Inducing lexical entries for an incremental semantic grammar

Abstract. We introduce a method for data-driven learning of lexical entries in an inherently incremental semantic grammar formalism, Dynamic Syntax (DS). *Lexical actions* in DS are constrained procedures for the incremental projection of compositional semantic structure. Here, we show how these can be induced directly from sentences paired with their complete propositional semantic structures. Checking induced entries over an artificial dataset generated using a known grammar demonstrates that the method learns lexical entries compatible with those defined by linguists, with different versions of the DS framework induced by varying only general tree manipulation rules. This is achieved without requiring annotation at the level of individual words, via a method compatible with work on linguistic change and routinisation.

1 Introduction

Dynamic Syntax (DS) is an inherently incremental semantic grammar formalism [1, 2] in which semantic representations are projected on a word-by-word basis. It recognises no intermediate layer of syntax (see below), and generation and parsing are interchangeable. Given these properties, it seems well suited for dialogue processing, and can in principle model common dialogue phenomena such as unfinished or co-constructed utterances, mid-utterance interruption and clarification etc. [3]. However, its definition in terms of semantics (rather than the more familiar syntactic phrase structure) makes it hard to define or extend broad-coverage grammars: expert linguists are required. On the other hand, as language resources are now available which pair sentences with semantic logical forms (LFs), the ability to automatically induce DS grammars could lead to a novel and useful resource for dialogue systems. Here, we investigate methods for inducing DS grammars, and present an initial method for inducing lexical entries from data paired with complete, compositionally structured, propositional LFs.

From a language acquisition perspective, this problem can be seen as one of constraint solving for a child: given (1) the constraints imposed through time by her understanding of the meaning of linguistic expressions (from evidence gathered from e.g. her local, immediate cognitive environment, or interaction with an adult), and (2) innate cognitive constraints on how meaning representations can be manipulated, how does she go about separating out the contribution of each individual word to the overall meaning of a linguistic expression? And how does she choose among the many guesses she would have, the one that best satisfies these constraints?

This paper represents an initial investigation into the problem: the method presented is currently restricted to a sub-part of the general problem (see below). Future work will adapt it to a more general and less supervised setting where the input data contains less structure, where more words are unknown in each sentence, and applicable to real-world datasets – but the work here forms a first important step in a new problem area of learning explicitly incremental grammars in the form of constraints on semantic construction.

2 Previous work on grammar induction

Existing grammar induction methods can be divided into two major categories: supervised and unsupervised. Fully supervised methods, which use a parsed corpus as the training data and generalise over the the phrase structure rules to apply to a new set of data, has achieved significant success, particularly when coupled with statistical estimation of the probabilities for production rules that share the same LHS category (e.g. PCFGs [4]). However, such methods at best only capture part of the grammar learning problem, since they presuppose prior linguistic information and are not adequate as human grammar learning models. Unsupervised methods, on the other hand, which proceed with unannotated raw data and hence are closer to the human language acquisition setting, have seen less success. In its pure form —positive data only, without bias— unsupervised learning has been demonstrated to be computationally too complex ('unlearnable') in the worst case [5]. Successful cases have involved some prior learning or bias, e.g. a fixed set of known lexical categories, a probability distribution bias [6] or a hybrid, semi-supervised method with shallower (e.g. POS-tagging) annotation [7].

More recently another interesting line of work has emerged: supervised learning guided by *semantic* rather than syntactic annotation – more justifiably arguable to be 'available' to a human learner with some idea of what a string in an unknown language could mean. This has been successfully applied in Combinatorial Categorial Grammar [8], as it tightly couples compositional semantics with syntax [9, 10] and [11] also demonstrates a limited success of a similar approach with partially semantically annotated data that comes from a controlled experiment. Since these approaches adopt a lexicalist framework, the grammar learning involves inducing a lexicon assigning to each word its syntactic and semantic contribution.

Such approaches are only lightly supervised, using sentence-level propositional logical form rather than detailed word-level annotation. Also, grammar is learnt ground-up in an 'incremental' fashion, in the sense that the learner collects data and does the learning in parallel, sentence by sentence. Here we follow this spirit, inducing grammar from a propositional meaning representation and building a lexicon which specifies what each word contributes to the target semantics. However, taking advantage of the DS formalism, a distinctive feature of which is *word-by-word* processing of semantic interpretation, we bring an added dimension of incrementality: not only is learning sentence-by-sentence incremental, but the grammar learned is word-by-word incremental, commensurate with psycholinguistic results showing incrementality to be a fundamental feature of human parsing and production [12, 13]. Incremental parsing algorithms have correspondingly been proposed [14–16], however, to the best of our knowledge, a learning system for an explicitly incremental grammar is yet to be presented – this work is a step towards such a system.

