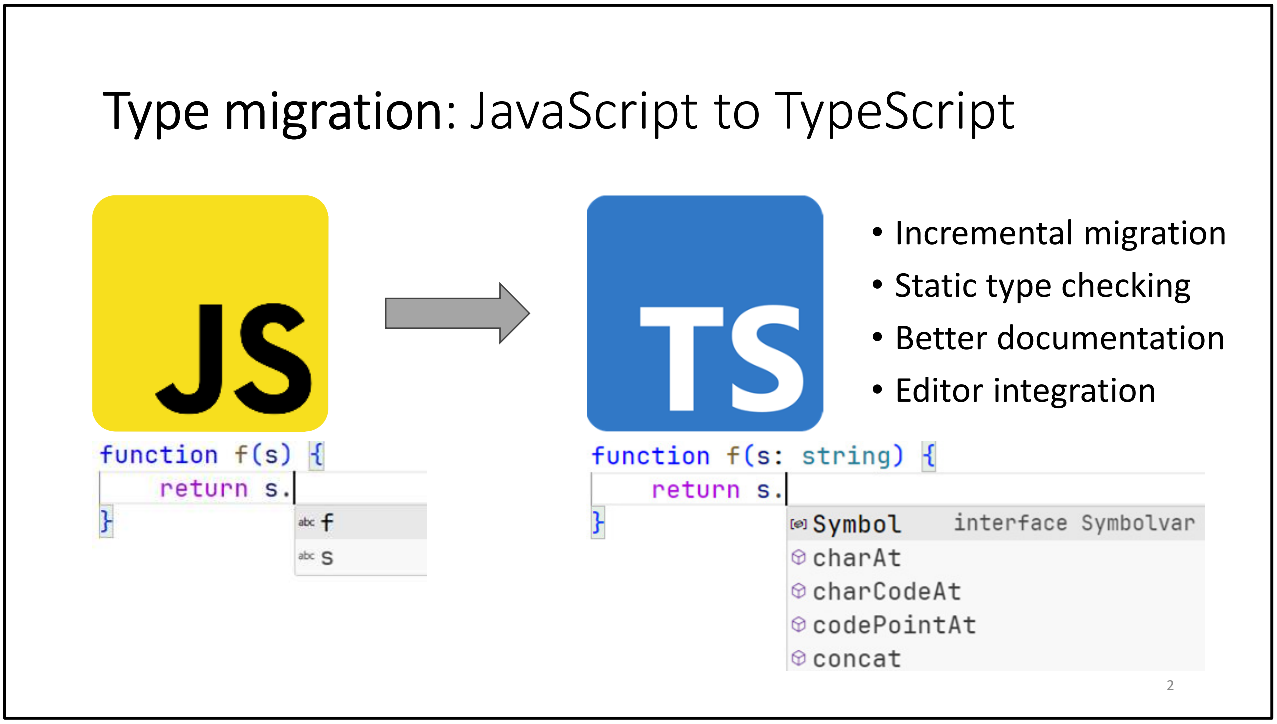
# Predicting TypeScript Type Annotations and Definitions With Machine Learning

Ming-Ho Yee
Northeastern University

September 13, 2023
Thesis Proposal

- Hello everyone, and thanks for coming to my thesis proposal.
- Today, I'll be talking about how we can use machine learning to predict TypeScript type annotations and definitions.




*0:10 to here (1:00 for this slide)*

- So, let's say we have a code base in JavaScript, and we want to migrate it to TypeScript.
    - We can do this by incrementally adding type annotations to our code.
    - As the program becomes more typed, we benefit from static type checking, better documentation, and editor integration.
- For example, on the left is a small code fragment.
    - The code is untyped, so the text editor can't provide any useful information.
- On the other hand, in the typed version of the code, s is a string.
    - As a result, the text editor can show the methods that are available on s.
- So, there are clear benefits for using TypeScript, and a migration path to get from JavaScript to TypeScript.
    - Unfortunately, manual type migration is a laborious process

## Machine learning for type prediction

*Predict the most likely type annotation for the given code fragment*

### Classification

```
function f(x) {
    return x + 1;
}
```

| Type of x | Probability |
|---|---|
| number | 0.4221 |
| any | 0.2611 |
| string | 0.2558 |
| *other* | |

### Large language models for code

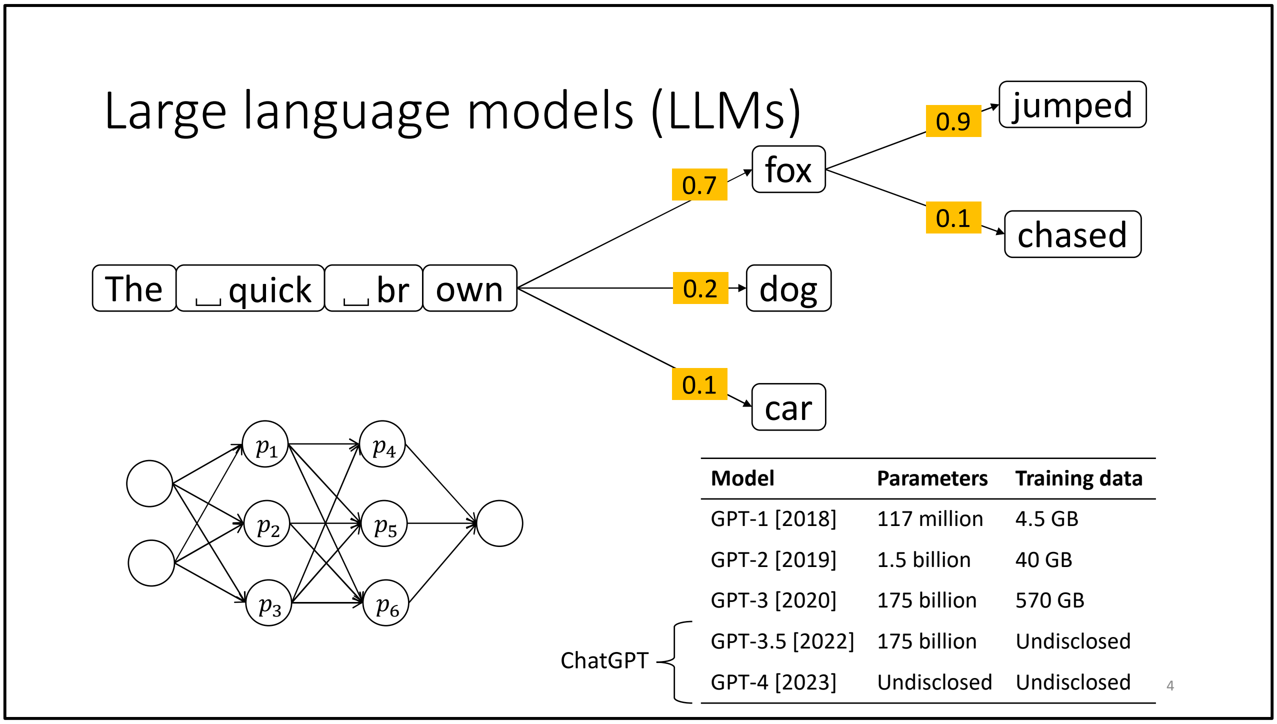
```
function f(x: _hole_) {
    return x + 1;
}
```

```
function f(x: number) {
    return x + 1;
}
```

3

*1:00 to here (1:30 for this slide)*

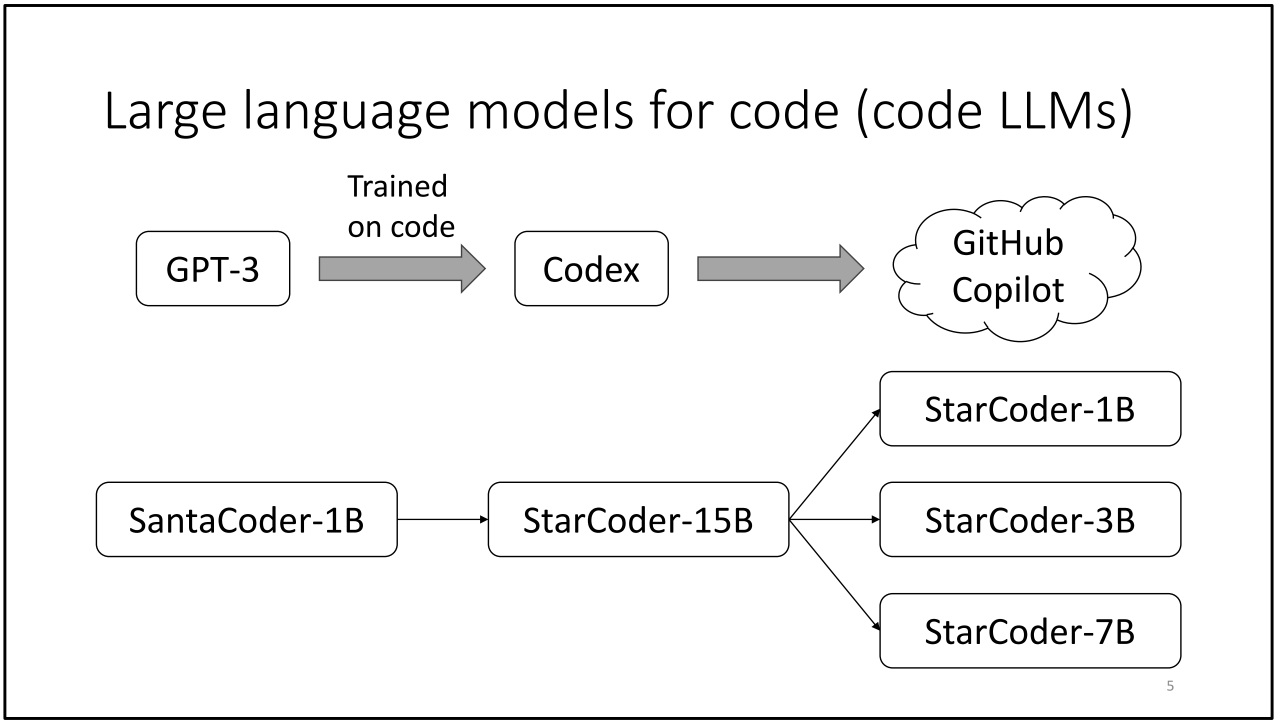
- To automate type migration, there has been research in using machine learning approaches.
    - The idea is to frame type migration as type prediction: “Predict the most likely type annotation for the given code fragment.”
- For this talk, I’ll group these approaches into two categories.
    - First are classification approaches.
        - These are older approaches where models are trained specifically for type prediction.
        - Given a code fragment, for each identifier, they produce a list of the most likely type annotations and their probabilities.
        - You can think of the output as a table of predictions, one for each identifier.
        - In the example, we can see a list of type predictions for x, which is the only identifier that can be annotated.
    - The other approach is to use large language models for code.
        - These models are trained for general-purpose code generation, but have become very popular for coding tasks in general.
        - Given a code fragment, they predict what code comes next.
        - Some models support fill-in-the-middle, which allows code generation to occur at arbitrary locations rather than at the very end.
        - For example, I’ve inserted a hole where the type annotation for x should be, and the model uses the surrounding context to predict number.




