OpenTau

Using OpenAI Codex for Gradual Type Inference

Federico Cassano and Noah Shinn

Manually type-annotating code is a mundane and time-consuming process.

Type inference systems automate that process by inferring types based on the program’s context of execution.

However, current type inference systems for **gradually-typed** languages often annotate with lossy types, such as the *any* type.

TypeScript Built-in Inference Example

Too many *any* types. These types are too permissive, which makes type inference less effective than expected.

```
const kClosest: (points: any, K: any) => any = (points, K) => {
    let len: any = points.length, l: number = 0, r: number = len - 1;
    while (l <= r) {
        let mid: any = helper(points, l, r);
        if (mid === K)
            break;
        if (mid < K) {
            l = mid + 1;
        }
        else {
            r = mid - 1;
        }
    }
    return points.slice(0, K);
};
```

How can we solve this problem using AI?

We can use natural language models for code synthesis, such as OpenAI Codex and Facebook InCoder.

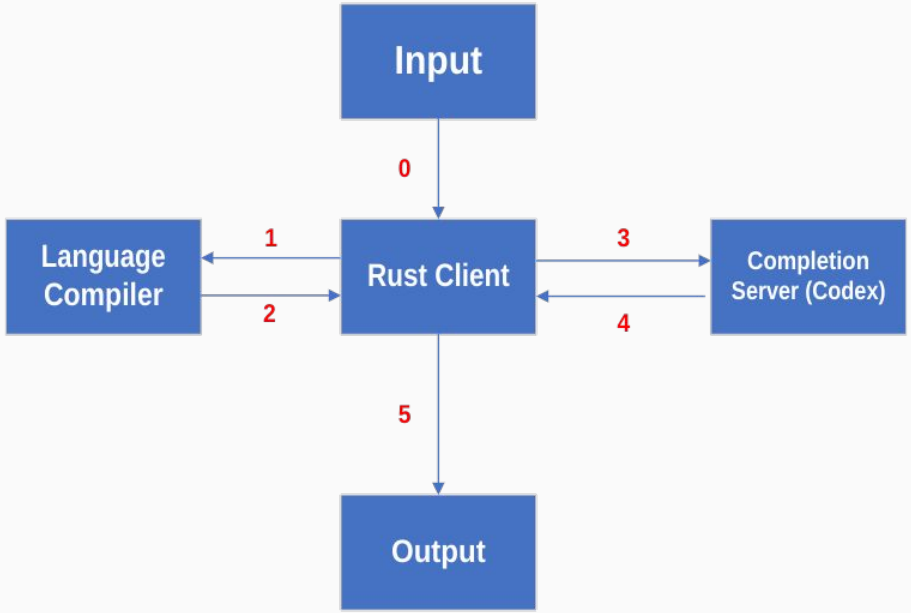
We fill in type holes and we ask the model to edit the type holes with real types.

A heuristic for descriptive types

Type	Score	Correct
missing (example: `let x = 1`)	N/A	False, terminate
literal type (example: `let x: 3 = 1`)	N/A	False, terminate
`_hole_`	N/A	False, terminate
`unknown`	+10	True, continue
`any`	+5	True, continue
`undefined`	+2	True, continue
`Function` (interface type)	+5	True, continue

A Protocol for Generalizing NLM-based Type-Inference Across Different Languages

We used TypeScript as our main example. However, we created a protocol, similar to LSP, that is able to generalize the usage of language models for type-inference. We implemented a client in Rust that interfaces with the TypeScript compiler, and is able to interface with other compilers, such as Python's. The compiler is responsible for AST modifications and type quality quantification.



Scaling The Completion Algo

Querying Codex with whole files does not scale as there is a max token limit. Instead, we would like to type-annotate code snippets, and transplant the type annotations into the full file.

We derived a parallelized bottom-up tree traversal algorithm that completes code blocks starting from the innermost scope. We transplant the type annotations from the inner scope into the outer scope, then we stub any inner functions in the outer scope. We recur until we reach top-level, where we reassemble. We also provide usages of functions in the prompt. The algorithm completes each level in parallel for faster completions.

```
function hello(name) {
  function inner() {
    const msg = "Hello, " + name + "!";
    return msg;
  }
  console.log(inner());
  return inner();
}
```

```
function inner(): string {
  const msg: string = "Hello, " + name + "!";
  return msg;
}
// Usages of inner below:
console.log(inner());
return inner();
```

```
function hello(name) {
  function inner(): string;
  console.log(inner());
  return inner();
}
```

```
function hello(name: string): string {
  function inner(): string;
  console.log(inner());
  return inner();
}
```

```
function hello(name: string): string {
  function inner(): string {
    const msg: string = "Hello, " + name + "!";
    return msg;
  }
  console.log(inner());
  return inner();
}
```