3 Dynamic Syntax

Dynamic Syntax is a parsing-directed grammar formalism, which models the word-by-word incremental processing of linguistic input. Unlike many other formalisms, DS models the incremental building up of *interpretations* without presupposing or indeed

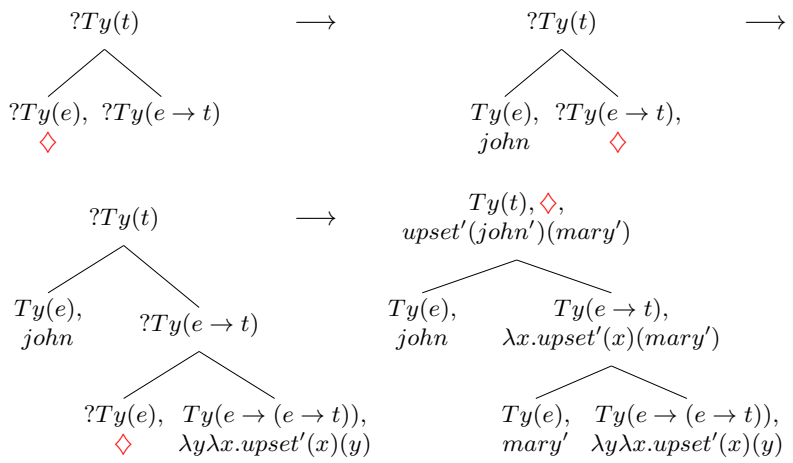


Fig. 1. Incremental parsing in DS producing semantic trees: *"John upset Mary"*

recognising an independent level of syntactic processing. Thus, the output for any given string of words is a purely *semantic* tree representing its predicate-argument structure; tree nodes correspond to terms in the lambda calculus, decorated with labels expressing their semantic type (e.g. $Ty(e)$) and formula, with beta-reduction determining the type and formula at a mother node from those at its daughters (Figure 1).

These trees can be *partial*, containing unsatisfied requirements for node labels (e.g. $?Ty(e)$ is a requirement for future development to $Ty(e)$), and contain a *pointer* $\Diamond$ labelling the node currently under development. Grammaticality is defined as parsability: the successful incremental construction of a tree with no outstanding requirements (a *complete* tree) using all information given by the words in a sentence. The input to our induction task here is therefore sentences paired with such complete, *semantic* trees, and what we try to learn are constrained lexical procedures for the incremental construction of such trees.

3.1 Actions in DS

The central tree-growth process is defined in terms of conditional *actions*: procedural specifications for monotonic tree growth. These take the form both of general structure-building principles (*computational actions*), putatively independent of any particular natural language, and of language-specific actions induced by parsing particular lexical items (*lexical actions*). The latter are what we here try to learn from data.

Computational actions These form a small, fixed set. Some merely encode the properties of the lambda calculus itself and the logical tree formalism (LoFT, [17]) – these we term *inferential* actions. Examples include THINNING (removal of satisfied requirements) and ELIMINATION (beta-reduction of daughter nodes at the mother). These actions are entirely language-general, cause no ambiguity, and add no new information to the tree; as such, they apply non-optionally whenever their preconditions are met.

Other computational actions reflect DS's predictivity and the dynamics of the framework. For example, replacing feature-passing concepts, e.g. for long-distance dependency, *ADJUNCTION introduces a single unfixed node with underspecified tree position; LINK-ADJUNCTION builds a paired ("linked") tree corresponding to semantic conjunction and licensing relative clauses, apposition and more. These actions represent possible parsing strategies and can apply optionally at any stage of a parse if their preconditions are met. While largely language-independent, some are specific to language type (e.g. INTRODUCTION-PREDICTION in the form used here applies only to SVO languages).

