# Predicting Haskell Type Signatures From Names

**Bowen Wang** and **Brian Hempel** and **Ravi Chugh**
Department of Computer Science, University of Chicago
Crerar Library, 5730 S Ellis Ave, Chicago, IL, USA 60637
{bowenwang1996,brianhempel,rchugh}@uchicago.edu

**Abstract**

Neural Program Synthesis has been a fast-growing field with many exciting advances such as translating natural language into code. However, those applications often suffer from the lack of high-quality data, as it is fairly difficult to obtain code annotated with natural language that transcribes its functionality precisely. Therefore, we want to understand what information we can automatically extract from source code to facilitate program synthesis, without explicit supervision. Specifically, we study the problem of predicting type signatures given identifier names. We focus on Haskell, a strongly typed functional language, and collect data from Haskell repositories on GitHub. We propose a structured prediction model that not only explicitly leverages the tree structure of Haskell types, but also takes advantage of context information. Our model achieves 27.28% signature accuracy and 61.65% structural accuracy on the test set.

## 1 Introduction

Researchers have recently demonstrated success in applying deep learning to program synthesis tasks previously deemed insurmountable: some nontrivial programs can be synthesized from from natural language alone (Rabinovich, Stern, and Klein 2017; Yin and Neubig 2017). Nevertheless, the lack of data for the task remains a limiting factor (indeed, it is the biggest obstacle in any application of deep learning).

Desired program behavior can be specified in different forms, such as natural language, input-output examples, and type signatures. Despite the vast amount of code available on the Internet, such specifications are not easy to collect and harness. Even though programmers often write comments when coding, they usually focus on high-level ideas rather than transcribing the functionality of the code verbatim.

Compared to comments explaining code, type signatures are more readily and consistently available, and though they often describe coarser-grained properties about code than comments, they are checked for correctness. Being able to leverage type signatures could, thus, help address the data problem for program synthesis tasks. Our goal in this paper is to investigate the question: **Can we predict the type of an expression given its name?** A successful model for this task could, in the future, be used as part of an interactive programming system that, e.g., suggests type annotations while the user types a name, or flags names that are not semantically consistent with their corresponding type annotations.

Formally, let $\Sigma$ be the space of identifier names and $T$ be the space of type signatures. We aim to learn a mapping $f : \Sigma \to T$. However, such a naive formulation would not be applicable in practice as such an $f$ cannot take into consideration the new types defined by users. It would be better to predict the type signature given some context that contains the types that appear in the type signature. As a result, $f$ should be a function of $\Sigma$, parametrized by context $c$.

**Challenges** Even with context, predicting types from names is a quite challenging task. First, unlike a normal sequence-to-sequence task such as machine translation, the rigidity of types requires exact matches rather than approximations. Second, each program may define its own types, which are not in the vocabulary of other programs. While the NLP community has developed some methods to deal with out-of-vocabulary words (Vinyals, Fortunato, and Jaitly 2015; Gu et al. 2016; See, Liu, and Manning 2017), those methods are often applied in settings where, unlike ours, out-of-vocabulary words are rare.

Furthermore, in normal sequence-to-sequence tasks like machine translation, the similarity and differences in the semantics of input usually correspond to those of the output, e.g., two sentences of similar meaning in English should be translated to sentences of similar meaning in Spanish. However, in our setting, the opposite sometimes holds. For example, two set functions `intersect` and `union` may both have the same type `Set a -> Set a -> Set a` despite having opposite meaning. This phenomenon appears quite often in programs, suggesting a more complicated semantic relationship between input names and output types.

**Our Approach** In principle, our approach works for any strongly typed programming language. In this paper, we choose to work with Haskell (Jones 2003), a functional language with a powerful type system where types generally specify finer-grained invariants than types in languages like C and Java. To simplify data collection and processing, we consider only top-level function and variable definitions, but our approach can be extended to non-top-level identifiers with some modifications of context.

We model the tree structure of Haskell types directly and thus guarantee the well-formedness of generated types. We also make use of context information, which in our approach consists of some number of previous type signatures. We measure the performance of our models on (a) signature accuracy, which measures the percentage of correct signature predictions, and (b) structural accuracy, which measures the percentage of predictions that have the same tree structure as the ground truth. Our model achieves 27.28% signature accuracy and 61.65% structural accuracy.

