Machine Learning Approaches in Programming Language Type Inference

Presentation: Jiancheng Qian

Overview

- Code Naturalness and Type Inference

- Related Work
  - *Deep Learning Type Inference* (Hellendorn, FSE 2018)
  - *Typilus: Neural Type Hints* (Allamanis, PLDI 2020)
  - *LambdaNet: probabilistic type inference using graph neural networks* (Jiayi Wei, ICLR 2020)

- Limitation, Challenges and My Own Opinions

Code Naturalness and Type Inference

- Code Naturalness:
  - **The naturalness hypothesis:** *Programming languages, in theory, are complex, flexible and powerful, but, "natural" programs, the ones that real people actually write, are mostly simple and rather repetitive; thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*

    – A. Hindle, On the Naturalness of Software, 2018
  - **Machine learning 'Language' models:**
    - Code generating models
    - Representational models
    - Pattern mining models

Code Naturalness and Type Inference

- Applications of code naturalness:
  - Recommender systems
  - Code completion
  - Finding defects
  - Code to natural language/natural language to code
  - Pattern mining
  - Code translation
  - Name/type inference
  - Traceability
  - …

Code Naturalness and Type Inference

- Type Inference

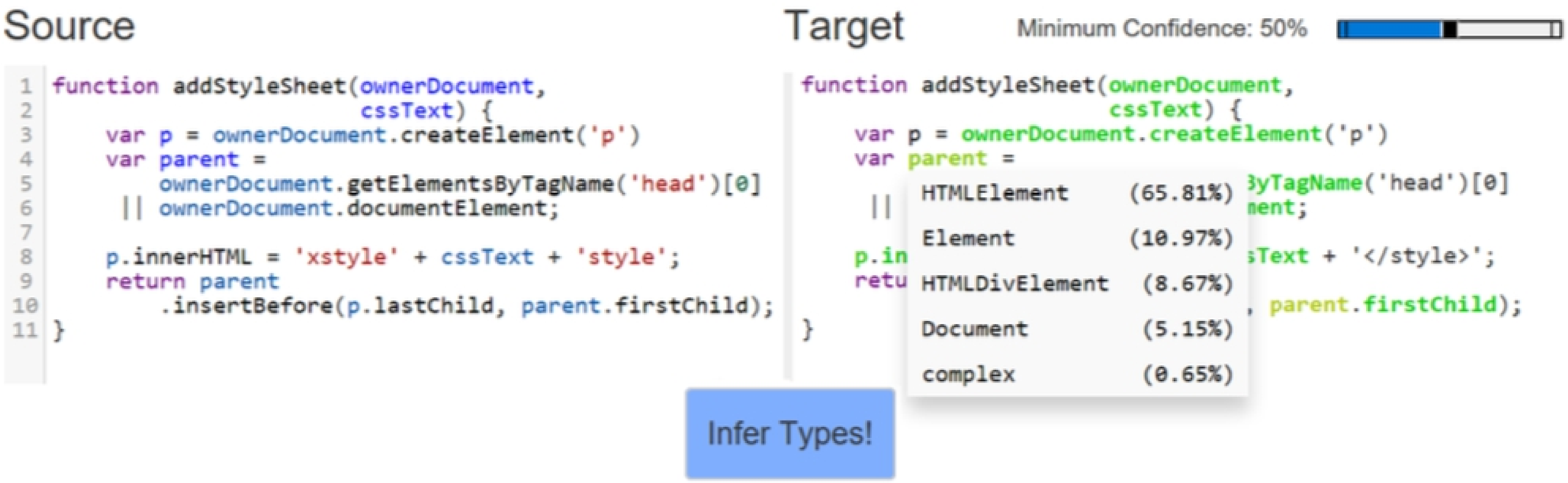


Figure: Deep Learning Type Inference (Hellendorn, FSE 2018)

Code Naturalness and Type Inference

- Why we need type inference:
  - Dynamic languages like Python/JavaScript
  - Enhance maintainability and readability
  - Avoid 15% of the bugs, lower fault occurrence
  - Automation in code maintenance and testing
- Traditional methods:
  - Type annotation (e.g. *mypy* and *pytype*, *TypeScript*)

```
function addStyleSheet(ownerDocument,
                        cssText) {
    var p = ownerDocument.createElement('p')
    var parent =
            ownerDocument.getElementsByTagName('head')[0]
         || ownerDocument.documentElement;

    p.innerHTML = 'x<style>' + cssText + '</style>';
    return parent.insertBefore(p.lastChild, parent.firstChild);
}
```

```
function addStyleSheet(ownerDocument : Document,
                        cssText : string) : any {
    var p : HTMLElement = ownerDocument.createElement('p')
    var parent : HTMLElement =
            ownerDocument.getElementsByTagName('head')[0]
         || ownerDocument.documentElement;

    p.innerHTML : string = 'x<style>' + cssText + '</style>';
    return parent.insertBefore(p.lastChild, parent.firstChild);
}
```

