CONTEXT-SENSITIVE PREDICTION OF HASKELL TYPE SIGNATURES FROM NAMES

Brian Hempel
Department of Computer Science
University of Chicago
Chicago, IL 60637
brianhempel@uchicago.edu

Corrections and explorations after submission displayed in blue.

1 ABSTRACT

Identifiers in programs contain semantic information that might be leveraged to build tools to help programmers write code. This work explores using RNN models to predict Haskell type signatures given the name of the entity being typed. A large corpus of real-world type signatures is gathered from online sources for training and evaluation. In real-world Haskell files, the same type signature is often immediately repeated for a new name. To attempt to take advantage of this repetition, a varying attention mechanism was developed and evaluated. The RNN models explored show some facility at predicting type signature structure from the name, but not the entire signature. The varying attention mechanism provided little gain.

2 INTRODUCTION

The aim of this project is to take a step towards a larger vision of using natural language to generate specifications which can be transformed into function code by a program synthesizer.

$$\text{NL/Code} \underset{ML}{\longrightarrow} \text{Specs} \underset{Synth}{\longrightarrow} \text{Code}$$

Type signatures are a particular light-weight form of specification. Given the name of a function or value in the programming language Haskell, the goal of this project is to predict the type of the name. This setup approximates an autocomplete system in which a programmer may type the function name and is then presented with suggestions for the function's type signature.

Type signatures in Haskell are moderately complex and are usually provided as a declaration separate from the actual definition of the named function or value. For example, the declaration:

```
maybeFind :: (a → Bool) → [a] → Maybe a
```

declares that the value named `maybeFind` is a function that takes two arguments, the first is function that returns a boolean given a value of some generic type `a`, the second is a list of values of that same type `a`, and the return type is `Maybe a`, an option type possibly containing a value of type `a`.

This work treats the type signature as a token stream to be predicted by a RNN seeded with an encoding of the name. To facilitate reusability, names and type identifiers are broken into words for which embeddings are learned. To take advantage of previous type declarations in a file, a varying attention mechanism is employed. The system is evaluated on a large corpus of real-world Haskell code.

3 RELATED WORK

Global language models of code may be queried in a token-by-token (or node-by-node) fashion to facilitate code completion (Hindle et al., 2012; Raychev et al., 2016; Bielik et al., 2016; White et al., 2015; Nguyen & Nguyen, 2015) (an approach also taken by class projects at other universities (Ginzberg et al., 2017; Das & Shah, 2015)). To generate larger pieces of code, another line of

work attempts to translate natural language specifications into programs (Gu et al., 2016; Gvero & Kuncak, 2015; Raghothaman et al., 2016) including a recent approach (Yin & Neubig, 2017) that adapts the particular sequence-to-sequence method of Bahdanau et al. (Bahdanau et al., 2014) for the purpose of generating moderately sized ASTs. Modular networks have also had some success (Rabinovich et al., 2017).

This work may be viewed in between these two lines of work: like language modeling, the input to the predictor comes directly from the code; like NL-to-code translation, the natural language semantic content of the identifiers is to be leveraged to produce a result.

4 METHODS

This work models the problem as a serial next-token prediction problem, with tokens predicted by a single-layer LSTM-RNN. The RNN is seeded with an encoding of the name to be assigned a type. The first token prediction is triggered by applying a special start-of-signature token; prediction proceeds apace from there. This work makes particular choices concerning the encoding of tokens for generalizability, and incorporation of context to inform the prediction. These choices are detailed below.

As predicting an entire type signature given only a name may not be generally achievable, this work additionally explores predicting just the structure of the type signature.

4.1 GATHERING TYPE SIGNATURES

The top 1000 Haskell language repositories by "stars" were downloaded from Github. The repositories owned by the Haskell and Glasgow Haskell Compiler (GHC) organizations were placed into the training set, and the remaining repositories were randomly assigned to form a total of 800 training, 100 development, and 100 test repositories. All files ending in `.hs` were run through the Haskell parser in GHC 8.0.1 to extract top-level type declarations, from which uncommon meta-data (such as `infix` declarations) was removed. Multiple names simultaneously assigned a single type were split into separate declarations. Thus each extracted declaration consisted of a name and a type signature for that name. 54,186 files (78%) were successfully parsed. Typeclass constraints were removed from each type signature using a Python-based parser and normalization of type variable names was performed (*i.e.* within each signature, the first type variable encountered was renamed to `a`, the second to `b`, etc.). A small number of (0.6%) of signatures were long (> 256 characters) or not parsable in Python and were discarded. The final dataset consists of 304,272, 20,962, and 25,619 signatures for training, development, and test, respectively.

For structure-only prediction, the type signatures were further normalized. Arrows, parentheses, brackets, and commas were considered "structural" tokens and were left unchanged. Between structural tokens, runs of consecutive non-structural tokens were merged into a single type and assigned a normalized name. For example, the type signature `Maybe a → ( a → Maybe b ) → Maybe b` would be normalized to the structure `A → ( B → C ) → C`.

