LAMBDANET论文阅读笔记

施赢凯

type inference

引言

近年来，动态语言越来越流行，如python，java，而由于静态类型的缺失会导致一系列的不足，例如在编译阶段就能发现错误以及能够代码的补全完成，所以采用了渐进类型，而由于这种类型采用了大量的动态语言结构，编译器不能很好的完成推断类型注释，所以越来越多的算法被采用来解决这类类型推断，通过阅读了几篇的论文，我发现其中有几种方法采用了深度机器学习，我所看得到有采用双向循环神经网络的DeepTyper，采用了图神经网络的La'mbdar net以及R-GNN中的一系列，当然这次主要的还是了解图神经网络在类型推断中的注释以及运用。

文章内容的引出

* 首先文章介绍了为了解决动态语言由于缺少静态类型，很难在编码阶段发现错误，所以引入了渐进类型的程序语言设计，能够使得我们同时使用动态类型和静态类型，而由于大量的动态语言结构设计，所以我们需要进行类型推断，本文章主要集中于Typescript。先前的工作主要集中在使用结构化推断模型（例如，Raychev et al. (2015) 和 Xu et al. (2016)）进行动态类型语言的类型注释的预测。这些方法虽然不使用深度学习，但受限于非常有限的预测空间。Hellendoorn et al. (2018) 和 Jangda & Anand (2019) 使用深度学习模型（RRNs 和 Tree-RNNs）将程序建模为序列和AST树，用于TypeScript 和 Python 程序。Malik et al. (2019) 使用不同的信息源，将文档字符串作为输入的一部分。然而，这些方法都受限于从固定词汇中预测类型。所以文章提出了一种新的类型推断模型，并将其命名为LAMBDANET。首先根据输入的程序采用轻量化的代码构建一个类型依赖图,中，节点表示类型变量，标记的超边编码它们之间的关系。除了像传统类型推断中表达逻辑约束（例如子类型关系）那样，类型依赖图还包含涉及命名和变量使用的上下文提示。在这样的类型依赖图的基础上，我们的方法使用图神经网络（GNN）计算每个类型变量的向量嵌入，然后利用类似指针网络的架构进行类型预测（Vinyals等人，2015）。图神经网络本身需要处理各种超边类型，其中一些具有可变数量的参数，我们为其定义了适当的图传播操作。我们的预测层将类型变量的向量嵌入与候选类型的向量表示进行比较，从而灵活地处理在训练期间未观察到的用户定义类型。此外，我们的模型通过构建变量级别而不是标记级别的预测来预测一致的类型分配。在这文章过成中，他做出了这样的贡献：（1）提出了一种用于TypeScript的概率类型推断算法，利用深度学习从程序的类型依赖图表示中进行预测。（2）描述了一种使用图神经网络计算类型变量向量嵌入的技术，并提出了一种类似指针网络的方法来预测用户定义的类型。（3）在数百个真实的TypeScript项目上进行了实验评估，并实现了较大的成果。

LAMBDANET的具体实现

类型依赖图的构建

如何进行类型推断呢，文中提出这样的解决方案，1）类型约束，2）上下文提示，3）类型依赖图 ,4）用户自定义类型。其中程序的表示有多种，本文用来解决问题的主要是图神经网络，因此在这篇文章中所采用的是将程序构造成一个类型依赖图，下面就将介绍如何构造一个类型依赖

图。
其中G = (N, E) 表示类型依赖图是一个超图，其中节点 N 代表类型变量，有标签的超边 E 编码它们之间的关系。通过对TypeScript程序进行静态分析，可以从源代码的中间表示中提取类型依赖图，使得每个程序子表达式都能关联到唯一的变量，直观地说，类型依赖图编码了类型变量的属性以及它们之间的关系。每个超边对应于表 1 中显示的谓词之一。谓词（即超边）分为两类，即逻辑和上下文，前者可以视为对类型变量施加硬约束，而后者则编码从变量、函数和类的名称中提取的有用提示。所有的超边均可以从下面这张表给出，其中超边的类型也被其划分为三类：

