# CS388: Natural Language Processing

## Lecture 20: Language and Code


Greg Durrett

TEXAS

The University of Texas at Austin



credit: Deepmind

## Announcements

- Project 3 back
  - Common issues: relatively surface level analysis for part 1, relatively surface level fix for part 2 and little analysis of results, writing clarity issues
- Check-ins due Thursday

## This Lecture

- Semantic parsing
  - Logical forms
  - Parsing to lambda calculus
  - Seq2seq semantic parsing
- Language-to-code
- Applications in software engineering

## Semantic Parsing

## Model Theoretic Semantics

- Key idea: can ground out natural language expressions in set-theoretic expressions called *models* of those sentences
- Natural language statement S => interpretation of S that models it

  *She likes going to that restaurant*

  - Interpretation: defines who *she* and *that restaurant* are, make it able to be concretely evaluated with respect to a *world*
- This is a type of truth-conditional semantics: reduce a sentence to its truth conditions (configuration of the world under which it is true)
- Our modeling language is *first-order logic*
- Entailment (statement A implies statement B) reduces to: in all worlds where A is true, B is true

## First-order Logic

- Powerful logic formalism including things like entities, relations, and quantifications

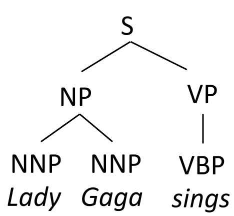
  *Lady Gaga sings*

- sings is a *predicate* (with one argument), function f: entity → true/false
- sings(Lady Gaga) = true or false, have to execute this against some database (*world*)
- Quantification: “forall” operator, “there exists” operator

  $\forall x$ sings(x) $\vee$ dances(x) → performs(x)

  *“Everyone who sings or dances performs”*

## Montague Semantics




| Id | Name | Alias | Birthdate | Sings? |
|---|---|---|---|---|
| e470 | Stefani Germanotta | Lady Gaga | 3/28/1986 | T |
| e728 | Marshall Mathers | Eminem | 10/17/1972 | T |

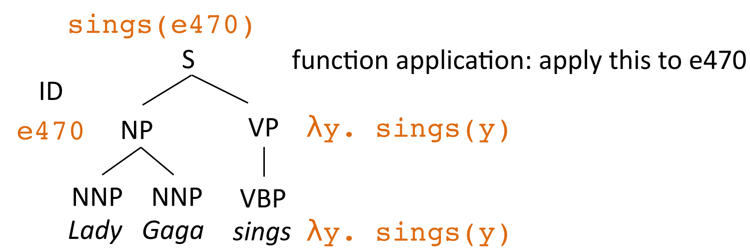
Database containing entities, predicates, etc.

- Richard Montague: operationalized this type of semantics and connected it to syntax
- Denotation: evaluation of some expression against this database

`[[`*Lady Gaga*`]] = e470` denotation of this string is an entity

`[[sings(e470)]] = True` denotation of this expression is T/F

## Montague Semantics

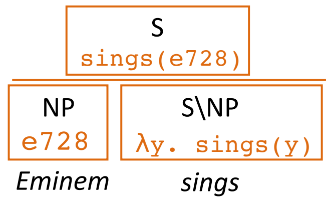



takes one argument (y, the entity) and returns a logical form `sings(y)`

- We can use the syntactic parse as a bridge to the lambda-calculus representation, build up a logical form (our model) *compositionally*

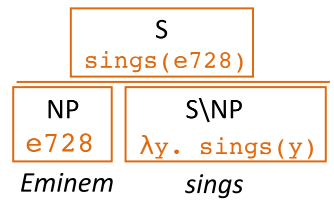
# Combinatory Categorial Grammar

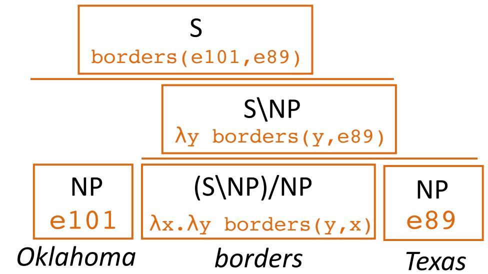
- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Parallel derivations of syntactic parse and lambda calculus expression
- Syntactic categories (for this lecture): S, NP, "slash" categories
- S\NP: "if I combine with an NP on my left side, I form a sentence" — verb
- When you apply this, there has to be a parallel instance of function application on the semantics side




