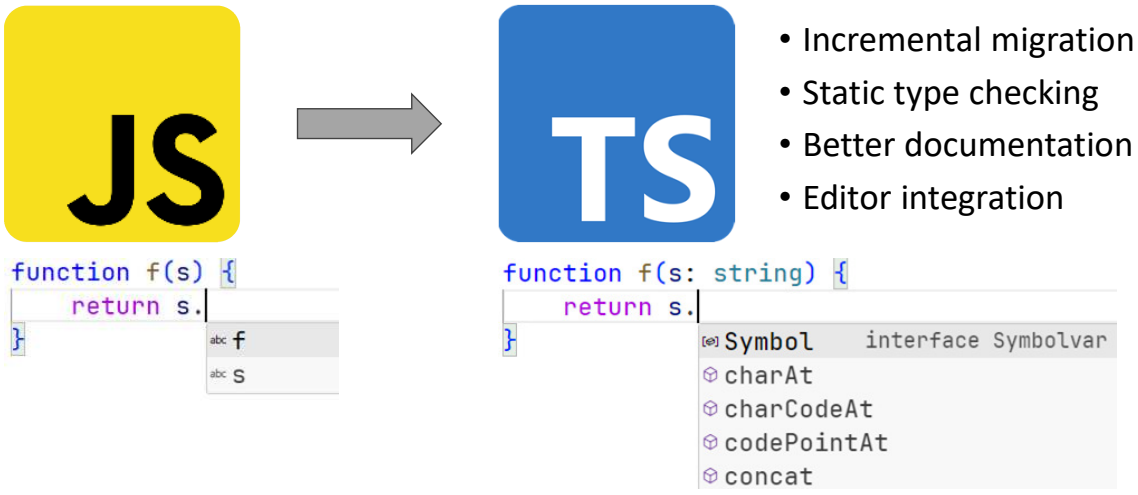
Do Machine Learning Models Produce TypeScript Types That Type Check?

Ming-Ho Yee[1] and Arjun Guha[1,2]
[1]Northeastern University
[2]Roblox

July 20, 2023
ECOOP 2023

- 0:15 for this slide
- Hello everyone, and thanks for coming to my presentation.
- Today, I want to talk about TypeScript, machine learning models that predict TypeScript types, and how we can evaluate these systems.

Type migration: JavaScript to TypeScript



- 1:00 to finish this slide
- So, let's say you have a code base in JavaScript, and you want to migrate it to TypeScript.
  - You can do this by incrementally adding type annotations to your code.
  - As your code becomes more typed, you benefit from static type checking, better documentation, and editor integration.
- For example, here's a small code fragment.
  - The code is untyped, so my text editor can't provide me with useful information.
- But if we look at a typed version of that fragment, we see that s is annotated as a string.
  - As a result, my text editor can show me the methods available on a string.
- 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

DeepTyper [ESEC/FSE 2018] LambdaNet [ICLR 2020] InCoder [ICLR 2023]

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

Type of x	Probability
number	0.4221
any	0.2611
string	0.2558
other	

Type of x	Probability
string	0.9512
number	0.0474
Function	0.0006
other	

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

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

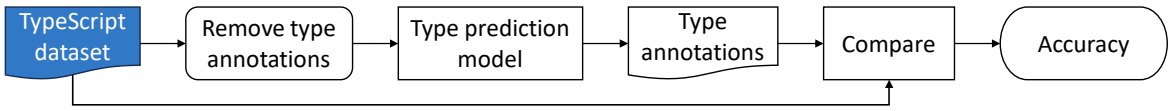
3

- 3:00 to finish this slide
- To automate type migration, there has been research in using machine learning approaches.
  - The idea is to frame the problem as type prediction: “Predict the most likely type annotation for the given code fragment.”
- For our paper, we studied three systems, but any system can be adapted for our framework.
  - DeepTyper (ESEC/FSE 2018) was an early work and uses a recurrent neural network.
  - LambdaNet (ICLR 2020) uses a graph neural network.
  - InCoder (ICLR 2023) is a general-purpose, multi-language transformer
- DeepTyper and LambdaNet are similar:
  - 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 results.
  - I’ve only shown the type predictions for x, which is the only identifier that can be annotated.
- InCoder is a bit different, because it supports a “fill-in-the-middle” task.
  - We can insert a hole into where the type annotation should go, and InCoder will condition on the text before and after the hole as context and predict what goes into the middle.
  - InCoder returns an updated code fragment, with the hole filled in as “number.”
- How do we know these models are doing the right thing? What do we do for evaluation?