4.2 TOKEN REPRESENTATION

To facilitate generalizability to unseen names, all tokens were segmented into words based on the camel case convention and the presence of underscores. All words were further "stemmed" by truncation to the first four characters. An embedding was assigned to each stemmed word. A token's representation was considered to be the average of the embeddings of its constituent stemmed words (duplicate words weighted by frequency). Formally, if $seg(y)_i$ is the ith stemmed word in the segmentation of y, and $emb(seg(y)_i)$ is the embedding for that word, then:

$$emb_{tok}(y) = \frac{1}{|seg(y)|} \sum_{i=1}^{|seg(y)|} emb(seg(y)_i)$$

Defining token embeddings in terms of constituent words reduces the number of needed embeddings considerably. The 35,112 unique non-structural tokens in the training data are composed of only

4,902 stemmed words. Similarly, the training data assigns types to 196,382 unique names which are composed of only 21,317 stemmed words.

4.3 Name Encoding

The goal is to predict a type signature given the name of a function or value. Names are segmented as above, and, again, the average of their embeddings is taken as their representation. Half of this representation is used as the initial hidden state and the other half as the initial memory cell of the LSTM. Consequently, words in names and words in types have separate embeddings, as embeddings for words in names must be twice as large (words in types have embeddings of the same dimensions as the LSTM hidden state).

4.4 Loss Heuristic

Token predictions from the RNN are made by a simple dot product between the token embeddings and and the RNN output. To train the network using log loss requires that the expected token be assigned a probability. The output can be treated as a probability distribution using $\mathrm{softmax}$:

$$P(y_t) = \mathrm{softmax}\left((\boldsymbol{W}^{(tok)})^\top \boldsymbol{h}_t\right) = \mathrm{softmax}\left((\boldsymbol{W}^{(word)} W^{(word \to token)})^\top \boldsymbol{h}_t\right)$$

where $\boldsymbol{h}_t$ is the output of the RNN at time t and $\boldsymbol{W}^{(tok)}$ is an embedding matrix with the token embeddings in the columns. As each token embedding is a linear combination of several word embeddings, $\boldsymbol{W}^{(tok)} = \boldsymbol{W}^{(word)} W^{(word \to token)}$, where $W^{(word \to token)}$ is a sparse matrix of dimensions $|V_{word}| \times |V_{tok}|$ that performs this linear combination.

Because of limitations of the learning framework used, $\boldsymbol{W}^{(word)} W^{(word \to token)}$ could not be efficiently calculated after each update to $\boldsymbol{W}^{(word)}$. Therefore, for driving loss during training, a heuristic was used instead:

$$P(y_t) = \mathrm{softmax}\left((\boldsymbol{W}'^{(word)})^\top \boldsymbol{h}_t\right)$$

where $\boldsymbol{W}'^{(word)} = \boldsymbol{W}^{(word)}$ when the expected token was a single word; otherwise the embedding of the expected token was appended to $\boldsymbol{W}^{(word)}$ to form $\boldsymbol{W}'^{(word)}$.

4.5 Incorporating Context

Repeated type signatures are common. To facilitate use of this information, the predicted embedding $\boldsymbol{h}_t$ in the above definitions is instead replaced with $\boldsymbol{h}_t + w_c \boldsymbol{c}_t$, where w_c is a learned scalar and $\boldsymbol{c}_t$ is a context vector defined as follows:

$$\boldsymbol{c}_t = \sum_{sig_{prior} \in context} \sum_{y_i \in sig_{prior}} k * attn(sig_{cur}, sig_{prior}, i, t) emb_{tok}(y_i)$$

Context is the three prior signatures within a file; or as many as are available if near the beginning of the file.

$$attn(sig_{cur}, sig_{prior}, i, t) = \prod_{j=1}^{k} sim\big((sig_{cur})_{t-j}, (sig_{prior})_{i-j}\big)$$

$$k = \min(4, |sig_{cur}|, |sig_{prior}|) \qquad sim(x, y) = cosine\_sim\big(emb_{tok}(x), emb_{tok}(y)\big)$$

That is, $\boldsymbol{c}_t$ is a linear combination of the tokens in several prior signatures in the same file. The weight for a context token is based the similarity of its prior tokens and the tokens immediately prior to the token being predicted. The approach may be thought of as a soft n-gram model. The weight is multiplied by k to capture the intuition that a longer match is more likely to be relevant. The attention weights are not normalized to sum to one—the weight should only be high when it is appropriate to copy tokens. Thus the magnitude of $\boldsymbol{c}_t$ varies for each prediction.

Possibly should have been taken to the 1/k power (geometric mean) and then also learn coefficients for the strength of each value of k; lots of variations could be tried here.

Note that in the above formulation, the name being assigned a type is considered part of the signature (as y_0). The half of its embedding for seeding the RNN hidden layer is used. y_1 is the start of type symbol. Thus, k is always at least 2.

Unkown tokens:
Test data is 6.1% unpredictable tokens (tokens not present in training). 27.3% of signatures are unpredictable because they contain a token not present during training.

