

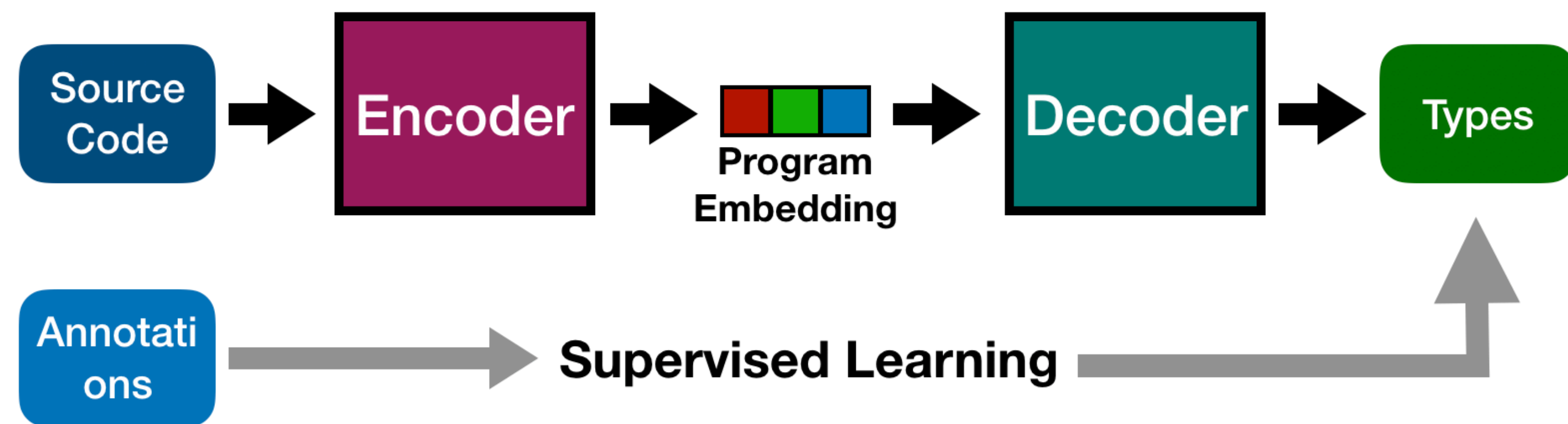


LambdaNet: Probabilistic Type Inference using Graph Neural Networks

Jiayi Wei, Maruth Goyal, Greg Durrett, Isil Dillig

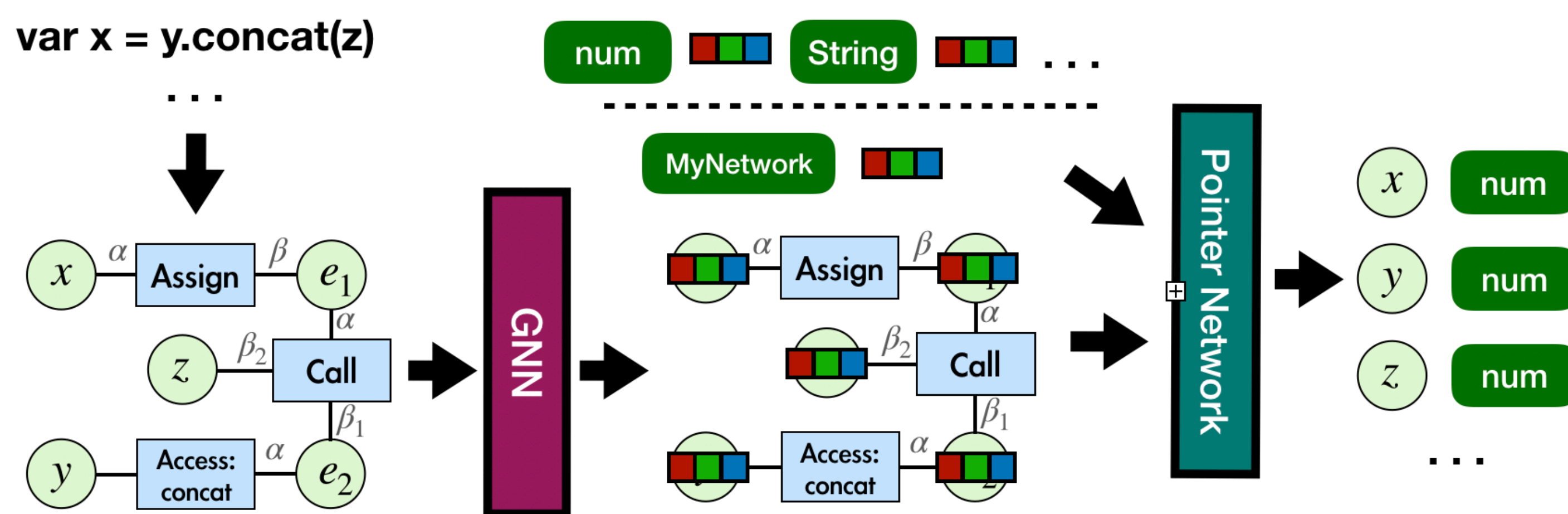
The University of Texas at Austin

A learning-based approach to automatically predict type annotations for dynamic typed codebases



LambdaNet workflow

- Uses program analysis to convert programs into an intermediate representation called *type dependency graphs*
- Computes variable embedding with graph neural networks
- Uses pointer network to find compatible type assignments