*2:30 to here (2:00 for this slide)*

- Let's talk about large language models, or LLMs.
- As an example, consider this fragment of English text.
    - We'll use this as input, also called a prompt, and the model will predict what words follow.
    - Technically, models operates on tokens, not words.
        - Let's tokenize this input, and I'll point out that a token may be smaller than a word.
    - Now, given these four tokens, the model returns a probability distribution over tokens.
        - "fox" is the most likely, followed by "dog", then "car.
    - We can select "fox" and append it to the prompt to create a new prompt, and get another probability distribution for the next token.
    - In this example, I just selected the most likely token at each step, but there are many different strategies.
    - In practice some kind of sampling is done, so the results will be nondeterministic.
- Language models are implemented as neural networks.
    - I won't go into detail, but you can think of it as a graph of nodes and edges.
    - Each node has a weight, also called a parameter, and these are the values that are adjusted during training.
    - This example has 6 parameters, but a large language model can have millions or billions of parameters.
- For example, let's look at the GPT family of models.

- You may recognize GPT-3.5 and GPT-4 as the models that power ChatGPT.
- We see that the parameter counts and training data are increasing exponentially, from millions to billions.
- Something like GPT-3 requires a datacenter-class GPU to run.




*4:30 to here (1:30 for this slide)*

- Large language models for code, or code LLMs, take the same idea.
    - For example, GPT-3 was trained on code to produce Codex, which is the model that powers GitHub Copilot, a service for code completion and programming assistance.
- GPT-3 and Codex are proprietary, so we don't know what they were trained on.
- For my research, I have used open code LLMs, such as SantaCoder and its successor StarCoder, where the parameters and training data are openly available.
    - However, StarCoder has 15 billion parameters, so just downloading it requires about 60 GB of disk space, and then you need a datacentre GPU to run it.
    - Fortunately, there are smaller versions of StarCoder, with 1, 3, and 7 billion parameters.
    - StarCoder-1B is small enough to run on consumer hardware.

# Fill in the middle (FIM)

## Training

```
<fim_prefix>function fact(n) {
<fim_suffix>return n * fact(n-1);
}<fim_middle>if (n == 0) return 1;
```

## Inference

```
function f(x: number) {
    return x + 1;
}
```

6

*6:00 to here (1:30 for this slide)*

- So far, we discussed left-to-right generation, where text is generated at the end of the prompt.
- Let’s now look at fill-in-the-middle.
- Training is done with a special format.
    - Let’s say we have this factorial example, and we want to train the model to generate the second line, conditioned on the surrounding lines.
    - We insert some special tokens that mark the prefix, middle, and suffix.
    - Then we move the middle to the very end, which has transformed the prefix and suffix into just a single prefix.
    - In other words, turned fill-in-the-middle into a left-to-right generation problem.
- For inference, when we use the model to generate text, we use the same format.
    - Let’s say we want to type annotate the parameter x.
    - We insert the special tokens, marking the prefix, suffix, and middle, and rearrange.
    - The model predicts that number should come after the special middle token.
    - So we extract that and reverse the transformation to get our result.

# Limitations of existing approaches

### Evaluation

```
function f(x: string) {
    return x * 1;
}
```

Do Machine Learning Models Produce TypeScript Types That Type Check? [ECOOP 2023]
Yee and Guha

### Fill in the Middle

```
function f(x: _hole_) {
    return x + 1;
}
```

Type Prediction With Program Decomposition and Fill-in-the-Type Training
[submitted to NeurIPS 2023]
Cassano, Yee, Shinn, Guha, and Holtzen

### Type Definitions

```
interface Point {
    x: number,
    y: number
}
```

Generating TypeScript Type Definitions With Machine Learning [proposed work]

7

*7:30 to here (1:30 for this slide)*

- Now that I've covered some background material, let's return to the original problem of type prediction.
- I have identified limitations of existing approaches, and my thesis aims to address them.
- First, there is the question of how to actually evaluate these systems.
    - The typical practice is to compute accuracy, but in my ECOOP paper, I argue that we should type check the type annotations.
- Second, there are challenges when using large language models with fill-in-the-middle out of the box.
    - My colleagues and I submitted a paper to NeurIPS where we address these challenges.
- Finally, I propose research on a new problem, that to my knowledge, has not been worked on.
    - This is the problem of generating type definitions.
    - Once you have predicted types, sometimes there are references to undefined types.
    - So I'd like to generate them.

# Thesis

Machine learning can be used to partially migrate JavaScript programs to TypeScript, by predicting type annotations and generating type definitions.

Do Machine Learning Models Produce TypeScript Types That Type Check? [ECOOP 2023]
Yee and Guha

Type Prediction With Program Decomposition and Fill-in-the-Type Training
[submitted to NeurIPS 2023]
Cassano, Yee, Shinn, Guha, and Holtzen

Generating TypeScript Type Definitions With Machine Learning [proposed work]

8

*9:00 to here (1:30 for this slide)*

- This now brings me to my thesis:
    - Machine learning can be used to partially migrate JavaScript programs to TypeScript, by predicting type annotations and generating type definitions.
- I want to go through each part of this thesis statement:
    - My research uses open-source code LLMs, specifically SantaCoder and StarCoder.
    - I'm focusing on a partial migration that predicts type annotations and generates type definitions.
        - I believe a full migration will involve other tasks like refactoring, and is beyond the scope of a single PhD.
    - Finally, I restrict my research to JavaScript and TypeScript, two of the most popular languages on GitHub and StackOverflow.
- To support my thesis, I make three contributions:
    - The two papers mentioned previously, as well as the proposed work.
- The rest of this talk will cover these three topics, and I'll start with the ECOOP paper.

Do Machine Learning Models Produce TypeScript Types That Type Check? [ECOOP 2023]

# Evaluating type prediction models

*What is the likelihood that a predicted type annotation is correct?*

$$\text{Accuracy} = \frac{\text{correct predictions}}{\text{total predictions}}$$

```
type S = number;
function f(w, x, y, z) { ... }
```

| Identifier | Ground truth | Prediction | |
|---|---|---|---|
| w | number | number | ✅ |
| x | A \| B | B \| A | ❌ |
| y | S | number | ❌ |
| z | number | any | ❌ |

**Accuracy: 0.25**

Limitations of accuracy:

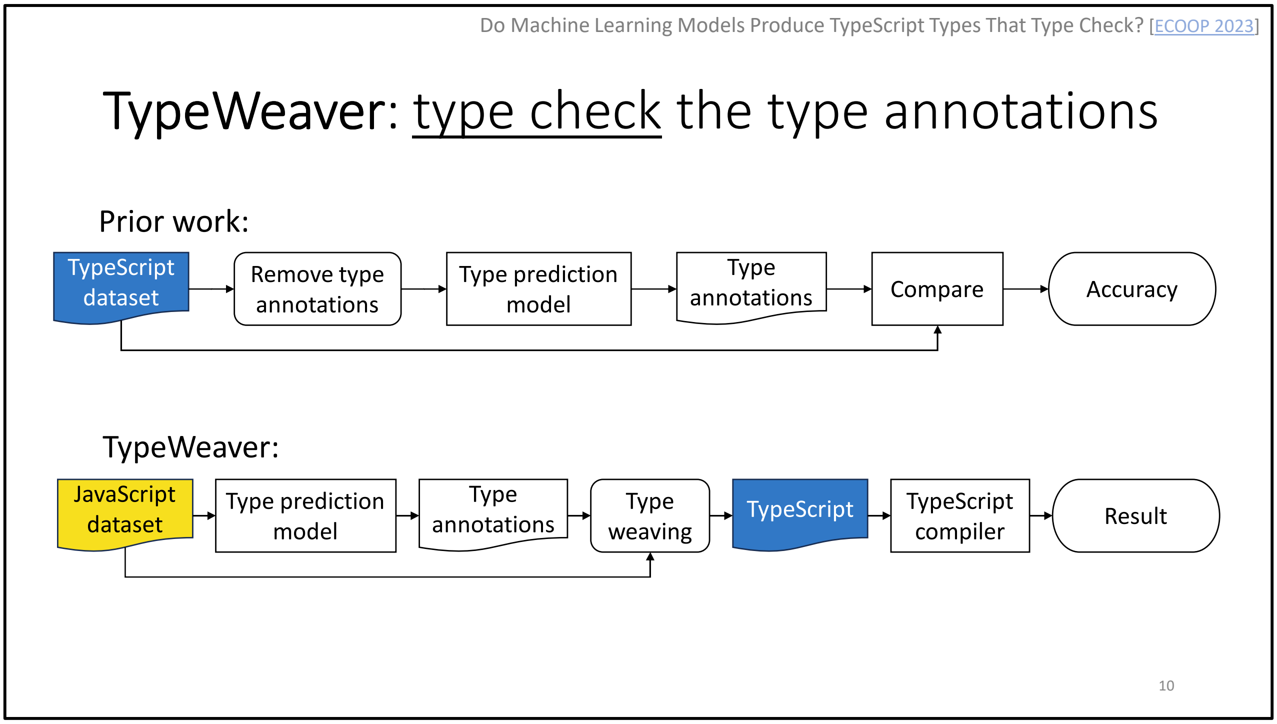
- Requires exact match
- Requires ground truth
- Predictions may not type check