- FIXED表示的表示这是一种固定的、确定的关系或约束。即，它指定了具体的关系，例如Bool(α)表示α被用作布尔值，或者$Name_l(\alpha)$表示α具有名称l
- NARY：表示这是一种n元关系，其中n是超边涉及的参数数量。有三种NARY关系：
  - NARY Function(α, β1, . . . , βk, β∗)：表示一个函数关系，其中α是一个函数类型，参数为β1到βk，返回类型为β。
  - NARY Call(α, β∗, β1, . . . , βk)：表示一个函数调用关系，其中α是函数，参数为β1到βk，返回类型为β∗。
  - NARY Object l1,...,lk(α, β1, . . . , βk)：表示一个对象的关系，其中α是对象类型，属性为l1到lk，类型分别为β1到βk。
- NPAIRS：表示这是一个多对关系，其中涉及多个成对的元素。在这个上下文中，有一个NPAIRS关系：
  - NPAIRS Usagel((α∗, β∗),(α1, β1), . . . ,(αk, βk))：表示涉及名称l的多对用法关系，其中每对都包含一个α和β。

Table 1: Different types of hyperedges used in a type dependency graph.

Type	Edge	Description
	Logical	
FIXED	$\text{Bool}(\alpha)$	α is used as boolean
FIXED	$\text{Subtype}(\alpha, \beta)$	α is a subtype of β
FIXED	$\text{Assign}(\alpha, \beta)^\dagger$	β is assigned to α
NARY	$\text{Function}(\alpha, \beta_1, \ldots, \beta_k, \beta^*)$	$\alpha = (\beta_1, \ldots, \beta_k) \to \beta^*$
NARY	$\text{Call}(\alpha, \beta^*, \beta_1, \ldots, \beta_k)$	$\alpha = \beta^*(\beta_1, \ldots, \beta_k)$
NARY	$\text{Object}_{l_1,\ldots,l_k}(\alpha, \beta_1, \ldots, \beta_k)$	$\alpha = \{l_1 : \beta_1, \ldots, l_k : \beta_k\}$
FIXED	$\text{Access}_l(\alpha, \beta)$	$\alpha = \beta.l$
	Contextual	
FIXED	$\text{Name}_l(\alpha)$	α has name l
FIXED	$\text{NameSimilar}(\alpha, \beta)$	α, β have similar names
NPAIRS	$\text{Usage}_l((\alpha^*, \beta^*), (\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$	usages involving name l

† Although assignment is a special case of a subtype constraint, we differentiate them because these edges appear in different contexts and having uncoupled parameters for these two edge types is beneficial.

在这篇论文中也给出了其中的一个程序和对应所得到的类型依赖图中的一部分，如下图所示，所采用的也就是在上面描述的内容,借助下面的内容描述我们可以来了解这张类型依赖图所能够容纳的内容。

```
1   var c1: τ8 = class MyNetwork {
2     name: τ1;  time: τ2;
3     var m1: τ9 = function forward(x: τ3, y: τ4):τ5 {
4       var v1: τ10 = x.concat;  var v2: τ11 = v1(y);
5       var v3: τ12 = v2.TIMES_OP; var v4: τ13 = v3(NUMBER);
6       return v4;
7     }
8   }  // more classes...
9   var f1:τ14 = function restore (network: τ6): τ7 {
10    var v3: τ15 = network.time;
11    var v4: τ16 = readNumber(STRING);
12    network.time = v4;  // more code...
13  }
```

Figure 2: An intermediate representation of the (unannotated version) program from Figure 1. The τ_i represent type variables, among which τ_8–τ_{16} are newly introduced for intermediate expressions.