Figure: Deep Learning Type Inference (Hellendorn, FSE 2018)

Related Work

- Similar to NLP
- Type inference neural models:
  - 1) Preprocessing: Embed the code/functions into tokens/graph
    - **Word2vec**
    - **AST**
    - **Type Dependency Graph**
  - 2) Train the neural model, obtain the vector representation of identifiers/nodes
    - **RNN**
    - **GNN**
  - 3) Prediction
    - Calculate the probability distribution of type for input tokens/nodes

Related Work

- Machine learning models:

Title	Input	Machine Learning Model	Prediction
DeepTyper(2018)	Sequential tokens	2 Bi-RNN	
NL2Type(2019)	Vector representation of function	Bi-LSTM	
DTLPy(2019)	Vector representation of function	2 LSTM/GRU	
Typilus(2020)	AST	GNN	KNN
Type4Py(2020)	AST + Sequential tokens	RNN	KNN
LambdaNet(2020)	Type Dependency Graph	GNN	MLP
TypeWriter(2020)	AST based extraction of code + comments	RNN	Feedback directed search

Deep Learning Type Inference (Hellendorn, FSE 2018)

- *DeepTyper*
  - First work to implement neural network in type inference
  - Training using aligned corpus (TypeScript) and prediction on JS code
  - Explore the probability for open-world type suggestion tasks

Deep Learning Type Inference (Hellendorn, FSE 2018)

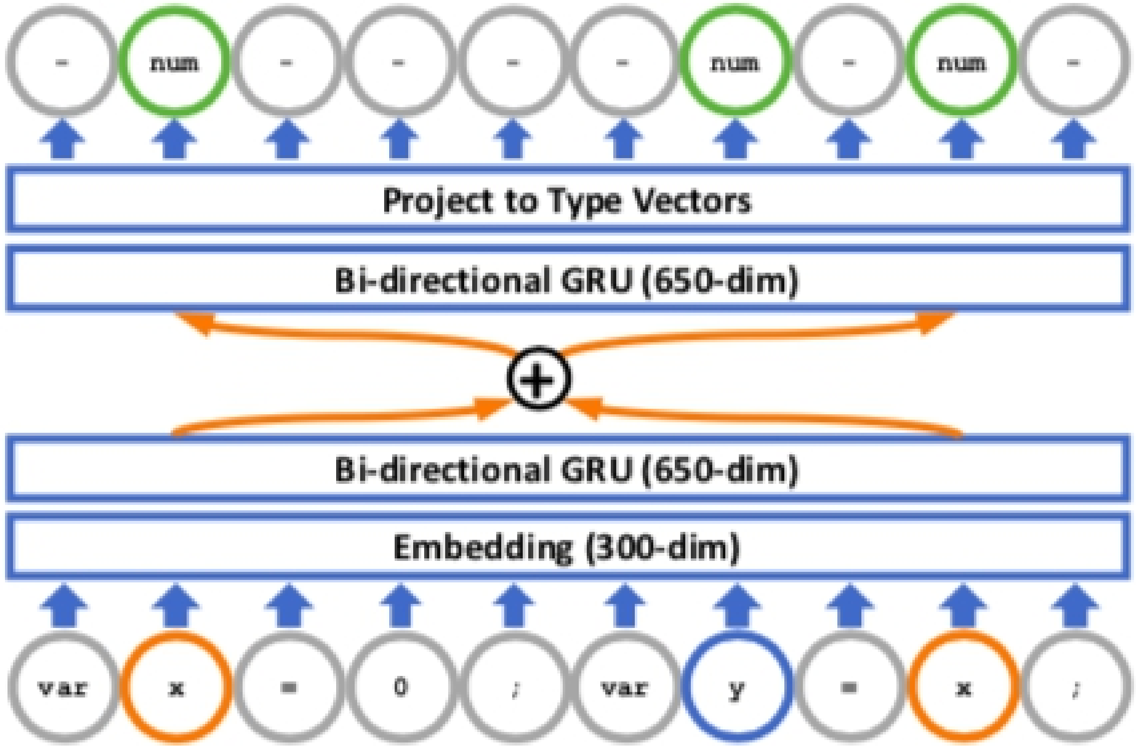
- Input a code sequence $s_1, \ldots, s_N$
- Embedding s_t to the vector representation $\mathbf{x}_{s_t}$
- Hidden state of token $\mathbf{h}_t = RNN\left(\mathbf{x}_{s_t}, \mathbf{h}_{t-1}\right)$
- For token s_t, calculate the probability of type τ:

$$P_{s_t}(\tau) = \frac{\exp(\hat{\tau}_t^T \mathbf{r}_\tau + b_\tau)}{\sum_{\tau'} \exp(\hat{\tau}_t^T \mathbf{r}_{\tau'} + b_{\tau'})}$$

- $\mathbf{r}_\tau$ is the vector representation of well-known type τ and $\hat{\boldsymbol{\tau}}_t$ is the projection vector of token s_t:

$$\hat{\tau}_t = \mathbf{h}_t^{bi} + \frac{1}{|V(t)|} \sum_{i \in V(t)} \mathbf{h}_i^{bi}$$

- $V(t)$ is the locations which bound to the same identifier at location t



Typilus: Neural Type Hints (Allamanis, PLDI 2020)

- *Typilus:*
  - A graph-based deep neural network to the type prediction problem by considering source code syntax and semantics
  - Using GNN to catch the relations between distant identifiers
  - A novel training loss to evaluate the learned type features in the type space, can accurately predict types that were rare, or even unseen
  - Trained on Python codes with *mypy/pytype* corpus

Typilus: Neural Type Hints (Allamanis, PLDI 2020)

- Overview of *Typilus*

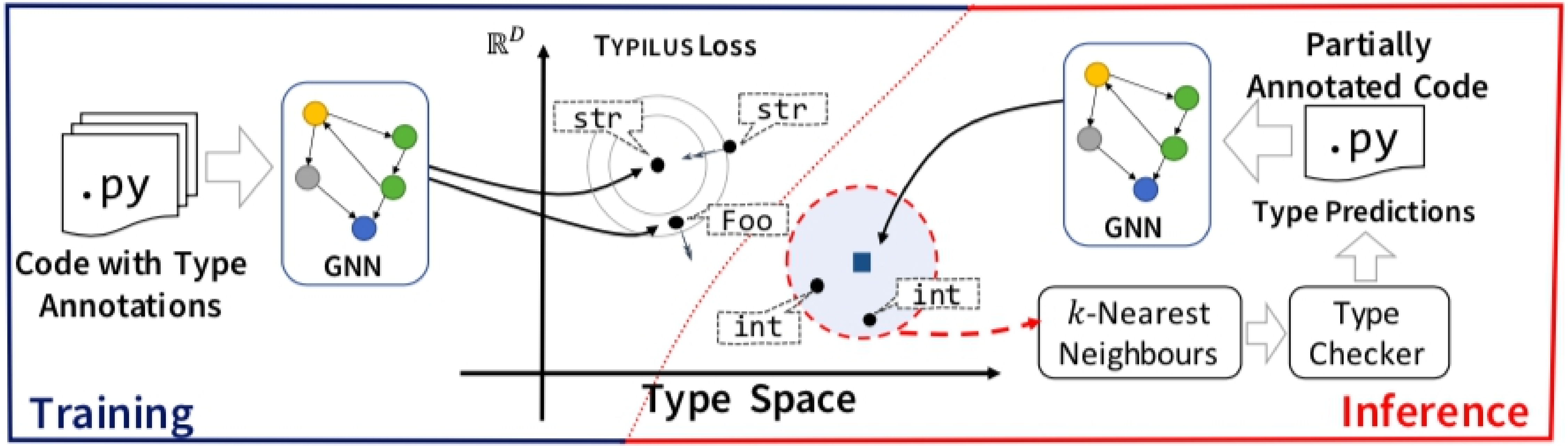


Figure: Allamanis, 2020

Typilus: Neural Type Hints (Allamanis, PLDI 2020)

- AST(abstract syntax tree) embedding for code
- A sample of foo = get_foo(i, i+1)