9

*10:30 to here (2:00 for this slide)*

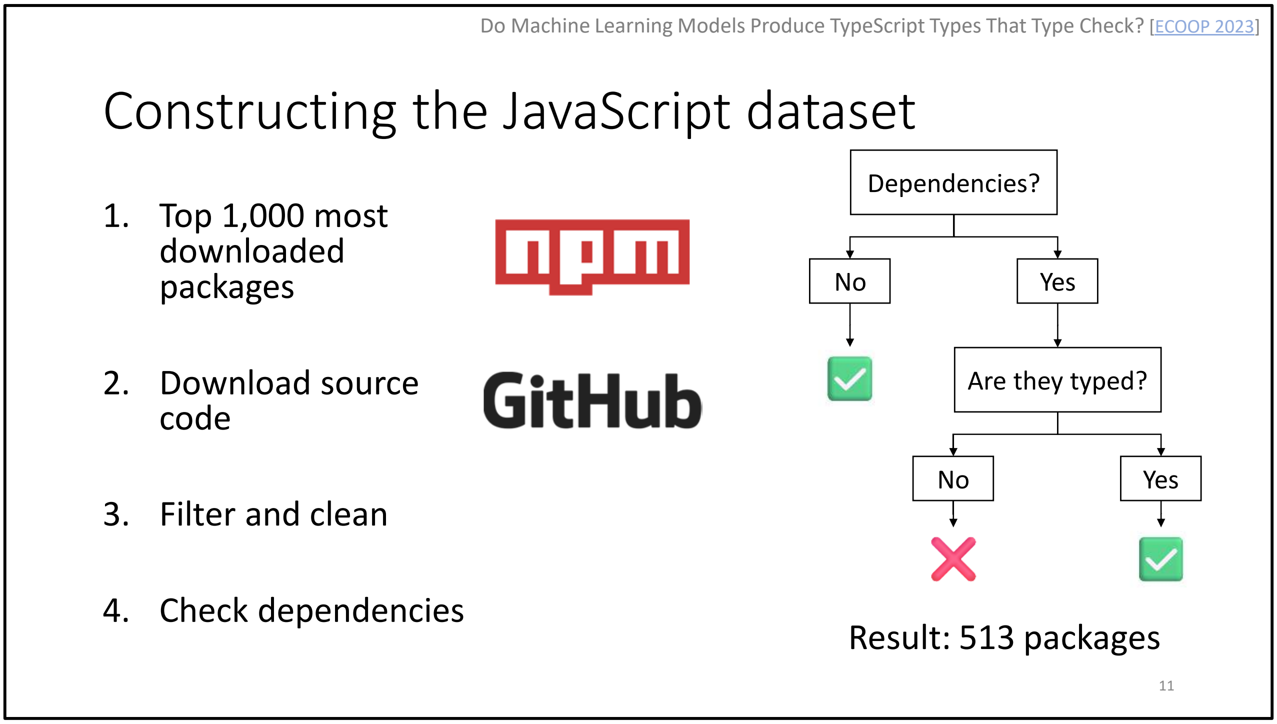
- The problem I address in the first paper is how to evaluate type prediction models.
- Prior work has used accuracy as an evaluation metric.
    - In other words, what is the likelihood that a predicted type annotation is correct?
    - This is defined as the number of correct predictions divided by the total number of predictions.
    - Correct means an exact textual match, and requires a ground truth of existing, handwritten type annotations.
- As an example, let's look at this code fragment, which has a type alias S = number.
    - Only the first prediction matches the ground truth, so the overall accuracy is 25%.
- But there are some limitations to accuracy:
    - It requires an exact match.
        - In the example, the type "A or B" is equivalent to "B or A," but is counted as incorrect.
        - Likewise, S is an alias for number, but because they're textually different, it's considered an incorrect match.
    - Another problem is that accuracy requires a ground truth.
        - So we can only do this evaluation on TypeScript code where someone has already annotated the code.
        - We could not do this with JavaScript.
    - Finally, the predicted types might not even type check.
        - In this example, we have the opposite problem.

    - The program type checks, but an accuracy of 25% suggests this is a bad result.
- So maybe we should use a metric that is more useful for a programmer trying to migrate JavaScript code to TypeScript.




*12:30 to here (1:30 for this slide)*

- As the first contribution to my thesis, I propose type checking the type annotations, and built TypeWeaver to do this.
- Let's first walk through the existing evaluation workflow:
    - We start with a TypeScript dataset.
    - The type annotations are removed, and the untyped code is given to a type prediction model, which produces type annotations.
    - The predicted type annotations are then compared to the original type annotations, and accuracy is computed.
- TypeWeaver takes a different approach:
    - We start with a JavaScript dataset, because I want the evaluation to reflect how these systems are used in practice, where you migrate from JavaScript to TypeScript.
    - Next, the dataset is given to a type prediction model.
    - Then, there's a step called type weaving, which combines the type annotations with the original JavaScript code to produce TypeScript.
    - This allows running the type checker on the code to get a result.
- I'll be covering pieces of this pipeline for this section of the talk.




*14:00 to here (1:30 for this slide)*

- To construct the evaluation dataset, I started with the top 1,000 most downloaded packages from the npm Registry.
- Next, I downloaded the package source code from GitHub.
    - This is to make sure I get the original code that developers work on.
    - I don't want compiled or minified code.
- Then I apply several filtering and cleaning steps.
    - For example, some packages do not contain any code, or were implemented in some other language, so I filter those out.
- Finally, I check the package dependencies.
    - This is important, because when type checking a package, I need to handle its dependencies.
    - If there are no dependencies, then I can use the package as-is.
    - If the package has dependencies, I check if those dependencies are typed.
        - By typed, I mean that someone has written type declarations and uploaded them to DefinitelyTyped.
        - DefinitelyTyped is a community-maintained repository of type declarations, so that JavaScript packages can be used in TypeScript projects.
    - If there are no type declarations for a dependency, then I discard the package.
    - In other words, I ensure that if a package has dependencies, then all those dependencies are typed.
- This results in a final dataset of 513 packages.

Do Machine Learning Models Produce TypeScript Types That Type Check? [ECOOP 2023]

# Type weaving: JS + type annotations = TS

```
function f(x: string, y: number): string {
    return x + y;
}
```

```
FunctionDeclaration
  Identifier
  Parameter
    Identifier
  Parameter
    Identifier
  Block
    ReturnStatement
      …
```

| Token | Type | Probability |
|---|---|---|
| function | | |
| f | string | 0.6381 |
| ( | | |
| x | string | 0.4543 |
| , | | |
| y | number | 0.4706 |
| ) | | |
| { | | |
| return | | |
| x | number | 0.3861 |
| + | | |
| y | number | 0.5039 |
| ; | | |
| } | | |

12

*15:30 to here (2:00 for this slide)*

- Now I want to talk about type weaving, the step where we combine predicted type annotations with JavaScript to produce TypeScript.
    - This is needed for the classification approaches that generate type annotations.
- As an example, let's say we have this JavaScript function as input, and the table of type predictions.
    - For this example, I'm only showing the top, most likely type annotation for each identifier, and I've cleaned up the table.
    - In general, you can assume more columns for additional, less likely type annotations.
- The problem we have is that we can't directly type check these results. We need type weaving.
    - First, we use the TypeScript compiler to parse the JavaScript to get an abstract syntax tree.
    - Now we traverse the syntax tree, and every time we encounter a declaration node, we look up the type prediction from the table, and update the program.
        - In this example, we find the function f has return type string, x is string, and y is number.
        - There are other types in this table, assigned to other identifiers, but we ignore them for simplicity.
- The result is an annotated TypeScript file, which can be type checked.
    - By the way, this program actually type checks, even though it was not intended.
    - The number is coerced to a string and + is string concatenation, so the function returns a string.

