Learning Expressive Contextual Grammars in Lambda Calculus

Lucas E. Morales {lucasem}

1 Introduction

Program induction can be approached as a search on a stochastic grammar over programs. Programs may be represented as an expression defined in a grammar where all terminal nodes are valid components for a program interpreter, and where invalid programs cannot be produced. The class of valid inputs for the program interpreter is the program language.

In the simplest case, the program grammar itself consists only of typed literals which themselves constitute the primitive lexicons of the program language. This approach is sufficient for computation, as at least one Turing complete program language can be entirely defined by a simply typed grammar [Turing, 1936][Church, 1941][Steedman, 2000]. We refer to this approach as the *primitive* grammar. Though sufficient, the primitive grammar as a search space over programs given a particular domain of program learning tasks lacks does not utilize any domain-specific knowledge which would aid in solving tasks quickly and expressively. Domains-specific knowledge, then, can be considered a grammatical construct rather than one of the program language itself — and its purpose is to aid in the expressivity of the search space.

In addition to this perspective on domain-specific knowledge, it follows that different domains should yield different expressive program grammars. The set of non-primitive grammars are therefore defined in a contextual manner, where the choice of grammar depends on the domain of the task in question. We refer to this class of non-primitive grammars as *contextual* grammars. In this paper, we introduce a new system designed to learn contextual grammars.

2 Program Language and Grammars

For simplicitly and completeness we use a simply typed variable-free lambda calculus — a variant of typed combinatory logic — as the foundation for the program language [Liang et al., 2010]. We define "curried" primitives of certain combinators

and of other elements which are natural for whatever tasks the program may need to solve. We use the following primitive combinators:

$$\begin{aligned} \mathbf{I}x &\to x & \mathbf{C}fgx &\to (fx)g \\ \mathbf{S}fgx &\to (fx)(gx) & \mathbf{B}fgx &\to f(gx) \end{aligned}$$

These primitive combinators permit a variable-free representation, where execution of the program with certain inputs will *route* the inputs to their appropriate destination in the program structure. This is because programs themselves are just combinators.

We then represent these programs as binary trees where every non-terminal node refers to a function application of its left child with its right child. See diagram (c) in Figure 1 for an example of a program with this representation. The expansion of the example $x \vdash x^2 + 1$ is as follows:

$$\begin{aligned} C\ (B + (S * I))\ 1\ x &\to (B + (S * I)\ x)\ 1 \\ &\to +\ (S * I\ x)\ 1 \\ &\to +\ ((*\ x)\ (I\ x))\ 1 \\ &\to +\ (*\ x\ x)\ 1 \end{aligned}$$

Extensionally, the program language is simply typed. To effectively use each primitive combinator as an individual lexical item, we permit polymorphic types. The non-terminal nodes in the binary tree representation are *typed* applications of combinators. The different type variants are primitive $\mathcal{T}_0$ (e.g. $\mathcal{T}_0 = \{\text{real}, \text{boolean}\}$), variable, or functional. A functional type $\tau_1 \to \tau_2$ refers to a relation from a source type τ_1 to a destination type τ_2.

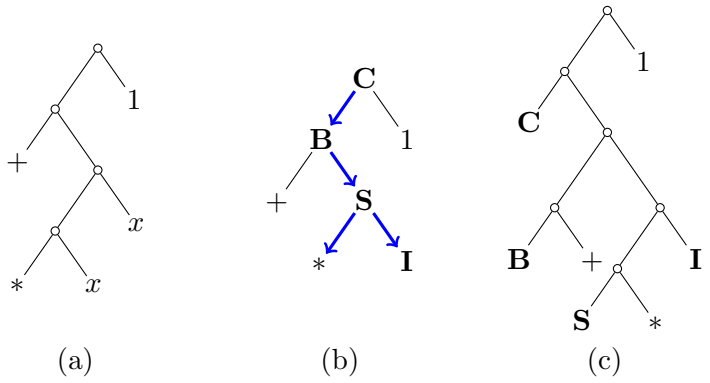

Figure 1: Binary tree representations for $\lambda x. + (*xx)1$, equivalent to the combinatory logic expression "$C(B + (S * I))1$": (a) lambda calculus; (b) variable-free, with arrows to visualize the path of an input; (c) combinators as terminals.

The *primitive* grammar, then, follows trivially from this description of the program language as a Combinatory Categorical Grammar [Steedman, 2000]. Every *contextual* grammer simply introduces additional primitive combinators, where each new primitive combinator is some non-primitive combinator in the primitive grammar.

3 Learning an Expressive Grammar

In this section, we outline an iterative procedure to construct an expressive contextual grammar given a set of tasks in a single program learning domain. Each task is a function which takes a program and returns a truth value of whether the task was solved. The task can be conceptualized as a pairing of inputs and output where the program must reproduce the output given the corresponding inputs.

