Predicting Haskell Type Signatures From Names

Bowen Wang

July 26, 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 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 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. The structured prediction model outperforms the unstructured prediction model in terms of both signature accuracy (27.28% vs. 23.98%) and structural accuracy (61.65% vs. 41.44%).

1 Introduction

Recent years have witnessed the great success of deep learning in various domains, including image classification [8], machine translation [32], speech recognition [23] 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 [33, 22]. 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 [22] — 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, and 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 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, the only nontrivial function that has type signature `(a, b) → a` is `fst`, which has the following definition:

```
fst       :: (a,b) -> a
fst (x,_) =  x
```

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]` $\rightarrow$ `[a]` $\rightarrow$ `[a]`.

Our Goals 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?** A successful model for this task could, in the future, be used as part of an interactive programming system that, for example, automatically suggests type annotations while the user is typing, or flags identifiers that are not semantically consistent with a user-defined type annotation.

Formally, let Σ be the space of identifier names and T be the space of type signatures. We aim to learn a mapping $f : \Sigma \rightarrow T$. 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. Therefore, 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 Σ, parametrized by context c.

Challenges Even with context, predicting type signatures from identifier names is a quite challenging task. The four main challenges are:

1. Unlike a normal sequence-to-sequence task such as machine translation, the rigidity of type signatures requires exact matches rather than approximations.

2. As mentioned before, users can, and often do, define new types as they wish, which makes prediction even more daunting. While the NLP community has developed some methods to deal with the notorious problem of out-of-vocabulary words [30, 6, 25], those methods are often applied in settings where out-of-vocabulary words are rare. In our task, since programmers who use strongly typed languages often define new types, there are often a large number of out-of-vocabulary types in a codebase.

3. To make things worse, with respect to identifier names, 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`.

4. 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. This phenomenon appears quite often in various programs, suggesting a more complicated semantic relationship between the input names and the output signatures.

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 [11], a functional language with a powerful type system. In this work, to make data collection and processing simpler, we only consider top-level function and variable definitions, although our approach can be extended to non-top-level identifiers with some modifications of context.

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 [28]. 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 in our

approach consists of some number of previous type annotations. 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. The structured prediction model has both higher signature accuracy (27.28%) and higher structural accuracy (61.65%), as expected. Both models beat the strong baseline proposed in [9], which simply copies the previous signature in the same file.

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 [12]. The application of deep learning to natural language processing brings about powerful language models [16], thereby enabling researchers to leverage natural language information for program synthesis. Thus, unlike some of the traditional program synthesis techniques like combinatorial search [27], synthesis from input-output examples [7, 26], and type-directed synthesis [19, 20], neural program synthesis makes much more use of natural language. Several recent works focus on synthesizing code from natural language descriptions [5, 14, 33, 22] and mostly take the approach of structured prediction to better model the structure of a program. 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 [5]. Ling et al. propose latent predictor network to allow character-level generation of code pieces [14]. Yin and Neubig use a generation model that follows the grammar for Python abstract syntax trees (ASTs) [33]. Rabinovich et al. use a modular neural network to generate output according to the abstract syntax description language [22], 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.

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 [21]. Murali et al. combine neural program generation with combinatorial search by training on program sketches [17].

The problem studied in this paper, which falls into the realm of neural program synthesis, was first proposed by Hempel [9]. To improve the performance of the model, we gathered a larger dataset (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

Attention [2] in the encoder-decoder model allows the decoder to focus on certain parts of the encoder output at each step of decoding and has proved to be highly effective at sequence-to-sequence tasks [2, 15, 29]. The complexity of our structured prediction model, however, requires us to combine hidden states from and compute attention on multiple modules. Zoph and Knight concatenate hidden states when computing attention from multiple sources [35] while Zadeh et al. first compute multiple attentions for different hidden states and combine them at the end [34]. We mostly follow the approach used in [35] to 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. To address this problem, researchers have considered copying OOV words from input. Vinyals et al. propose the pointer network to copy

Frequency of Type in Signatures	#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 $t \in T$.

from input according to the attention probability [30]. Later work such as [6] and [25] 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. [25] to copy types from context. However, unlike the text summarization task studied in [25], 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 [25] 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

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% for validation, and 10% for 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. 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 3.2.1.