# Combinatory Categorial Grammar

- Steedman+Szabolcsi (1980s): formalism bridging syntax and semantics
- Syntactic categories (for this lecture): S, NP, "slash" categories
  - S\NP: "if I combine with an NP on my left side, I form a sentence" — verb
  - (S\NP)/NP: "I need an NP on my right and then on my left" — verb with a direct object




# CCG Parsing

$$\begin{array}{cccc}
\text{What} & \text{states} & \text{border} & \text{Texas} \\
\hline
(S/(S\backslash NP))/N & N & (S\backslash NP)/NP & NP \\
\lambda f.\lambda g.\lambda x.f(x) \wedge g(x) & \lambda x.state(x) & \lambda x.\lambda y.borders(y,x) & texas \\
 & & \multicolumn{2}{c}{\overline{\qquad\qquad\qquad}>} \\
 & & \multicolumn{2}{c}{(S\backslash NP)} \\
 & & \multicolumn{2}{c}{\lambda y.borders(y, texas)}
\end{array}$$

- "What" is a ***very*** complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (*border Texas*)

Zettlemoyer and Collins (2005)

# CCG Parsing

$$\begin{array}{cccc}
\text{What} & \text{states} & \text{border} & \text{Texas} \\
\hline
(S/(S\backslash NP))/N & N & (S\backslash NP)/NP & NP \\
\lambda f.\lambda g.\lambda x.f(x) \wedge g(x) & \lambda x.state(x) & \lambda x.\lambda y.borders(y,x) & texas \\
\multicolumn{2}{c}{\overline{\qquad\qquad\qquad}>} & \multicolumn{2}{c}{\overline{\qquad\qquad\qquad}>} \\
\multicolumn{2}{c}{S/(S\backslash NP)} & \multicolumn{2}{c}{(S\backslash NP)} \\
\multicolumn{2}{c}{\lambda g.\lambda x.state(x) \wedge g(x)} & \multicolumn{2}{c}{\lambda y.borders(y, texas)} \\
\multicolumn{4}{c}{\overline{\qquad\qquad\qquad\qquad\qquad\qquad}>} \\
\multicolumn{4}{c}{S} \\
\multicolumn{4}{c}{\lambda x.state(x) \wedge borders(x, texas)}
\end{array}$$

- "What" is a ***very*** complex type: needs a noun and needs a S\NP to form a sentence. S\NP is basically a verb phrase (*border Texas*)
- **Why are we talking about this in this lecture? Because this lambda calculus expression is basically executable code.**

Zettlemoyer and Collins (2005)

## CCG Parsing

- These question are *compositional*: we can build bigger ones out of smaller pieces

*What states border Texas?*

*What states border states bordering Texas?*

*What states border states bordering states bordering Texas?*

Zettlemoyer and Collins (2005)

## Training CCG Parsers

- Training data looks like pairs of sentences and logical forms

| | |
|---|---|
| *What states border Texas* | $\lambda$x. `state(x)` $\wedge$ `borders(x, e89)` |
| *What borders Texas* | $\lambda$x. `borders(x, e89)` |

...

- Unlike PCFGs, we don't know which words yielded which fragments of CCG
- Very hard to build a conventional parser for this problem

Zettlemoyer and Collins (2005)

## Semantic Parsing as Translation

*"what states border Texas"*

↓

```
lambda x ( state ( x ) and border ( x , e89 ) ) )
```

- Write down a linearized form of the semantic parse, train seq2seq models to directly translate into this representation (similar to code generation like GitHub Copilot)
- What are some benefits of this approach compared to grammar-based?
- What might be some concerns about this approach? How do we mitigate them?

Jia and Liang (2016)

## Applications

- GeoQuery (Zelle and Mooney, 1996): answering questions about states (~80% accuracy)
- Jobs: answering questions about job postings (~80% accuracy)
- ATIS: flight search
- Can do well on all of these tasks if you handcraft systems and use plenty of training data: these domains aren't that complex and models these days can produce well-formed outputs

## Code Generation

- Suppose we are going to generate source code like in Codex/GitHub Copilot. What differs from generating natural language?
- In spite of these differences, the "obvious" thing is to do some pre-training and see how far we get!