Lexical actions The lexicon associates words with lexical actions, which like computational actions, are each a sequence of tree-update actions in an IF..THEN..ELSE format, and composed of explicitly procedural *atomic actions* like `make`, `go`, `put` (and others). `make` creates a new daughter node. `go` moves the pointer to a daughter node, and `put` decorates the pointed node with a label. Fig. 2 shows a simple lexical action for *John*. The action says that if the pointed node (marked as ♢) has a requirement for type e, then decorate it with type e (thus satisfying the requirement); decorate it with formula $John'$ and finally decorate it with the bottom restriction $\langle\downarrow\rangle\bot$ (meaning that the node cannot have any daughters). In case the IF condition $?Ty(e)$ is not satisfied, the action aborts, meaning that the word 'John' cannot be parsed in the context of the current tree.

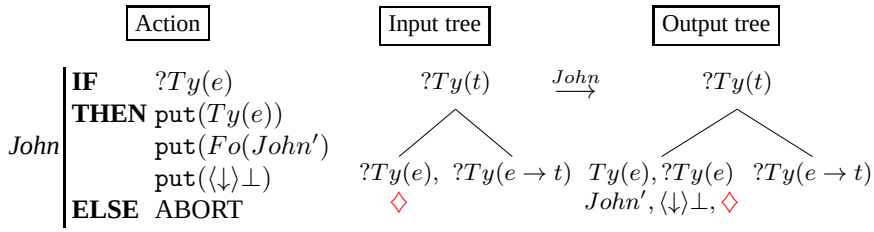
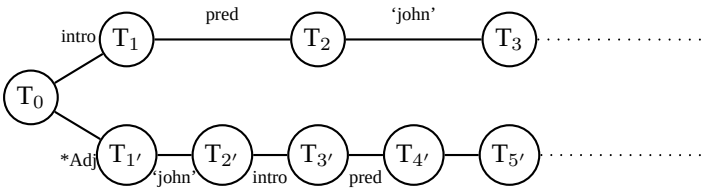


Fig. 2. Lexical action for the word 'John'

3.2 Graph Representation of DS Parsing

Given a sequence of words $(w_1, w_2, ..., w_n)$, the parser starts from the *axiom* tree T_0 (a requirement $?Ty(t)$ to construct a complete tree of propositional type), and applies the corresponding lexical actions $(a_1, a_2, \ldots, a_n)$, optionally interspersing computational actions – see Figure 1. [18] shows how this parsing process can be modelled on a *Directed Acyclic Graph* (DAG), rooted at T_0, with partial trees as nodes, and computational and lexical actions as edges (i.e. transitions between trees):



In this DAG, *intro*, *pred* and **Adj* correspond to the computational actions INTRODUCTION, PREDICTION and *-ADJUNCTION respectively; and 'john' is a lexical action. Different paths through the DAG represent different parsing strategies, which may succeed or fail depending on how the utterance is continued. Here, the path $T_0 - T_3$ will succeed if 'John' is the subject of an upcoming verb ("John upset Mary"); $T_0 - T_4$ will succeed if 'John' turns out to be a left-dislocated object ("John, Mary upset").

This DAG is taken to represent the *linguistic context* available during a parse, used for ellipsis and pronominal construal [19, 20]. It also provides us with a basis for implementing a best-first probabilistic parser, by taking the current DAG, plus a backtracking history, as the *parse state*. Given a conditional probability distribution $P(a_i|f(t))$ over possible actions a_i given a set of features of the current partial tree $f(t)$, the DAG is then incrementally constructed and traversed such that at any node (partial tree), the most likely action (edge) is traversed first, with backtracking allowing other possibilities to be explored. Estimation of this probability distribution is not a problem we address here – we assume a known probability distribution for the known grammar fragment.

4 Learning lexical actions

4.1 Assumptions and Problem Statement

Assumptions. Our task here is data-driven learning of lexical actions for unknown words. Throughout, we will assume that the (language-independent) *computational* actions are known. To make the problem tractable at this initial stage, we further make the following simplifying assumptions: (1) The supervision information is structured: i.e. our dataset pairs sentences with the DS tree that expresses their predicate-argument structure – rather than just a less structured Logical Form as in e.g. [9] (Note that this does not provide word-level supervision: nodes do not correspond to words here) (2) The training data does not contain any pronouns or ellipsis (3) We have a seed lexicon such that there are no two adjacent words whose lexical actions are unknown in any given training example. As we will see this will help determine where unknown actions begin and end. We will examine the elimination of this assumption below. Relaxing any of these assumptions means a larger hypothesis space.

Input. The input to the induction procedure to be described is now as follows:

- the set of computational actions in Dynamic Syntax, G.
- a seed lexicon L_s: a set of words with their associated lexical actions that are taken to be known in advance.
- a set of training examples of the form $\langle S_i, T_i \rangle$, where S_i is a sentence of the language and T_i – henceforth referred to as the *target tree* – is the complete semantic tree representing the compositional structure of the meaning of S_i.