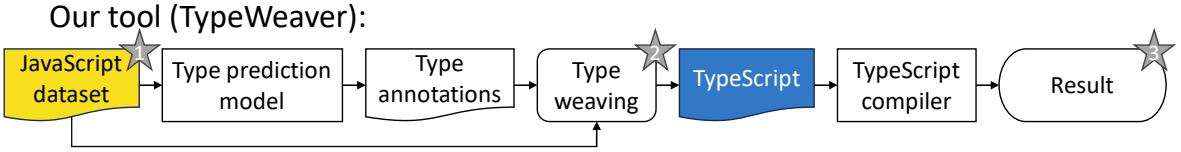
TypeWeaver: type check the type annotations

Prior work:

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



Our tool (TypeWeaver):



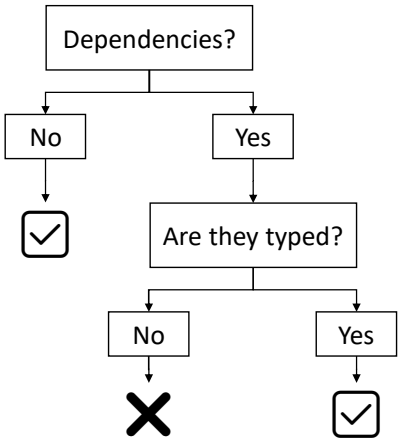
This talk

4

- 5:00 to finish this slide
- In our paper, we propose type checking the type annotations; we created TypeWeaver to do this.
- To recap, let's 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.
    - Accuracy is 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.
- In our paper, we propose and use a different workflow.
  - We start with a JavaScript dataset, and feed that to the type prediction model.
    - We want our evaluation to reflect how type migration would be done in practice, which is to migrate JavaScript to TypeScript.
  - Next, we perform a step called type weaving, which combines the type annotations with the original JavaScript code to produce TypeScript.
  - This allows us to run the type checker on the code, and we get a result.
- For the rest of this talk, I'll be discussing three contributions: our dataset, type weaving, and our results.

Constructing the JavaScript dataset

1. Top 1,000 most downloaded packages
2. Download source code
3. Filter and clean
4. Check dependencies



Result: 513 packages

5

- 7:00 to finish this slide
- Our first contribution is our dataset.
- We start with the top 1,000 most downloaded packages from the npm Registry.
- Next, we download the source code from GitHub.
- Then we apply several filtering and cleaning steps.
    - For example, some packages do not contain any code, or were implemented in some other language, so we filter those out.
- Finally, we check the package dependencies.
    - If there are no dependencies, then we're all set, and we can use the package as-is.
    - If the package has dependencies, we must ensure that we handle those dependencies.
        - Fortunately, we only need the *type declarations* for the dependencies, not the entire source code.
        - There is a community-maintained repository called DefinitelyTyped, where developers contribute type declarations for popular packages.
    - If we can't find type declarations for a dependency, then we discard the package.
    - In other words, we ensure that if a package has dependencies, then all those dependencies are typed.
- This leaves us with a final dataset of 513 packages.