3.2 Data Preprocessing

Now we discuss how we process the dataset to prepare for training.

3.2.1 Normalize Signatures

Multi-signatures are de-sugared to consecutive single-signatures. For example, if we have multi-signature

```
union, intersection :: Set a -> Set a -> Set a
```

we de-sugar it to

```
union :: Set a -> Set a -> Set a
intersection :: Set a -> Set a -> Set a
```

To simplify the task a bit further, we remove the qualification of types. For example, `Prelude.Maybe` is simplified to `Maybe`.

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. A good example is the `lookup` function in `Data.Map`, which has signature

```
lookup :: k -> Map k a -> Maybe a
```

where `k` indicates that the type variable is the key type. The signature is then normalized to

```
lookup :: a -> Map a b -> Maybe b
```

where `k` is normalized to `a` and `a` is normalized to `b`.

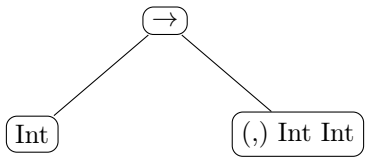
3.2.2 Type Classes

One of the important features unique to Haskell are typeclasses, which allows types that share a common set of operations (methods), as described explicitly with types and implicitly with intended invariants [31]. For example, one might want to overload the `(+)` operator on a number of different types including `Int`, `Float`, `Double` and so on. Haskell programmers deal with 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 typeclasses would add more complexity (make type signatures longer and thus harder to predict, for example) 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 operator `(+)`, which is `Num a ⇒ a → a → a`, would simply be `a → a → a` after removing typeclass constraints. The signature of `lookup` function discussed in section 3.2.1 also has typeclass constraint removed. The original signature is `Ord k ⇒ k → Map k a → Maybe a`.

3.2.3 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 [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 [2]: <`Int`; →; `(`; `Int`; `,`; `Int`; `)`> whereas under the latter approach, the type `Int → ( Int , Int )` is a tree shown below.



Notice that type application `(,) Int Int` remains as is, since we only treat arrow as a special type constructor and construct the tree structure accordingly. 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.

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

[2] We use the notation <`a`; →; `b`> to represent type signature `a → b`.

3.2.4 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 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 (seq2seq) model similar to the classic seq2seq model [28]. In section 6, we explicitly model the type signatures as binary trees and the model is, in principle, 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. Unlike [9], which merges context and identifier names to feed to the encoder, we use separate encoders for context in both models.

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 [28]. 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. LSTM (Long Short-Term Memory) is a type of recurrent neural network that uses gated connection to allow better gradient flow [10]. Bidirectional LSTM [24] is a variation of LSTM that has both forward and backward hidden states. The update at time step t is given by

$$\overrightarrow{h_t}, \overleftarrow{h_t} = LSTM(\overrightarrow{h_{t-1}}, \overleftarrow{h_{t-1}}, e_t)$$

where e_t is the embedding of x_t. We concatenate the forward and backward final hidden states 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. At time step t, the decoder produces the prediction by first computing the update

$$h_t = LSTM(h_{t-1}, e_t)$$

where e_t is the embedding of the previous prediction token, and then feeding h_t to a softmax layer on top of the feed-forward network to obtain the probability distribution $p(w)$ of the output tokens (types). The prediction token is then given by $y_t = \text{argmax}(p(w))$.

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

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. [15].

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 $\text{score}(h_t, \bar{h}_s)$ is 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), as suggested in [15].

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. Later in section 5.4 when we include context type signatures, $\tilde{h}_t$ becomes $\tanh(W_c[c_t; c'_t; h_t])$ where c'_t is the context vector for context type signatures.

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, Hempel shows that if the prediction model simply copies the previous type signature in the same file, it can achieve more than 20% accuracy [9], 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 have the same type signature. [4] 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.

There are many ways 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 in the file, we simply take all the 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.

[4] There is a number of such files in the dataset, some of which look like auto-generated files.