Target. The output is the lexical actions associated with previously unknown words. We take these to be conditioned solely on the semantic type of the pointed node (i.e. their IF clause takes the form IF $?Ty(X)$). This is true of most lexical actions in DS (see examples above), but not all. This assumption will lead to some over-generation:

inducing actions which can parse some ungrammatical strings.The main output of the induction algorithm is therefore the THEN clauses of the unknown actions: sequences of DS atomic actions such as *go*, *make*, and *put* (see Fig. 2). We refer to these sequences as *lexical hypotheses*. We first describe our method for constructing lexical hypotheses with a single training example (a sentence-tree pair). We then discuss how to generalise over and refine these outputs in an incremental fashion as we process more training examples.

4.2 Hypothesis construction

DS is *strictly monotonic*: actions can only *extend* the tree under construction, deleting nothing except satisfied requirements. Thus, hypothesising lexical actions consists in an incremental search through the space of all monotonic extensions of the current tree T_{cur} that subsume (i.e. can be extended to) the target tree T_t. Not all possible trees and tree extensions are well-formed (meaningful) in DS, making the search constrained to a degree. The constraints are: (1) Lexical actions add lexical content —formula & type labels together— only to *leaf* nodes with the corresponding type requirement. Non-terminal nodes can thus only be given type *requirements* (later receiving their type and content via beta-reduction); (2) Leaf nodes can only be decorated by one lexical action, i.e. once a leaf node receives its semantic content, no lexical action will return to it (anaphora, excluded here, is an exception); (3) Once a new node is created, the pointer must move to it immediately and decorate it with the appropriate type requirement.

The process of hypothesis construction proceeds by locally and incrementally extending T_{cur}, using sequences of `make`, `go`, and `put` operations as appropriate and constrained as above, each time taking T_{cur} one step closer to the target tree, T_t, at each stage checking for subsumption of T_t.This means that lexical actions are not hypothesised in one go, but left-to-right, word-by-word.

Hypothesis construction for unknown words is thus interleaved with parsing known words on the same DAG: Given a single training example, $\langle(w_1, \ldots, w_n), T_t\rangle$, we begin parsing from left to right. Known words are parsed as normal; when some unknown w_i is encountered, $(w_i, \ldots, w_n)$ is scanned until the next known word w_j is found or the end of the word sequence is reached (i.e. $j = n$). We then begin hypothesising for $w_i, \ldots, w_{j-1}$, incrementally extending the tree and expanding the DAG, using both computational actions, and hypothesised lexical tree extensions until we reach a tree where we can parse w_j. This continues until the tree under development equals the target tree. All such possibilities are searched depth-first via backtracking until no backtracking is possible, resulting in a fully explored hypothesis DAG. Successful DAG paths (i.e. those that lead from the axiom tree to the target tree) thus provide the successful hypothesised lexical sub-sequences; these, once refined, become the THEN clauses of the induced lexical actions.

For unknown words, possible tree extensions are hypothesised as follows. Given the current tree under construction T_{cur} and a target tree T_t, possible sequences of atomic actions (e.g. `go`, `put`, `make`) are conditioned on the node N_t in T_t which corresponds to – has the same address as – the pointed node N_{cur} in T_{cur}. If N_t is a leaf node, we hypothesise `put` operations which add each label on N_t not present on N_{cur}, thus unifying them. Otherwise, we hypothesise adding an appropriate type requirement followed by `make`, `go` and `put` operations to add suitable daughters.

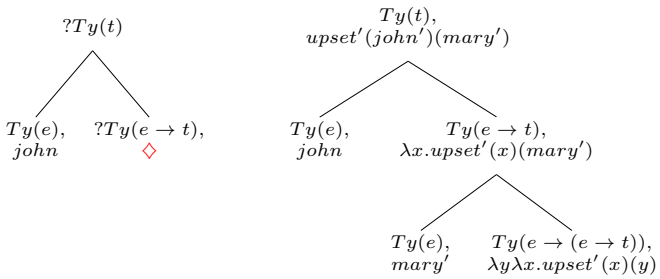


Fig. 3. The tree under development T_{cur} (left) and the target tree T_t (right)

Figure 3 shows an example. Here, T_t is the complete tree on the right, and T_{cur} the partial tree on the left. Since T_{cur}'s pointed node corresponds to a non-leaf node in T_t, we hypothesise two local action sequences: one which builds an argument daughter with appropriate type requirement ($\texttt{make}(\downarrow_0); \texttt{go}(\downarrow_0); \texttt{put}(?Ty(e))$), and another which does the same for a functor daughter ($\texttt{make}(\downarrow_1); \texttt{go}(\downarrow_1); \texttt{put}(?Ty(e \rightarrow (e \rightarrow t))$).