59%

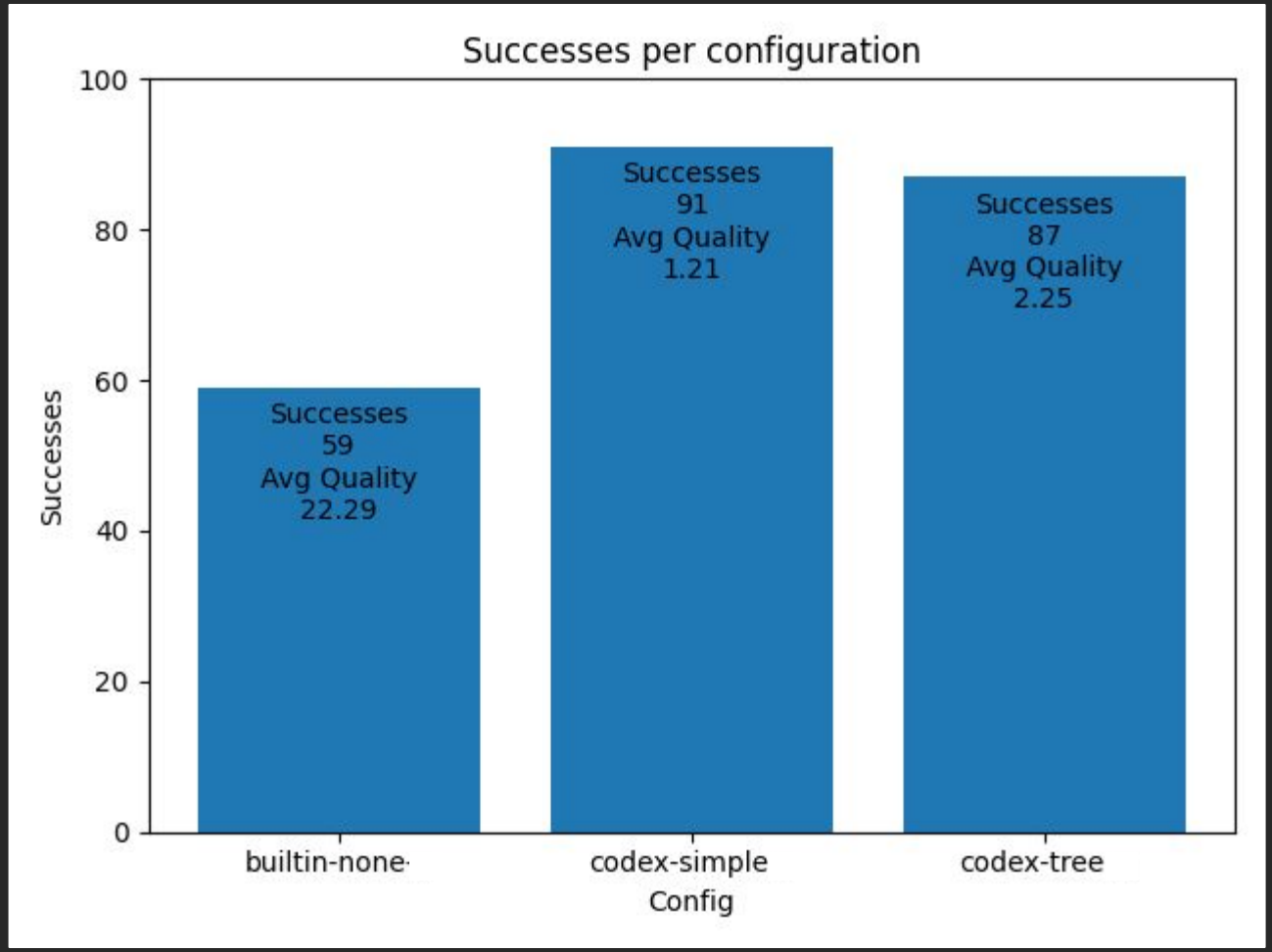
TypeScript built-in type inference

Mostly over-permissive types

91%

OpenTau type inference

Most descriptive types



Hyperparameters used for Codex:
# of completions: 10
temperature: 0.8
retries : 0

NOTE:
Higher “Avg Quality” is worse.

Built-in

```
const kClosest: (
  points: any,
  K: any
) => any = (points, K) => {
  let len: any = points.length,
    l: number = 0,
    r: number = len - 1;
  while (l <= r) {
    let mid: any = helper(points, l, r);
    if (mid === K) break;
    if (mid < K) {
      l = mid + 1;
    } else {
      r = mid - 1;
    }
  }
  return points.slice(0, K);
};
```

VS

Codex

```
const kClosest: (
  points: [number, number][],
  K: number
) => [number, number][] = (points, K) => {
  let len: number = points.length,
    l: number = 0,
    r: number = len - 1;
  while (l <= r) {
    let mid: number = helper(points, l, r);
    if (mid === K) break;
    if (mid < K) {
      l = mid + 1;
    } else {
      r = mid - 1;
    }
  }
  return points.slice(0, K);
};
```

Future Directions

1. Use Facebook’s InCoder model for faster execution time and larger sets of completions.
2. Python type-inference implementation
3. Compare metrics for TypeScript vs Python and Codex vs InCoder

Thanks!

Link to project:
https://github.com/GammaTauAl/opentau

Link to project evaluation:

https://github.com/GammaTauAl/opentau-test

FAQ: Why called OpenTau?
The “tau” greek letter is used to represent an unknown type in type theory. “Open” because it’s open source.