## 2 Related Work

**Neural Program Synthesis** Neural program synthesis, which uses deep learning to tackle program synthesis problems, has become a rapidly growing field (Kant 2018). This brings about powerful natural language models (Melis, Dyer, and Blunsom 2018) to a task traditionally attacked with type specifications or input-output examples.

Several approaches synthesize code from natural language descriptions (Dong and Lapata 2016; Ling et al. 2016; Yin and Neubig 2017; Rabinovich, Stern, and Klein 2017) and mostly model the structure of a program explicitly. Dong and Lapata propose the sequence-to-tree (seq2Tree) model, which uses a sequence-to-sequence model to generate tree output by adding special tokens to model depth-first tree generation (Dong and Lapata 2016). Ling et al. propose latent predictor network to allow character-level generation of code pieces (Ling et al. 2016). Yin and Neubig use a generation model that follows the grammar for Python abstract syntax trees (ASTs) (Yin and Neubig 2017). Rabinovich et al. use a modular neural network to generate output according to the abstract syntax description language (Rabinovich, Stern, and Klein 2017), an approach most similar to ours. However, there are some crucial differences between our approach and the approaches mentioned above: (1) our task does not require external labels and thus has abundant data available; and (2) none of the approaches above considers context, which is an essential component of our model.

Researchers have also combined neural networks with traditional synthesis techniques, such as searching and programming by examples. Robust Fill (Devlin et al. 2017) combines structured prediction with searching based on examples to synthesize spreadsheet programs. Polosukhin and Skidanov propose tree beam search to refine the seq2Tree generation results (Polosukhin and Skidanov 2018). Murali et al. combine neural program generation with combinatorial search by training on program sketches (Murali et al. 2018).

**Copying** Dealing with out-of-vocabulary (OOV) words has always been an obstacle in NLP applications such as text summarization and machine translation. To address this problem, researchers have considered copying OOV words from input. Vinyals et al. propose the pointer network to copy from input according to the attention probability (Vinyals, Fortunato, and Jaitly 2015). (Gu et al. 2016) and (See, Liu, and Manning 2017) allow both generation and copying by computing the probability of generation at each time step. We generally follow the mechanism proposed by See et al. (See, Liu, and Manning 2017) to copy types from context. However, unlike the text summarization task studied in (See, Liu, and Manning 2017), our prediction task may actually benefit from token repetitions, as signatures like `Int → Int` and `a → a → a` are quite common. Therefore, we do not use the coverage loss proposed in (See, Liu, and Manning 2017) to punish repetitions.

| Freq. of Type in Sigs. | #Types | #Signatures |
|---|---|---|
| 10193 | 1 | 10193 |
| 1000 – 3613 | 17 | 33106 |
| 100 – 1000 | 192 | 43947 |
| 10 – 100 | 3644 | 76239 |
| 2 – 10 | 50333 | 150140 |
| 1 | 161294 | 161294 |

Table 1: Distribution of types in the dataset. For a given row, the first column denotes a range $[n, m)$. The second column denotes the size of the set $T$ of types $t$ that appear in between $n$ and $m$ signatures. The third column denotes the number of signatures $x :: t$ such that $t \in T$.

## 3 Data Pipeline

In this section, we describe how we collect and preprocess data for our task.

### 3.1 Data Collection

To construct the dataset, we crawled all the existing Haskell repositories with 10 or more stars on GitHub. This yielded 1,932 repositories and, using a standard 80/10/10 split (i.e, 80% data for training, 10% for validation, and 10% for testing), we had 1,546 repositories for training, 193 for validation and 193 for testing. All `.hs` files in the repositories were passed to a Haskell parser, yielding 401,882 signatures for training, 39,040 signatures for validation, and 33,997 signatures for testing. The dataset has 192,606, 18,149, and 18,212 unique types in training, validation, and testing set, respectively. Among all types in the dataset, the most common one is `IO ()`, which appears 10,193 times.

Table 1 gives a more detailed view of the distribution of types in the dataset. Most types appear in fewer than 10 signatures in the dataset and 161,294 types appear only once in the dataset. The data shown in Table 1 are from the entire dataset, including training, validation, and testing. Note that the data describe types after normalization, as will be discussed in Section 3.2.