# Type prediction front end

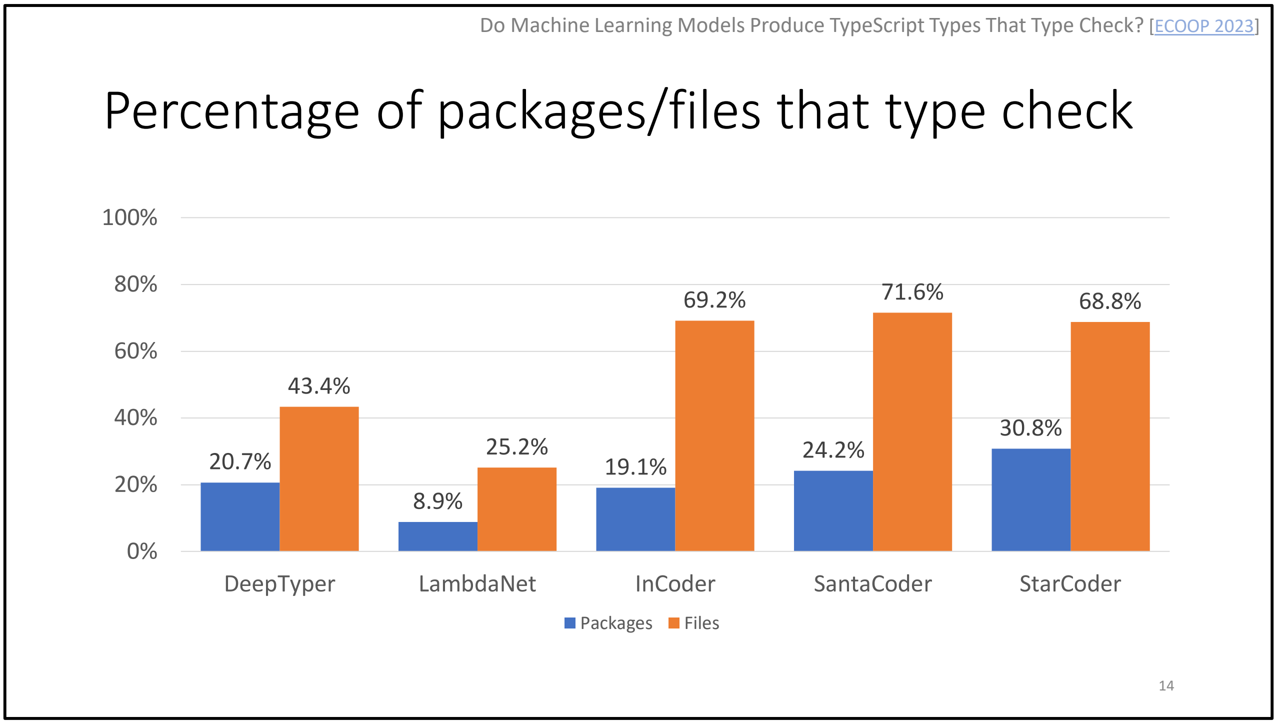
Result

```
function sum_list(l: any[]) {
    let sum = 0;
    for (let i = 0; i < l.length; i++) {
        sum += l[i];
    }
    return sum;
}
```



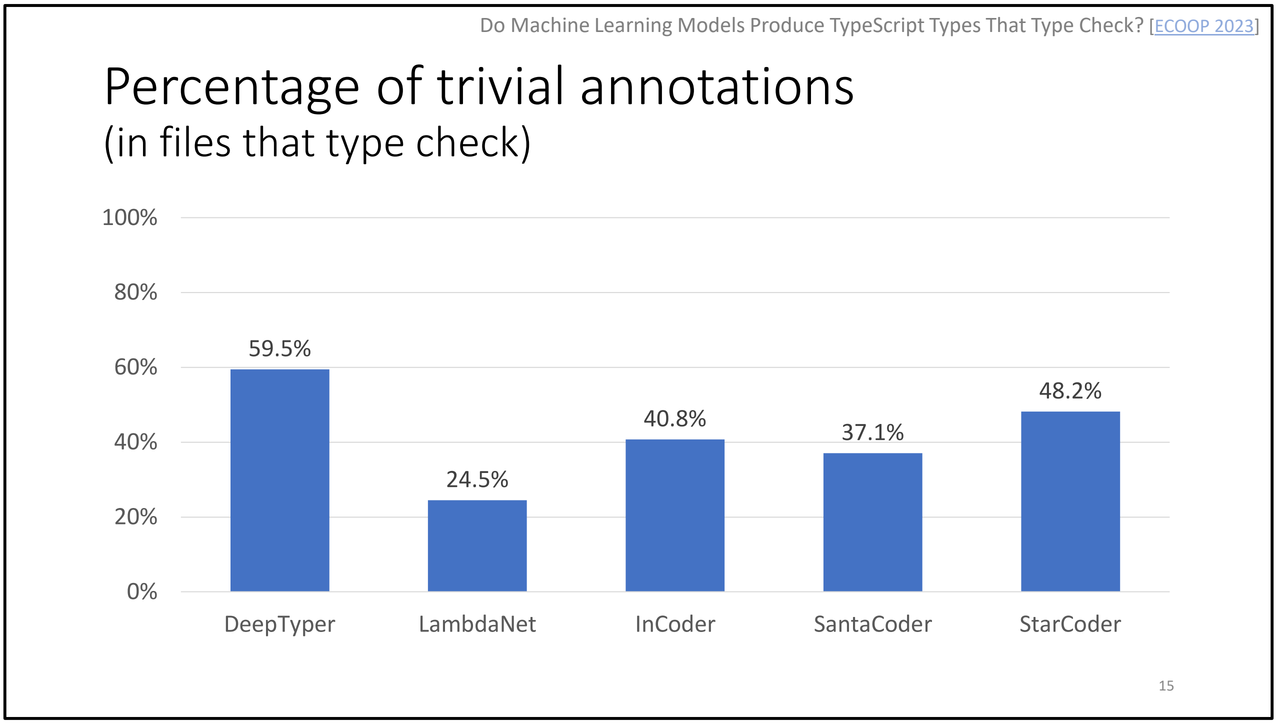
*17:30 to here (1:30 for this slide)*

- That was type weaving, which is for models that generate tables of type predictions.
- Now I want to talk about the type prediction front end, which is required when we use large language models for type prediction
- The front end takes a JavaScript program as input.
    - Next, it inserts a holes at the type annotation locations, one location at a time.
    - Then the program is transformed into the appropriate format for fill-in-the-middle.
    - The model returns a completion.
    - In this case, the model has generated a lot of code, more than just a type annotation.
    - So we'll use a parser to extract the first type annotation, and then insert it into the original program.
- In fact, this problem of generating extra code happens a lot, and I'll come back to this when discussing the next paper.



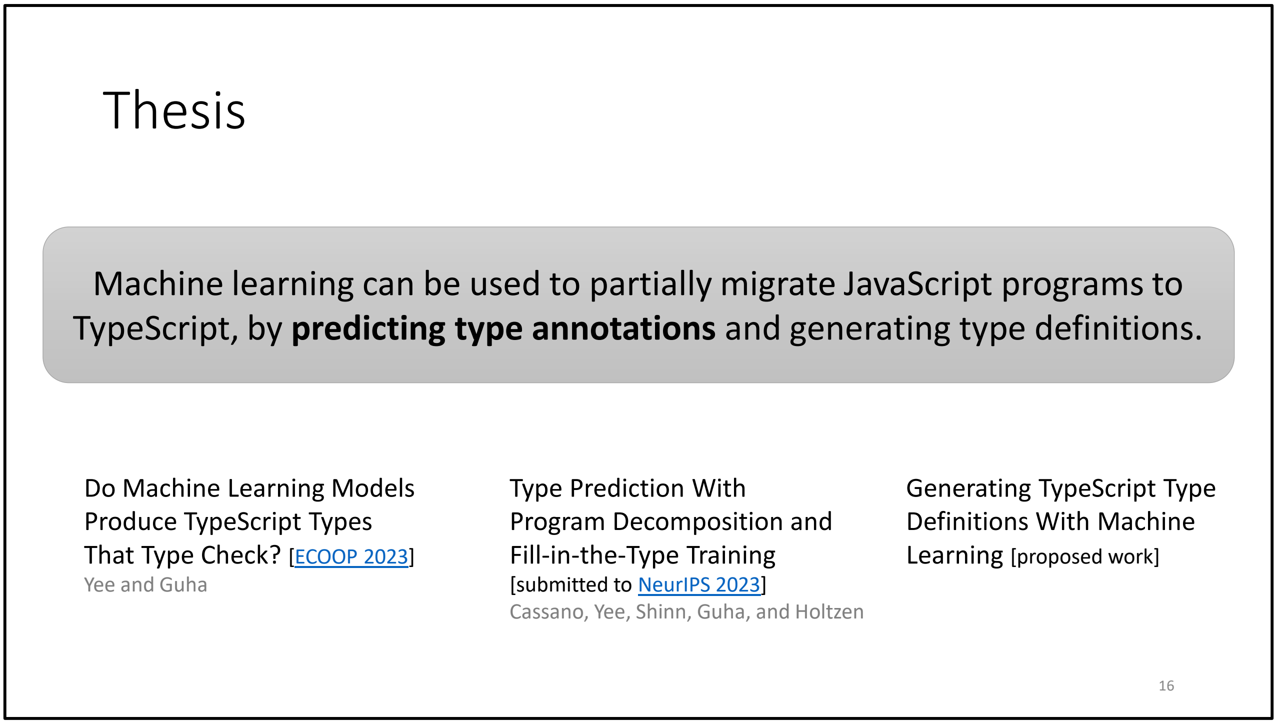

*19:00 to here (2:00 for this slide)*

- Now we can type check packages and look at results.
- For the paper, I evaluated three type prediction systems.
    - The first two are DeepTyper and LambdaNet.
        - These are "classification" approaches and require type weaving.
        - DeepTyper, an early system from 2018, uses a bidirectional recurrent neural network architecture.
        - LambdaNet, from 2020, uses a graph neural network.
    - The third system was InCoder from 2023, which is a code LLM that supports fill-in-the-middle.
- Since then, I evaluated two more recent code LLMs that support fill-in-the-middle: SantaCoder and its successor StarCoder.
- Now we can ask the question: what percent of our dataset type checks with these systems?
    - It's promising that the newer models are showing improvement.
    - But overall, I would say this result is disappointing, but not surprising.
    - Requiring an entire package to type check is a very high standard to meet, and even a single incorrect type annotation will cause the entire package to fail.
- So let's ask a more fine-grained question, and look at whether files type check.
    - The intuition is that if you're migrating a package, it makes sense to do so one file at a time, and triage based on which files do or do not type check.
    - And now the results are more encouraging.