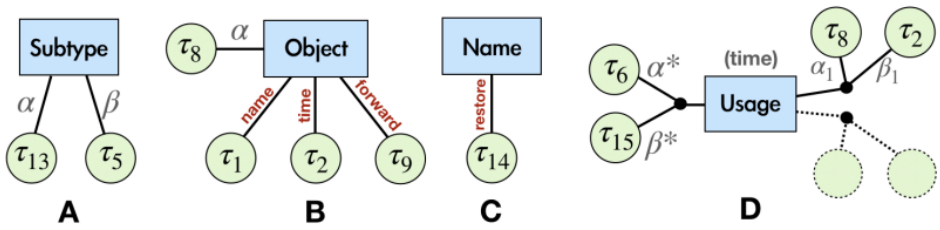

图 3 展示了从图 2 中间表示中提取的类型依赖图 G 中的一些超边。如图 3(A) 所示，这段代码中提取了 Subtype(τ13, τ5) 的谓词，因为与返回表达式 v4 关联的类型变量必须是封闭函数返回类型的子类型。类似地，如图 3(B) 所示， $Object_{name,time,forward}$(τ8, τ1, τ2, τ9) 的谓词，因为 τ8 是一个对象类型，其名称、时间和forward 函数成员分别与类型变量 τ1、τ2、τ9 相关联。 与对类型变量施加硬约束的 Subtype 和 Object 谓词相反，图 3 中显示的下两个超边编码了从变量名称中获取的上下文提示。图 3(C) 表明类型变量 τ14 与一个名为 restore 的表达式关联。尽管这种命名信息对于 TypeScript 的结构类型系统是不可见的但在之后仍然能够做为 GNN 架构的有用输入特征。 除了存储与每个类型变量相关的唯一变量名之外，类型依赖图还编码了变量和类名称之间的相似性。许多程序变量的名称模仿它们的类型：例如，称为 MyNetwork 的类的实例可能经常被称为 network 或 network1。为了捕捉这种对应关系，我们的类型依赖图还包含一个称为 NameSimilar 的超边，如果它们对应的标记化名称具有非空交集，则连接类型变量 α 和 β。 如表 1 所示，还有一种名为 Usage 的最终类型的超边，用于促进对象类型的类型推断。特别是，如果有一个对象访问 var y = x.l，我们提取谓词 Usagel((τx, τy),(α1, β1), . . . ,(αk, βk))，将 x 和 y 的类型变量与所有包含属性/方法 βi 的类 αi 连接起来，其名称为 l。图 3 展示了从图 2 代码中提取的 Usage 超边。

神经网络架构：

正好如我在另一篇文章中看到的，图神经网络主要是分三个步骤，消息沿着依赖图传递，消息的聚合，信息的更新，最后再判断最有可能的类型分布。 给定一个类型依赖图 G = (N, E)，我们首先计算每个 n ∈ N 的矢量嵌入 vn，使得这些向量隐式编码类型信息。因为我们的程序抽象是一个图，自然的选择是使用图神经网络架构。从高层来看，该架构接受每个节点 n 的初始向量 v_n^0，对图神经网络执行 K 轮消息传递，然后返回每个类型变量的最终表示。

更详细地说，设 v_n^t 表示第 t 步时节点 n 的矢量表示，其中每一轮包括消息传递和聚合两个步骤。消息传递步骤计算一个矢量值的更新，发送到与之相连的连接节点 p1, . . . , pa 的每个超边 e ∈ E 的第 j 个参数。一旦所有消息都被计算出来，聚合步骤通过组合发送给 n 的所有消息计算一

个新的嵌入 v_n^t：

$$\mathbf{m}_{e,p_j}^t = \mathrm{Msg}_{e,j}(\mathbf{v}_{p_1}^{t-1}, \ldots, \mathbf{v}_{p_a}^{t-1}) \quad \mathbf{v}_n^t = \mathrm{Aggr}(\mathbf{v}_n^{t-1}, \{\mathbf{m}_{e,n}^t | e \in \mathcal{N}(n)\})$$

- 初始化 在图神经网络中，节点对应于类型变量，每个类型变量与程序变量或常量相关联。我们将代表常量（或者是变量）的节点称为常量（或者是变量）节点，并且初始化过程根据 n 是否为常量节点而有不同的工作方式。 如果是常量节点，由于已知每个常量的类型，将每个类型为 τ（例如，string）的常量节点的初始嵌入设置为可训练的向量 cτ，并在 GNN 迭代过程中不进行更新。 如果是变量节点，那么在初始化过程中对其类型没有信息；因此，使用通用的可训练初始向量对所有变量节点进行初始化（即，它们初始化为相同的向量，但在 GNN 迭代中更新为不同的值）。