Advantages of our approach

- Can predict user-defined types that are not seen during training
- Achieves 75.6% accuracy (14.1% improvement over prior work)
- Predicts consistent types for each program variable

Comparison with DeepTyper

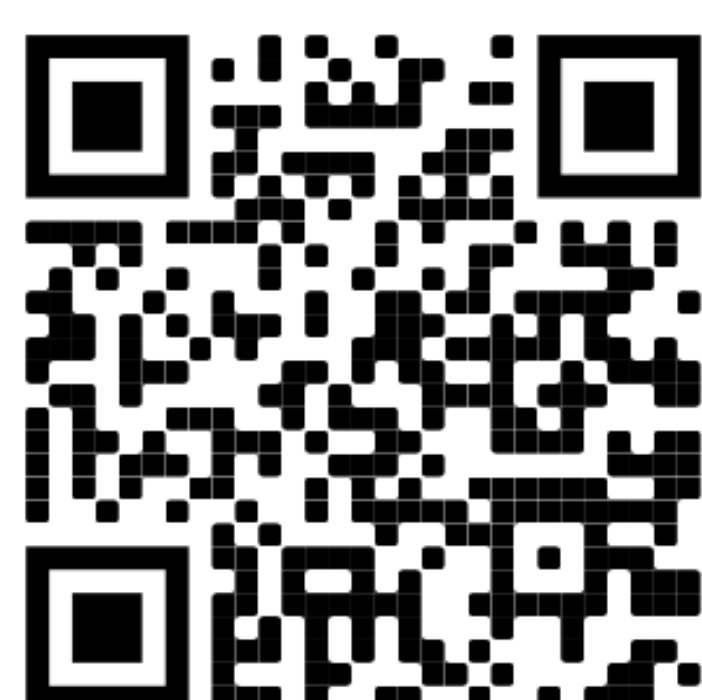
- Fix both Tools to predict only (the same set of) library types

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

Predicting User-Defined types

- TypeScript compiler is sound by incomplete
- SimilarName uses name similarity between type vars and types