The first iteration starts with the primitive grammar, and each subsequent iteration acts on the grammar learned in the previous iteration. In each iteration, probable expressions that solve tasks are found and used to construct a new grammar.

3.1 Stochastic Extension

We follow the E.C. algorithm described in [Dechter et al., 2013] to define and enumerate a stochastic grammar over programs. For a language $\mathcal{L}$ with primitive combinators $C = \{c_i\}_{i=1}^n$ fitting the description in the previous section, the probability of an expression is the product of the probabilities of each terminal in its tree. The stochastic grammar only needs to associate each primitive combinator of $\mathcal{L}$ with a distribution $\mathcal{D} = \{p_i\}_{i=1}^n$ of prior probabilities for each combinator.

The probability of a primitive combinator c in a terminal node depends on the *requesting* type $\tau(c)$ according to the type its parent requested when the terminal was selected. For example, if the requesting type is the functional $\tau_0 \to \mathit{Bool}$ from a variable type to a boolean, then $(<\ 2)$ is applicable (its type is $\mathit{Real} \to \mathit{Bool}$) but $(+\ 2)$ is not (its type is $\mathit{Real} \to \mathit{Real}$). If the requesting type was $\tau_0 \to \tau_0$, then $(+\ 2)$ would be applicable and $(<\ 2)$ would not. The set of primitive combinators whose types unify with τ is C_τ.

An expression $e \in \mathcal{L}$ with the set of primitive combinators in its terminal nodes C_e has probability $\mathbb{P}(e) = \prod_{c \in C_e} \mathbb{P}(c|\tau(c))$. The posterior is determined using Bayes' rule:

$$\mathbb{P}(c_i|\tau(c_i)) = \frac{p_i}{\sum_{c_j \in C_{\tau(c_i)}} p_j}$$

An expression is most probable if it maximizes the likelihood of each terminal primitive. For a set of expressions $\mathcal{E}_N$ of finite size N, computing the exact maximizer is an explosive task that is in need of optimization. We use the maximum likelihood

estimator from the E.C. algorithm, which quickly estimates a combinator's likelihood using the frequency with which it unifies with the requesting type of terminals in $\mathcal{E}_N$.

The E.C. algorithm also has a method for constructing $\mathcal{E}_N$, which we adopt. This method, referred to as the best-first enumeration of programs, is essentially a recursive procedure which treats every program, much like the binary tree representation shown in Section 2, as either a primitive combinator or a both a left-program and right-program (where the left program is applied to the right program). This procedure maintains type information and probabilties as it steps down a tree of possible programs in the path of maximum probability. While it enumerates possible programs it also keeps record of the probabilities assigned to each terminal — and for not-yet-expanded nodes, it associates a probability according to a guaranteed upper-bound probability which applies to all trees.

3.2 Selecting Expressive Combinators and Updating the Grammar

The E.C. algorithm, mentioned above, also describes a method for finding the most compressive set of solutions after enumerating the best N programs, which we adopt. The goal of compressibility results in a choice of modular combinators that are expressive for the particular set of tasks which are solved by expressions in $\mathcal{E}_N$. Compression, in this case, is measured inversely with number of unique sub-trees in the expressions which solved a task. Tasks for which only one solution was produced must identify with that solution. Tasks which had many solutions in $\mathcal{E}_N$ then identify according to whichever expression maximizing the compression over the entire solution set. An exact solution to this problem is computationally infeasible, so rather than computing the number of unique trees in the complete solution set, we choose a solution to identify with the particular task by computing the number of unique trees only in the preceding task's solution and the solution candidate for the task. This method was shown effective in [Dechter et al., 2013].

We use the sequitur algorithm on the solution set produced above to construct a new compressive grammar [Nevill-Manning and Witten, 1997]. The stochastic extension for the new grammar is constructed by re-writing each solution according to the new grammar's rules and estimating the probabilities for each node production in the solution set using the same approach described in the previous subsection.

4 Contextual Learning

The set of primitive combinators which define a program grammar lack structure — they are simply a flat collection of distinct combinators. We abandon this core idea in favor of a system inspired by intuitive traits of human cognition. We refer to a set of combinators as an item of knowledge. A grammar must not be free to use all items of knowledge: a particular context must motivate certain items of

knowledge to be readily available while setting others further apart, depending on their relationships to those in contextual proximitiy.

We develop a novel framework for contextualizing knowledge in an abstract manner. Knowledge is represented as distinct atoms of information and their relations in the structure of a connected network, which we refer to as the *knowledge network*. This network is constructed by preferential "least effort" attachment [Barabási and Albert, 1999][i Cancho and Solé, 2003], where a new item of knowledge joins the network with relations to the contextual knowledge from which it was learned. This network is scale-free — it conforms to a mathematical pattern similar to that of Zipf's Law [Zipf, 1949], where the degrees of connectivity for nodes in the network follow a class of power law distributions.