This method produces, for each training example $\langle S_i, T_i \rangle$, a hypothesis DAG representing all possible sequences of actions that lead from an Axiom tree to the associated target tree, T_i, using known lexical actions for known sub-strings of S_i, new hypothesised lexical actions for unknown sub-strings, and the known computational actions of DS. Such a DAG is in effect a mapping from the unknown word sub-strings of S_i into sequences of local action hypotheses plus general computational actions that may have applied between words.

This method does not in principle require us to know any of the words in a given training example in advance if we employed some method of 'splitting' sequences associated with more than one adjacent word (a tactic employed in [10] as well as [11]). We will discuss this possibility in the next section, but currently, since producing a compact lexicon requires us to generalise over these action sequence hypotheses, our method opts for the simpler alternative of assuming that in any pair of adjacent words one of them is known.

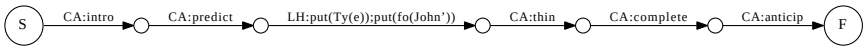
4.3 Hypothesis generalisation and refinement

Hypotheses produced from a single training example are unlikely to generalise well to other unseen examples: words occur at different syntactic/semantic positions in different training examples. We therefore require a method for the incremental, example-by-example refinement and generalisation of the action hypotheses produced for the same unknown word in processing different $\langle S, T_t \rangle$ pairs as above.

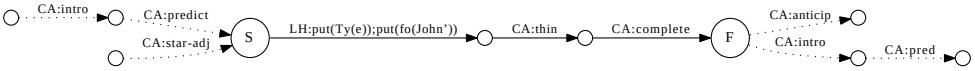
DS's general computational actions can apply at any point before or after the application of a lexical action, thus providing strategies for adjusting the syntactic context in which a word is parsed. We can exploit this property to generalise over our lexical hypotheses: by partitioning a sequence into sub-sequences which can be achieved by computational actions, and sub-sequences which must be achieved lexically. Removing the former will leave more general lexical hypotheses.

However, we need a sequence generalisation method which not only allows computational action subsequences to be removed when this aids generalisation, but also

First Training Example: ‘john’ in subject position:



Second Training Example: ‘john’ on unfixed node, i.e. left-dislocated object:



Third training example: ‘john’ before parsing relative clause ‘who. . . ’:

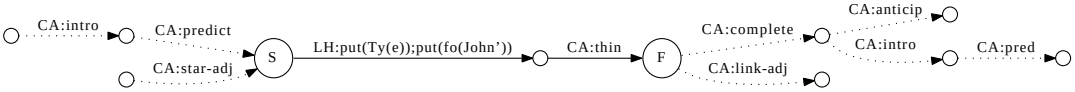


Fig. 4. Incremental intersection of candidate sequences

allows them to become lexicalised when generalisation is not required – i.e. when all observed uses of a word involve them[1]. For example, the parsing of relative clauses in current DS grammars involves the computational action LINK-ADJUNCTION building a paired tree. Parse DAGs for sentences including relatives will therefore include LINK-ADJUNCTION in all successful paths. If every observed use of the relative pronoun *who* is now associated with a sequence containing LINK-ADJUNCTION, this computational action can become part of its lexical entry, thus increasing parsing efficiency.

Generalisation through sequence intersection. The hypothesis construction process above produces a set of parse/hypothesis DAGs, D_i, from a corresponding set of training examples, $\langle S_i = (w_1, \ldots, w_n), T_i \rangle$. Each of these provides a mapping, $CS_i(w)$, from any unknown word $w \notin L_s$ in S_i (where L_s is the seed lexicon), into a set of sequences of (partial) trees, connected by *candidate sequences* of actions (see Figure 4) made up of both computational actions and the local lexical hypotheses (marked as “LH” in Figure 4). Given our first simplifying assumption from Section 4.1 above, these candidate sequences must always be bounded on either side by *known* lexical actions. As we process more training examples, the set of candidate sequences for w grows as per: $CS(w) = \bigcup_{n=1}^{i} CS_i(w)$. The problem now is one of generalisation over the candidate sequences in $CS(w)$.