### 3.2 Data Preprocessing

We preprocessed the dataset in several ways before training.

**Normalizing Signatures** Multi-name type signatures (e.g., `a, b :: Int`) are de-sugared to consecutive single-name signatures (e.g., `a :: Int` and `b :: Int`). We also remove the qualification of types (e.g., `Prelude.Maybe` is simplified to `Maybe`). Furthermore, we choose canonical names for type variables in left-to-right order (i.e, the first type variable is renamed to `a`, the second to `b`, etc.) to avoid unnecessary noise in the data. For example, the type of the `lookup` function in `Data.Map` is rewritten from `k → Map`

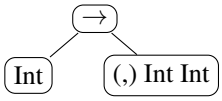
`k a` → `Maybe a` (where `k` indicates that the type variable is the key type) to `a` → `Map a b` → `Maybe b` (where `k` is rewritten to `a` and `a` is rewritten to `b`).

**Type Classes** Somewhat akin to interfaces in Java and C# interface, Haskell type classes describe types that share a common set of operations (a.k.a. methods), explicitly with types and implicitly with intended (unchecked) invariants (Wadler and Blott 1989). For example, the `Num` type class describes the set of types (including `Int`, `Float`, and `Double`) for which a binary `(+)` operator is defined.

Although type class information may provide valuable semantic information, explicitly modeling type classes would add additional complexity (type signatures would be longer and harder to predict) and reduce the generalizability of our model to languages with different type systems. Therefore, we choose to remove type class constraints. For example, the signature of the operator `(+)` is rewritten from `Num a` ⇒ `a` → `a` → `a` to `a` → `a` → `a`. The signature of `lookup`, discussed above, has been similarly processed from the original version: `Ord k` ⇒ `k` → `Map k a` → `Maybe a`.

**Token Streams and ASTs** We treat the input (identifier names) and the target (type signatures) differently. For input names, since Haskell programmers usually follow the camel case naming convention (over 99% names in our dataset do), we segment the names into tokens accordingly.[1] To achieve better generalizability, we also stem each token using the NLTK library (Bird and Loper 2004). The stemmer converts upper case letters to lower case letters to avoid superficial differences such as that between "md5" and "MD5". It also stems verbs and nouns to the original form. For example, "Zoned" is stemmed to "zone" and "Suffixes" is stemmed to "suffix". Stemming reduces the number of unique identifier tokens in the training set from 44,543 to 30,361.

For types, we model their tree structure directly — e.g., `Int` → `( Int , Int )` is viewed as a tree as shown on the right. Notice that type application `(,) Int Int` remains a single tree node; we treat arrow as the only special type constructor. However, if there is an arrow contained with in the type application — consider `(,) (Int` → `Int) Int` for example — then the subtree `Int` → `Int` will be modeled as a tree.




**Qualified Identifier Names** To gain more information than contained in identifier names alone, we also consider two ways of qualifying an identifier name: prepending the identifier name with module name or the entire path to the file containing the identifier. For example, if the function `fromJust` is defined in the `Maybe` module, and the `Maybe` module is a file in the `Prelude` directory, then the module-qualified name of `fromJust` is < `Maybe`; `fromJust` >, whereas the path-qualified name of `fromJust` is < `Prelude`; `Maybe`; `fromJust` >. When the identifier names are qualified by module names, they may still not be unique. In the above example, there could be another directory `MyPrelude` that contains the same `Maybe` module. On the other hand, when identifier names are qualified by the entire path leading to the file that contains the identifier, it is almost globally unique (the exception is that two different project owners might have exactly the same path names).

[1] Our segmentor can also handle other naming conventions.

# 4 Structured Prediction Model

To move towards syntactic and semantic soundness in type signature generation, we explicitly model the binary tree structure of Haskell types. More specifically, a Haskell type can be described by the following grammar:

```
Type = NonArrowType | Arrow Type Type
```

where `NonArrowType` refers to types whose type constructor is not arrow. For example, `Maybe (Int` → `Int)` is a non-arrow type even though it contains an arrow, whereas `Int` → `Int` is an arrow type. We choose to differentiate → from other constructors because of its unique position in the Haskell type system: a type represents a function if and only if it has arrow as its type constructor.

