Predicting Haskell Type Signatures From Names

Bowen Wang

June 28, 2018

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 extract from source code to facilitate program synthesis without explicit supervision. To ground our task more concretely, 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 use two different approaches: unstructured prediction, which is based on the sequence-to-sequence model, and structured prediction, which models the tree structure of the type signatures directly. Surprisingly, even though the structured prediction model is better at predicting the structures of the type signatures (60.46% structural accuracy), the unstructured prediction model achieves much better signature accuracy (23.71%).

1 Introduction

Recent years have witnessed the great success of deep learning in various domains, including image classification [8], machine translation [26], speech recognition [19] and so on. Such success has inspired researchers in programming languages and natural language processing to apply deep learning techniques to tackle program synthesis tasks previously deemed insurmountable, such as synthesizing code from natural language specification alone [27, 18]. Even though the application of deep learning to program synthesis has gained some success — we can now synthesize some nontrivial programs from natural language alone [18] — we still face the biggest obstacle in any application of deep learning: lack of data. In some sense, the task of program synthesis can be viewed as translating human intent into code. While there are many different forms of human intent used in programming, such as natural language, input-output examples, partial specifications, it is extremely hard to gather high-quality data for human intent along with the huge amount of code available on the Internet. For example, even though programmers often write comments when coding, they usually focus on high-level ideas rather than transcribing the functionality of the code in verbatim, thereby making it difficult to extract useful information for downstream tasks, such as program synthesis.

The difficulty in collecting high-quality data for program synthesis pushes us to think from another perspective: what useful information can we extract from programs per se for program synthesis? This paper tries to provide a possible answer: names and type signatures.

While most modern programming languages provide some notion of type, the role of type systems varies significantly across different languages. In strongly typed languages, i.e, languages where typechecking is used to prevent runtime exceptions, type signatures often contain important information that specify the behavior of an expression. In some cases, a type signature almost uniquely defines the semantics of a function. For example, consider the following Haskell type signature `Maybe a → a`. While placeholders like `undefined` and `error` could satisfy the given type signature, the only nontrivial function of type `Maybe a → a` is `fromJust`, which has the following definition:

```
fromJust          :: Maybe a -> a
fromJust Nothing  = errorWithoutStackTrace "Maybe.fromJust: Nothing"
fromJust (Just x) = x
```

Similarly, if a function has type `(a, b)` → `a`, it is very likely that the function takes the first element of the pair, which is exactly what Haskell function `fst` does. In addition to the importance of type signatures, we can often tell the corresponding type signature from the name of an identifier, thanks to the rich information encoded in identifier names. For instance, the name `listDiff` suggests that the function takes two lists and returns their difference, and is thus likely to have the type signature `[a]` → `[a]` → `[a]`. The connection between identifier names and their corresponding type signatures leads us to the question: **can we predict the type of an expression, given its name?**

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

Challenge Even with context, predicting type signatures from identifier names is a quite challenging task. Unlike a normal sequence-to-sequence task such as machine translation, the rigidity of type signatures requires exact matches rather than approximations. Moreover, as mentioned before, users can often define new types as they wish, which makes the prediction an even more daunting task. While the NLP community has developed some methods to deal with the notorious problem of out-of-vocabulary words [24, 6, 20], those methods are often applied in settings where out-of-vocabulary words are rare. In our task, since programmers who use strongly typed languages can often freely define new types as they wish, there are often many more out-of-vocabulary types appearing in a codebase that is not seen before. To make things worse, not all programmers follow the naming convention of a particular language and some might use names that are uninformative, such as `foo` or `bar`, or names that are generic and thus ambiguous, such as `applyFunc`, `get`, and `toFun`. Furthermore, unlike some other tasks that leverage the semantics of natural language such as machine translation, the semantic information functions in a different way in our task. 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, i.e, two sentences of similar meaning in English should be translated to sentences of similar meaning in Spanish. However, in our tasks, the opposite sometimes holds. Consider the following two set functions:

```
intersect :: Set a -> Set a -> Set a
union :: Set a -> Set a -> Set a
```

In terms of names, they are opposite of each other, yet they share the same type signature. Such phenomenon appears quite often in various programs, suggesting a more complicated semantic relationship between the input names and the output signatures. All these factors conspire to make the prediction a quite challenging task.

Our Approach While our approach works for any strongly typed programming language, to ground our task more concretely, we focus on predicting type signatures for Haskell, a functional language with a powerful type system. In addition, to make data collection and processing simpler, we only consider top-level function and variable definitions, although our approach can be easily extended to non-top-level identifiers.

We consider two different approaches to this problem, unstructured prediction and structured prediction. Unstructured prediction treats both the name and type signatures as a sequence of tokens and builds the model on top of the classic sequence-to-sequence model [23]. Structured prediction explicitly models the tree structure of the type signatures and thus is guaranteed to generate well-formed type signatures. Both models make use of context information, which consists of some number of previous type annotations. We mainly 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. Despite the lack of structure modeling, the unstructured prediction model is surprisingly better at predicting entire signatures, achieving 23.71% signature accuracy. The structured prediction model, on the other hand, does achieve better structural accuracy (60.46%) as expected.