## Generating Code

## CodeT5

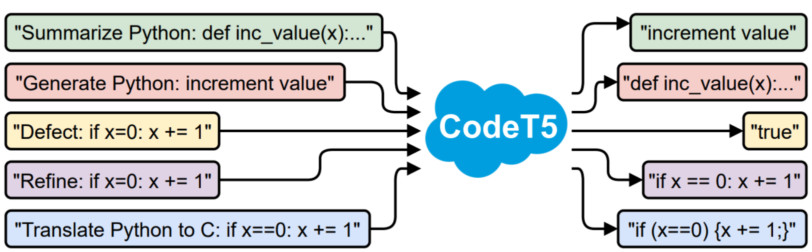
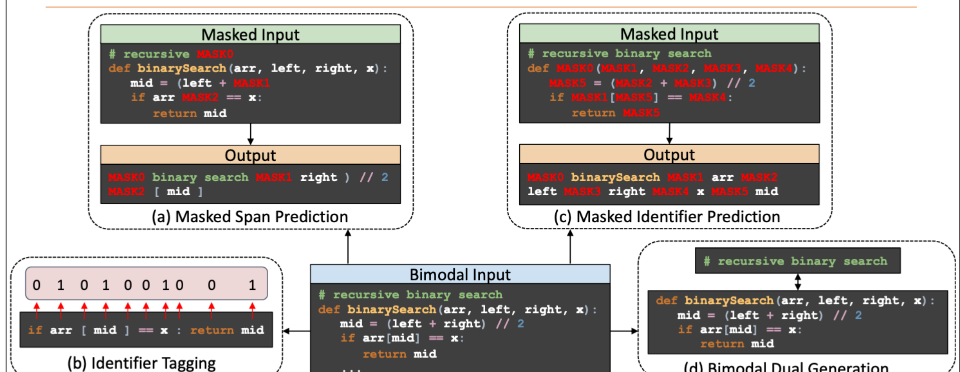



Figure 1: Illustration of our CodeT5 for code-related understanding and generation tasks.

- Key idea: code analogue of T5 that should be able to map language to source code

Wang et al. (2021)

## CodeT5




- Predict (a) spans; (c) identifiers; (d) language from code and vice versa
- What's different from normal T5?

Wang et al. (2021)

# CodeT5

- Pre-trained on data from several language and NL
- Applied to several generation tasks: code summarization, generation, and translation (between programming languages)

| | PLs | W/ NL | W/o NL | Identifier |
|---|---|---|---|---|
| CodeSearchNet | Ruby | 49,009 | 110,551 | 32.08% |
| | JavaScript | 125,166 | 1,717,933 | 19.82% |
| | Go | 319,132 | 379,103 | 19.32% |
| | Python | 453,772 | 657,030 | 30.02% |
| | Java | 457,381 | 1,070,271 | 25.76% |
| | PHP | 525,357 | 398,058 | 23.44% |
| Our | C | 1M | - | 24.94% |
| | CSharp | 228,496 | 856,375 | 27.85% |
| | Total | 3,158,313 | 5,189,321 | 8,347,634 |

- Also used for classification like bug detection (can be fine-tuned like BERT-style models)

Wang et al. (2021)

# CodeT5

- Generation task from CONCODE (Iyer et al., 2018):

```
public class SimpleVector implements Serializable {
  double[] vecElements;
  double[] weights;

  NL Query: Adds a scalar to this vector in place.
  Code to be generated automatically:
  public void add(final double arg0) {
    for (int i = 0; i < vecElements.length; i++){
      vecElements[i] += arg0;
    }
  }
```

- What do you think about this evaluation?

| Methods | EM | BLEU | CodeBLEU |
|---|---|---|---|
| GPT-2 | 17.35 | 25.37 | 29.69 |
| CodeGPT-2 | 18.25 | 28.69 | 32.71 |
| CodeGPT-adapted | 20.10 | 32.79 | 35.98 |
| PLBART | 18.75 | 36.69 | 38.52 |
| CodeT5-small | 21.55 | 38.13 | 41.39 |
| +dual-gen | 19.95 | 39.02 | 42.21 |
| +multi-task | 20.15 | 35.89 | 38.83 |
| CodeT5-base | 22.30 | 40.73 | 43.20 |
| +dual-gen | **22.70** | **41.48** | **44.10** |
| +multi-task | 21.15 | 37.54 | 40.01 |

Table 3: Results on the code generation task. EM denotes the exact match.

Wang et al. (2021)

# Codex

- GPT-3 additionally fine-tuned on code (although they state that pre-training on NL isn't really helpful)
  - Modified tokenizer to handle whitespace better. Otherwise, no real modifications!
- Up to 12B parameter models fine-tuned on Python
- One challenge is evaluation. How to go beyond BLEU/EM?