## 4.1 Type Constructors And Kinds

The rich type system of Haskell allows users to define new types with type constructors. For example, type `Maybe` is defined by:

```
data Maybe a = Just a | Nothing
```

with type constructor `Maybe` and data constructors `Just` and `Nothing`. Here `Maybe` is a type constructor with kind $* \rightarrow *$, which means it maps a ground type (of kind $*$) to another ground type.

We explicitly model the kind of each type constructor by augmenting the type constructors with their kinds to avoid collapsing type constructors with different arity to the same name. For example, if there are two `Maybe`s in the dataset with kinds $*$ and $* \rightarrow *$, the first `Maybe` is represented as `Maybe#0` while the second `Maybe` is represented as `Maybe#1`.

## 4.2 Model Architecture

To model the binary tree structure of Haskell types, our model consists of several modules, each with its own functionality. The final model is shown in Figure 1.

**Name Encoder** The name encoder, which is a bidirectional LSTM, processes the qualified function names and returns a vector representation of the given name.

**Context Encoder** The context encoder has two parts: context name encoder $E_{cname}$ and context signature encoder $E_{csig}$. $E_{cname}$ is, similar to the name encoder, a bidirectional LSTM. $E_{csig}$, on the other hand, is a tree LSTM encoder similar to that used in (Dong and Lapata 2016). It parses the type signature as a binary tree and processes each node in depth-first order. Given a context containing $n$ name-signature pairs, $E_{cname}$ processes the names and outputs the hidden state $h_{cname}$ by averaging the hidden states of the names processed. Similarly, $E_{csig}$ takes in the type signatures as trees and produces two outputs: $h_{csig}$, the average final hidden states of the type signatures, and $h_{cstates}$,

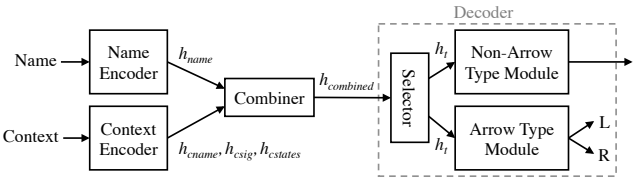



Figure 1: **Structured prediction model.** Given input name and context, the model produces the input name hidden state $h_{name}$, context name hidden state $h_{cname}$, context signature final state $h_{csig}$, and context signature hidden states $h_{cstates}$. The combiner combines $h_{name}$, $h_{cname}$, and $h_{csig}$ to produce the combined hidden state $h_{combined}$ for decoder to decode from. The decoder uses the module selector to decide which module to use to generate output.

the hidden states of all the nodes in the context. $h_{cstates}$ will later be used to compute the attention mask for the decoder.

**Combiner** The combiner module, similar to that proposed in (Zoph and Knight 2016), combines the hidden states produced by the different encoders to obtain a single hidden state for the decoder to decode from.

The combiner, given the hidden states $h_{name}$, $h_{cname}$, $h_{csig}$, produces the combined hidden state $h_{combined}$ by

$$h_{combined} = \tanh(W[h_{in}; h_{cname}; h_{csig}])$$

where $W$ is a matrix of learnable parameters and $[a; b]$ denotes the concatenation of $a$ and $b$.

**Decoder Overview** The grammar presented in Section 4.1 suggests a two-module decoder: one module for handling non-arrow types and one module for handling arrow types. Since the decoder needs to know, at each step of generation, which module to use, we have another module dedicated to module selection.

**Module Selector** At each step of generation with previous node token $x_t$ and previous hidden state $h_t$, the selector first computes an embedding $e_t$ of $x_t$ and then compute the probability for selecting each module by $p_{base}, p_{arrow} = \mathrm{softmax}(f_T(e_t, h_t))$ where $f_T$ is a two-layer feed-forward network with ReLU (rectified linear unit) nonlinearity. The module with larger probability is chosen for the next step of generation.