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 1. 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 and $[a; b]$ denotes the concatenation of a and b.

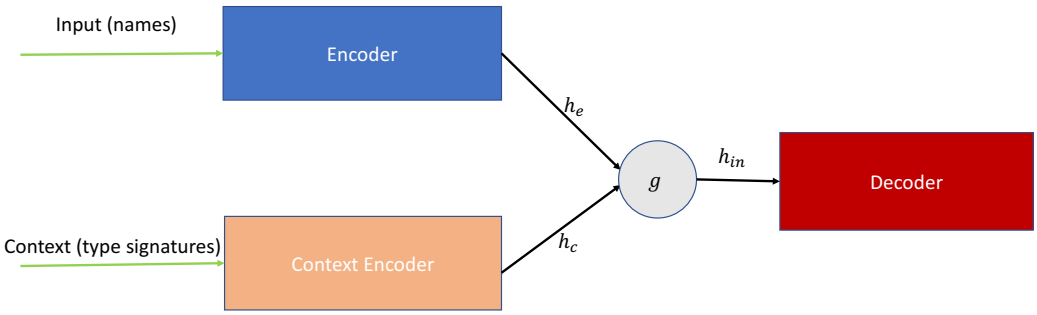


Figure 1: Model for incorporating context.

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 [30, 6, 25], most of which proposed some way of copying unknown words from certain context. We mostly follow the approach used by See et al. [25]. However, there are some crucial differences: the pointer-generator network proposed by See et al. [25] 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 text, 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-formed 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 collapsing type constructor with different arity to the same name. 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 2.

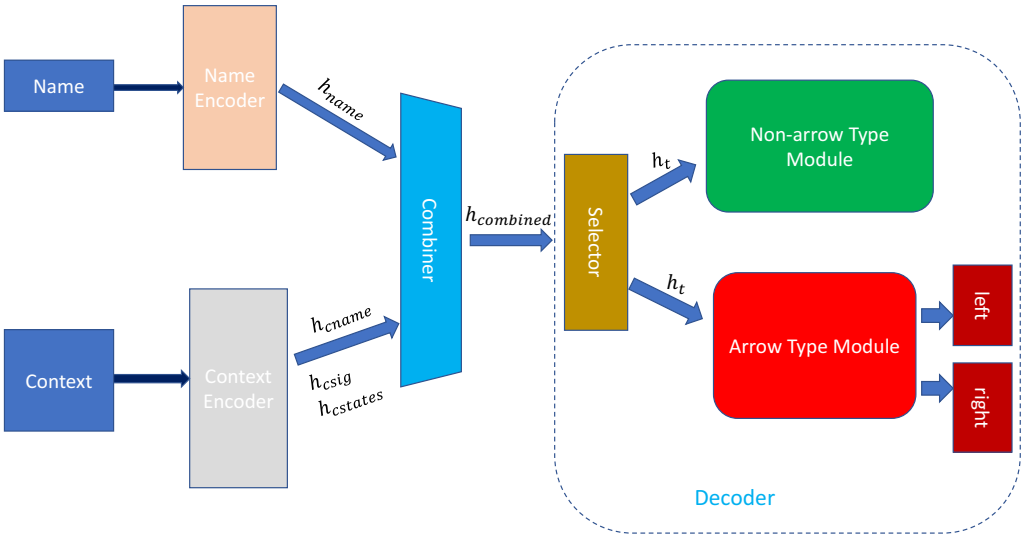


Figure 2: **Structured prediction model.** Given input name and context, the model produces the input name hidden state h_{name}, context name hidden state h_{cname}, context signature final state h_{csig}, and context signature hidden states $h_{cstates}$. The combiner combines h_{name}, h_{cname}, 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.

6.2.1 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 depth-first search 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 averages 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.

6.2.3 Combiner

The combiner module, similar to that proposed in [35], 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 (rectified linear unit) 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 [35]. 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 [25], 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 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 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}.

