with temporal and hierarchical decomposition models. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI*, 115–121. IEEE Computer Society.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *Proceedings of AAAI-1994*, 1123–1128.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Geib, C., and Goldman, R. 2011. Recognizing plans with loops represented in a lexicalized grammar. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, 958–963.

Geib, C. W., and Kantharaju, P. 2018. Learning combinatory categorial grammars for plan recognition. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 3007–3014. AAAI Press.

Geib, C. 2004. Assessing the complexity of plan recognition. In *Proceedings of AAAI-2004*, 507–512.

Geib, C. W. 2009. Delaying commitment in probabilistic plan recognition using combinatory categorial grammars. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1702–1707.

Geib, C. W. 2016. Lexicalized reasoning about actions. *Advances in Cognitive Systems* Volume 4:187–206.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Greibach, S. A. 1965. A new normal-form theorem for context-free phrase structure grammars. *J. ACM* 12(1):42–52.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2):223–254.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:2001.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language classification of hierarchical planning problems. In *ECAI 2014 - 21st European Conference on Artificial Intelligence*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 447–452. IOS Press.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2016. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS*, 158–165. AAAI Press.

IPC. `http://ipc.icaps-conference.org`.

Kuhlmann, M.; Koller, A.; and Satta, G. 2015. Lexicalization and generative power in CCG. *Computational Linguistics* 41(2):215–247.

Kwiatkowski, T.; Goldwater, S.; Zettlemoyer, L. S.; and Steedman, M. 2012. A probabilistic model of syntactic and semantic acquisition from child-directed utterances and their meanings. In *EACL*, 234–244.

Langley, P., and Choi, D. 2006. Learning recursive control programs from problem solving. *Journal of Machine Learning Research* 7(Mar):493–518.

Nau, D.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Shivashankar, V.; Alford, R.; and Aha, D. W. 2017. Incorporating domain-independent planning heuristics in hierarchical planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 3658–3664. AAAI Press.

Shivashankar, V.; Kuter, U.; Nau, D. S.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes)*, 981–988. IFAAMAS.

Steedman, M. 2000. *The Syntactic Process*. MIT Press.

Tate, A. 1977. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 888–893. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc.

Veloso, M. M. 1992. Learning by analogical reasoning in general problem solving. Technical report, DTIC Document.

White, M., and Rajkumar, R. 2008. A more precise analysis of punctuation for broad-coverage surface realization with ccg. In *Proceedings of the Workshop on Grammar Engineering Across Frameworks*, 17–24. Association for Computational Linguistics.