- 消息传递 消息的传递是在边上进行的，而在这张依赖图中，由于边类型的不同，采用的Msg传递函数是不一样的，

  - FIXED（固定）：由于这些边对应于固定arity谓词（每个参数的位置很重要），我们通过首先连接所有参数的嵌入向量，然后将结果向量馈送到一个2层MLP（多层感知机）以计算第 j 个参数的消息。此外，由于类型为 Access 的超边具有一个标识符，我们还将标识符嵌入为一个向量，并将其视为额外的参数。
  - NARY（N元）：由于 NARY 边连接可变数量的节点，因此需要一种可以处理这一挑战的架构。在当前的 LAMBDANET 实现中，使用一种适合批处理的简单架构。具体来说，对于一个 NARY 边 $E_{l1,\ldots,lk}(\alpha, \beta 1, \ldots, \beta k)$（对于 Function 和 Call，标签 lj 表示参数位置），计算 α 的消息集合为 $\{MLP_\alpha(v_{l_j}||v_{\beta_j})|j = 1 \ldots k\}$，每个 βj 的消息计算为 $MLP_\beta(v_{l_j}||v_\alpha)$。其中为 α 计算了 k 种不同的消息，而每个 βj 的消息仅依赖于 α 的向量嵌入和其位置 j，而不依赖于其他 βj 的向量嵌入。
  - NPAIRS（多对）：这是与 $Usage_l((\alpha*, \beta*), (\alpha 1, \beta 1), \ldots, (\alpha k, \beta k))$ 相关的特殊类别。这种类型的边源自形如 b = a.l 的表达式，并用于将 a 和 b 的类型变量连接到所有包含带有标签 l 的属性/方法 βi 的类 αi。直观地说，如果 a 的类型嵌入与类型 C 非常相似，那么 b 的类型很可能与 C.l 的类型相同。基于这种推理，使用基于点积的注意力来计算 α∗ 和 β∗ 的消息。具体来说，我们使用 α∗ 和 αj 作为注意力键，使用 βj 作为注意力值来计算 β∗ 的消息（在计算 α∗ 的消息时切换键值角色）：

$$\mathbf{m}_{e,\beta*}^t = \sum_j w_j \mathbf{v}_{\beta_j}^{t-1} \qquad \mathbf{w} = \mathrm{softmax}(\mathbf{a}) \qquad a_j = \mathbf{v}_{\alpha_j} \cdot \mathbf{v}_{\alpha*}$$

PS（通过查阅得知）：dot-product based attention" 是一种基于点积的注意力机制，常用于神经网络中的注意力模型。在这种机制中，注意力的权重由查询向量与键向量之间的点积计算得到。这样的注意力机制允许网络在聚合步骤中关注于与节点相关的重要信息，而不同的消息在计算中获得的权重取决于它们与节点的相似度。这提高了网络对不同信息的适应能力。其中softmax正是用来计算权重的。

- 消息的聚合 聚合步骤将发送到节点n的所有消息组合在一起，以计算新的嵌入v_n^t。为了实现这个目标，这里的模型采用的是GAT。

$$v_n^t = \text{Aggr}(v_n^{t-1}, \{m_{e,n}^t | e \in \mathcal{N}(n)\}) = v_n^{t-1} + \sum_{e \in \mathcal{N}(n)} w_e \mathbf{M}_1 m_{e,n}^t \quad (1)$$

  其中w_e是来自边e的消息的注意力权重。具体来说，权重 w_e通过softmax(a)计算，其中$ae = LeakyReLu(v_{t-1}^n \cdot M_2 m_{te,n})$，其中$m_{te,n}$是从节点n的邻居e接收的消息。而M1和M2是可训练的矩阵，两个权重矩阵是模型在训练过程中学习到的参数，它们用于计算注意力权重和更新节点嵌入。。与原始的GAT架构类似，将LeakyReLu的斜率设置为0.2，但这里使用点积来计算注意力权重，而不是线性模型。

- 标识符的嵌入 正如在上面提到的，文中将标识符的嵌入也用作一个向量，为什么呢，标识符嵌入常用于将代码中的标识符（如变量名、函数名）转化为向量表示，以供深度学习模型使用。这样的嵌入可以帮助模型更好地理解代码的结构和语义，从而提高任务的性能。文中将变量名根据驼峰命名法和下划线规则分解为单词标记，并为在训练集中出现多次的所有单词标记分配一个可训练向量。对于所有其他标记，将它们随机映射标记之一，其中i在我们当前的实现中从0到50变化。这种映射在每次运行GNN时都是随机构建的，因此有助于我们的神经网络区分不同的标记，即使它们是稀有标记。我们沿着我们体系结构的其余部分端到端地训练这些标识符嵌入。