2 Related Work

2.1 Neural Program Synthesis

With the advance in deep learning in recent years, neural program synthesis, which uses deep learning to tackle traditional program synthesis problems, has become a rapidly growing field [10]. The application of deep learning to natural language processing brings about powerful language models [13], thereby enabling researchers to leverage natural language information for program synthesis. Thus, unlike some of the traditional program synthesis techniques like combinatorial search [22], synthesis from input-output examples [7, 21], and type-directed synthesis [15, 16], neural program synthesis makes much more use of natural language. Several recent works focus on synthesizing code from natural language descriptions [5, 11, 27, 18] and mostly take the approach of structured prediction to better model the structure of a program. Dong and Lapata use a sequence-to-sequence model to generate tree output by adding special tokens to model depth-first tree generation [5]. Ling et al. propose latent predictor network to allow character-level generation of code pieces [11]. Yin and Neubig use a generation model that follows the grammar for python ASTs [27]. Rabinovich et al. use a modular neural network to generate output according to the abstract syntax description language [18], an approach most similar to ours.

More recently, researchers started to combine traditional synthesis techniques, such as searching and programming by examples, with neural networks in hope of further extending neural program synthesis capabilities. Robust Fill [4] combines structured prediction with searching based on examples to synthesize spreadsheet programs. More recently, Polosukhin and Skidanov propose tree beam search to refine the seq2tree generation results [17]. Murali et al. combine neural program generation with combinatorial search by training on program sketches [14].

The problem studied in this paper, which falls into the realm of neural program synthesis, was first proposed by Hempel [9]. In comparison, our dataset, though also collected from GitHub, is larger (1932 repos vs. 1000 repos, 401, 882 vs. 304,272 signatures). In terms of methodology, unlike the simple encoder-decoder model considered in [9] which ignores the structure of type signatures, we propose the structured prediction model to capture the tree structure of Haskell type signatures. We also make much better use of context information through attention and copying and our models have much better performance than the model discussed in [9].

2.2 Multi-attention

The complexity of our structured prediction model requires us to combine hidden states from and compute attention on different modules. Zoph and Knight concatenate hidden states when computing attention from multiple sources [29] while Zadeh et al. first compute multiple attentions for different hidden states and combine them at the end [28]. We mostly follow the approach used in [29] attend to multiple different sources.

2.3 Copying

Dealing with out-of-vocabulary (OOV) words has always been an obstacle in NLP applications such as text summarization and machine translation. For example, when summarizing a piece of text on President Donald Trump, the vocabulary gathered from training data might not have the world "Trump", and therefore the summarization will fail miserably without copying. 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 [24]. Later work such as [6] and [20] 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. [20] to copy types from context. However, unlike the text summarization task studied in [20], our prediction task may actually benefit from token repetitions, especially in the structured prediction model, as signatures like `Int` $\rightarrow$ `Int` and `a` $\rightarrow$ `a` $\rightarrow$ `a` are quite common. Therefore, we do not use the coverage loss proposed in [20] to punish repetitions.

3 Data Pipeline

As is true with every deep learning application, the data pipeline is of paramount importance. In this section, we describe how we collect and preprocess data for our task.

3.1 Data Collection

There is currently no existing dataset for this problem. To construct the dataset, we crawled all the existing Haskell repositories on with 10 or more stars on github. We ended up with 1932 repositories and, using a standard 80/10/10 split (i.e, 80% data for training, 10% each for validation and testing), we had 1546 repositories for training, 193 for validation and 193 for testing. All the `.hs` files in the repositories were passed to a Haskell parser and we ended up with 401,882 signatures for training, 39,040 signatures for validation, and 33,997 signatures for testing [1]. The dataset has 192,606, 18,149, and 18,212 unique type signatures in training, validation, and testing set, respectively. Among all type signatures in the dataset, the most common one is `IO ()`, which appears 10,189 times.

3.2 Data Preprocessing

Before we introduce the models, it is important that we discuss how we process the dataset to prepare for training. One of the important feature unique to Haskell is the typeclass, which allows polymorphism in an ad-hoc fashion [25]. For example, one might want to overload the `(+)` operator on a number of different types including `Int`, `Float`, `Double` and so on. Haskell programmer can easily deal with the overloading in a principled way by creating the `Num` typeclass, which has `(+)` as a method. Then for each type that `(+)` is applicable, one just needs to implement an instance of `(+)` for that particular type. In this way, Haskell programmers can easily extend operators like `(+)` to user-defined types by implementing the `Num` class instance. However, explicitly modeling typeclass would add more complexity and reduce the generalizability of our model to other functional languages that do not employ typeclass. Therefore, as in Hempel [9], we choose to remove typeclass constraints from type signatures. For example, the signature of the bind operator `>>=`, which is `Monad m ⇒ m a → (a → m b) → m b`, would simply be `m a → (a → m b) → m b` after removing typeclass constraints. We also normalize the type variables to lower case letters (i.e, the first type variable is renamed to a, the second to b, etc) to avoid unnecessary noise in the data. Under such normalization, the signature of `>>=` becomes `a b → (b → a c) → a c`.