We use this framework iteratively where each iteration is a *phase*. Within each phase lies many iterations of the expressive grammar learner described in the previous section. Each phase has a set of tasks which correspond to the same particular domain of tasks, though different phases may refer to the same domain. At the end of each phase, we interact with the knowledge network.

4.1 Abstraction on Grammars

Our framework is best conceptualized as a knowledge abstraction — on par with typical non-hierarchical sets or addressed memory. We refer to a small motivated subnet of the knowledge network as the *context*, which is constrained to always be approximately scale-free. We provide a simple interface for interacting with knowledge where all interactions either retrieve information related to the current context or adjust the context within the knowledge network. The interactions are GET(), EXPLORE(), ORIENT($\mathcal{I}$), and ADD($\mathcal{I}$), where $\mathcal{I}$ refers to an item of knowledge. The GET() method returns the set of all items within the context. The EXPLORE() method is like GET(), but instead of only yielding items within the context, it additionally yields all items that are one edge away from the context. The ORIENT($\mathcal{I}$) method moves the context with a focus around the specified item. The ADD($\mathcal{I}$) method automatically adds a new item of knowledge to the network, and adjusts the context to include the new item. The automatic attachment follows a least-effort procedure where a connection between the new node and another node $\mathcal{I}_j \in \{\mathcal{I}_i\}_{i=1}^{k}$ in the context (of size k) is established based on the result of a Bernoulli trial parameterized by θ_j, which is calculated as

$$\theta_j = \frac{m_j}{M}$$

where m_j is the count of accesses to $\mathcal{I}_j$ since the last context-switch and where $M = \sum_{i=1}^{k} m_i$ is the total count of knowledge item accesses since the last context-switch.

4.2 Interacting with Knowledge

The initial grammar employed by the grammar learner at the beginning of each phase is defined using the set of combinators in the current context:

$$C = \bigcup \textsc{Get}() = \bigcup_{i=1}^{k} \mathcal{I}_i$$

At the end of each phase, we must determine whether the context should be adjusted (and if so, where to adjust it) and whether to add a new item of knowledge. We refer to the new set of combinators constructed by the grammar learner as C^*.

Consider the the set of combinators beyond the context:

$$C' = \bigcup \textsc{Explore}()$$

If the most probable combinator c_i in C^* is also in C', we adjust the context according to the item of knowledge which contains c_i:

$$\exists \mathcal{I}. c_i \in \mathcal{I}. \textsc{Orient}(\mathcal{I})$$

We now recompute C' with the new state of the knowledge network. We check the P most probable combinators in C^*, referred to as $C^P \subseteq C^*$, for presence in C', and add a new item of knowledge of every combinator not in C':

$$\exists \mathcal{I}. \left(\mathcal{I} = C^P \setminus C'\right). |\mathcal{I}| > 0. \textsc{Add}(\mathcal{I})$$

The resulting system, parameterized by P, is such that the current context may appropriately be identified as domain-specific knowledge, where an item knowledge is a set of combinators which are used as primitives for a grammar defined on the context.

5 Discussion and Future Work

In this paper, we introduced a system for learning expressive contextual grammars in a variable-free variant of lambda calculus defined as a simply typed combinatory logic. This system has yet to be implemented and tested to verify its utility in constructing expressive grammars in different problem domains. The novel knowledge abstraction given in Section 4.1 could be used for contextual learning in other classes of problems in artificial intelligence rather than just program induction: if items of knowledge maintained records of which mechanisms could consume it (e.g. program inductor), multiple mechanisms could be connected to the same network (and confined to the same context), permitting a natural method for multi-modal learning. For example, a sound texture mechanism and a vision mechanism could hear and see rainfall and associate them in the same context, effectively resulting in a "symbol"

for the multi-model object of rainfall. Additionally, mechanisms could *act*, rather than simply percieve, resulting in a production system which would act according to its context. Finally, if communication between mechanisms were supported, hierarchies of mechanisms could be constructed. In this case, the mechanisms would be more fittingly referred to as *agents* [Minsky, 1988].

References

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

Alonzo Church. *The calculi of lambda-conversion*. Number 6 in Annals of Mathematical Studies. Princeton University Press, 1941.

Eyal Dechter, Jonathan Malmaud, Ryan P Adams, and Joshua B Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.

Ramon Ferrer i Cancho and Ricard V Solé. Least effort and the origins of scaling in human language. *Proceedings of the National Academy of Sciences*, 100(3): 788–791, 2003.

Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

Marvin Minsky. *Society of mind*. Simon and Schuster, 1988.

Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res.(JAIR)*, 7:67–82, 1997.

Mark Steedman. *The syntactic process*, volume 24. MIT Press, 2000.

Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

George Kingsley Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Addison-Wesley, 1949.