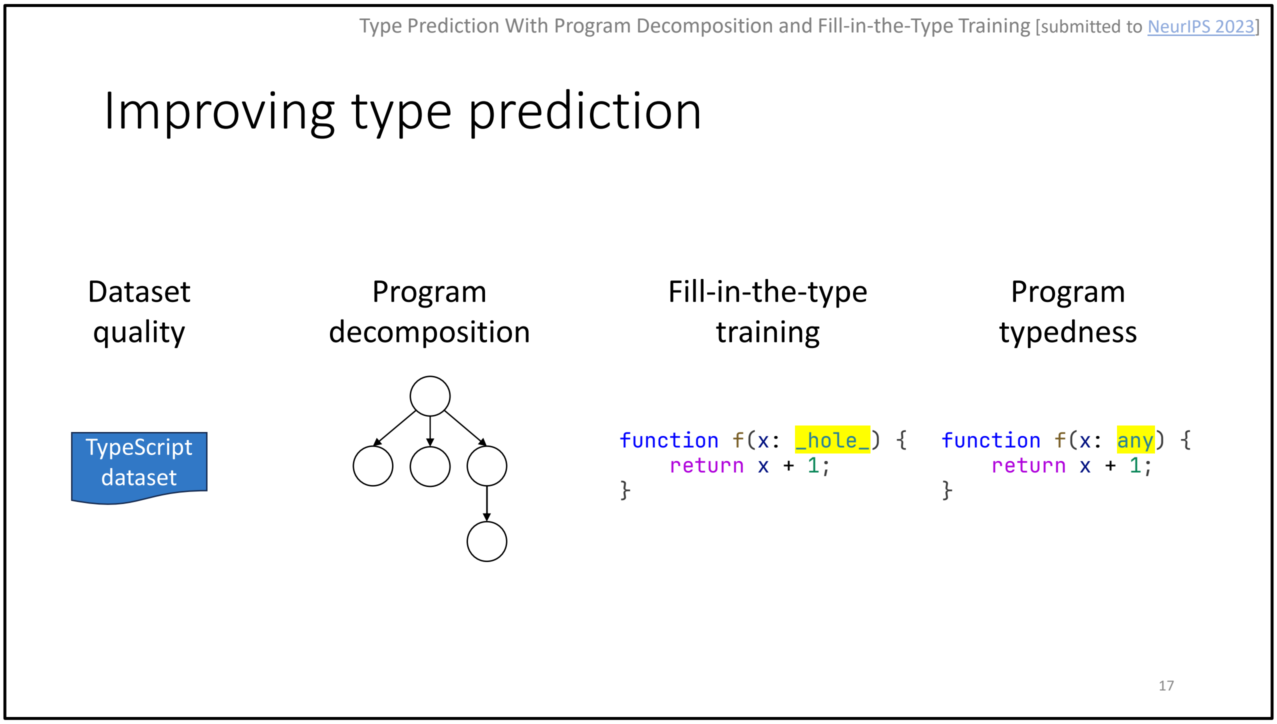

*21:00 to here (1:00 for this slide)*

- But one question we should ask: what happens if a system type annotated everything with "any"?
    - This would type check, but the type annotations aren't very helpful.
    - We prefer type annotations that are precise, and contain useful information to the programmer.
- So going back to the results for the files that type check, let's see what percentage of type annotations are trivial.
    - By trivial, I mean "any," array of any, or the generic Function type.
- The results are okay.
    - It looks like the systems generally predict non-trivial types.
    - But there is room for improvement.



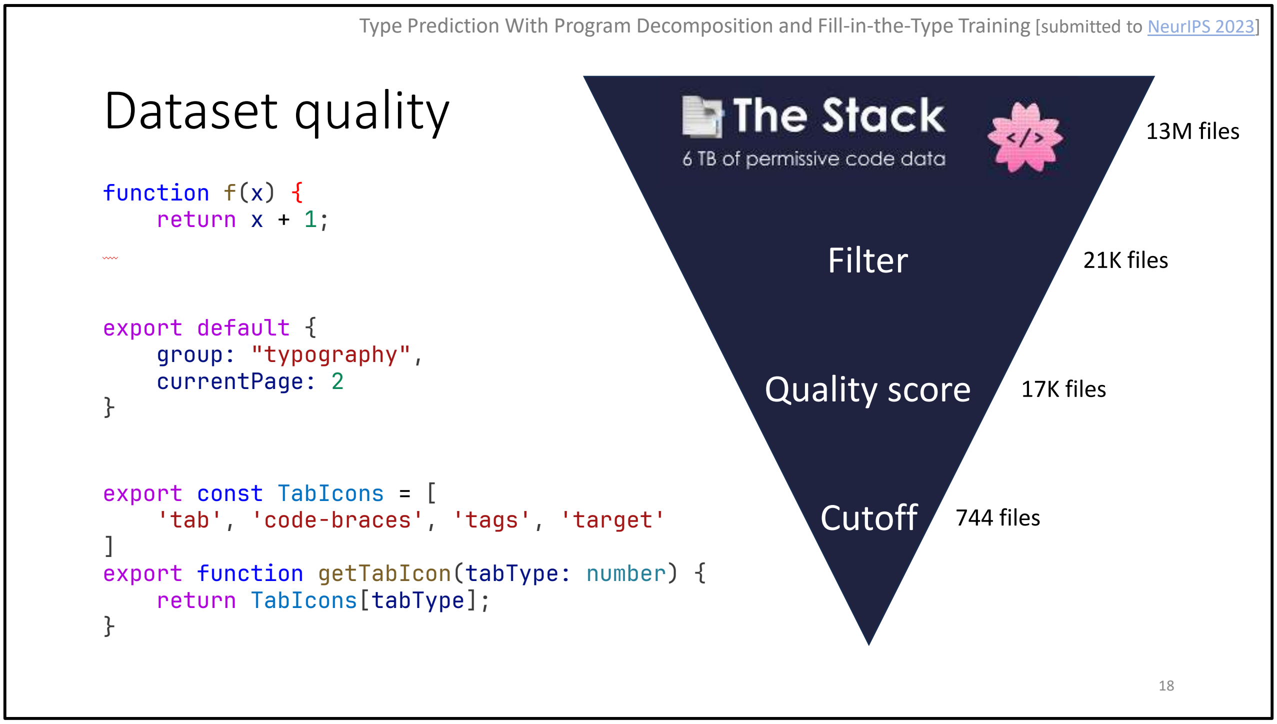

*22:00 to here (0:30 for this slide)*

- That was the first part of my talk on the ECOOP paper for evaluating type prediction systems.
- Now I'm moving on to the second part of the talk.
    - I'll be discussing how we can predict type annotations.
    - This is the paper my colleagues and I submitted to NeurIPS, "Type Prediction With Program Decomposition and Fill-in-the-Type Training."




*22:30 to here (1:00 for this slide)*

- The work in this paper builds on lessons we learned from TypeWeaver, with the goal of improving type prediction.
    - There are four parts to this work.
- First, we revisit the dataset, with an emphasis on better dataset quality.
    - This time we use a TypeScript dataset.
    - This is a trade-off: using TypeScript may be a less realistic evaluation, but it avoids the problem of JavaScript files that cannot be migrated without refactoring.
- Second, instead of trying to type annotate an entire program at a time, we decompose the program into smaller subprograms.
    - We can represent this as a tree, and we type annotate each subprogram at a time.
- Third, we use fill-in-the-type for type prediction.
    - This is a variant of fill-in-the-middle that we designed.
- Fourth, to avoid the problem of a system predicting “any” for all type annotations, we introduce a metric to measure the amount of type information in a prediction.
- I’ll cover these four improvements in this section of the talk.




*23:30 to here (3:00 for this slide)*

- First, let's start with the dataset
    - As I said before, I want a TypeScript dataset because some JavaScript files cannot be typed without refactoring.
- Let me show some more examples of programs that I'd like to exclude from a dataset.
    - First, we have a function that has a syntax error, so it will always fail to type check.
    - Second, we have a file that exports a dictionary. There is nothing to type annotate, so it will always type check.
    - Third, we have a very short file that isn't trivial.
        - We can annotate it in a way that it will type check, or choose a different annotation so that it fails to type check.
        - But the code isn't really doing anything interesting.
- To construct the dataset, I started with The Stack, a dataset of 6TB of permissively licensed code, in 30 programming languages.
    - TypeScript is one of those languages, and The Stack contains 13 million TypeScript files.
- Next, I do some filtering to remove files that are not appropriate for evaluation.
    - This includes files that do not already type check, have no type annotation sites, no functions, fewer than 50 LOC, and fewer than 5 LOC per function.
    - This reduced the dataset to 21,000 files.
- Then, I computed a quality score for the remaining files.
    - This score was designed to maximize the number of type annotation sites and type definitions, as well as the LOC per function.