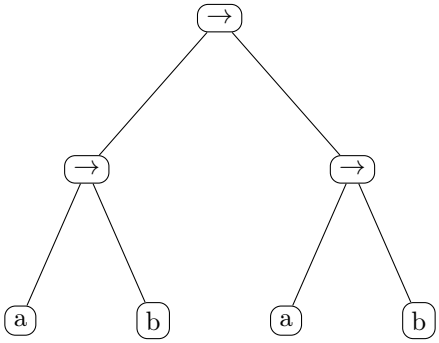
With the final probability over tokens $p(w)$ available, the next token a_{next} is selected as $\text{argmax}(p(w))$. If a_{next} is not a ground type, we recursively call the decoder with the corresponding number of steps to make sure a valid type is generated.

6.2.7 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. [5] 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. This 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 contains information vital for the generation of the right subtree. For example, consider the type signature `(a → b) → a → b`, which has the following tree structure:



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 [28], 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 keep generating subtrees infinitely. 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 that have depth larger than 6.

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 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 λ is a hyperparameter. Note here that we follow [1] and add a λ multiplier on the topology loss, although combining the two loss functions with a weighted sum, i.e, $\lambda loss_{token} + (1 - \lambda) loss_{topo}$ might also be worth considering.

7 Evaluation

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 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 [13] to optimize for both unstructured and structured models. These hyperparameters are chosen according to [22] and [33]. The learning rate

[5] 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	23.98	41.44	62.15
Structured	**27.28**	**61.65**	100
Human	25.33	57.33	100

Table 2: Overview of final results.

for unstructured prediction model is set to 3×10^{-4} and is reduced to half of its value when the dev loss plateaus, i.e, when the dev loss does not decrease for two epochs. For structured prediction model, we use a fixed learning rate of 2×10^{-4}. The models are implemented in PyTorch [18] and the whole implementation has roughly 3000 lines. The code and dataset are available at https://github.com/bowenwang1996/predictTypeSignature.

We measure the performance of our models on the following three 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 6.1. For example, if the ground truth is `Int` $\rightarrow$ `Int` and the prediction is `String` $\rightarrow$ `Int`, the prediction is structurally correct.

3. **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.

7.1 Summary of Results

An overview of the final results is presented in Table 2. Table 2 also includes the results reported in [9], which does not measure structural accuracy and well-formedness. [6]

The results mostly align with our expectations: the structured prediction model performs the best in all measures. To better understand the performance of each model, we study the prediction results from the following perspectives:

Overlap between baseline, unstructured prediction, and structured prediction. Figure 3 shows that there are 4962 correct predictions shared by all three models, which account for more than 50% of the correct predictions of each model.

Performance of models based on frequency of type signatures. We also investigated the performance of each model based on the frequency of type signatures in the dataset (see Table 1). The result is shown in Table 3. Note that, in Table 3, the numbers refer to the occurrences of type signatures in a certain category. If a type signature belongs to, say, category 2, and it occurs 500 times in the test set, it is counted as 500 towards #Signatures instead of 1.

Somewhat surprisingly, both unstructured prediction model and structured prediction model are able to predict some number of category 6 type signatures correctly, which means that both models have learned, to some extent, how to generate type signatures without context information. It is also worth noting that both unstructured prediction model and structured prediction model are much better at predicting `IO ()` than the baseline, which again reveals that the two models have learned some semantic information for names like `main`.

Performance of models based on depth and length of type signatures We also study the distribution of number of correct predictions according to the size of type signatures. Here we consider two measures of size: depth and length. Depth refers to the depth of a type signature as a tree. For example, `Int` has depth 1 and `Int` $\rightarrow$ `Int` has depth 2. The distribution of correct predictions according to depth is shown in Figure 4a. Length, on the other hand, refers to the number of tokens a type signature has when considered as a sequence of tokens. For example, `Int`

[6] The dataset used in [9] is different but similar enough for comparison.