Generalisation over these sequences proceeds by removing *computational* actions from the beginning or end of any sequence. We implement this via a single packed data-structure which we term the *generalisation DAG*, as shown in Figure 4: a representation of the full set of candidate sequences via their intersection (the central common path) and differences (the diverging paths at beginning and end), under the constraint that these differences consist only of computational actions. Nodes here therefore no longer

[1] see e.g. [21] for how syntactic change can be explained through a process of calcification or routinisation, whereby repeated use of certain parsing strategies leads to these strategies becoming fixed within some lexical domain.

represent single trees, but sets of trees. As new candidate sequences are added from new training examples, the intersection is reduced. Figure 4 shows this process over three training examples containing the unknown word 'john' in different syntactic positions. The 'S' and 'F' nodes here mark the start and finish of the current intersection subsequence –initially the entire sequence. As new training examples arrive, the intersection – the maximal common path – is reduced as appropriate. Lexical hypotheses thus remain as general as possible, with initial/final action sub-sequences which depend on syntactic context being delegated to computational actions, but computational actions that *always* precede or follow a sequence of lexical hypotheses will become lexicalised, as desired.

Eventually, the intersection is then taken to form the THEN clause of the new learned lexical entry. The IF clause is a type requirement, obtained from the pointed node on all partial trees in the 'S' node beginning the intersection sequence. As lexical hypotheses within the intersection are identical, and lexical hypotheses are constrained to add type information before formula information (see Section 4.2), any type information must be common across these partial trees. In Figure 4 for 'john', this is $?Ty(e)$, i.e. a requirement for type e, common to all three training examples.[2]

Lexical Ambiguity. Of course, it may well be that a new candidate sequence for w cannot be intersected with the current generalisation DAG for w (i.e. the intersection is the null sequence). Such cases indicate differences in lexical hypotheses rather than in syntactic context – either different formula/type decoration (polysemy) or different structure (multiple syntactic forms) – and thus give rise to *lexical ambiguity*, with a new generalisation DAG and lexical entry being created.

Splitting Lexical Items. Our assumption that no two adjacent words are unknown in any training example reduces the hypothesis space: candidate sequences correspond to single words and have known bounds. Relaxing this assumption can proceed in two ways: either by hypothesising candidate action sequences for multi-word sequences, and then hypothesising a set of possible word-boundary breaks (see e.g. [10, 11]); or by hypothesising a larger space of lexically distinct candidate sequences for each word. Due to the incremental nature of DS, hypotheses for one word will affect the possibilities for the next, so connections between lexical hypotheses for adjacent words must be maintained as the hypotheses are refined; we leave this issue to one side here.

5 Testing and Evaluation

We have tested a computational implementation of this method over a small, artificial data set: following [22] we use an existing grammar/lexicon to generate sentences with interpretations (complete DS trees), and test by removing lexical entries and comparing the induced results. As an initial proof of concept, we test on two unknown words: 'cook' (in both transitive and intransitive contexts) and 'John' (note that results generalise to all words of these types); the dataset consists of the following sentences paired

[2] As we remove our simplifying assumptions, IF conditions must be derived by generalising over all features of the partial trees in the start node. We do not address this here.

with their semantic trees: (1) 'John likes Mary' (2) 'John, Mary likes' (3) 'Mary likes John' (4) 'Bill cooks steak' (5) 'Pie, Mary cooks' (6) 'Bill cooks'. (1), (2) and (3) have 'John' in subject, left-dislocated object, and object positions respectively. The structurally ambiguous verb 'cooks' was chosen to test the ability of the system to distinguish between its different senses.

Original		**Induced**	
IF	$?Ty(e)$	**IF**	$?Ty(e)$
THEN	$put(Ty(e))$	**THEN**	$put(Ty(e))$
	$put(Fo(John'))$		$put(Fo(John'))$
	$put(\langle\downarrow\rangle \perp)$		$put(\langle\downarrow\rangle \perp)$
ELSE	ABORT		$delete(?Ty(e))$
		ELSE	ABORT

Fig. 5. Original and Induced lexical actions for 'John'

The original and learned lexical actions for a proper noun ('John') are shown in Figure 5. The induced version matches the original with one addition: it deletes the satisfied $?Ty(e)$ requirement, i.e. it lexicalises the inferential computational action THINNING (see Figure 4: THINNING occurs in all observed contexts, hence its lexicalisation).

