Using OpenAI Codex for Gradual Type Inference

Milestone II Report

Federico Cassano, Noah Shinn

Report

Our high-level approach to type-inference via Codex is the following:

- 1. We create a tree of nested code-blocks T in our JavaScript program. Where the top-level is the root of the tree and each function, class or method has a node below the parent block. The node of the tree is identified by id-α, the α-renamed name of the function/class/method. The root node id is defined as id-$\alpha = root$.
- 2. For each id-α, we find usages of the identifier and we save them in the node for prompt-engineering.
- 3. We start traversing the tree using a bottom-up traversal, starting from the leaves. To each node we apply the steps described below:
  - 1. If this is not a leaf node, we extract the completions for all the direct children of this node and we type-weave the types of the children into this node. This process creates permutations for all completions of all direct children of this node. For each permutation, we create a prompt $\mathcal{P}_i$. We use the type-weaving procedure:
    $\mathcal{W}$: Original File, Nettle File $\rightarrow$ Resulting File.
  - 2. For each prompt, we insert the saved usages of id-α at the bottom of the prompt, to help Codex infer the types.
  - 1. For each prompt, we insert the identifier `_hole_` in place of missing types in our prompts. To do this, we use a compiler $\mathcal{K}$: File $\rightarrow \mathcal{P}$.
  - 2. We define an instruction $\mathcal{I}$, which is the constant string:
    $\mathcal{I}$ = "Substitute the identifier _hole_ with the correct type."
  - 3. For each prompt computed for the node, query the `davinci-edit` or `incoder` model using the prompt $\mathcal{P}_i$ and instruction $\mathcal{I}$. We receive back a set of completions $\mathcal{C}_{\text{id-}\alpha}$, $0 \leq |\mathcal{C}_{\text{id-}\alpha}| \leq n$, where n is a pre-defined maximum number of completions.
  - 4. We use a cheap and admissible heuristic $h : c \rightarrow$ (Boolean, $\mathcal{N}$) that determines if a given completion c is *correct* (a *correct* completion however, may still not type-check) and the quality of the type annotations q, where the lower the q the better.
  - 5. We apply h to all elements in $\mathcal{C}_{\text{id-}\alpha}$ and add the completions that produced True to an ordered set $\mathcal{Q}_{\text{id-}\alpha}$ sorted by q.
- 4. Using the command: `tsc --allowJs --checkJs --noEmit --target es2022 <file.ts>` we run a full type-check on every completion in $\mathcal{Q}_{root}$, terminating as soon as the command returns code `0`, meaning that the completion is type-correct. By terminating early on the sorted set, we guarantee that our solution is optimal with respect to $\mathcal{Q}_{\text{root}}$. We let c^* be the optimal type-correct solution.
- 5. We produce c^* if it exists, otherwise we produce an error.

Implementation of the Pipeline

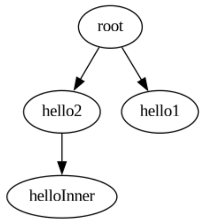
Initial Tree Generation We have implemented a procedure that generates a code-block tree from a given program.

For instance, given this input:

```
function hello2(name) {
  const helloInner = (name) => {
    return "Hello " + name;
  };
  return helloInner(name);
}

const hello1 = (name) => {
  return "Hello " + name;
};
```

The procedure is going to output:



Where each node in the tree is annotated using the definition:

```
pub struct CompNode {
  pub children_idxs: Vec<usize>, // pointers to the children
  pub name: String, // the alpha-renamed id
  pub code: String, // the code of this node
  pub completed: Vec<String>, // the completions of this node (starts empty)
  pub usages: String, // the usages of the alpha-renamed id of this node
}
```

Type-Weaving Procedure We have implemented a type-weaving procedure $\mathcal{W}$. This procedure is used to transplant the type-annotations from one program to another without being affected by structural differences in either programs. The procedure uses scope-based α-renaming in order to extract the types from matching identifiers in different scopes. This gives us fine-grained control for selecting which scopes should have types extracted and which scopes should not.