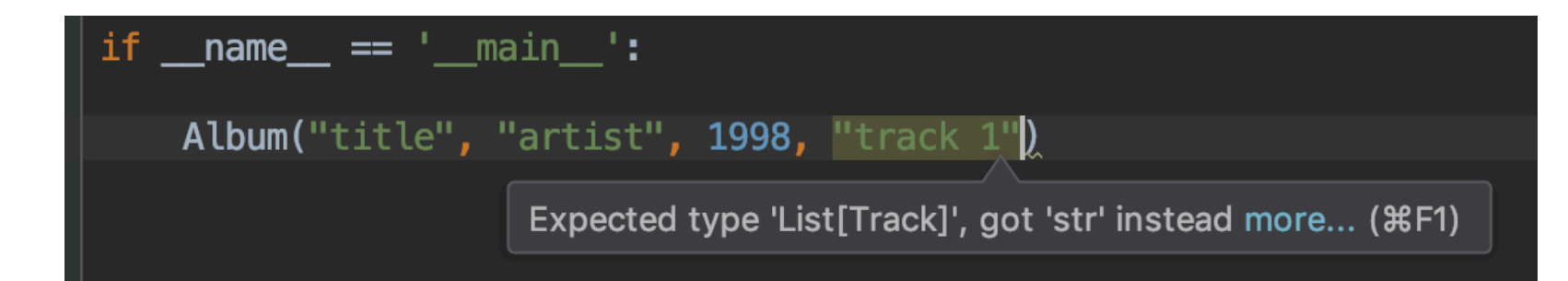
Model	Top1 Accuracy (%)			Top5 Accuracy (%)		
	$\mathcal{Y}_{\text{user}}$	\mathcal{Y}_{lib}	Overall	$\mathcal{Y}_{\text{user}}$	\mathcal{Y}_{lib}	Overall
TS COMPILER	2.66	14.39	8.98	-	-	-
SIMILARNAME	24.1	0.78	15.7	42.5	3.19	28.4
LAMBDANET (K=6)	53.4	66.9	64.2	77.7	86.2	84.5



Learn more about our ICLR'20 paper!

Why are type annotations useful?

- Detect errors statically
- Serve as API documentations
- Help IDEs to provide code completions

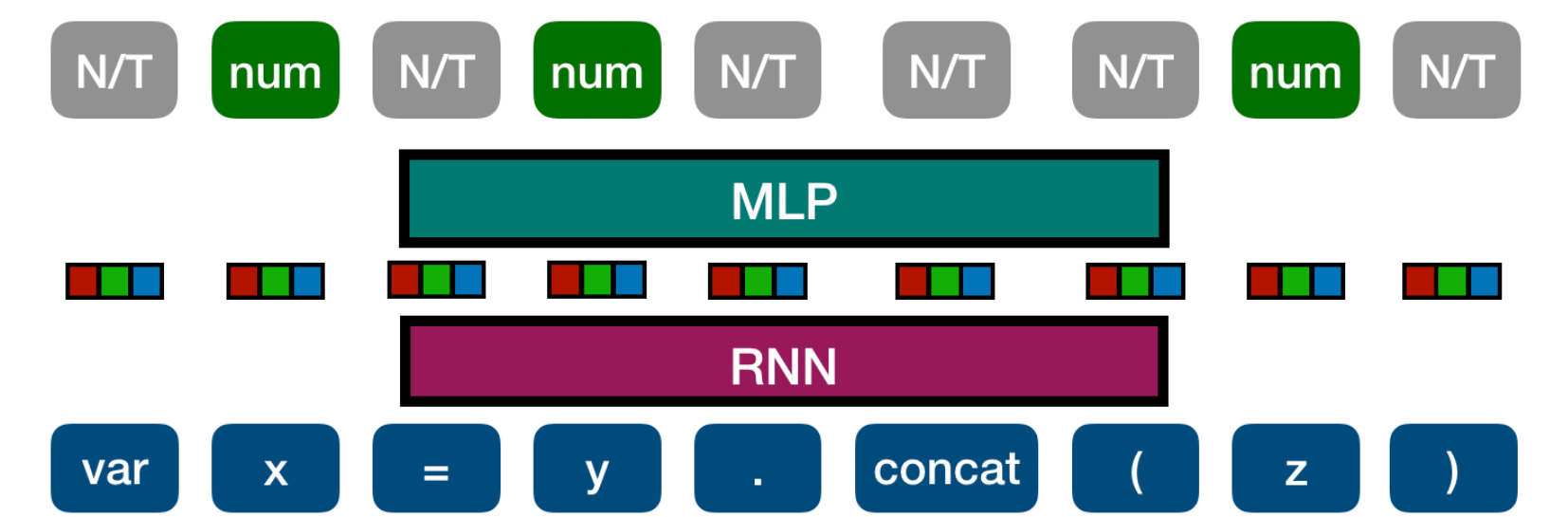


Type inference for dynamic code is super hard

- Dynamic language features
 - Most codebase cannot be fully typed
- Absence of principle types
 - There can be multiple ways to type the same code
- Traditional type inference algorithms won't work
 - Motivates us to take a learning-based approach

```
eval("1+a.b")
obj["method"+i]
obj.f = g
a.concat(b)
Strings?
Matrices?
```

What does prior work do?