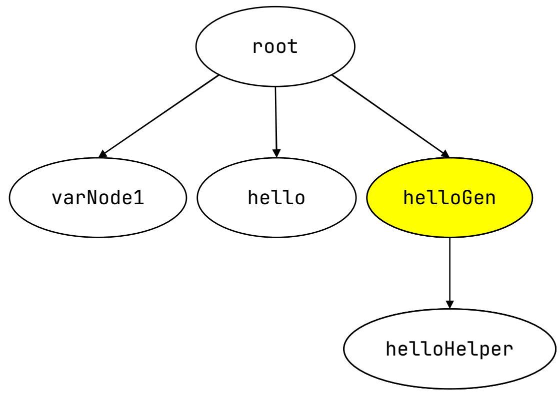
    - It also minimizes dynamic features like eval, trivial and builtin type annotations.
    - By filtering on this score, I was left with 17,000 files.
- Finally, I apply a time-based cutoff (Dec 31 2021).
    - Files before this cutoff are potentially used for training, and files after are used for evaluation.
- This results in a final dataset of 744 files.

# Program decomposition

```
let greeting = "Hello";
let suffix = "!";

// Produces a greeting for the given name
const hello = (name) => {
    return greeting + " " + name;
};

function helloGen(name): () => string {
    const helloHelper = (): string => {
        return hello(name) + suffix;
    };
    return helloHelper;
}
```






*26:30 to here (2:00 for this slide)*

- Now that we have our dataset, let’s take a file from that dataset and try to migrate it.
    - First we need to decompose it into smaller subprograms.
- If you recall from the introduction, a program like the one here is given to a language model and then tokenized.
    - But a problem is that language models can only accept a limited number of tokens as input.
    - This limit is called the “context window,” and some common limits are 2000 or 4000 tokens.
- To support these limits, we would like to split a program into smaller subprograms.
- Our approach is to follow the declaration hierarchy in a program.
- Going by the example, we start with a root node that represents the top-level of the program.
    - Next, we have a special varNode for the top-level variable declarations.
    - Then we have the hello function as a new node.
        - The comment is included in the node, because it provides additional context.
    - Next, we have helloGen.
    - Finally, helloHelper is a child node of helloGen, because it is an inner definition of helloGen.
- Now we’ve decomposed the program into a tree, and this induces a bottom-up traversal order.
    - We want to predict types for helloHelper before helloGen.
    - Then, when the model predicts types for helloGen, the context already contains the types for helloHelper.

Type Prediction With Program Decomposition and Fill-in-the-Type Training [submitted to NeurIPS 2023]

# Fill-in-the-type training

Fill in the middle

```
<fim_prefix>function sumThree(a: number, b:
<fim_suffix>}
<fim_middle>number, c: number): number {
    return a + b + c;
```

Fill in the type

```
<fim_prefix>function sumThree(a: number, b:
<fim_suffix>, c) {
    return a + b + c;
}<fim_middle>number
```

20

*28:30 to here (2:00 for this slide)*

- Now that we've decomposed our program, we want to run type prediction for each subprogram.
    - The problem is that we can't use fill-in-the-middle out of the box.
- Recall this example from earlier.
    - We would like to fill in the type annotation for "l."
    - But when we use fill-in-the-middle, the large language model generates more code than needed.
        - It generates an entire function implementation, not just the type annotation.
    - We have a front end that extracts the type annotation, but we really would like the model to generate just a single type.
- The reason for this is because of the way fill-in-the-middle was trained.
    - Let's look at a simpler example as a reminder.
        - We start with a type annotated function.
        - Next, we randomly select a middle span.
        - We transform it to the fill-in-the-middle format.
        - Now we train the model on this entire sequence.
- We want to tweak this process slightly, and we call it "fill-in-the-type" because it is trained specifically for type prediction and not arbitrary code generation.
    - The main difference is that instead of selecting an arbitrary span, we select just a type annotation.
    - Next, we remove type annotations from the suffix.
        - This is so we get an input that reflects the input during inference, where the prefix

is type annotated but the suffix is not.

- This example shows the case where “a” is already annotated, “b” is what we’re training the model to annotate, and “c” hasn’t been annotated yet.
- Now we can insert the special tokens and rearrange, and this is the format we give to the model for training.
    - The hope is that the model is trained to produce only a single type annotation when it sees this input.

Type Prediction With Program Decomposition and Fill-in-the-Type Training [submitted to NeurIPS 2023]

# Program typedness

Both programs type check

```
function f(x: any) {
    return x + 1;
}
```

Score: 500

```
function f(x: number) {
    return x + 1;
}
```

Score: 0

| Type annotation | Score |
|---|---|
| unknown | 1.0 |
| any | 0.5 |
| Function | 0.5 |
| undefined | 0.2 |
| null | 0.2 |

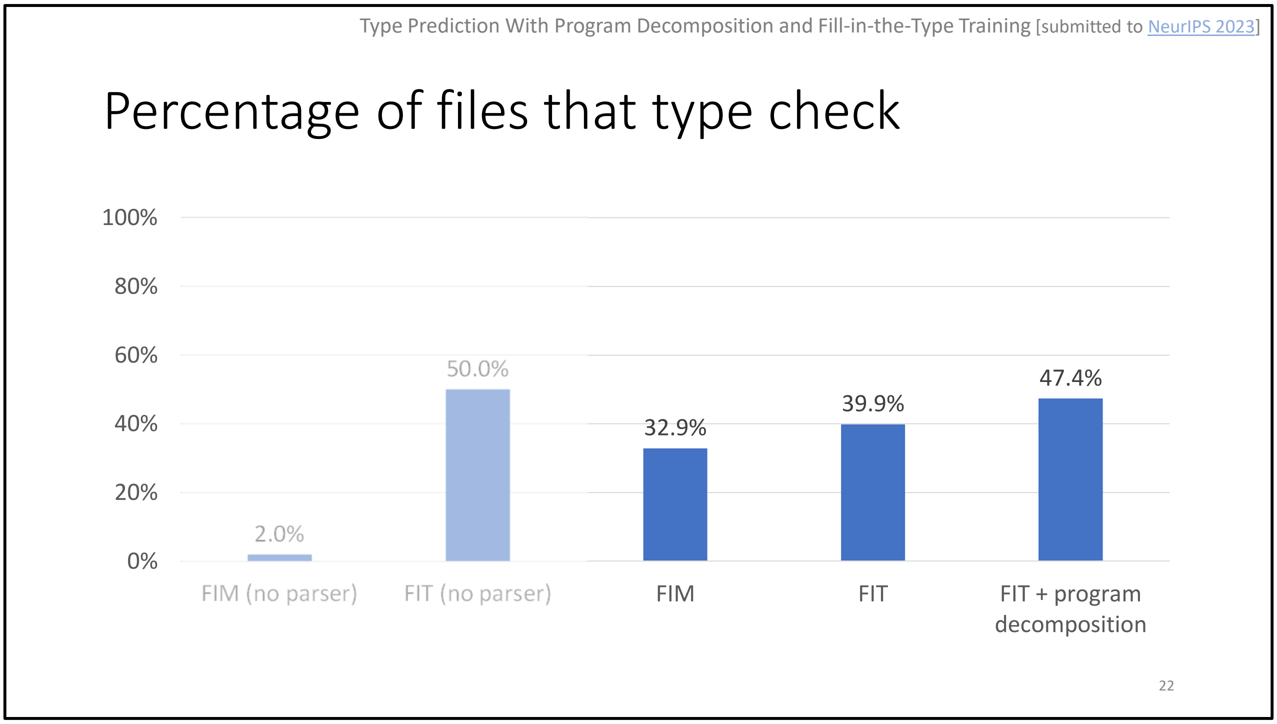
We also use this metric during type prediction

21

*30:30 to here (2:30 for this slide)*

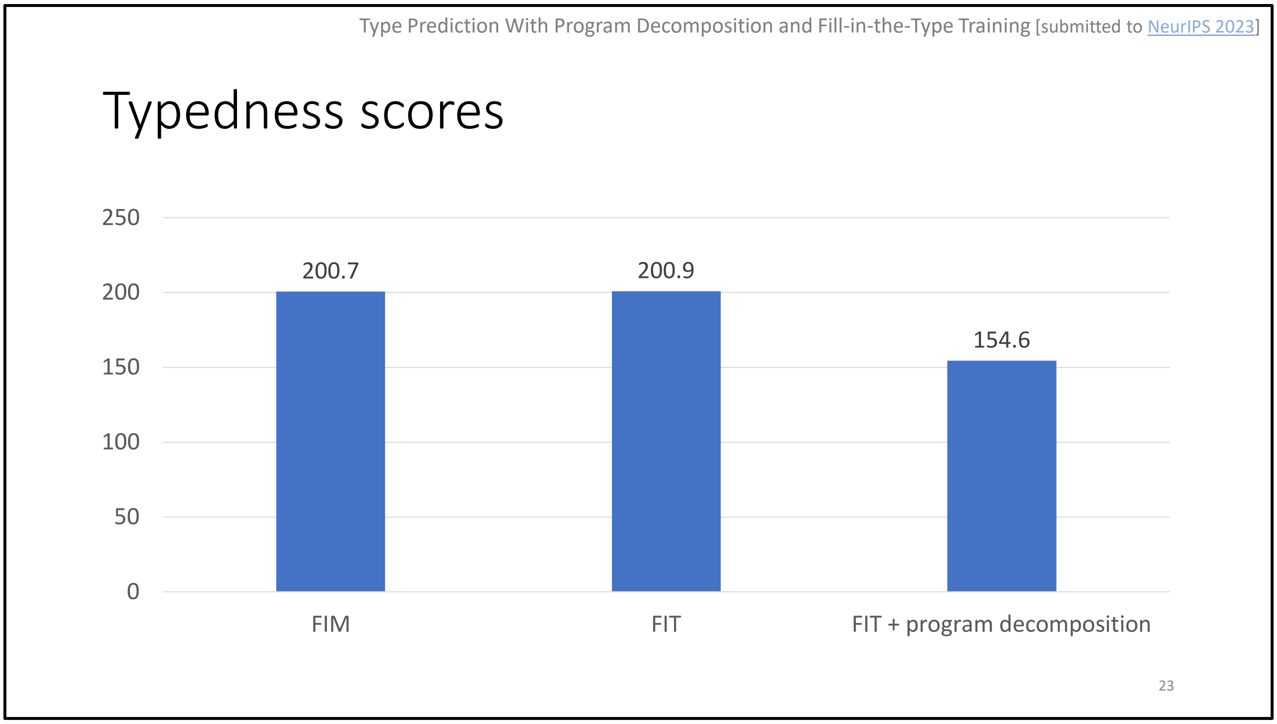
- So we’ve taken a file from our dataset, decomposed it, and predicted types for it using fill-in-the-type.
    - Recall from TypeWeaver that we had a concern: what if a type prediction system generated “any” for all types?
    - We introduce a metric called program typedness to account for this.
- For example, consider the two functions on the slide.
    - Other than the type annotation for x, they are both the same.
    - If the evaluation metric is whether the program type checks, then both functions are considered equally good.
    - However, we prefer the function on the right, as “number” has more information than “any.”
- Our approach is to count the number of undesirable type annotations, and use them to compute a score.
    - The table on the right has annotations we consider to be undesirable, and a score for how “bad” they are.
    - “unknown” is the worst because it causes type errors: an unknown value can only be assigned to the any type.
    - “any” and “Function” are the next worst.
        - “any” will always type check, and “Function” will type check as long as its value is used as a function.
    - “undefined” and “null” are undesirable but do carry some information; they suggest that