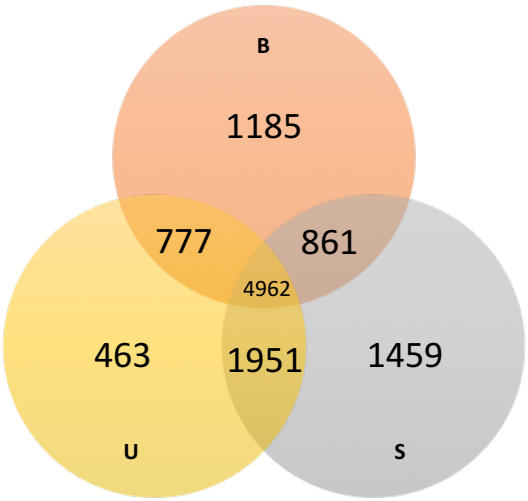


Figure 3: Overlap of correct prediction between baseline (B), unstructured prediction (U), and structured prediction (S).

Category \ Model	Baseline	Unstructured	Structured	#Signatures
Category 1 (10193)	108	666	726	866
Category 2 (1000 - 3613)	481	543	558	957
Category 3 (100 - 1000)	3403	3438	3607	4978
Category 4 (10 - 100)	1945	2081	2405	5674
Category 5 (2 - 10)	2030	1396	1824	11397
Category 6 (1)	0	29	113	10125
Total	7967	8153	9233	33997

Table 3: Number of correct predictions based on frequency of type signatures.

has length 1 while `Int` $\rightarrow$ `Int` has length 3. The distribution of correct predictions according to length is shown in Figure 4b. In both Figure 4a and Figure 4b, rows where none of the prediction models gets anything right are redacted; this is why the total number of signatures are less than 33997, the size of the test set.

The general trend, as shown in Figure 4a and Figure 4b, is that the larger (deeper or longer) the type signature is, the harder it is for the model to make a correct prediction.

Depth \ Model	Baseline	Unstructured	Structured	#Signatures
1	5058	5352	5583	10042
2	1916	2011	2656	11802
3	688	666	889	7745
4	212	111	92	2791
5	68	8	13	948
6	16	3	0	383
7	3	1	0	146
8	5	1	0	68
9	1	0	0	38
Total	7967	8153	9233	33963

(a) Distribution of correct predictions according to depth of type signatures.

Length \ Model	Baseline	Unstructured	Structured	#Signature
1	3156	3190	3200	4633
2	1419	1839	2086	3567
3	860	701	859	3606
4	923	1377	1750	4914
5	380	206	259	2764
6	329	475	638	3230
7	194	82	153	2235
8	169	81	106	1571
9	101	59	52	1400
10	93	37	38	950
11	69	19	12	1022
12	54	14	13	685
13	28	8	6	635
14	74	35	35	434
15	40	22	20	464
16	27	3	0	333
17	13	2	4	285
18	6	0	0	166
19	10	2	2	197
20	6	0	0	138
21	1	0	0	133
22	0	1	0	76
23	2	0	0	85
24	4	0	0	63
26	1	0	0	42
27	1	0	0	49
28	1	0	0	33
29	1	0	0	34
34	1	0	0	12
42	2	0	0	6
51	1	0	0	3
54	1	0	0	2
Total	7967	8153	9233	33767

(b) Distribution of correct predictions according to length of type signatures.

Figure 4: Distribution of correct predictions according to size of type signatures.

Performance of models based on the length of identifier names. We also consider the influence of the length of identifier names on the prediction results. In particular, we consider the distribution of correct predictions according to the length of module-qualified identifier names. Here, the length refers to the length of the list of segmented and stemmed tokens contained in an identifier name. The result is shown in Table 4.

Length \ Model	Baseline	Unstructured	Structured	#Signature
2	667	838	1029	4001
3	2676	2502	2869	10764
4	1873	1551	1896	8920
5	1348	1530	1696	5756
6	676	859	904	2616
7	362	429	429	1090
8	173	206	195	451
9	71	103	99	214
10	63	72	64	101
11	22	28	22	42
12	14	13	8	16
13	14	14	14	18
14	8	8	8	8
Total	7967	8153	9233	33997

Table 4: Distribution of correct predictions according to length of module-qualified identifier names.

Now that we have discussed the general results, we shall turn to analyze the performance of each model.

7.1.1 Strong Baseline: Copy-Previous-Signature

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.