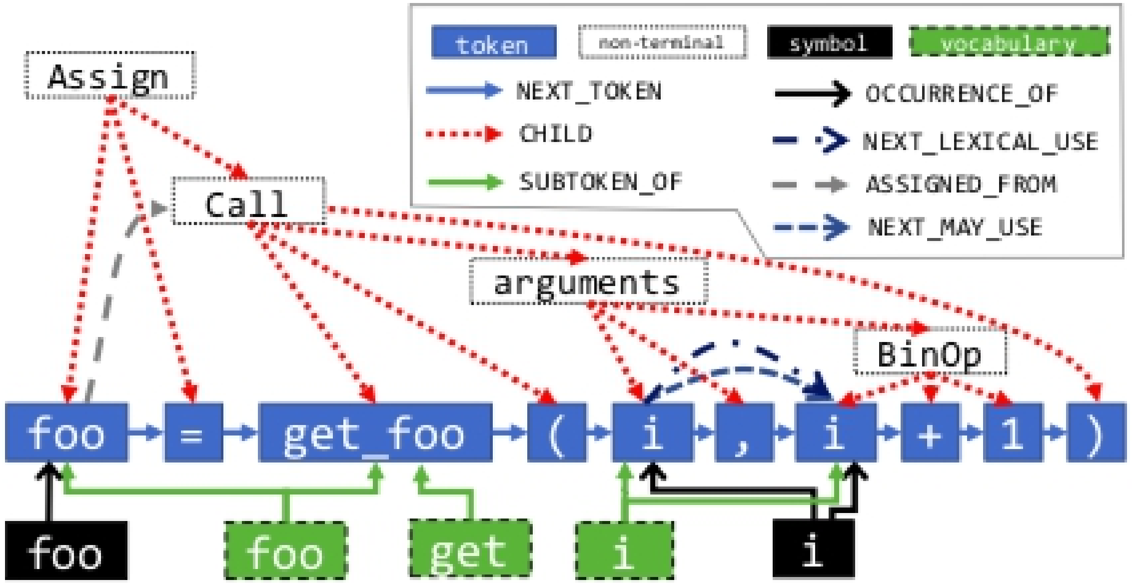


Figure: Allamanis, 2020

Typilus: Neural Type Hints (Allamanis, PLDI 2020)

- Input code sequences S with each identifier $s \in S$, learn the vector representation (type embedding): $\mathrm{e}(S)[s] = \boldsymbol{r}_s \in \mathbb{R}^D$, where e is the GNN network serves as a map
- Define the well-known types $\mathcal{T} = \{\tau_i\}$, learn their representation $\tilde{\boldsymbol{r}}_{\tau_i}$
- Calculate the classification loss for s:

$$\mathcal{L}_{\text{CLASS}}(\boldsymbol{r}_s, \tau) = \underbrace{-\log \frac{\exp\left(\boldsymbol{r}_s \tilde{\boldsymbol{r}}_\tau^T + b_\tau\right)}{\sum_{\tau_j \in \mathcal{T}} \exp\left(\boldsymbol{r}_s \tilde{\boldsymbol{r}}_{\tau_j}^T + b_{\tau_j}\right)}}_{-\log P(s:\tau)}$$

- Similarity learning: calculate the similarity loss for s, $\boldsymbol{r}_{s^+}$ is the representation vector of same type as s, while $\boldsymbol{r}_{s^-}$ is a different type and :

$$\mathcal{L}_{\text{SPACE}}(s) = \sum_{s_i^+ \in P_+(s)} \frac{\left\|\boldsymbol{r}_{s_i^+} - \boldsymbol{r}_s\right\|}{|P_+(s)|} - \sum_{s_i^- \in P_-(s)} \frac{\left\|\boldsymbol{r}_{s_i^-} - \boldsymbol{r}_s\right\|}{|P_-(s)|},$$

$$P_+(s) = \left\{x_i^+ : \left\|\boldsymbol{r}_{s_i^+} - \boldsymbol{r}_s\right\| > d_{\min}^-(s) - m\right\}$$

$$P_-(s) = \left\{x_i^- : \left\|\boldsymbol{r}_{s_i^-} - \boldsymbol{r}_s\right\| < d_{\max}^+(s) + m\right\}$$

- Combine the two loss functions for the learning objective:

$$\mathcal{L}_{\text{TYPILUS}}(s, \tau) = \mathcal{L}_{\text{SPACE}}(s) + \lambda \mathcal{L}_{\text{CLASS}}\left(W\boldsymbol{r}_s, \text{ER}(\tau)\right)$$

Typilus: Neural Type Hints (Allamanis, PLDI 2020)

- The implementation of type space enables the model to predict unseen types without well-known annotation