For instance, the given program:

```
function hello(name: string): string {
  function inner(): string {
    let my_string: string = "Hello " + name;
    return my_string;
  }
  return inner();
}
```

Has the following α-renamed type-table:

α-renamed id	Type
`hello`	`(name: string) => string`
`hello$inner`	`() => string`
`hello$inner$my_string`	`string`

Using this program as our *nettle* (the reference program that donates types to be transplanted), we can transplant types into the following program, starting from scope `hello`:

```
function hello(name) {
  function inner() {
    let my_string = "Hello " + name;
    return my_string;
  }
  return inner();
}
```

We will receive back the program:

```
function hello(name) {
  function inner(): string {
    let my_string: string = "Hello " + name;
    return my_string;
  }
  return inner();
}
```

This allows us to ignore any types given to the `hello` function, while transplanting types given to the `hello$inner` function and the `hello$inner$my_string` variable.

Compiler We have implemented $\mathcal{K}$ by interfacing with the TypeScript compiler API for inserting type-holes into identifiers that lack type annotations.

For instance, given this input:

```
function hello(name: string) {
  let msg = "Hello";
  return msg + ", " + name + "!";
}
```

$\mathcal{K}$ will output the following:

```
function hello(name: string): _hole_ {
  let msg: _hole_ = "Hello";
  return msg + ", " + name + "!";
}
```

Heuristic We have implemented our heuristic h. The heuristic traverses the program's abstract syntax tree identifying different types, which will be scored. Some types terminate the heuristic early and denote that the program cannot possibly be correct. The scores are summed to compose q using the following table:

Type	Score	Correct
missing (example: `let x = 1`)	+0	False, terminate
literal type (example: `let x: 3 = 1`)	+0	False, terminate
`_hole_`	+0	False, terminate
`unknown`	+10	True, continue
`any`	+5	True, continue
`undefined`	+2	True, continue
`Function` (interface type)	+5	True, continue

For example, with the completion:

```
function hello(name: string): any {
  let msg: undefined = "Hello";
  return msg + ", " + name + "!";
}
```

h will output $(\text{True}, 7)$, as the presence of one `any` and `undefined` gives a summed score of $5 + 2 = 7$

While, with the completion:

```
function hello(name: string): _hole_ {
  let msg: string = "Hello";
  return msg + ", " + name + "!";
}
```

h will terminate early and output $(\text{False}, 0)$, as the presence of one `_hole_` type terminates the heuristic early.

Additionally, h checks if the model didn't add anything other than just types (such as additonal code blocks and comments) to the original prompt. If that condition isn't met, $(\text{False}, 0)$ will be produced.

Client Finally, we have implemented a client in Rust that manages the pipeline and queries the Codex API. The client will communicate with the compiler K, which is written in TypeScript and will send the outputs to the model. The client can be downloaded from https://github.com/GammaTauAI/opentau and can be utilized by using the following terminal interface:

```
USAGE:
    client [OPTIONS] --tokens <TOKENS> --file <FILE> --output <OUTPUT>

OPTIONS:
    -c, --cache <CACHE>
            The Redis URL for the cache
        --disable-rate-limit
            Whether or not to prevent rate limits. You may want to set this to false if You are
            using your own model. By default, we try to prevent rate limits, by using this flag you
            can disable this behavior
    -e, --endpoint <ENDPOINT>
            The url of the codex endpoint [default: https://api.openai.com/v1/edits]
    -f, --file <FILE>
            The target file path
        --fallback
            Whether to fallback to "any" or not
    -h, --help
            Print help information
    -l, --lang <LANG>
            The target language. Either `ts` or `py` [default: ts]
    -m, --max-type-quality <MAX_TYPE_QUALITY>
            The maximum type-quality score for a completion to be valid (lower means better quality)
            [default: 9999999]
    -n, --n <N>
            The number of completions to return [default: 3]
    -o, --output <OUTPUT>
            Output file directory path
    -r, --retries <RETRIES>
            The number of request to send to Codex [default: 1]
    -s, --strategy <STRATEGY>
            Completion strategy. Either: {"simple": simple completion, "tree": tree completion}
            [default: simple]
        --stop-at <STOP_AT>
            The maximum number of type-checkable completions to return [default: 1]
    -t, --tokens <TOKENS>
            Codex tokens to use, separated by commas
        --temp <TEMP>
            The temperature to use for the completion [default: 1]
    -V, --version
            Print version information
```

Type-correct solutions will be written to the specified directory.

The appendix at the end of the paper provides a set of prompts and completions that our client produced.