It is worth emphasizing the effectiveness of the rule-based baseline that simply copies the previous type signature. As shown in Table 2, 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 surprisingly 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.

7.1.2 Unstructured Prediction

As shown in Table 2, the best result we obtained for unstructured prediction is 23.98% signature accuracy, 41.44% structural accuracy, and 62.15% well-formedness (we rank the result in terms of signature accuracy, as it is the most important measure). In addition, we also study the influence of context on the performance of the model (on the test set), as context is an important component of the model. The result is presented in Table 5. It is clear from Table 5 that the use of context makes a huge difference. Even if we just use one type signature as context, the signature accuracy jumps from 5.15% to 17.13%. It is also worth noting that the best performance is achieved with 3 context type signatures and adding more type signatures to context only lowers the performance.

7.1.3 Structured Prediction

As shown in Table 2, the best result for unstructured prediction is 27.28% signature accuracy, 61.65% 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

	Unstructured		Structured	
Context Num	Sig. Acc. (%)	Struct. Acc. (%)	Sig. Acc. (%)	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 5: Performance of prediction models, by number of annotations in the context.

λ	Signature Accuracy (%)	Structural Accuracy (%)
0.5	26.40	61.75
1	**27.28**	61.65
5	26.56	61.60
10	24.86	61.56
100	24.41	**62.12**

Table 6: Structured prediction model performance, by λ.

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 Figure 5. As with unstructured prediction model, the usefulness of context is again manifest.

Then we keep the number of context annotations used to 3 and vary λ. The result is show in Table 6.

Somewhat surprisingly, varying λ does not impact the performance of the model significantly. It remains to be seen what might cause such unexpected behavior.

7.2 A Human Perspective

In addition to tackling 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 had one researcher (the author) complete 300 predictions. For each prediction task, the researcher was given the module name, the identifier name, and three previous annotations (including both name and signature). The tasks were 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. [7]

In terms of signature accuracy, the human performance (25.33%) is the second highest in Table 2. However, it should be noted that the human benchmark is produced by a single researcher who also designed 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 author 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.

7.3 Additional Analysis

7.3.1 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

[7] We didn't measure the performance of baseline, unstructured prediction model, and structured prediction model on the 300 examples, primarily because we expect the numbers to be similar to the ones shown in Table 2.

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 5. Table 5 reveals that, if we do not include any context, both unstructured and structured prediction model perform quite poorly (5.15% and 6.84% 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, 6.84% $\rightarrow$ 24.96% 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 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.

We investigated the relationship 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 7.

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 7: Relationship 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 models shown in Table 5.

7.3.2 Name Qualifications

As mentioned in section 3.2.4, we consider two ways of qualifying identifier names — module-qualified name and full-path qualified name. We investigate whether the two ways of qualification have an impact on the performance of the model. The result is shown in Table 8. For both unstructured prediction and structured prediction, using qualification yields better results and module-qualified names perform better than path-qualified names.

7.4 Case Studies

In this section, we choose several examples from the test set and dissect the performance of the models on those examples.

7.4.1 Example 1

In the example below, the structured prediction model correctly predicts the signature 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. The difference is that, unlike the structured prediction model, the unstructured prediction model does not take the structure of the type signatures into consideration, thereby risking generating ill-formed signatures like `Str ->`. In comparison, 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`).

```
Full Path: hoogle/src/General/Str.hs
Module: Str
Context:
strUnpack :: Str -> String
strReadFile :: FilePath -> IO Str
```

Model	Qualification	Signature Accuracy (%)	Structural Accuracy (%)
Unstructured	No qualification	21.22	38.02
Unstructured	Module-qualified	23.71	42.23
Unstructured	Path-qualified	22.66	42.35
Structured	No qualification	26.01	**61.75**
Structured	Module-qualified	**27.28**	61.65
Structured	Path-qualified	25.23	60.06

Table 8: Impact of different ways of qualification on the performance of prediction models.

```
strSplitInfix :: Str -> Str -> Maybe ( Str , Str )

Name: StrNull

Prediction (unstructured): Str ->
Prediction (Structured): Str -> Bool
```