**Non-Arrow Type Module** The non-arrow type module first computes the new hidden state $h_t = LSTM(\tilde{e}_{t-1}, h_{t-1})$ where

$$\tilde{e}_{t-1} = attn(e_{t-1}, h_{name}, h_{cname}, h_{csig})$$

is the attention output of the previous embedding $e_{t-1}$. The attention mechanism is similar to that used in (Zoph and Knight 2016). We concatenate $h_{name}$, $h_{cname}$ and $h_{csig}$ to feed into the attention module and compute the attention accordingly. Based on $h_t$, the module computes $p_{vocab} = \mathrm{softmax}(f_T(h_t))$ where $f_T$ represents a feed-forward network. Then, depending on whether there is context available, the module behaves differently.

When context is available, the module has two modes: generation and copy. Thus, it needs to compute the generation probability, which, similar to that in (See, Liu, and Manning 2017), is given by

$$p_{gen} = \mathrm{sigmoid}(w_h h_t + w_n h_{cname} + w_s h_{csig} + w_e e_{t-1})$$

where $w_h$, $w_n$, $w_s$ and $w_e$ are learnable parameters. The copy probability of tokens, $p_{copy}$, is given by the normalized attention score of $h_t$ on $h_{cstates}$ (see Section 4.2). The final probability over the tokens is given by

$$p(w) = p_{vocab} * p_{gen} + p_{copy} * (1 - p_{gen}).$$

When the context is not available, the module computes the output token probability as the softmax output of a feed-forward network on the attention output of $h_t$ on the input hidden states $h_{in}$.

With the final probability over tokens $p(w)$ available, the next token $a_{next}$ is selected as $\mathrm{argmax}(p(w))$. If $a_{next}$ is not a ground type, we recursively call the decoder with the corresponding number of steps to make sure a valid type is generated.

**Arrow Type Module** The arrow type module is in charge of generating a type using the `Arrow Type Type` production rule. It first computes the new hidden state $h_t = LSTM(e_{t-1}, h_{t-1})$ where $e_{t-1}$ is the embedding of the previous token generated by the decoder and $h_{t-1}$ is the previous hidden state. Then the module recursively generates the left subtree and the right subtree by calling the decoder with new hidden state and the arrow embedding, i.e,

$$\begin{aligned} Tree_{left},\ h_{left} &= Decoder(h_t, e_{arrow}) \\ h_{right} &= \tanh(W[h_{left}, h_t]) \\ Tree_{right},\ h_{final} &= Decoder(h_{left}, e_{arrow}) \end{aligned}$$

where $e_{arrow}$ is the embedding for the arrow token.[2] The generation starts with a special start token. Note that the hidden state for generating the right tree is a combination of the left hidden state and the parent hidden state. This arrangement follows the idea of parent feeding used in (Dong and Lapata 2016) and also allows the model to carry information from the left subtree to facilitate the generation of the right subtree.

**Controlling Recursion Depth** Unlike the classic sequence-to-sequence model (Sutskever, Vinyals, and Le 2014), our model does not use a special end token to signal the end of generation. Instead, the generation stops when the entire tree has been generated. However, due to the recursive nature of the decoder, it is possible that, when not well-trained, the decoder could keep generating subtrees indefinitely. To prevent this, we add a hyperparameter `rec_depth` to control the recursion depth of the decoder. We set `rec_depth` to 6 in practice as there is only a small fraction (1.8%) of type signatures in the entire dataset that have depth larger than 6.

[2]Note that we do not explicitly generate the arrow token as it is embedded in the tree structure of the output.

| Model | Sig. Acc. (%) | Struct. Acc. (%) |
|---|---|---|
| Baseline | 23.39 | 49.78 |
| seq2Seq | 5.15 | 20.45 |
| seq2Seq + context | 23.98 | 41.44 |
| seq2Tree | 6.84 | 49.52 |
| seq2Tree + context | **27.28** | **61.65** |

Table 2: Overview of final results.

**Loss Function** Similar to (Alvarez-Melis and Jaakkola 2017), we consider the topology loss

$$loss^t_{topo} = -\log(p(m^t_i))$$

where $p(m_i)$ denotes the probability of choosing module $i$ ($i = 0, 1$) at time step $t$. The topology loss pushes the model to learn the structure of type signatures. The final loss for each step $t$ is

$$loss_t = loss^t_{token} + \lambda loss^t_{topo}$$

where $loss^t_{token}$ is the usual negative log-likelihood loss and $\lambda$ is a hyperparameter.