Mark Chen et al. (2021)

# HumanEval

- Generate standalone Python functions from docstrings **and execute them!**

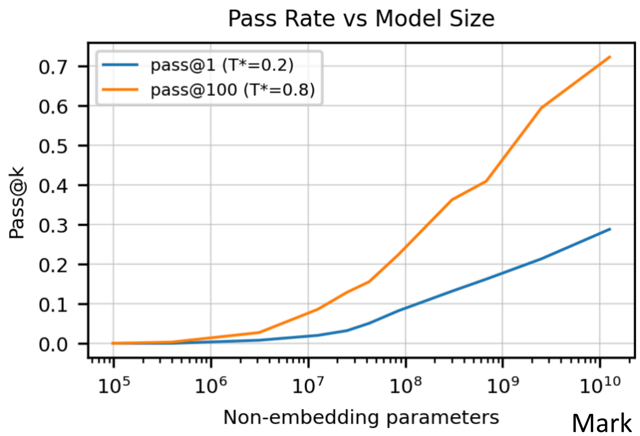
```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

- Handwritten benchmarks evaluated for correctness ("pass@k": generate k, see if one of them works)

Mark Chen et al. (2021)

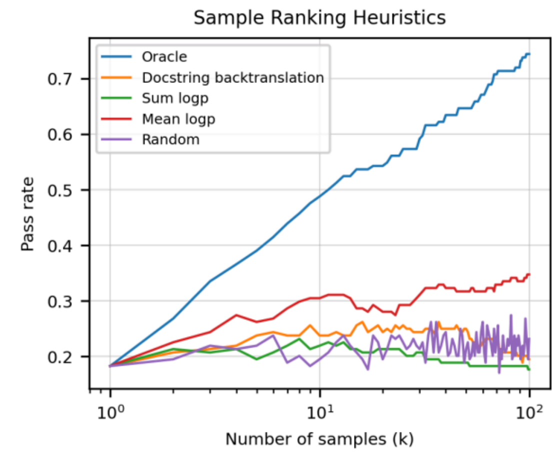
# HumanEval



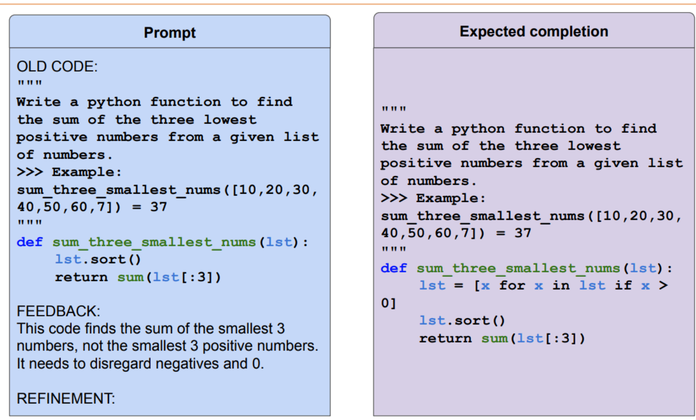

Mark Chen et al. (2021)

# HumanEval

- Another setting: can we generate a bunch of samples and then pick the correct one? This would be useful for rejection sampling
- Other experiments: additional fine-tuning on competitive programming problems, docstring generation



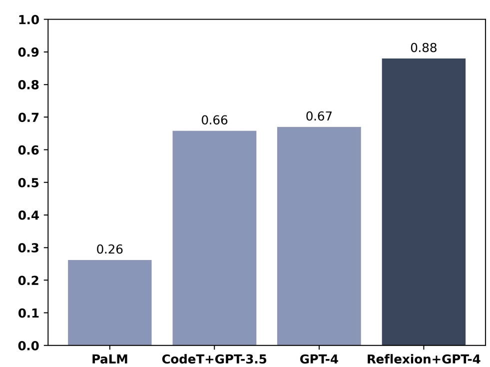

# NL Feedback




Improving Code Generation by Training with Natural Language Feedback Angelica Chen et al. (2023)

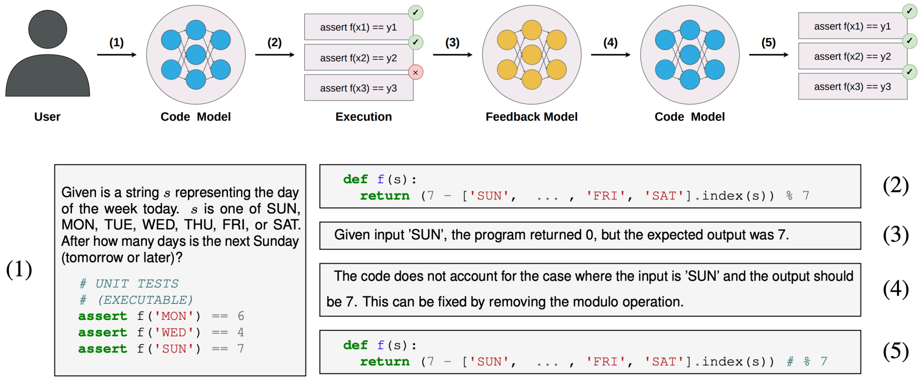
# Reflexion

- Interact with environment, generate a "reflection" about that interaction, then condition on that interaction for the next round
- Very little details about this, but very strong results on HumanEval!




https://twitter.com/johnjnay/status/1639362071807549446 Shinn et al. (2023)