7.4.2 Example 2

In the example below, all the type signatures in the context are the same. The context, combined with the identifier name that is reasonably similar to the names in the context, leads us to think that the identifier `testClassParserTypeParameterImplements` has type `Test` and indeed it does. In this case, all predictions (unstructured, structured, and human) are correct, which again reinstates the importance of context.

```
Full Path: cobalt/compiler/test/tests/cobalt/parsers/expr_parsers/ClassParserTest.hs
Module: ClassParserTest
Context:
testClassParserTypeParameter :: Test
testClassParserTypeParameterExtends :: Test
testClassParserTypeParameterExtendsImplements :: Test

Name: testClassParserTypeParameterImplements

Prediction (unstructured): Test
Prediction (Structured): Test
Prediction (Human): Test
```

7.4.3 Example 3

However, context is not always helpful. In the example below, if we just look at the context, we might tend to think that whatever comes next should also have type signature `Int`. That is not the case, however, as the identifier name is `convertToBool'`, which is unlikely to have type signature `Int` because of the `ToBool` part. As a result, none of the predictions is correct and the ground truth is `[Int]` → `[Bool]`.

```
Full Path: xmonad-contrib/XMonad/Layout/ImageButtonDecoration.hs
Module: ImageButtonDecoration
Context:
minimizeButtonOffset :: Int
maximizeButtonOffset :: Int
closeButtonOffset :: Int

Name: convertToBool

Prediction (unstructured): Int
```

```
Prediction (Structured): Int -> Int
Prediction (Human): Int -> Bool
```

7.4.4 Example 4

The example below illustrates the difference between human prediction and machine prediction. From a human perspective, it is relatively easy to observer 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.

```
Full Path: hoogle/src/General/Str.hs
Module: Str
Context:
parseLogLine :: ( String -> Bool ) -> LBS.ByteString -> Maybe ( Day , SummaryI )
logSummary :: Log -> IO [ Summary ]
strPack :: String -> Str

Name: strUnpack

Prediction (unstructured): String -> String
Prediction (Structured): Str -> Str
Prediction (Human): Str -> String
```

8 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 structured prediction model is better in terms of both signature accuracy (27.28% vs. 23.98%) and structural accuracy (61.66% vs. 41.44). 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. In addition, we also obtain a human benchmark by randomly selecting 300 tasks from the test set and letting one researcher (the author) manually predict the type signatures. We find that the human benchmark (25.33% signature accuracy, 57.33% structural accuracy) is actually worse than the performance of the structured prediction model, which illustrates both the difficulty of the task and the power of the structured prediction model.

Despite the relative success of the structured prediction model, there are multiple ways we can further improve the signature accuracy of our prediction model. More specifically, the signature accuracy can be potentially 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 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. In addition, we can also combine rule-based prediction (`main` should have type `IO ()`, for example) and model-based prediction to further improve signature accuracy. From a programmer's perspective, arranging identifiers with the same type signature close to each other can certainly help improve the accuracy of our prediction models. Avoiding using uninformative names and following the naming convention of the language used would also help.

Our evaluation metrics can also be augmented. In this work, we only consider 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. Furthermore, when breaking down the prediction results according to various metrics of identifiers and type

signatures, we did not use the size of the expression itself as a metric, which is mainly caused by the fact that the body of expressions were discarded at the data collection stage. Measuring the size of expression AST can shed light on whether the prediction is correlated with the size of the expression.

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, if we were able to achieve a much higher accuracy, we can imagine an interactive system that auto-completes the type signature as the user types in the name. Then from the name and the type signature, as well as some additional specifications like comments or I/O examples, the system can start to synthesize the function body itself. Another possible application, provided that we can achieve a better accuracy on this task, 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.

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. http://people.cs.uchicago.edu/~brianhempel/context-sensitive_prediction_of_haskell_type_signatures_from_names.pdf.

[10] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[11] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

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

[13] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] 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.

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

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

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

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

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

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

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

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

[23] 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.

[24] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 1997.

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

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

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

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

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

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

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

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

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

[34] 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.

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