## 5 Evaluation

We conducted multiple experiments to understand the performance of our proposed model. However, due to time and resource limitations, we were unable to tune some hyperparameters. Throughout the experiments, we use a embedding size of 128 and hidden size of 256. The LSTMs all have one layer. We use Adam (Kingma and Ba 2014) for optimization. These hyperparameters are chosen according to (Rabinovich, Stern, and Klein 2017) and (Yin and Neubig 2017).

We measure the performance of our models on the following two criteria:

1. **Signature Accuracy** refers to the percentage of correct signature predictions. A predicted signature must match the ground truth exactly to be considered correct.
2. **Structural Accuracy** refers to the percentage of predicted signatures that have the same tree structure as the ground truth under the grammar presented in Section 4.1. For example, if the ground truth is `Int → Int` and the prediction is `String → Int`, the prediction is structurally correct.

### 5.1 Summary of Results

An overview of the final results is presented in Table 2. The baseline simply copies the previous signature in the same file. For comparison, we also implemented a unstructured prediction model using seq2Seq instead of seq2Tree. The results presented in Table 2 use the module-name qualification discussed in Section 3.2.

As expected, the structured prediction model (seq2Tree + context) has the best performance in terms of both signature accuracy and structural accuracy, which reveals the usefulness of explicitly modeling the structure of Haskell types. The inclusion of context information also proves to be crucial in boosting the performance of both the seq2Seq model (5.15% → 23.39%) and the seq2Tree model (6.84% → 27.28%). While the structured prediction model has a reasonably good performance, it is also worth noting that the non-ML baseline, which simply copies the previous signature in the same file, is almost as good as the unstructured prediction model (seq2Seq + context). The surprising performance of the baseline seems to suggest a strong connection between top-level definitions that are spatially close.

| | seq2Seq + context | | seq2Tree + context | |
|---|---|---|---|---|
| #Ann | Sig. / Struct. Acc. (%) | | Sig. / Struct. Acc. (%) | |
| 0 | 5.15 | 20.45 | 6.84 | 49.52 |
| 1 | 17.13 | 30.48 | 24.96 | **62.04** |
| 2 | 23.01 | 39.07 | 27.04 | 61.73 |
| 3 | **23.98** | **41.44** | **27.28** | 61.65 |
| 4 | 23.03 | 40.79 | 23.98 | 61.51 |
| 5 | 22.62 | 39.22 | 25.52 | 61.00 |

Table 3: Performance by number of annotations in context.

As seen in Table 2, context information, which provides information about new types that are not seen in the training data, proves to be highly effective in improving the performance of the models. The results presented in Table 2 use the three previous signatures in the same file as context; Table 3 shows the difference in performance of the models if we vary the amount of context information available. Somewhat surprisingly, even if we just use the previous type signature as context, the performance of both models (seq2Seq and seq2Tree) improve dramatically (5.15% → 17.13% for seq2Seq, 6.84% → 24.96% for seq2Tree), again indicating the effectiveness of context information. However, we also notice that the performance starts to decrease once the number of context signatures reaches three. We suspect that this is because the previous three type signatures are usually sufficient for the model to see the new types and understand what types are "important" for the prediction; having more signatures in the context makes it harder to the model to learn to copy the correct types and leads to overfitting.

### 5.2 Case Studies

We briefly describe two example predictions (from `hoogle/src/General/Str.hs`) drawn from the test set.

**Example 1** In the example below, the structured prediction model correctly predicts the type while the unstructured prediction model fails. It is clear that both unstructured prediction model and structured prediction model learn something from the context — they both generate the `Str →` part. Unlike the structured prediction model, however, the unstructured prediction model is at risk for generating ill-formed signatures like `Str →`. The structured prediction model is able to correctly generate the whole signature, which also reveals that it has learned to not only copy from context, but also generate types according to the connotations of the name (null relates to `Bool`).

```
Context:
  strUnpack :: Str → String
  strReadFile :: FilePath → IO Str
```

```
  strSplitInfix :: Str → Str → Maybe (Str, Str)

Prediction (second is correct):
  StrNull :: Str →      (unstructured)
  StrNull :: Str → Bool (structured)
```