  - a value is uninitialized or missing.
  - All other types have a score of 0.
- So to compute the typedness score for a program, we iterate over the type annotations in a program, assign scores, and then sum.
- Finally, we normalize to a number between 0 and 1000, where 0 is the best, and 1000 means every type annotation is “unknown,” the worst type.
- In this example, the function on the left has a typedness score of 500.
- The typedness score actually serves two purposes.
  - We use it to evaluate how typed a program is.
  - But we also use it during inference, as a search metric.




*33:00 to here (1:30 for this slide)*

- Now we can talk about experiments and results.
- Our first experiment is to compare fill-in-the-middle with fill-in-the-type.
    - For this experiment, we fine-tuned SantaCoder on TypeScript to get the first result, and then we further fine-tuned it for fill-in-the-type to get the second result.
    - In this experiment, we modified the type prediction front end.
        - Instead of using a parser to extract a type annotation, it simply accepts whatever was returned.
    - We use a subset of the dataset, with only 50 files.
    - In this experiment, fill-in-the-type is significantly better than fill-in-the-middle.
- For the second experiment, we repeat the first experiment with the full dataset and the parser enabled.
    - The numbers don't exactly match the first experiment because this is using the full dataset.
    - But we see that fill-in-the-type is more successful than fill-in-the-middle at producing types that type check.
- Finally, we run program decomposition with fill-in-the-middle.
    - This experiment performs better than fill-in-the-type without program decomposition.




*34:00 to here (0:30 for this slide)*

- Next, let's look at the typedness scores.
- Again, fill-in-the-middle and fill-in-the-type do not do program decomposition.
    - They both have similar typedness scores.
- Program decomposition also uses the typedness score to search for better predictions.
    - So it achieves a much better typedness score.

# Thesis

Machine learning can be used to partially migrate JavaScript programs to TypeScript, by predicting type annotations and **generating type definitions**.

Do Machine Learning Models Produce TypeScript Types That Type Check? [ECOOP 2023]
Yee and Guha

Type Prediction With Program Decomposition and Fill-in-the-Type Training
[submitted to NeurIPS 2023]
Cassano, Yee, Shinn, Guha, and Holtzen

Generating TypeScript Type Definitions With Machine Learning [proposed work]

24

*35:00 to here (0:30 for this slide)*

- That was the second paper.
- Now we have reached the last part of the talk.
- The last part of my thesis is about using machine learning to generate type definitions.

# Problem definition

```
function dist(p1: Point, p2: Point) {
    const dx = p2.x - p1.x;
    const dy = p2.y - p1.y;
    return Math.sqrt(dx*dx + dy*dy);
}

interface Point {
    x: number,
    y: number
}
```



*35:30 to here (1:00 for this slide)*

- To illustrate the problem, let's look at an example.
- Here is a function "dist" that computes the distance between two points.
- With the work I have already done, I can use a machine learning model to predict type annotations for p1 and p2.
    - I actually tried this, and the model got it right: p1 and p2 are annotated as Point.
- However, this example will not type check, because Point is not defined.
- I want to use machine learning to generate this Point definition.
- The intuition is that a large language model has been trained on a lot of code, so that it associates the type annotation "Point" with this function definition.
    - During training, the model must have encountered the Point type and its definition.

# Approach

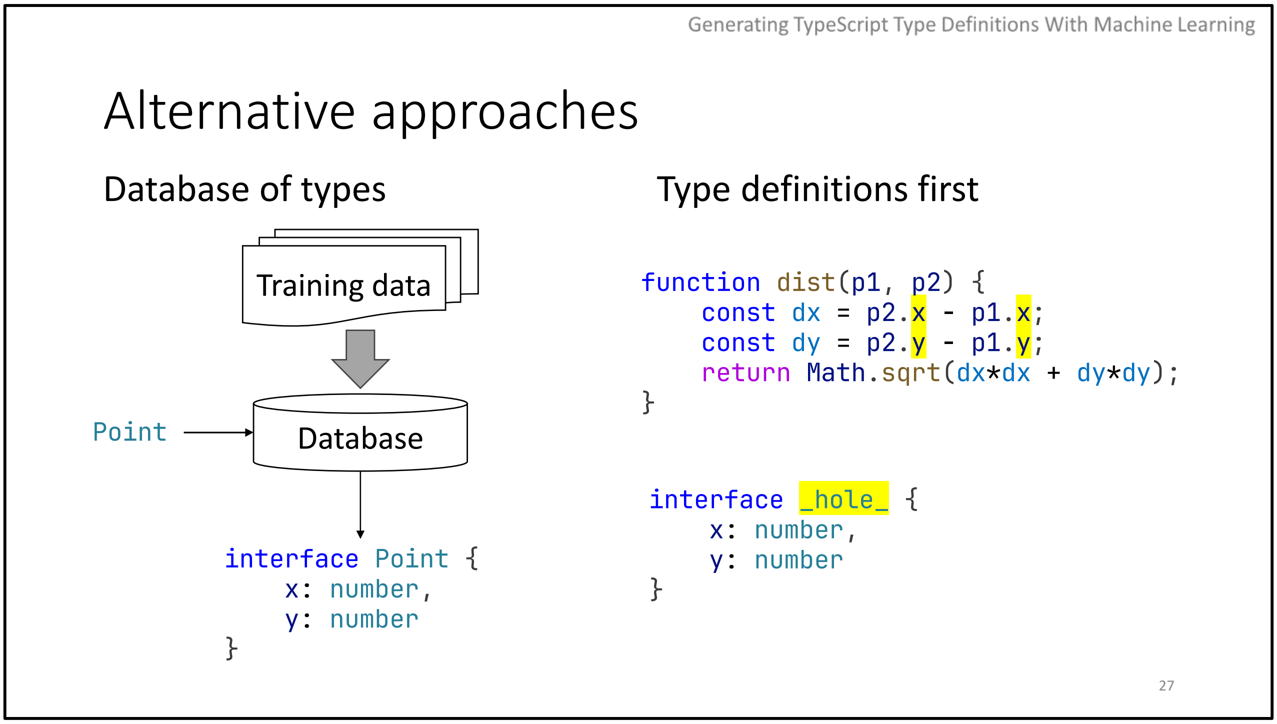
```
<commit_before>function dist(p1: Point, p2: Point) {
    const dx = p2.x - p1.x;
    const dy = p2.y - p1.y;
    return Math.sqrt(dx*dx + dy*dy);
}
<commit_msg>Add the type definition for Point
<commit_after>interface Point {
    x: number,
    y: number
}

function dist(p1: Point, p2: Point) {
    const dx = p2.x - p1.x;
    const dy = p2.y - p1.y;
    return Math.sqrt(dx*dx + dy*dy);
}
```



*36:30 to here (2:00 for this slide)*

- My proposed approach is to fine-tune StarCoder-1B, similar to how fill-in-the-middle works
- As I discussed in the introduction, StarCoder is an open-source large language model for code.
    - The 1 billion parameter version is small enough to use with a consumer GPU.
- One nice thing about StarCoder is that it was trained on a variety of formats, expanding its capabilities.
    - For example, it was trained on Git commits, in the following format.
    - The special tokens denote original code before the commit, the commit message, and the updated code after the commit.
    - This way, the model learns to associate the commit message, which is a natural language instruction, with code before and after following that instruction.
- For example, I can take a TypeScript program and treat it as the commit after.
    - Then I can remove types and make it the code before a commit.
    - Finally, I can make the commit message the instruction “Add type definitions and interfaces.”
- I don’t think it’s feasible to train a new model from scratch or fine-tune StarCoder on a new format.
- And even within this format, there’s a lot of room to experiment.
    - I could try a different commit message.
    - I could go from partially typed to fully typed.
    - Or I could handle type annotations first, and then add type definitions later.