Type weaving: JS + type annotations = TS

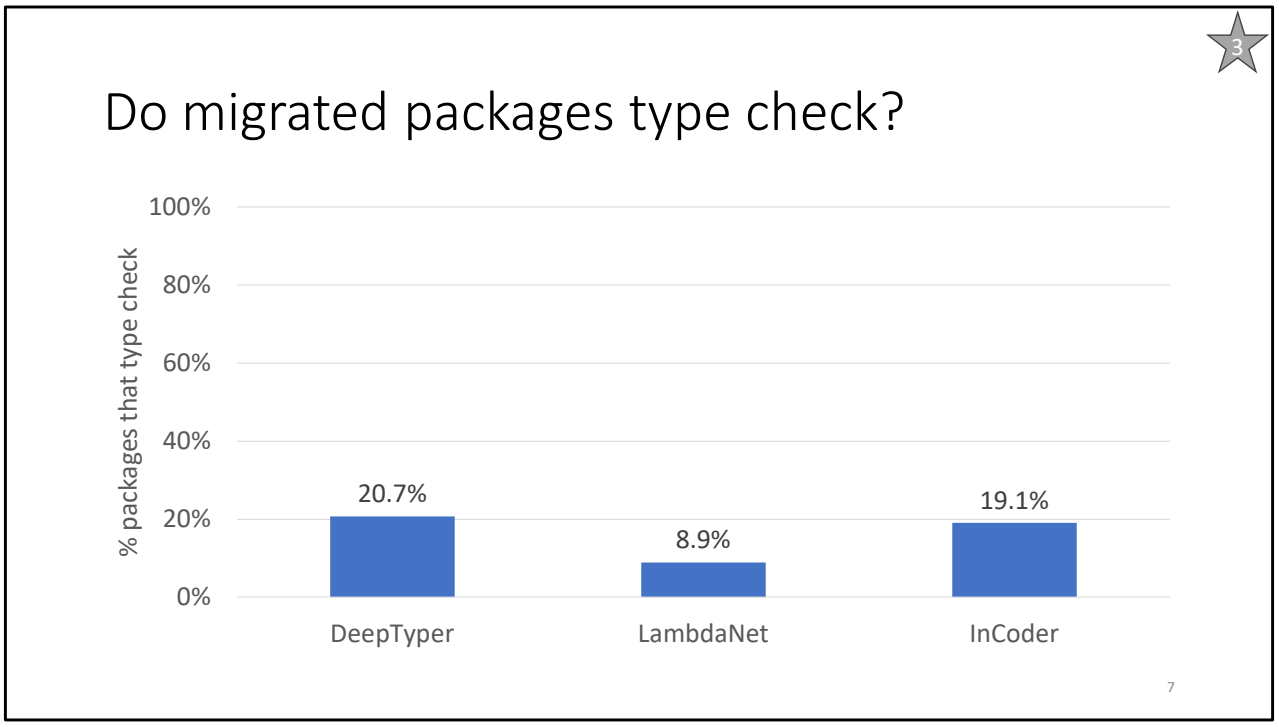
```
function f(x: string, y: number): string {
    return x + y;
}
```

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

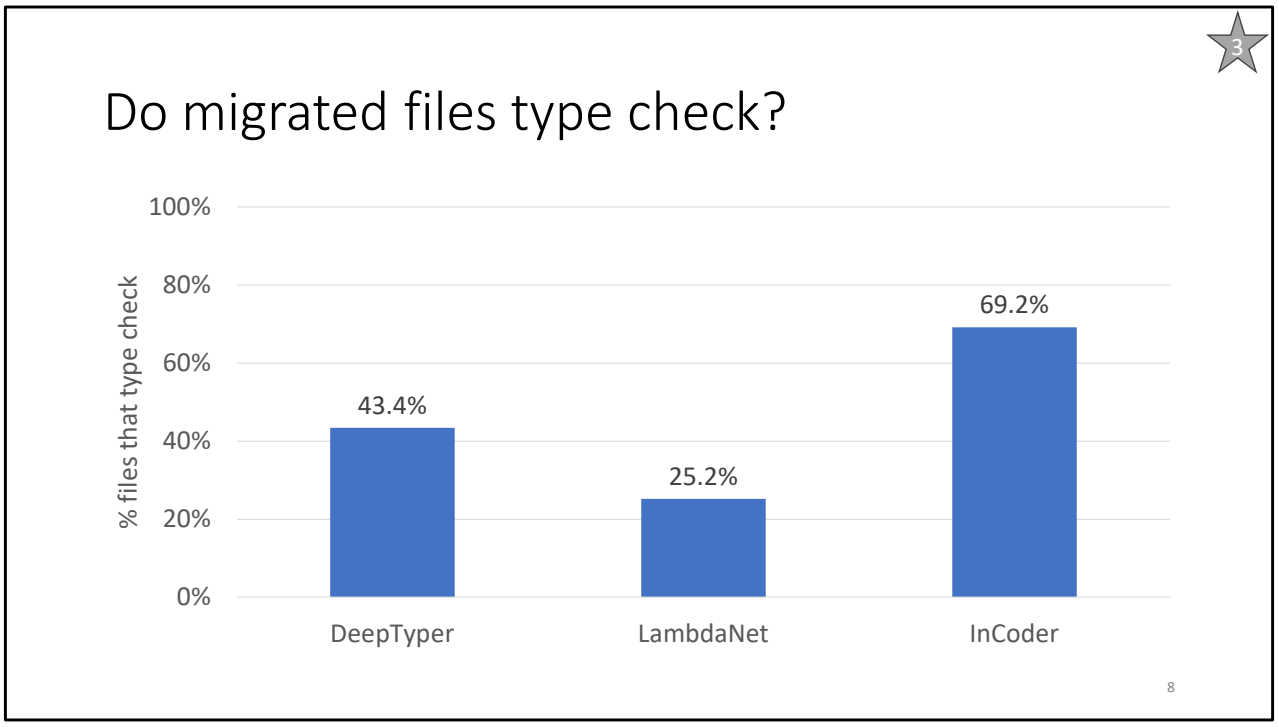
Token	Type	Probability
function		
f	string	0.6381
(		
x	string	0.4543
,		
y	number	0.4706
)		
{		
return		
x	number	0.3861
+		
y	number	0.5039
;		
}		