- 预测层： 预测层是神经网络中的一部分，用于将网络的中间表示映射到最终的输出，通常用于进行任务相关的预测。在文中提到的上下文中，预测层主要用于进行类型预测。预测层的设计在这里描述了如何为每个类型变量 n和每个候选类型c∈Y(g) 计算兼容性分数。具体而言，采用多层感知机（MLP）来计算兼容性分数$s_{n,c} = \text{MLP}(v_n, u_c)$，其中 u_c是类型 c 的嵌入向量。 这个设计中，有两种情况考虑：

如果c∈Y lib，即 c 是库类型，那么对每个库类型 c，有一个可训练的向量v c用于表示该库类型。如果c∈Y user(g)，即 c 对应于图g 的类型依赖图中的节点n_c，那么直接使用n_c的嵌入向量v_{n_c} 作 u_c 。 形式上，这个设计类似于指针网络，其中在前向传播期间使用计算的嵌入来预测指向这些类型的“指针”。 在获得这些兼容性分数后，作者应用 softmax 层将其转换为概率分布 $P_n(c|g) = \frac{\exp(s_{n,c})}{\sum_{c'} \exp(s_{n,c'})}$. 在测试时，可以通过选择具有最高概率的类型来进行类型分配，或者选择前 N 个最高概率的类型。这一过程使得模型能够在给定图G的上下文中预测最可能的类型分配。

对于LAMBDANET的评估

文中为了评估lambdanet这个模型，提出了三个问题，并且围绕着问题进行解答：（1）我们的方法与先前的工作相比如何？（2）我们的模型能够多好地预测用户定义的类型？（3）我们模型的各个组成部分有多有效?

与先前的工作对比，文中选择了同为深度学习算法模型的deeptyper,其是采用双向循环网络的一种模型，DeepTyper将程序视为标记序列，并使用双向RNN进行类型预测。由于DeepTyper只能从固定的词汇表中预测类型，将LAMBDANET和DeepTyper的预测空间都固定为libY，并测量它们的相应的Top-1准确性。同时为了比较的更加准确，DeepTyper进行过整改，对于每个变量进行单次预测（通过在进行预测之前对同一变量的所有出现的RNN内部状态进行平均处理）。最后的比

较结果是如下图所示的。我们很明显的看到，无论在声明准确性还是出现准确性都是优于deeptyper，有着挺高的提升。

Model	Top1 Accuracy (%)	
	Declaration	*Occurrence*
DeepTyper	61.5	67.4
LAMBDANET$_{\text{lib}}$ (K=6)	75.6	77.0

此外lambdanet的一个特点是能够预测用户定义的类型。在第二个实验中，将LAMBDANET的预测空间扩展到包括用户定义的类型，并使用两个简单的基准进行性能校准。第一个基准是TypeScript编译器的类型推断，是完备但不完整的。第二个基准是SIMILARNAME，根据变量名称与相应类型之间的相似性进行预测。实验结果表明LAMBDANET在Top-1方面达到了64.2%，在Top-5方面达到了84.5%，明显优于两个基准。这表明，我们融合逻辑和上下文信息的类型预测方法比基于规则的独立集成方法更为有效。

Model	Top1 Accuracy (%)			Top5 Accuracy (%)		
	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*
TS COMPILER	2.66	14.39	8.98	-	-	-
SIMILARNAME	24.1	0.78	15.7	42.5	3.19	28.4
LAMBDANET (K=6)	53.4	66.9	64.2	77.7	86.2	84.5

为了回答第三个问题，文中设计了第三个实验，(a)改变了消息传递迭代的次数（左侧）和(b)禁用了架构设计的各种功能（右侧）。从左表可以看出，随着消息传递迭代次数的增加，准确性一直在提高，直到6次；这种收益表明我们的网络学会了在较长距离上执行推理。右表显示了我们设计选择对总体结果的影响。这些都表明其中的任何组成部分对于lambdanet的功能具有较大的影响。

K	Top1 Accuracy (%)		
	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*
6	53.4	66.9	64.2
4	48.4	65.5	62.0
2	47.3	61.7	58.8
1	16.8	48.2	41.9
0	0.0	17.0	13.6

Ablation (K = 4)	Top1 Accuracy (%)		
	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*
LAMBDANET	48.4	65.5	62.0
No Attention in NPAIR	44.1	57.6	54.9
No *Contextual*	27.2	52.6	47.5
No *Logical**	24.7	39.2	36.2
Simple Aggregation	40.2	66.9	61.5

代码的复现部分，仍在尝试，会补充。