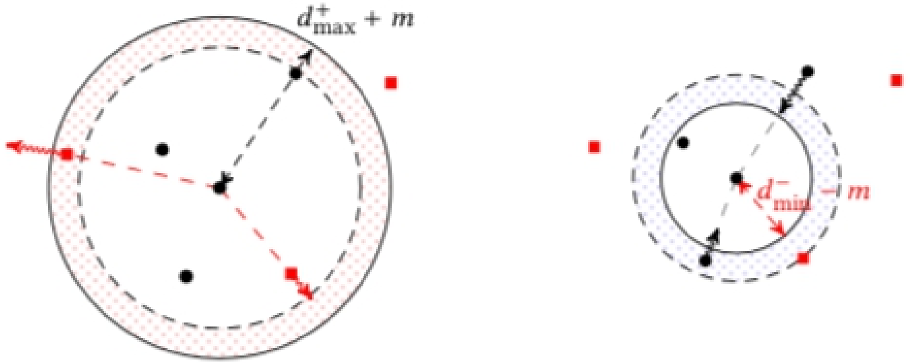


Figure 2. Graphic depiction of the two terms of the similarity objective in Eq. 3. *Left*: all dissimilar points (red squares), *i.e.* P_{-} within distance $d^{+}_{max} + m$ of the query point are pushed away. *Right*: all similar points (black circles) that are further than $d^{-}_{min} - m$ from the query point, *i.e.* P_{+}, are pulled towards it. The margin distance m is shaded.

Figure: Allamanis, 2020

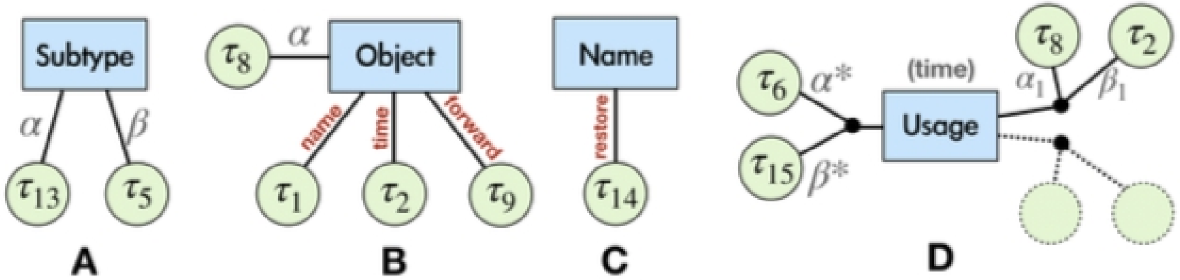
LambdaNet: probabilistic type inference using graph neural networks (Jiayi Wei, ICLR 2020)

- *LambdaNet*:
  - Using a novel graph representation – *type dependency graph* for the JavaScript code
  - Using GNN to learn the features of input nodes
  - An MLP prediction layer to compute a compatibility score with well-known types for each output vector

LambdaNet: probabilistic type inference using graph neural networks (Jiayi Wei, ICLR 2020)

- Type dependency graph

```
var c1: τ8 = class MyNetwork {
   name: τ1;  time: τ2;
   var m1: τ9 = function forward(x: τ3, y: τ4):τ5 {
      var v1: τ10 = x.concat;  var v2: τ11 = v1(y);
      var v3: τ12 = v2.TIMES_OP; var v4: τ13 = v3(NUMBER);
      return v4;
   }
}  // more classes...
var f1:τ14 = function restore (network: τ6): τ7 {
  var v3: τ15 = network.time;
  var v4: τ16 = readNumber(STRING);
  network.time = v4;  // more code...
}
```



Types of hyperedges: bool, subtype, function, call, object, name, usage,…

Figure: LambdaNet: probabilistic type inference using graph neural networks, *Jiayi*, 2020

Challenge in Type Inference

- Depends on existing corpus (like mypy/pytype/TypeScript)
- Predicting unknown/complicated types
  - Some types are user-defined and specialized in some projects
    - Need to know the stability among different projects/datasets
  - For some nested types like `List[List[List[int]]]`, `Optional[Map[str, int]]` are hard to predict
- Cross programming language use
  - E.g. Typilus performs bad on JS comparing to Python code
  - Need to find more universal methods since PL is developing fast
    - Focus on the dynamic type features for different language and build dependency graphs
    - The impact of network structure is still unknown

My Own Opinions

- Recent work (since 2020) are focusing on the graph representation, while not enough research on the network structure (most are using RNN/single layer GNN)
- Working on a NAS(Neural architecture search) attempting to find the network structures implications
- We can using a unsupervised or self-supervised learning method to enhance the ability for training without an existing type annotation corpus