We treat the input (function names) and the target (type signatures) differently. For input names, since Haskell programmers usually follow the camel case naming convention, we segment the names into tokens accordingly [2]. To achieve better generalizability, we also stem each token using the NLTK library [3]. 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 type signatures, we considered two different approaches: viewing type signatures as a sequence of string tokens without considering any structures (section 5) or exploiting the tree structure underlying Haskell type signatures (section 6). For example, under the former approach, the type `Int → ( Int , Int )` is a sequence of seven tokens: `Int`, →, `(`, `Int`, `,`, `Int`, `)` whereas under the latter approach, the type `Int → ( Int , Int )` is a tree shown in Figure 1.

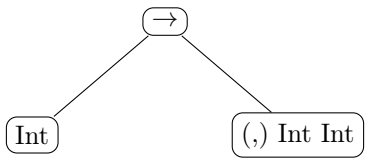

Figure 1: Tree representation of type `Int → ( Int , Int )`

3.2.1 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`

[1] The dataset will be made publicly available upon publication.
[2] Our segmentor can also handle other naming conventions

module, and the `Maybe` module is a file in the `Prelude` directory, then the module-qualified name of `fromJust` is the concatenation of token representations of `Maybe` and `fromJust`, whereas the path-qualified name of `fromJust` is the concatenation of token representations of `Prelude`, `Maybe`, and `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 contain 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 [3].

4 Overview of Two Approaches

We generally view the task of predicting type signatures given (qualified) identifier names as a machine translation problem with the identifier names being the source and the type signatures being the target. In section 5, we ignore the structure of type signatures and use a sequence-to-sequence to model similar to classic seq2seq model [23]. In section 6, we explicitly model the type signatures as binary trees and the model is, in principal, a sequence-to-tree model similar to the seq2Tree model proposed in [5]. However, the information encoded in the identifier names alone may not be sufficient to decode their type signatures and thus we also incorporate context information (names and type signatures that appear in the same file) in both the unstructured prediction model and the structured prediction model to achieve better accuracy.

5 Unstructured prediction

We begin with unstructured prediction, i.e., predicting the type signature as if it were a sequence. As mentioned in section 3.2, this requires us to segment the type signature into tokens and treat the type signature as a sequence of tokens. In this setting, the backbone of the model is the seq2seq model proposed in [23]. Later in this section, we discuss how we add attention and copying to the model as we take context into consideration.

5.1 Encoder

As described in 3.2, we segment and stem the function name s into a sequence of tokens $(x_i)_{1 \leq i \leq n}$. To encode the input sequence, we use a bidirectional LSTM. We concatenate the forward and backward vector of the final hidden state to obtain the initial hidden state for the decoder.

5.2 Decoder

The decoder mainly consists of a one-layer feed-forward network on top of an LSTM. A softmax layer is added on top of the feed-forward network to obtain the probability distribution of the output tokens (types).

5.3 Attention

Attention has proved to be highly successful at improving the performance of sequence-to-sequence tasks [2]. In essence, attention allows the decoder to "focus" on different parts of the input sequence based on the current hidden state, thereby achieving higher accuracy. We mostly follow the attention mechanism proposed by Loung et al. [12].

The attention alignment vector a_t is computed from the decoder hidden state h_t and each of the encoder hidden states $\bar{h}_s$ as

$$a_t(s) = \frac{\exp(\text{score}(h_t, \bar{h}_s)}{\sum_{s'} \exp(\text{score}(h_t, \bar{h}_{s'}))}$$

where $\text{score}(h_t, \bar{h}_s)$ a scoring function that measures how much h_t aligns with $\bar{h}_s$. We choose the scoring function to be

$$\text{score}(h_t, \bar{h}_s) = h_t^T W_a \bar{h}_s$$

for generality (W_a is a learnable parameter matrix).

[3] With the exception that two different project owners might have exactly the same path names

From the alignment vector a_t we can compute the context vector

$$c_t = \sum_s a_t(s) h_s,$$

which summarizes the attention information based on the current decoder hidden state. Given the context vector c_t and the decoder hidden state h_t, we compute the attentional hidden state $\tilde{h}_t = \tanh(W_c[c_t; h_t])$ where W_c is a matrix of weight parameters.

To obtain the output probability for each type at time step t, we compute

$$p_{vocab} = p(y_t | y_{<t}, x) = \text{softmax}(W_s \tilde{h}_t) \tag{1}$$

where W_s is another parameter matrix.