DeepTyper treats programs as token sequences and uses a bidirectional recurrent neural network to obtain embedding vectors. Only the vectors corresponding to variable tokens are then fed into a multi-layer perceptron to make type predictions.

An Motivating Example

```
1 class MyNetwork {
2   name: string; time: number;
3   forward(x: Tensor, y: Tensor): Tensor {
4     return x.concat(y) * 2;
5   }
6 }
7 // more classes ...
8 function restore (network: MyNetwork): void {
9   network.time = readNumber("time.txt");
10  // more code ...
11 }
```

Given an unannotated version of this Typescript program, a traditional rule-based type inference algorithm cannot soundly deduce the true type annotations (underlined).

Constructing Intermediate Program Representation

```
1 var c1: T8 = class MyNetwork {
2   name: T1; time: T2;
3   var m1: T9 = function forward(x: T3, y: T4): T5 {
4     var v1: T10 = x.concat; var v2: T11 = v1(y);
5     var v3: T12 = v2.TIMES_OP; var v4: T13 = v3(NUMBER);
6     return v4;
7   }
8 } // more classes ...
9 var f1: T14 = function restore (network: T6): T7 {
10  var v3: T15 = network.time;
11  var v4: T16 = readNumber("time.txt");
12  network.time = v4; // more code ...
13 }
```

Our intermediate program representation introduces fresh type variables for various places that require a type annotation. Note that T_8-T_{16} are introduced for intermediate expressions.