Building our own Codex

The InCoder model query system was implemented as a simple HTTP server that evaluates the InCoder model M and receives an untyped input and several hyperparameters from the language server and returns a list of `type infills`. Then, the `type infills` are inserted into the original untyped code to yield the final completion. The pipeline is shown below:

Given the following prompt $\mathcal{P}$: we have the untyped `input`

```
function hello(name: _hole_): _hole_ {
  let msg: string = "Hello";
  return msg + ", " + name + "!";
}
```

and the hyperparameters `max_to_generate=5`, `temperature=0.8`, and `max_retries=1` in a POST request, the server will split the `input` by the fake type, `_hole_` to yield the following list:

```
parts_list = [
  "function hello(name: ",
  "): ",
  ' { let msg: string = "Hello"; return msg + ", " + name + "!";}',
];
```

The `parts_list` and the given hyperparameters will form prompt $\mathcal{P}$' and will be given to M for the `infill` task. $M(\mathcal{P}')$ will yield a list of inferred types, such as

```
type_list = ["string", "string"];
```

Then, the `type_list` will be inserted into the original `parts_list` to build the final completion:

```
function hello(name: string): string {
  let msg: string = "Hello";
  return msg + ", " + name + "!";
}
```

The describes process will execute N number of times where N is the number of requested completions. Finally, the InCoder server will return the following JSON response:

```
{
  "choices": [
    "text": "<completion0>",
    "text": "<completion1>",
    ...
    "text": "<completionN>"
  ]
}
```

where `choices` is of length N.

This API mimics the Codex API in order to be an effective drop-in replacement, all we had to do in our client is changing the url.

In production, we execute the InCoder model on NVIDIA A100 GPUs using Northeastern's Discovery Cluster.

Reflect

Initial Progress Goals *"By now, we will have implemented type-inference for Python and a prompt engineering approach that will allow us to type-infer larger files."*

Current Achievements While we were initially planning to implement Python type-inference for milestone 2, we decided to focus on the InCoder completion implementation as it would allow us to have access to an infinite number of calls to the model without reaching certain rate limitations. However, after we briefly tested the InCoder server, we observed a trade-off between the effectiveness of the Codex model and the speed of the InCoder model. So far, the Codex model has shown a strong performance in the type-inference task, but the model can only be queried up to 20x per minute per API key. On the other hand, the InCoder model can be evaluated several times in less than a second without reaching a rate limitation, but the respective completions have proven to be significantly less accurate than the Codex model. In particular, the InCoder model struggles with the case of inferring a single type where appropriate while being given a max token length hyperparameter greater than 1. For example:

Given the following untyped input,

```
function hello(name: _hole_): _hole_ {
  let msg: string = "Hello";
  return msg + ", " + name + "!";
}
```

the correct types should be `string` and `string`. However, given a max token length hyperparameter of 5, the model may return the following typed completion,

```
function hello(name: stringstring): stringstringstring {
  let msg: string = "Hello";
  return msg + ", " + name + "!";
}
```

Clearly, the model is trying to infill `string` but is unable to effectively determine the appropriate length of the type infill. It is important to note that the hyperparameter for max token length must be greater than 1 to accommodate the possibility of larger type-infills, such as `[str, int]`, `LargeCustomTypeLongName` or first-class function types.

As for the prompt-engineering algorithm, it has been successfully implemented using a bottom-up tree traversal algorithm, described in the Report section.

Replan

Plan Moving Forward To address the max token hyperparameter problem, we plan to run a benchmark test for several temperature hyperparameters to find a list of the most optimal temperatures for the general `type infill` task. Then, we will update the InCoder server implementation to return a list of completions that were generated with varying temperatures. In addition, we plan to implement type-inference for Python using the same completion pipeline that is used for TypeScript. Then, we plan to formally benchmark our TypeScript and Python type-inference across a randomly selected 100-file subset of Leetcode solutions.