5.4 Incorporating Context

One of the key insights we have is that there should be connections between the type signatures of functions defined in the same file. Indeed, we observe that if the prediction model simply copies the previous type signature in the same file, it can achieve more than 20% accuracy, which is a quite high accuracy considering all the difficulties in predicting the correct signature mentioned in section 1 such as the requirement for exact match and ambiguity in the names. In addition, there are over 25% type signatures in the test set that contain out-of-vocabulary types, which means that the prediction accuracy can be at most 75% if we do not consider any unseen types that users defined or imported from other files or libraries (This could of course be done, but that requires integrating the system with each project's build system, which we try to avoid to keep the project simple and self-contained). Consider the following example: we are trying to predict the type signature of an identifier `tAvgPx`, in the module `FIX40`. The correct type signature is `FIXTag`, which is not in the vocabulary gathered from the training set. However, all the identifiers in this file has the same type signature. Thus, if we are able to copy type signatures from context, it is very likely that we can predict the type signature of `tAvgPx` correctly. By incorporating context into our prediction model, we hope to gather more information to enhance prediction and circumvent the notorious out-of-vocabulary word problem.

It is difficult to give a precise definition of context of an identifier. We take a naive approach and define the context of a top-level identifier f as the N preceding type signatures that appear in the same file as f, where N is a hyperparameter. If there are less than N such type signatures, we simply take the maximum of N and the number of type signatures that precede f. Of course there are more refined ways of considering the context for f. For example we could consider the types of identifiers used in the definitions of preceding top-level expressions. Such considerations are left for future work.

5.4.1 Context Signature Encoder

We first turn the context signatures into a sequence of tokens by simply turning each type signature in the context into a sequence of tokens and concatenate the sequences. An end token is added at the end of each signature to serve as a delimiter. A natural way to process the resulting sequence is to add another encoder. Similar to the encoder for the input sequence, we use a bidirectional LSTM to encode the context.

The final architecture we used is shown in figure 2. We use g to combine the final hidden states from the input name encoder and the context signature encoder to obtain the initial hidden state for the decoder. g is defined as follows:

$$g(h_e, h_c) = \tanh(W_d[h_e; h_c])$$

where W_d is a learnable parameter matrix.

5.4.2 Copying from context

As mentioned earlier, one of the advantages of incorporating context is that type signatures in the context provide a way of coping with the out-of-vocabulary types. Out-of-vocabulary prediction has always been one of the major difficulties in NLP applications and there have been numerous attempts at resolving this issue [24, 6, 20], most of which proposed some way of copying unknown

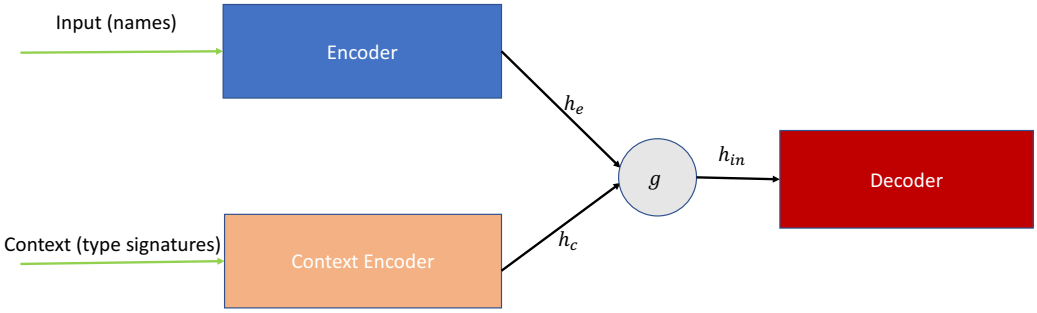

Figure 2: Model for incorporating context

words from certain context. We mostly follow the approach used by See et al. [20]. However, there are some crucial differences: the pointer-generator network proposed by See et al. [20] is used in the context of text summarization where the out-of-vocabulary words are copied directly from the input text. In our case, however, the out-of-vocabulary types live in a different space from the identifier names. Therefore, our copying mechanism are necessarily more complicated, as it involves dealing with two disparate kinds of data: names, which are texts, and type signatures, which consist of types.

To properly model generation and copying, we explicitly compute the probability of generation at time step t by agglomerating information from input name encoder and context encoder. More specifically, let c_t^* be the context vector computed from attention on the hidden states of the context encoder, and h_t^* be the context vector on the hidden states of the input name encoder. Then the probability of generation is given be

$$p_{\text{gen}} = \sigma(w_{h^*}^T h_t^* + w_{c^*}^T c_t^* + w_h^T h_t + w_x^T x_t + b)$$

where w_{h^*}, w_{c^*}, w_h, w_x and b are learnable parameters and x_t is the embedding of the decoder input.

Since the decoder either generates a token or copies a token from context, the final probability distribution over the extended vocabulary is given by

$$p(w) = p_{\text{gen}} \cdot p_{\text{vocab}}(w) + (1 - p_{\text{gen}}) \sum_{i:w_i=w} a_t(i),$$

where p_{vocab} is given by equation 1 and $a_t(i)$ is the amount of attention on the ith word of the context for the current output step t.

5.4.3 Loss Function

As with many sequence-to-sequence models, the unstructured prediction model use the negative log-likelihood loss, which has the following form

$$\text{loss} = \frac{1}{T} \sum_{t=0}^{T} -\log(p(y_t))$$

where y_t is the ground-truth token at time step t.