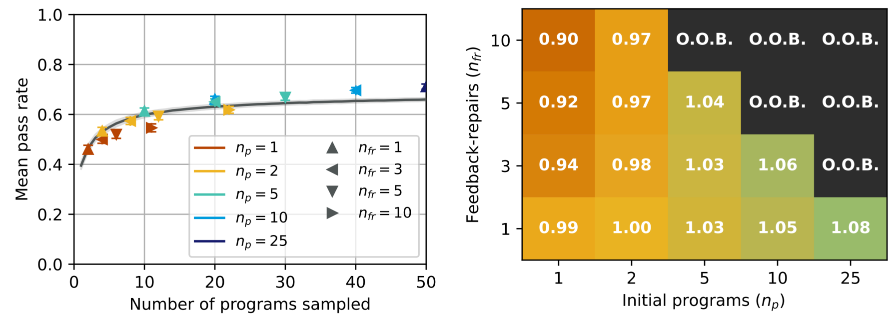
# Self-Repair




Is Self-Repair a Silver Bullet for Code Generation?

Theo Olausson et al. (2023)

# Self-Repair




- GPT-4 results on “APPS” dataset, frequently math-y programming puzzles
- Getting many initial programs and trying to repair each one once is the best strategy

Theo Olausson et al. (2023)

# State of LLM Program Generation

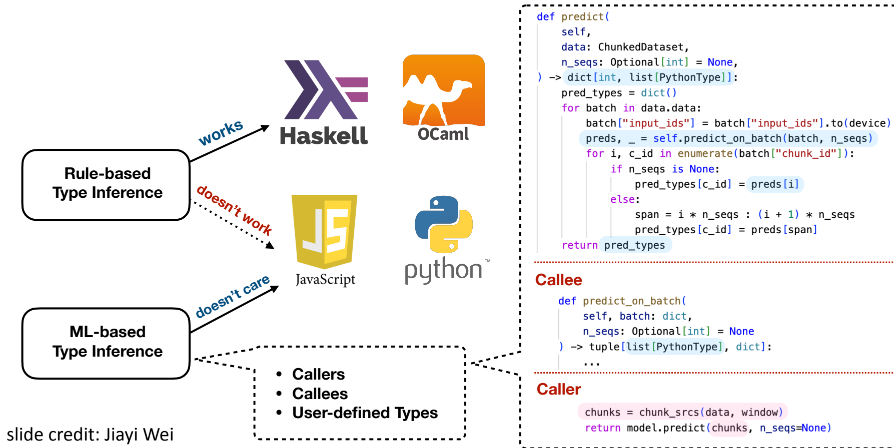
- Pre-training big models:
  - CodeLlama (with Python and Instruct variants)
  - OctoCoder (trained on GitHub commit data)
  - Many other efforts and likely more to come
- Loops to improve program generation:
  - Debugging from failed tests, compiler errors, etc.
  - Fine-tuned models to do these

# Applications in Software Development

# Applications

- Generating complete code is nice, but is very challenging: can't read the user's mind, if generated code has errors they may be time-consuming to spot
- There are a range of applications in software engineering: bug detection, type inference, etc. — solving these subproblems can still help save developers time
- One such problem: type inference

# Type Inference




slide credit: Jiayi Wei

# Type Inference

- Typing this code snippet:

```
chunks = chunk_srcs(data, window)
return model.predict(chunks, n_seqs=None)
```

...requires looking at this function:

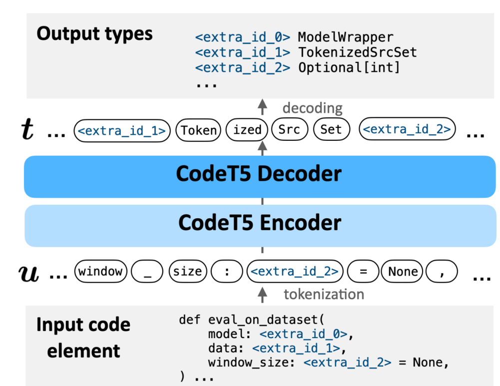
```
def predict(
    self,
    data: ChunkedDataset,
    n_seqs: Optional[int] = None,
) -> dict[int, list[PythonType]]:
    pred_types = dict()
    for batch in data.data:
        batch["input_ids"] = batch["input_ids"].to(device)
        preds, _ = self.predict_on_batch(batch, n_seqs)
        for i, c_id in enumerate(batch["chunk_id"]):
            if n_seqs is None:
                pred_types[c_id] = preds[i]
            else:
                span = i * n_seqs : (i + 1) * n_seqs
                pred_types[c_id] = preds[span]
    return pred_types
```

- Changes are non-local: even with GPT-4-length contexts, you usually can't have a whole project in Transformer context

slide credit: Jiayi Wei

# Type Inference

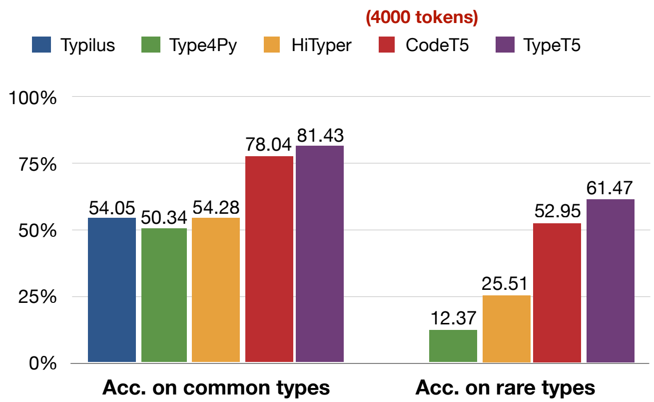
- Can use CodeT5 to predict the types...but what context do we feed it?
- Solution: use **static analysis** to determine relevant parts of the program
- Use the call graph to assemble a context for CodeT5 consisting of callers, callees, and skeletons of various files

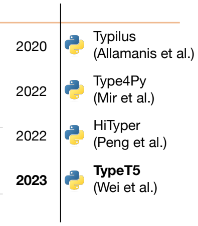



slide credit: Jiayi Wei

Jiayi Wei, Durrett, Dillig (ICLR 2023)

# Type Inference



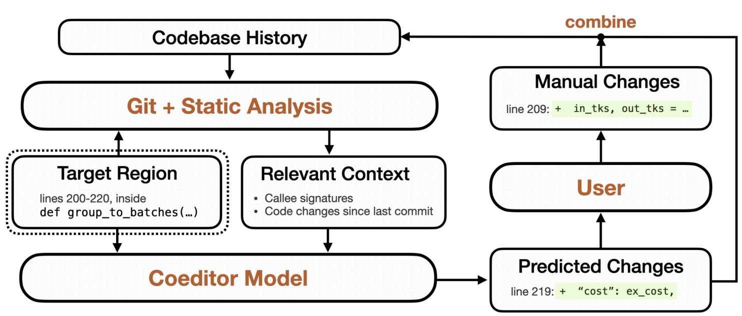

slide credit: Jiayi Wei

Jiayi Wei, Durrett, Dillig (ICLR 2023)

# Other Applications

- Bug detection: spot bugs in code
- Test generation
- Comments: code-to-comment translation, updating comments when code has changed, and more (see papers by Sheena Panthaplackel)
- Debugging: ask GPT-4 to fix code given an error message (see Greg Brockman's GPT-4 demo)
- Program synthesis: have some specification other than language (e.g., input-output examples, formal spec) and produce code to follow that

# Beyond Copilot




- Can autocomplete a user's refactoring change by using knowledge of what they've changed so far. Copilot doesn't support this

Jiayi Wei, Durrett, Dillig (2024)

# Takeaways

- Language was being interpreted into logical forms that looked like code for a long time (including in formal semantics)
- Rather than doing this with parsers, now we just use seq2seq models
  - Powerful enough models will almost always generate code that compiles. You don't need special constraints on the output.
- …and because of pre-training, rather than using customized DSLs, we just use source code because models have seen more of it