*38:30 to here (2:00 for this slide)*

- There are two alternative approaches I'd like to quickly discuss.
- First, there is the "database of types" approach.
  - The idea is to preprocess all the training data to extract type definitions and insert them into a database.
  - Then, if a program refers to some undefined type like Point, we can query the database and get the definition.
  - There are some problems with this approach.
    - There could be multiple definitions for a single type.
    - Or the database could be too large to be practical.
- The other alternative is "type definitions first."
  - In this approach, I'd use something like constraint-based type inference, to generate a type definition.
  - For example, both p1 and p2 have x and y properties.
  - So I can generate an anonymous type definition, and then use something like fill-in-the-middle to generate a type name.
  - The challenge with this approach is whether constraint-based type inference can produce precise types, and how to evaluate the quality of generated type names.

# Status report

## Completed

- Test harness
- Baseline experiments
- Initial fine-tuning
- Initial evaluation

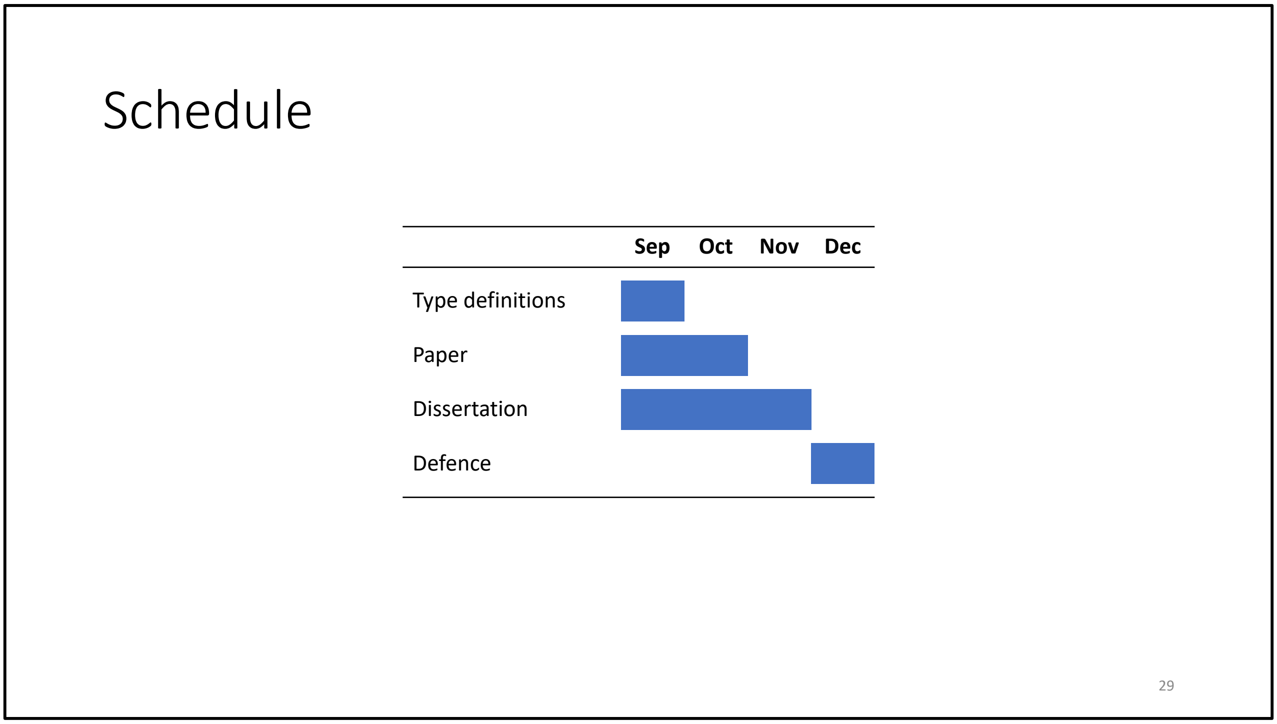
## Next steps

- Analyze results
- Different training formats
- More rigorous evaluation
- Ablation studies



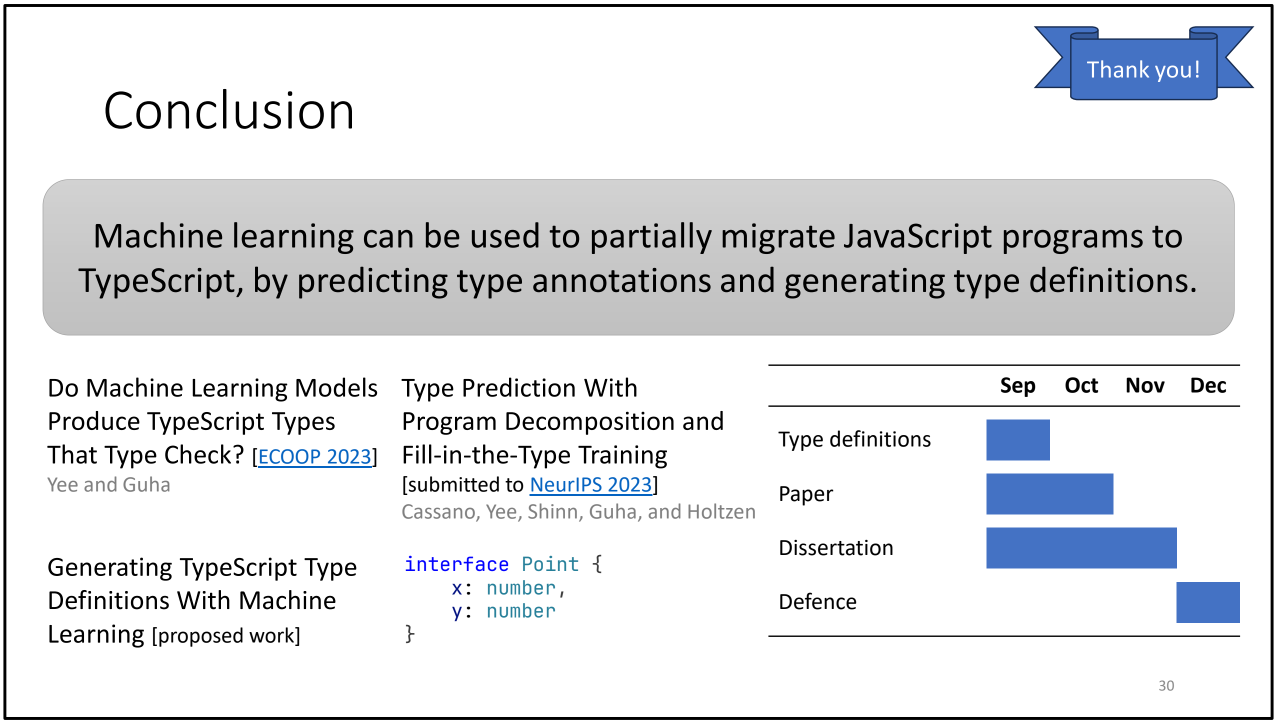
*40:30 to here (1:00 for this slide)*

- Before wrapping up, I'd like to share a status report.
- I have already started the work for generating type definitions.
  - I've set up the test harnesses and experiments, and fine-tuned and evaluated a model.
  - I still want to take a closer look at the results, and examine the types that were generated.
- My next steps are to try different training formats, perform a more rigorous evaluation, and do some ablation studies to better understand what components of the training dataset or format contribute to performance.




*41:30 to here (1:00 for this slide)*

- Here is my proposed schedule.
- I hope to complete the implementation by the end of this month.
- I also plan to start writing the paper and my dissertation.
- Depending on when the paper is completed, I could submit it to PLDI in November, ECOOP in December, or ICML in January.
- Then the goal is to defend in December.




*42:30 to here (1:00 for this slide)*

- To conclude, my proposed thesis is that I can use machine learning to partially migrate JavaScript programs to TypeScript, by predicting type annotations and generating type definitions.
- To support this thesis, I have already completed two papers.
    - The first paper, published at ECOOP, is about evaluating type prediction models.
    - The second paper, submitted to NeurIPS, is about training models for the specific task of type prediction.
- And the last part of my thesis is to generate type definitions, which I'm currently working on.
- You can see my planned schedule on the slide.
- If everything goes well, I expect to be back here in December to defend my thesis.
- Thank you everyone, and I'm happy to take questions now.

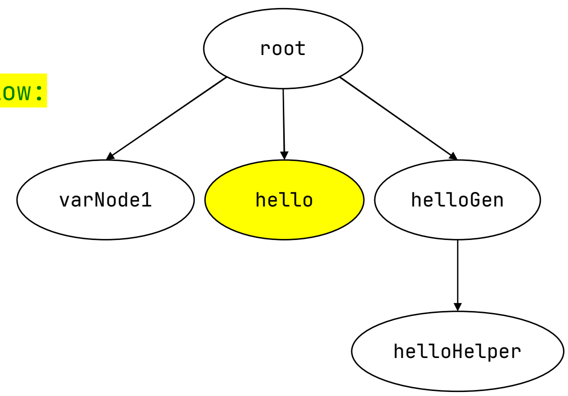
Type Prediction With Program Decomposition and Fill-in-the-Type Training [submitted to NeurIPS 2023]

# Program decomposition – with usages

```
let greeting = "Hello";
let suffix = "!";

/* Example usages of 'hello' are shown below:
    hello(name) + suffix */
// Produces a greeting for the given name
const hello = (name) => {
    return greeting + " " + name;
};

function helloGen(name) {
    const helloHelper = () => {
        return hello(name) + suffix;
    };
    return helloHelper;
}
```




31

- Program decomposition means we lose the global program context.
    - We try to mitigate this with usage comments.
- The “hello” function is used inside “helloGen.”
    - We extract the call to “hello” and include it in a comment for “hello.”
    - This way, the “hello” node has some context for how “hello” is used.