6 Structured Prediction

Unstructured prediction, which treats the type signatures as sequences of tokens, fails to exploit syntactic and semantic structure underlying Haskell type signatures, thereby generating many ill-founded type signatures. To move towards syntactic and semantic soundness in type signature generation, we explicitly model the binary tree structure of Haskell type signatures.

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. In contrast, `Int → Int` is an arrow type. One might argue that the grammar presented above seems arbitrary in that it distinguishes arrow from other type constructors. While it is certainly true that → is a type constructor, it has a unique position in the Haskell type system: a type represents a function if and only if it has arrow as its type constructor. We consider such distinction to be essential and therefore choose to explicitly model the arrow type constructor.

6.1 Type constructors and kinds

The rich type system of Haskell allows users to freely 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 $* \to *$, 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 naming collisions. For example, if there are two `Maybe`s in the dataset with kinds $*$ and $* \to *$, the first `Maybe` is represented as `Maybe#0` while the second `Maybe` is represented as `Maybe#1`.

6.2 Model Architecture

To model Haskell type signatures as binary trees according to the grammar presented in section 6, our model consists of several modules, each with its own functionality. The final model is shown in Figure 3.

6.2.1 Name Encoder

The name encoder processes the qualified function names and returns a vector representation of the given name. Similar to the input encoder used in section 5.1, the name encoder consists of a bidirectional LSTM.

6.2.2 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 [5]. It parses the type signature as a binary tree and processes each node in DFS order. Given a context containing n (name, signature) pairs, E_{cname} processes the names and outputs the hidden state h_{cname}. Notice that, unlike the unstructured prediction model which concatenates all context signatures due to the sequence-to-sequence modeling, E_{name} processes each signature separately and average the hidden states at the end to produce the context name hidden state h_{cname}. Similarly, E_{csig} takes in the type signatures as trees and outputs two things: h_{csig}, the average final hidden states of the type signatures, and $h_{cstates}$, the hidden states of all the nodes in the n type signatures. $h_{cstates}$ will later be used to compute the attention mask for the decoder.

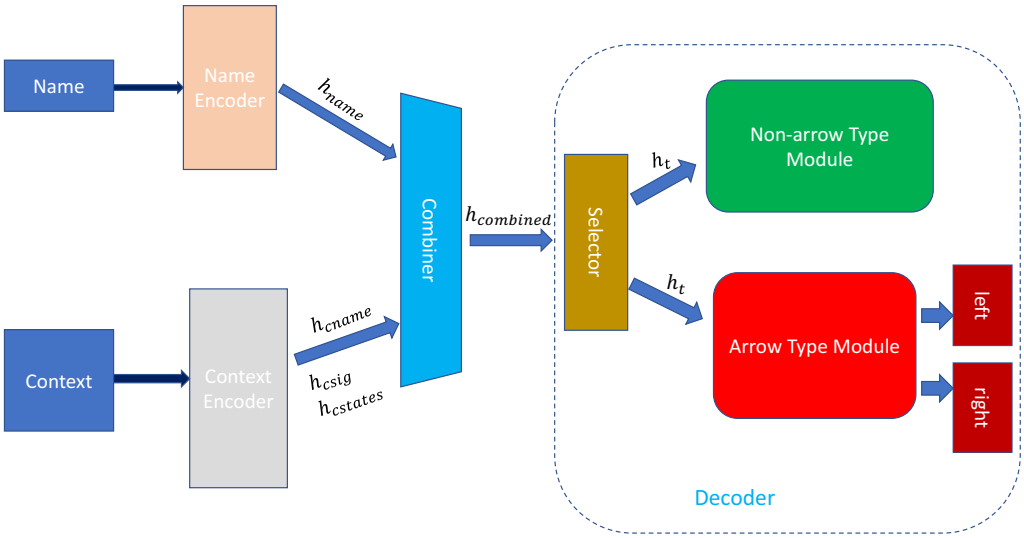

Figure 3: **Structured prediction model.** Given input name and context, the model produces the input name hidden state h_{name}, context name hidden state h_{cname}, context signature final state h_{csig}, and context signature hidden states $h_{cstates}$. The combiner combines h_{name}, h_{cname}, and h_{csig} to produce the combined hidden state $h_{combined}$ for decoder to decode from. The decoder selects which module to use according to the output of the module selector and starts generating output accordingly.

6.2.3 Combiner

The combiner module, similar to that proposed in [29], 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 $h_{combined}$. We consider two different ways of combining the hidden states:

1. (Weighted Sum)

$$h_{combined} = \tanh(W_{name}h_{name} + W_{cname}h_{cname} + W_{csig}h_{csig})$$

   where W_{name}, W_{cname}, and W_{csig} are learnable parameters.

2. (Projection)

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

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

6.2.4 Decoder Overview

Compared to the relatively simple encoder modules, the decoder is much more complex. The grammar presented in section 6.1 suggests a natural 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.

6.2.5 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 $p_{base}, p_{arrow} = \text{softmax}(f_T(e_t, h_t))$ where f_T is a two-layer feed-forward network with ReLu nonlinearity. The module with larger probability is chosen for the next step of generation.