Verbs provide a stronger test case. In the original conception of DS [1], the computational actions INTRODUCTION and PREDICTION (together, INTRO-PRED) were taken to build argument and functor daughters of the root $Ty(t)$ node in English, accounting for the strict SVO word order and providing a node of $Ty(e \rightarrow t)$ as trigger for the verb. However, more recent variants [23] have abandoned INTRO-PRED in favour of a more language-general LOCAL*-ADJUNCTION rule, motivated independently for Japanese and all languages with NP clustering and scrambling (see [2], chapter 6).[3] Such variants require markedly different lexical actions for verbs, triggered by $?Ty(t)$ and building a complete propositional template while merging in the other argument nodes already constructed.

We therefore test verb induction given different sets of computational actions (i.e. one with LOCAL*-ADJUNCTION, one with INTRO-PRED). Fig. 6 shows the results for a transitive verb: the induced actions match the original manually defined actions for both variants, given only this change in the general computational actions available. With INTRO-PRED, the induced action is triggered by $?Ty(e \rightarrow t)$ and does not need to build the root node's daughters; with LOCAL*-ADJUNCTION, the action builds a complete propositional template triggered by $?Ty(t)$, and merges the unfixed node introduced by LOCAL*-ADJUNCTION into its appropriate subject position.

Moreover, in the sequence intersection stage of our method, the action for the intransitive 'cook' (from training sentence (6), but not included here for reasons of space) was successfully distinguished from that of the transitive form in Fig. 6; the candidate sequences induced from sentences (4) and (5) were incompatible with those from (6), and thus resulted in a null intersection, giving rise to two separate lexical entries.

[3] This rule allows the addition of a second local *unfixed node* with its merge point restricted to any argument position. See [23, 2] for details.

Original

IF	$?Ty(e \rightarrow t)$
THEN	$make(\downarrow_1); go(\downarrow_1)$
	$put(Fo(\lambda y \lambda x.cook(x, y)))$
	$put(Ty(e \rightarrow (e \rightarrow t)))$
	$\texttt{put}(\langle\downarrow\rangle \perp)$
	$go(\uparrow); make(\downarrow_0); go(\downarrow_0)$
	$put(?Ty(e))$
ELSE	ABORT

Induced with Intro-Pred

IF	$?Ty(e \rightarrow t)$
THEN	$make(\downarrow_1); go(\downarrow_1)$
	$put(?Ty(e \rightarrow (e \rightarrow t)))$
	$put(Fo(\lambda y \lambda x.cook(x, y)))$
	$put(Ty(e \rightarrow (e \rightarrow t)))$
	$put(\langle\downarrow\rangle \perp)$
	$delete(?Ty(e \rightarrow (e \rightarrow t)))$
	$go(\uparrow); make(\downarrow_0); go(\downarrow_0)$
	$put(?Ty(e))$
ELSE	ABORT

Induced with Local*-Adj

IF	$?Ty(t)$
THEN	$make(\downarrow_0); go(\downarrow_0)$
	$put(?Ty(e))$
	$merge$
	$make(\downarrow_1); go(\downarrow_1)$
	$put(?Ty(e \rightarrow t))$
	$make(\downarrow_1); go(\downarrow_1)$
	$put(?Ty(e \rightarrow (e \rightarrow t)))$
	$put(Fo(\lambda y \lambda x.cook(x, y)))$
	$put(Ty(e \rightarrow (e \rightarrow t)))$
	$delete(?Ty(e \rightarrow (e \rightarrow t)))$
	$go(\uparrow); make(\downarrow_0); go(\downarrow_0)$
	$put(?Ty(e))$
ELSE	ABORT

Fig. 6. Original and Induced lexical actions for transitive 'cook'

6 Conclusions and Future work

In this paper we have outlined a novel method for the induction of new lexical entries in an inherently incremental and semantic grammar formalism, Dynamic Syntax, with no independent level of syntactic phrase structure. Methods developed for other non-incremental or phrase-structure-based formalisms could not be used here. Our method learns from sentences paired with semantic trees representing the sentences' predicate-argument structures: hypotheses for possible lexical action subsequences are formed under the constraints imposed by the known sentential semantics and by general facts about tree dynamics. Its success on an artificially generated dataset shows that it can learn new lexical entries compatible with those defined by linguists, with different variants of the DS framework inducible by varying only general tree manipulation rules.