That sets upper limits on the accuracy of the given approach—however current performance is far from those limits.

Table 1: Token prediction accuracy and whole signature prediction accuracy.

	FULL VOCABULARY			STRUCTURE ONLY		
	Copy	LSTM	+Attn	Copy	LSTM	+Attn
Token Accuracy	43.1	46.1	**47.6**	55.6	73.7	**75.5**
Signature Accuracy	**20.6**	3.5	5.8	35.7	37.7	**38.3**

Able to get up to 53.1% (token) and 10.0% (whole sig) by:
- On token collision, predict most common token
- Use a Snowball stemmer, with word suffixes (adds a few more embeddings: 5817 type words, 23380 ident words)
  - Type words: "Typed.TypedDefinition'" => ["Type", "-d.", "Type", "-d", "Definit", "-ion", "'"]
  - Ident words: same, but all words downcased
- Longer training (2.9M signatures, ADAM, learning rate = 0.0002)

5 Experimental Setup

The model described above was implemented and trained in DyNet (Neubig et al., 2017). DyNet's default LSTM variant uses peephole connections and couples the input and forget gates (Greff et al., 2015). 150 dimensions were used for the hidden layer size and type word embeddings (300 for name words). For each of signature and structure prediction, a model ignoring and a model incorporating context was trained. Context included up to three immediately prior signatures in the same file. Models were trained with Adam (Kingma & Ba, 2014) at a learning rate of 0.0002. Time did not permit exploration of hyperparameters. Training was not halted at any principled time, however all the models incorporating context were trained for a shorter amount of time compared to the corresponding non-attentive model, so the improvement shown above is not unfair. Accuracy on development also quickly plateaued, extra training is unlikely to alter the story. For each variation, the model checkpoint with highest whole signature accuracy on half of the development data was used for evaluation. The checkpoints used were trained on a minimum of 300,000 and maximum of 700,000 declarations.

6 Results and Analysis

Table 1 displays the experimental results. The "Copy" baseline simply copies the previous type signature token by token as the prediction. If there is no prior type signature, or when the current signature extends beyond the previous signature, the end-of-signature token is repeatedly predicted. For the RNN models, the prediction for the entire signature is produced by a simple greedy search, selecting only the most probable token at each step.

The copy-from-previous baseline performs surprisingly well. The RNN models are able to improve on the token prediction accuracy slightly, but the attention mechanism did not facilitate widespread copying of the previous type signature as hoped. For predicting only the structure of a type signature, the RNN without context is able to improve slightly on the copying baseline, using only the name of the function to predict the structure of the type signature. The additional information provided by the context provides a further slight improvement. However, both RNN models fail to significantly outperform simple copying of the previous signature.

7 Conclusions and Future Work

The structure prediction models were moderately successful at predicting some type signatures' structures from their names, but, overall, the RNNs presented did not perform significantly better than simply copying the previous type signature in a file. The proposed varying attention mechanism designed to facilitate such copying in the neural setting did not have a significant effect—an investigation of why should be performed and an alternative mechanism proposed.

Other improvements might also be made. A slower learning rate might be used—accuracy on the development dataset quickly plateaued during training. Larger embeddings and a deeper LSTM might also facilitate some improvements. The prediction model itself could be enriched as well. Token-by-token prediction does not match the actual syntax tree structure of the type: a tree prediction model might produce better results. Furthermore, the use of context might be improved. An IDE providing an autocomplete service can be expected to only suggest names that are in scope; similarly, the number of arguments applied to each type should be known. A smarter model could use this information to constrain its output. Finally, an important next step for this work is to include the prediction of typeclass constraints, as they commonly occur in Haskell programs.

REFERENCES

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning To Align and Translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016.

Subhasis Das and Chinmayee Shah. Contextual Code Completion using Machine Learning. *Stanford CS229 Class Project*, 2015.

Adam Ginzberg, Lindsey Kostas, and Tara Balakrishnan. Automatic Code Completion. *Stanford CS224n Class Project*, 2017.

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A Search Space Odyssey. *CoRR*, abs/1503.04069, 2015. URL http://arxiv.org/abs/1503.04069.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep API Learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016.

Tihomir Gvero and Viktor Kuncak. Synthesizing Java Expressions From Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the Naturalness of Software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

Anh Tuan Nguyen and Tien N. Nguyen. Graph-Based Statistical Language Model for Code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract Syntax Networks for Code Generation and Semantic Parsing. *CoRR*, abs/1704.07535, 2017. URL http://arxiv.org/abs/1704.07535.

Mukund Raghothaman, Yi Wei, and Youssef Hamadi. SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016.

Veselin Raychev, Pavol Bielik, and Martin T. Vechev. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, 2016.

Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. Toward Deep Learning Software Repositories. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, 2015.

Pengcheng Yin and Graham Neubig. A Syntactic Neural Model for General-Purpose Code Generation. *CoRR*, abs/1704.01696, 2017. URL http://arxiv.org/abs/1704.01696.