6.2.6 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 [29]. 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 $p_{vocab} = \text{softmax}(f_T(h_t))$ 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 [20], is given by

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

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 6.2.2). Thus, the final probability over the tokens are computed by

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

When the context is not available, the module computes the output token probability in a similar fashion to the decoder in unstructured prediction: the probability is computed 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_{final} available, the next token a_{next} is selected as $\text{argmax}(p_{final})$. 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.

6.2.7 Arrow Type Module

The arrow type module is in charge of generating a type using the `Arrow Type Type` grammar. 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 [4]. The generation starts with a special start token. Notice that the hidden state for generating the right tree is a combination of the left hidden state and the parent hidden state. Such arrangement follows the idea of parent feeding used in [5] and also allows the model to carry information from the left subtree to facilitate the generation of the right subtree, since the left subtree often contain information vital for the the generation of the right subtree. For example, consider the type signature `(a → b) → a → b`, the left subtree `a` *to* `b` contains the types that the right tree `a → b` needs and therefore we believe that carrying such information over is conducive to generating better type signatures.

6.2.8 Controlling Recursion Depth

Unlike the classic sequence-to-sequence model [23], our model does not use a special end token to signal the end of generation. Instead, the generation stops when a complete tree has been generated. However, due to the recursive nature of the decoder, it is possible that, when not well-trained, the decoder could keeps generating subtrees infinitely. To prevent this, we add a hyperparameter `rec_depth` to control the recursion depth of the decoder.

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

Model	Signature Accuracy (%)	Structural Accuracy (%)	Well-formedness (%)
Hempel [9]	10.00	–	–
Baseline	23.39	49.78	100
Unstructured	**23.71**	42.23	65.99
Structured	7.92	**60.46**	100
Human	25.33	57.33	100

Table 1: Overview of final results

6.2.9 Loss Function

Unlike the relatively simple unstructured prediction model that only use the negative log-likelihood loss on the target tokens, the structured prediction also incorporates a topology loss, similar to that proposed in [1],

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

where $p(m_i)$ denotes the probability of choosing module i $(i = 0, 1)$ at time step t. The structural loss pushes the model the 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 λ is a hyperparameter.

7 Experiments

We conducted multiple experiments for both unstructured prediction model and the structured prediction model to study the influence of some critical hyperparameters such as the number of context names and signatures, as well as different choices for module structures in the structured prediction model. However, due to time and resource limitations, we were unable to study the influence of 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 to optimize for both unstructured and structured models. The learning rate for unstructured prediction model is set to 3×10^{-4} and is reduced to half of its value when the dev loss plateaus. For structured prediction model, we use a fixed learning rate of 2×10^{-4}.

8 Results

We mainly measure the performance of our models on the following three criteria: signature accuracy, structural accuracy, well-formedness. **Signature accuracy** refers to the percentage of correct signature predictions. A predicted signature must match the ground truth exactly to be considered correct. **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 6.1. For example, if the ground truth is `Int` → `Int` and the prediction is `String` → `Int`, the prediction is structurally correct. **Well-formedness** refers to the percentage of the predicted signatures that follow the grammar presented in section 6.1. Note that this only applies to the unstructured prediction model, as structured prediction model is guaranteed to generate well-formed type signatures. An overview of the final results is presented in Table 1. Table 1 also includes the result reported in [9], which does not measure structural accuracy and well-formedness.

8.1 A Strong Baseline

As mentioned in section 5.4, there is a very strong rule-based baseline: copying the previous signature in the same file. In the case that there is no preceding type signatures, this baseline predicts nothing. The baseline is able to achieve 23.39% signature accuracy and 49.78% structural accuracy on the test set.

Context Num	Signature Accuracy (%)	Structural Accuracy (%)
0	5.15	20.45
1	17.13	30.48
2	23.01	39.07
3	23.23	40.20
4	23.03	40.79
5	22.62	39.22

Table 2: Impact of the number of context annotation used on the unstructured prediction model

Context Num	Signature Accuracy (%)	Structural Accuracy(%)
0	3.31	50.35
1	7.10	62.04
2	7.84	61.73
3	7.49	61.74
4	7.19	61.51
5	7.22	61.00

Table 3: Impact of the number of context annotations used on the structured prediction model

8.2 Unstructured Prediction Results

As shown in Table 1, the best result for unstructured prediction is 23.71% signature accuracy, 42.23% structural accuracy, and 65.99% well-formedness. In addition, we also study the influence of context on the performance of the model, as context is an important component of the model. The result is presented in Table 2.

8.3 Structured Prediction Results

As shown in Table 1, the best result for unstructured prediction is 7.92% signature accuracy, 60.46% structural accuracy. The structured prediction model is guaranteed to have 100% well-formedness. Similar to the structured prediction model, we are interested in how the amount of context information available affects the performance of the model. Since the loss function for structured prediction model consists of both the token loss and the topology loss, i.e, ($loss = loss_{token} + \lambda loss_{topo}$), we are interested in how the hyperparameter λ affects the performance of the model.

First we keep $\lambda = 1$ and vary the number of context annotations. The result is shown in Table 3.