6

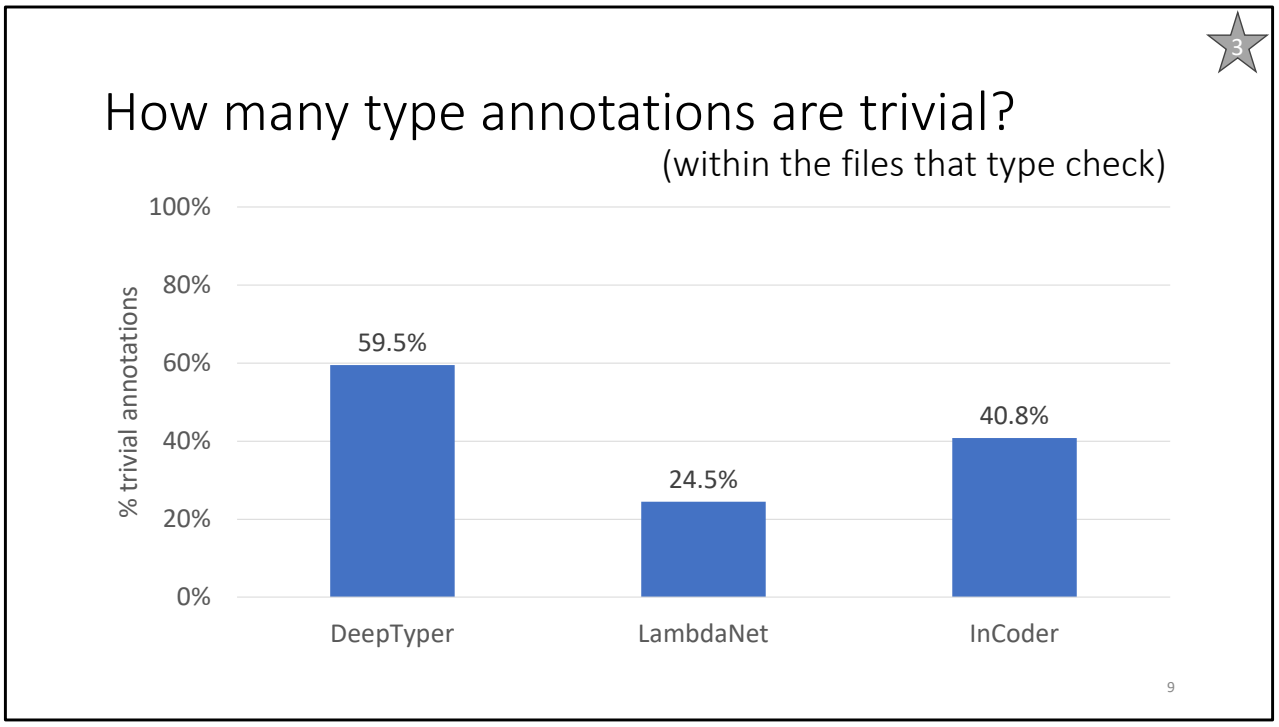
- 9:00 to finish this slide
- Our second contribution is type weaving: this is the process where JavaScript is combined with type annotations to produce TypeScript.
- There are some tricky implementation details, but the overall idea is simple.
    - Let's walk through an example. We start with JavaScript code.
    - Next, we take the type predictions from the model.
        - For this example, I only list the most likely prediction for each token, and I've also cleaned up the table a little.
        - I made up this example and gave it to a system.
    - 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.
    - And this program actually type checks, even though it's not what was intended.
    - The number is coerced to a string and + is string concatenation, so the function returns a string.
- Now we are ready to type check.