Building Type Dependency Graph

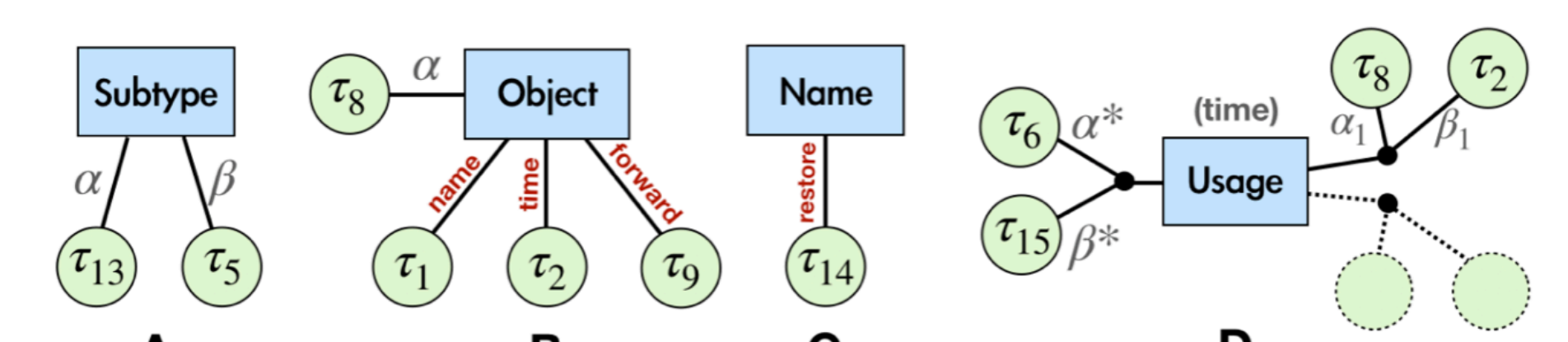
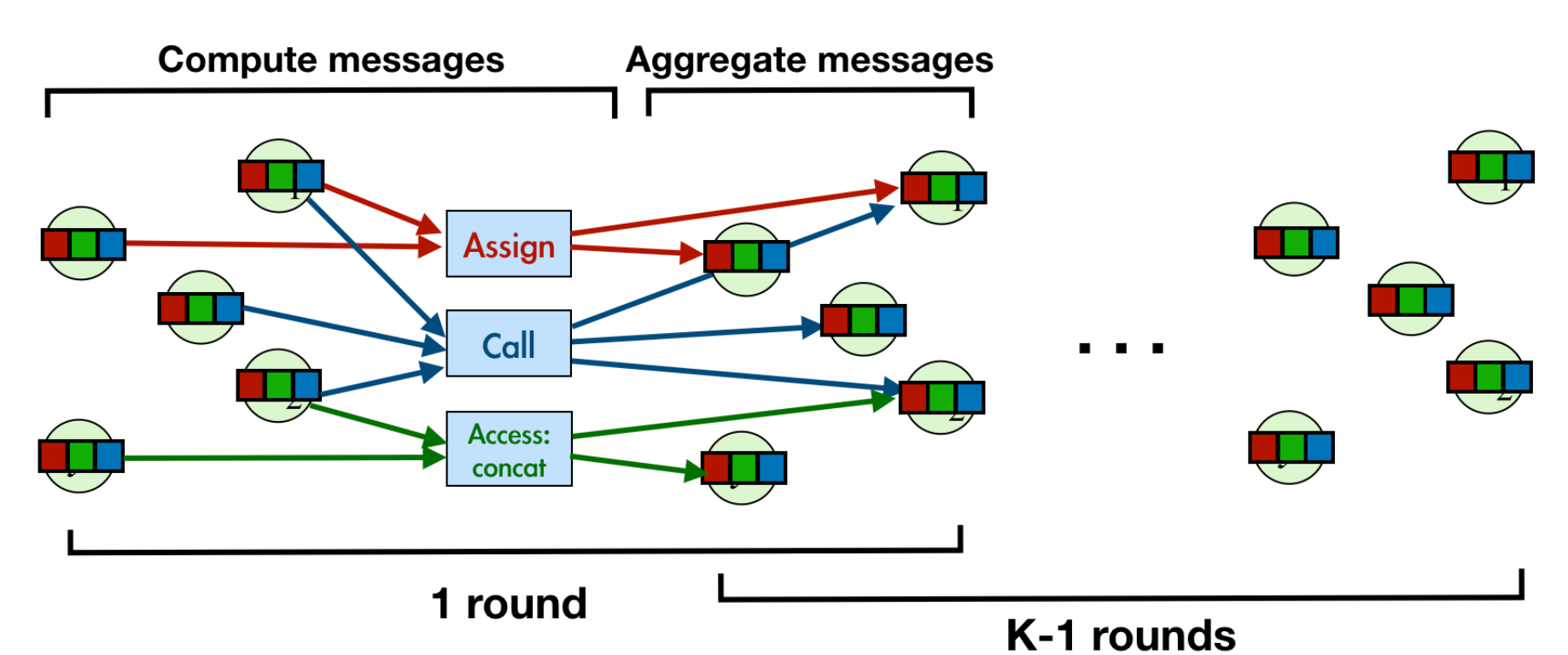


Figure 3: Example hyperedges for Figure 2. Edge labels in gray (resp. red) are positional arguments (resp. identifiers). (A) The return statement at line 6 induces a subtype relationship between T_{13} and T_5 . (B) MyNetwork T_8 declares attributes name T_1 and time T_2 and method forward T_9 . (C) T_{14} is associated with a variable whose name is restore. (D) Usage hyperedge for line 10 connects T_6 and T_{15} to all classes with a time attribute.

GNN Architecture



Our architecture performs K rounds of message-passing to compute embedding vectors for each type variable. We have a different network architecture for each edge type, and weights are shared between different edge instances of the same type.

Experimental Settings

- We collected 300 popular TypeScript projects from Github
 - 60 for testing, 40 for validation, the rest for training
 - Contain about 1.2 million lines of code in total
- Prediction space: All user-defined types + Some library types
 - Use training set to select 100 most frequent library types
- Hyperparameters:
 - 32-dimensional type embedding vectors
 - All MLPs use one hidden layer of 32 units
 - GNN layers have independent weights
- Used Adam to train the model
 - Learning rate linearly decreases from 10^{-3} to 10^{-4}
- We have made our code publicly available on Github