Then we keep the number of context annotations used to 3 and vary λ. The result is show in Table 4.

8.4 A Human Perspective

In addition to tackle the prediction task using machine learning techniques, we are also curious how a Haskell programmer, given the same information, would perform on the same task. More specifically, we have one researcher (the author) who is familiar with Haskell complete 300 predictions. For each prediction task, the researcher is given the module name, the identifier name, and three previous annotations (including both name and signature). The tasks are randomly selected from the test set used for the machine learning models. The resulting signature accuracy and structural accuracy are 25.33% and 57.33%, respectively.

9 Analysis

The experiment results are quite surprising. In this section, we shall compare and analyze the performances of unstructured prediction model and structured prediction model, as well as the baseline and the human benchmark.

λ	Signature Accuracy (%)	Structural Accuracy (%)
0.5	7.45	61.75
1	7.49	61.74
5	7.69	61.60
10	7.28	61.56
100	7.73	62.1

Table 4: Impact of λ on structured prediction model

9.1 Baseline

Even though discussed in section 5.4, it is worth emphasizing the effectiveness of the rule-based baseline that simply copies the previous type signature. As shown in Table 1, our machine-learning-based model barely beats such a naive baseline. Considering that our model uses context that contains up to three type annotations whereas the baseline only needs one annotation, the simple baseline works really well. Again, the performance of the baseline indicates the connection between type annotations that are spatially close to each other and the importance of context information.

9.2 Human Benchmark

In terms of signature accuracy, the human performance (25.33%) is the highest in Table 1. However, it should be noted that the human benchmark is produced by a single researcher who also designs the models on a small test set consisting of only 300 type annotations. Therefore, it is expected that the numbers we collected here cannot accurately represent how Haskell programmers in general would perform on this task. Nevertheless, the human benchmark provides some insight into the difficulty of the task — even the researcher who is very familiar with the task and understands the effectiveness of the baseline, i.e, copying the previous signature, only achieved 25% signature accuracy, which is not much higher than the performance of the baseline.

9.3 Unstructured Prediction

The unstructured prediction model achieved 23.71% signature accuracy, which, by itself, may not be surprising given that the rule-based baseline also has a signature accuracy of 23.39%. However, if we view the signature accuracy in the context of the other two measures, 42.23% structural accuracy and 65.99% well-formedness, the 23.71% signature accuracy seems really good. Conditioned on predicting a well-formed signature, the model has a 64% chance of producing a structurally correct signature. In comparison, the baseline, which copies the previous signature directly, although always produces a well-formed signature, only has a 50% chance of producing a structurally correct signature. Similarly, the unstructured prediction model produces the correct signature over 56% of the time given that it predicts the structure of the signature correctly. The results suggest that the unstructured prediction model has learned the semantics of type signatures well but falls a bit short on getting the syntax right.

9.4 Structured Prediction

The structured prediction model produced disappointing results. We expect that it should outperform the relatively simple unstructured prediction model by a large margin. However, the 7.92% signature accuracy suggests quite the opposite. We notice that the structured prediction model tends to make more "simple" mistakes than the unstructured prediction model. For example, when the context signatures are all `Int` → `HappyReduction`, it is pretty clear that the model should just copy that signature for the prediction. However, the model would sometimes produce results like `[Int]` → `HappyReduction`. On the other hand, the structured prediction model does have the highest structural accuracy in Table **??** and does much better in that regard than the unstructured prediction model, which is expected given that the structured prediction model explicitly models the tree structure of the type signatures. From Table 2 and Table 4 we can see that the important hyperparameters like context number and topology loss factor λ do not impact the performance of

the model significantly. It remains to be seen whether the structured prediction model can achieve a good signature accuracy.

9.5 Context

As mentioned in section 5.4, the context information is crucial because it not only provides information about new types that are not seen in the training data, but also allows one to boost the probability of the types that appear close to the current type annotation through attention. Indeed, the usefulness of context information is clearly shown in Table 2 and Table 3. Both tables reveal that, if we do not include any context, both unstructured and structured prediction model perform quite poorly (5.15% and 3.31% signature accuracy, respectively). In contrast, even if we just use the previous type annotation as context, the performance of both models improve dramatically (5.15% $\rightarrow$ 17.13% for unstructured prediction model, 3.31% $\rightarrow$ 7.10% for structured prediction model), again indicating the effectiveness of context information. However, we also notice that the performance starts to decrease once the number of context annotations reaches three. We suspect that this is because the previous three type signatures usually contain the sufficient information for the model see the new types and understand what types are "important" for the prediction and having more signatures in the context makes it harder to the model to learn to copy the correct types. Indeed, we investigated the relation between number of signatures in the test set that contain unseen types and the number of context annotations used (types that appear in the context are considered known types). The results are shown in Table 5.

Context Num	% signatures containing unseen types
0	25.43
1	12.16
2	10.69
3	10.06
4	9.66
5	9.39

Table 5: Relation between number of context annotations and number of unseen types

It is clear that adding context greatly reduces the number of signatures that have unseen types. The results also reveal that, when the number of context annotations reach two, adding more context information has diminishing returns, which echoes with the performance of the model shown in Table 2 and Table 3.