- So the first question to ask is: do migrated packages type check?
  - Unfortunately, the results are disappointing.
- DeepTyper and InCoder perform about the same, with a 20% success rate, while LambdaNet is closer to 9%.
- Type checking is a very high standard, because all type annotations need to be correct.
  - Even a single incorrect type annotation will cause a package to fail to type check.
- If you were someone trying to migrate their JavaScript project, this metric isn't very helpful.
  - It simply gives you a pass/fail result for an entire package.
- So let's ask a different, more fine-grained question.



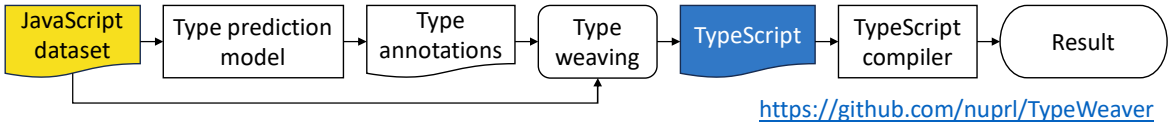
- Do migrated files type check?
- Instead of grading an entire package, we grade each file individually.
  - Our intuition is that each file is a separate module and can be type checked individually.
  - This seems to be a reasonable way to triage errors, when trying to migrate an entire package.
  - However, we still need to be aware that even if a module type checks successfully, we may need to adjust its type annotations if it isn't consistent with other modules.
- This result is much more promising: InCoder has a success rate of 69%.



- 12:00 to finish this slide
- Finally, there's one more chart I'd like to show.
- We need to be careful with our metric, because we can "cheat" by predicting "any" for all type annotations.
    - This result will type check, but the type annotations are useless, and no different from having an untyped program.
- So we counted the proportion of type annotations that are trivial.
    - These are the type annotations for the files that type checked.
    - This is "any" and related type annotations, like array of any, or the generic Function type with unspecified argument types.
- These results are interesting:
    - On the previous slides, DeepTyper performed better than LambdaNet.
    - But now we see that may be because DeepTyper predicted more trivial types.
    - InCoder was somewhere in between but still performed the best.

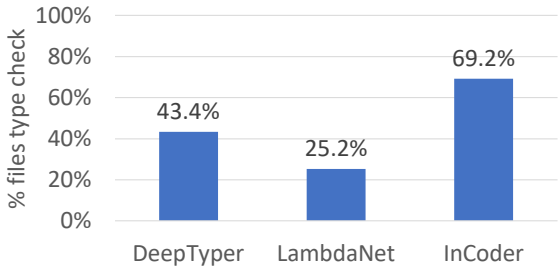
Conclusion: TypeWeaver

Thank you!



https://github.com/nuprl/TypeWeaver

Do machine learning models produce TypeScript types that type check?



Open questions:

- How should we evaluate type prediction models?
- Can slightly wrong annotations be useful?

10

- 13:00 to finish this slide
- So, to summarize our work:
  - We built TypeWeaver, a framework for evaluating type prediction systems.
  - It uses our JavaScript dataset, but it's possible to build your own dataset.
  - We run the dataset through type prediction models, and you can plug in your own model.
  - The model produces type annotations, and we perform type weaving to get TypeScript.
  - And this allows us to type check the resulting code.
- Now, we can finally answer the question: do machine learning models produce TypeScript types that type check?
  - If we're asking about packages, the results are a bit disappointing.
  - But if we look at individual files, the results are more promising.
- Of course, there are more avenues for future work and open questions to answer:
  - How should we evaluate type prediction models? Can we do better than type checking?
  - What if a system produces types that are "slightly wrong"? They would fail to type check, but how can we use those types?
- I'd be happy to chat if you have any thoughts about this.
- Thanks for listening to my talk, and I can take questions now.