**Example 2** The example below illustrates the difference between human understanding and machine prediction. It is relatively easy to observe that `strUnpack` is semantically opposite of `strPack` and should thus have the reverse signature `Str → String`. Both prediction models, however, fail to produce the correct answer.

```
Context:
  parseLogLine :: ( String → Bool )
               → LBS.ByteString
               → Maybe ( Day, SummaryI )
  logSummary :: Log → IO [ Summary ]
  strPack :: String → Str

Prediction (both wrong):
  strUnpack :: String → String (unstructured)
  strUnpack :: Str → Str       (structured)
```

## 6 Conclusion & Future Work

The structured prediction model we proposed, which leverages context information, achieves state-of-the-art performance for predicting types from names on our dataset. Our experiments also show that, the use of context information greatly improves the performance of the models.

There are multiple ways we can further improve the signature accuracy of our prediction model. For simplicity, our data processing phase considers type aliases (e.g., `type Name = String` to be different types. This makes the embedding for types larger and harder to learn, as we often see aliases that are not semantically close to each other in natural language). Also, when predicting the type signatures, we do not check to make sure the predicted types are actually in scope. Doing so is possible but requires much more effort to wrangle the Haskell compiler for our use. Furthermore, we could combine rule-based prediction (`main` should have type `IO ()`, for example) and model-based prediction to further improve signature accuracy.

In this paper, our evaluation metrics consider only "hard" measures like signature accuracy and structural accuracy. We could also include more "soft" measures like the top-$k$ accuracy, i.e, whether the ground truth is in the top-$k$ candidates produced by the model, and token error rate, i.e, the edit distance between the prediction and the ground-truth.

One possible application of our approach, provided that we can achieve better accuracy, is that the suggested type signature can be used as a metric of user's choice of identifier names. If the auto-completed type signature is not what the user expects, then it implies that the user might need to choose a better name.

Future work can also investigate the same task for different languages — especially those with different naming conventions and type system features — to understand the applicability of our models.

## References


Alvarez-Melis, D., and Jaakkola, T. S. 2017. Tree-structured decoding with doubly-recurrent neural networks. In *Proc. ICLR*.

Bird, S., and Loper, E. 2004. Nltk: the natural language toolkit. In *Proc. ACL on Interactive poster and demonstration sessions*.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.-r.; and Kohli, P. 2017. Robustfill: Neural program learning under noisy i/o. In *Proc. ICML*.

Dong, L., and Lapata, M. 2016. Language to logical form with neural attention. In *Proc. ACL*.

Gu, J.; Lu, Z.; Li, H.; and Li, V. O. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proc. ACL*.

Jones, S. P. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Kant, N. 2018. Recent advances in neural program synthesis. In *arXiv preprint arXiv:1802.02353*.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Ling, W.; Grefenstette, E.; Hermann, K. M.; Kočiskỳ, T.; Senior, A.; Wang, F.; and Blunsom, P. 2016. Latent predictor networks for code generation. In *Proc. ACL*.

Melis, G.; Dyer, C.; and Blunsom, P. 2018. On the state of the art of evaluation in neural language models. In *Proc. ICLR*.

Murali, V.; Qi, L.; Chaudhuri, S.; and Jermaine, C. 2018. Neural sketch learning for conditional program generation. In *Proc. ICLR*.

Polosukhin, I., and Skidanov, A. 2018. Neural program search: Solving programming tasks from description and examples. In *Proc. ICLR (Workshop Track)*.

Rabinovich, M.; Stern, M.; and Klein, D. 2017. Abstract syntax networks for code generation and semantic parsing. In *Proc. ACL*.

See, A.; Liu, P. J.; and Manning, C. D. 2017. Get to the point: Summarization with pointer-generator networks. In *Proc. ACL*.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Proc. NIPS*.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Proc. NIPS*.

Wadler, P., and Blott, S. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*.

Yin, P., and Neubig, G. 2017. A syntactic neural model for general-purpose code generation. In *Proc. ACL*.

Zoph, B., and Knight, K. 2016. Multi-source neural translation. In *Proc. ACL*.