9.6 Name qualifications

As mentioned in section 3.2.1, we consider two ways of qualifying identifier names — module-qualified name and full-path qualified name. We investigate whether the two ways of qualification has an impact on the performance of the model. The result is shown in Table 6. For unstructured prediction, using qualification yields better results and module-qualified names perform better than path-qualified names. For structured prediction, however, using names without any qualification works the best, although the performance is very close to that of module-qualified names.

10 Conclusion & Future Work

In this paper, we study the problem of predicting type signatures from identifier names. We use two different approaches: unstructured prediction, which ignore the structure of type signatures and treat both identifier names and type signatures as sequences of tokens, and structured prediction, which considers both the syntax and the semantics of type signatures. The unstructured prediction model has better performance in terms of signature accuracy (23.71%) while the structured prediction model has higher structural accuracy (60.46%). For both of our models, we study the influence of context information, which consists of previous type annotations of the signature to be predicted. Our experiments show that, for both structured and unstructured prediction, the use of context information greatly improves the performance of the models.

Model	Qualification	Signature Accuracy (%)	Structural Accuracy (%)
Unstructured	No qualification	21.22	38.02
Unstructured	Path-qualified	22.66	42.35
Unstructured	Module-qualified	23.71	42.23
Structured	No qualification	7.96	61.66
Structured	Module-qualified	7.92	60.46
Structured	Path-qualified	7.36	58.29

Table 6: Impact of different ways of qualification on the performance of prediction models

However, it remains unclear why the unstructured prediction model performs much better than the structured prediction model in terms of signature accuracy, especially given that structured prediction model yields a much higher structural accuracy. It could be the case that the architecture proposed in this paper cannot efficaciously address the prediction task. Future work can focus on understanding the problems with the current architecture and propose better ones to further improve the signature accuracy.

The signature accuracy can be potentially further improved by resolving aliases and checking the types that are in scope. In the data processing phase, we consider aliases to be different types to simplify the problem, but 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 (`Name = String` for example). Also, when predicting the type signatures, we didn't 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. Future work on the same topic may benefit from these two considerations.

More broadly, future work can also investigate the same task for different languages, especially those with different naming conventions to understand the applicability of our models.

A more interesting direction is to consider how the predicted type signatures can help with program synthesis in general. Our results are clearly not good enough for the models to be used for downstream tasks like synthesizing a whole program. However, we could imagine incorporating more natural language information such as comments or specifications would probably improve the performance and make the models useful for other tasks. In a similar vein, we can imagine a personalized interactive system where a programmer can actively correct mistakes made by the prediction model and give feedbacks to improve the model.

References

[1] David Alvarez-Melis and Tommi S Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. In *Proc. ICLR*, 2017.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *Proc. ICLR*, 2015.

[3] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proc. ACL on Interactive poster and demonstration sessions*, 2004.

[4] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proc. ICML*, 2017.

[5] Li Dong and Mirella Lapata. Language to logical form with neural attention. In *Proc. ACL*, 2016.

[6] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor OK Li. Incorporating copying mechanism in sequence-to-sequence learning. In *Proc. ACL*, 2016.

[7] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proc. POPL*, 2011.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. CVPR*, 2016.

[9] Brian Hempel. Context-sensitive prediction of haskell type signatures from names. 2017.

[10] Neel Kant. Recent advances in neural program synthesis. In *arXiv preprint arXiv:1802.02353*, 2018.

[11] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. In *Proc. ACL*, 2016.

[12] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Proc. EMNLP*, 2015.

[13] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. In *Proc. ICLR*, 2018.

[14] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *Proc. ICLR*, 2018.

[15] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proc. PLDI*, 2012.

[16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proc. PLDI*, 2016.

[17] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. In *Proc. ICLR (Workshop Track)*, 2018.

[18] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proc. ACL*, 2017.

[19] George Saon, Hong-Kwang J Kuo, Steven Rennie, and Michael Picheny. The ibm 2015 english conversational telephone speech recognition system. In *arXiv preprint arXiv:1505.05899*, 2015.

[20] Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. In *Proc. ACL*, 2017.

[21] Rishabh Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. In *Proc. VLDB*, 2016.

[22] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proc. ASPLOS*, 2006.

[23] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Proc. NIPS*, 2014.

[24] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proc. NIPS*, 2015.

[25] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*, 1989.

[26] Mingxuan Wang, Zhengdong Lu, Jie Zhou, and Qun Liu. Deep neural machine translation with linear associative unit. In *arXiv preprint arXiv:1705.00861*, 2017.

[27] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proc. ACL*, 2017.

[28] Amir Zadeh, Paul Pu Liang, Soujanya Poria, Prateek Vij, Erik Cambria, and Louis-Philippe Morency. Multi-attention recurrent network for human communication comprehension. In *Proc. AAAI*, 2018.

[29] Barret Zoph and Kevin Knight. Multi-source neural translation. In *Proc. NAACL*, 2016.