Our research now focusses on relaxing our simplifying assumptions and applying to real data. Firstly, we are developing the method to remove the assumptions limiting the number of unknown words. Secondly, the induction method here is more supervised than we would like; work is under way to adapt the same method to learn from sentences paired not with trees but with less structured LFs using Type Theory with Records [24] and/or the lambda calculus, for which corpora are available. Other work planned includes integrating this method with the learning of conditional probability distributions over actions, to provide a coherent practical model of parsing and induction with incremental updates of both the lexical entries themselves and the parameters of the parsing model.

References

1. Kempson, R., Meyer-Viol, W., Gabbay, D.: Dynamic Syntax: The Flow of Language Understanding. Blackwell (2001)
2. Cann, R., Kempson, R., Marten, L.: The Dynamics of Language. Elsevier, Oxford (2005)

3. Gargett, A., Gregoromichelaki, E., Kempson, R., Purver, M., Sato, Y.: Grammar resources for modelling dialogue dynamically. Cognitive Neurodynamics **3**(4) (2009) 347–363
4. Charniak, E.: Statistical Language Learning]. MIT Press (1996)
5. Gold, E.M.: Language identification in the limit. Information and Control **10**(5) (1967) 447–474
6. Klein, D., Manning, C.D.: Natural language grammar induction with a generative constituent-context mode. Pattern Recognition **38**(9) (2005) 1407–1419
7. Pereira, F., Schabes, Y.: Inside-outside reestimation from partially bracketed corpora. In: Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics. (1992) 128–135
8. Steedman, M.: The Syntactic Process. MIT Press, Cambridge, MA (2000)
9. Zettlemoyer, L., Collins, M.: Online learning of relaxed CCG grammars for parsing to logical form. In: Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL). (2007)
10. Kwiatkowski, T., Zettlemoyer, L., Goldwater, S., Steedman, M.: Inducing probabilistic CCG grammars from logical form with higher-order unification. In: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing. (2010) 1223–1233
11. Sato, Y., Tam, W.: Underspecified types and semantic bootstrapping of common nouns and adjectives. In: Proceedings of Language Engineering and Natural Language Semantics. (2012)
12. Lombardo, V., Sturt, P.: Incremental processing and infinite local ambiguity. In: Proceedings of the 1997 Cognitive Science Conference. (1997)
13. Ferreira, F., Swets, B.: How incremental is language production? evidence from the production of utterances requiring the computation of arithmetic sums. Journal of Memory and Language **46** (2002) 57–84
14. Hale, J.: A probabilistic Earley parser as a psycholinguistic model. In: Proceedings of the 2nd Conference of the North American Chapter of the Association for Computational Linguistics. (2001)
15. Collins, M., Roark, B.: Incremental parsing with the perceptron algorithm. In: Proceedings of the 42nd Meeting of the ACL. (2004) 111–118
16. Clark, S., Curran, J.: Wide-coverage efficient statistical parsing with CCG and log-linear models. Computational Linguistics **33**(4) (2007) 493–552
17. Blackburn, P., Meyer-Viol, W.: Linguistics, logic and finite trees. Logic Journal of the Interest Group of Pure and Applied Logics **2**(1) (1994) 3–29
18. Sato, Y.: Local ambiguity, search strategies and parsing in Dynamic Syntax. In: The Dynamics of Lexical Interfaces. CSLI (2010) to appear.
19. Cann, R., Kempson, R., Purver, M.: Context and well-formedness: the dynamics of ellipsis. Research on Language and Computation **5**(3) (2007) 333–358
20. Purver, M., Eshghi, A., Hough, J.: Incremental semantic construction in a dialogue system. In: Proceedings of the 9th International Conference on Computational Semantics. (2011) 365–369
21. Bouzouita, M.: At the syntax-pragmatics interface: clitics in the history of spanish. In: Language in Flux: Dialogue Coordination, Language Variation, Change and Evolution. College Publications, London (2008) 221–264
22. Pulman, S.G., Cussens, J.: Grammar learning using inductive logic programming. Oxford University Working Papers in Linguistics **6** (2001)
23. Cann, R.: Towards an account of the english auxiliary system: building interpretations incrementally. In: Dynamics of Lexical Interfaces. Chicago: CSLI Press (2011)
24. Cooper, R.: Records and record types in semantic theory. Journal of Logic and Computation **15**(2) (2005) 99–112