Timeline

- **12/13:**
  - Python type-inference
  - Benchmarking for accuracy across a 100-file dataset

Appendix

Our system was able to type-infer this program:

```
const findAllPeople = function (n, meetings, firstPerson) {
  meetings.sort((a, b) => a[2] - b[2]);
  const uf = new UnionFind(n);
  uf.connect(0, firstPerson);
  let ppl = [];
  for (let i = 0, len = meetings.length; i < len; ) {
    ppl = [];
    let time = meetings[i][2];
    while (i < len && meetings[i][2] === time) {
      uf.connect(meetings[i][0], meetings[i][1]);
      ppl.push(meetings[i][0]);
      ppl.push(meetings[i][1]);
      i++;
    }
    for (let n of ppl) {
      if (!uf.connected(0, n)) uf.reset(n);
    }
  }
  let ans = [];
  for (let i = 0; i < n; ++i) {
    if (uf.connected(0, i)) ans.push(i);
  }
  return ans;
};

class UnionFind {
  arr;

  constructor(n) {
    this.arr = Array(n).fill(null);
    this.arr.forEach((e, i, arr) => (arr[i] = i));
  }
  connect(a, b) {
    this.arr[this.find(a)] = this.find(this.arr[b]);
  }
  find(a) {
    return this.arr[a] === a ? a : (this.arr[a] = this.find(this.arr[a]));
  }
  connected(a, b) {
    return this.find(a) === this.find(b);
  }
  reset(a) {
    this.arr[a] = a;
  }
}
```

And annotate it with these types:

```
const findAllPeople: (
  n: number,
  meetings: number[][],
  firstPerson: number
) => number[] = function (n, meetings, firstPerson) {
  meetings.sort((a, b) => a[2] - b[2]);
  const uf: UnionFind = new UnionFind(n);
  uf.connect(0, firstPerson);
  let ppl: number[] = [];
  for (let i = 0, len = meetings.length; i < len; ) {
    ppl = [];
    let time: number = meetings[i][2];
    while (i < len && meetings[i][2] === time) {
      uf.connect(meetings[i][0], meetings[i][1]);
      ppl.push(meetings[i][0]);
      ppl.push(meetings[i][1]);
      i++;
    }
    for (let n of ppl) {
      if (!uf.connected(0, n)) uf.reset(n);
    }
  }
  let ans: number[] = [];
  for (let i = 0; i < n; ++i) {
    if (uf.connected(0, i)) ans.push(i);
  }
  return ans;
};

class UnionFind {
  arr: number[];
  constructor(n) {
    this.arr = Array(n).fill(null);
    this.arr.forEach((e, i, arr) => (arr[i] = i));
  }
  connect(a: number, b: number): void {
    this.arr[this.find(a)] = this.find(this.arr[b]);
  }
  find(a: number): number {
    return this.arr[a] === a ? a : (this.arr[a] = this.find(this.arr[a]));
  }
  connected(a: number, b: number): boolean {
    return this.find(a) === this.find(b);
  }
  reset(a: number): void {
    this.arr[a] = a;
  }
}
```

While TypeScript's type inference only managed to infer these types (too many `any`s and loose typing):

```
const findAllPeople = function (n: number, meetings: any[], firstPerson: any) {
  meetings.sort((a: number[], b: number[]) => a[2] - b[2]);
  const uf = new UnionFind(n);
  uf.connect(0, firstPerson);
  let ppl = [];
  for (let i = 0, len = meetings.length; i < len; ) {
    ppl = [];
    let time = meetings[i][2];
    while (i < len && meetings[i][2] === time) {
      uf.connect(meetings[i][0], meetings[i][1]);
      ppl.push(meetings[i][0]);
      ppl.push(meetings[i][1]);
      i++;
    }
    for (let n of ppl) {
      if (!uf.connected(0, n)) uf.reset(n);
    }
  }
  let ans = [];
  for (let i = 0; i < n; ++i) {
    if (uf.connected(0, i)) ans.push(i);
  }
  return ans;
};

class UnionFind {
  arr: any[];

  constructor(n: any) {
    this.arr = Array(n).fill(null);
    this.arr.forEach(
      (e: any, i: string | number, arr: { [x: string]: any }) => (arr[i] = i)
    );
  }
  connect(a: number, b: string | number) {
    this.arr[this.find(a)] = this.find(this.arr[b]);
  }
  find(a: string | number) {
    return this.arr[a] === a ? a : (this.arr[a] = this.find(this.arr[a]));
  }
  connected(a: number, b: number) {
    return this.find(a) === this.find(b);
  }
  reset(a: string | number) {
    this.arr[a] = a;
  }
}
```

Note that TypeScript's built-in inference type-annotated unbound arrow functions, while our system didn't. We believe that these functions should be left untyped, as the signature of the function that calls them should be typed and TypeScript's type-inference should enforce those rules. Our system will not battle with TypeScript's type-inference, it will try to work alongside it. Additionally, our system will not perform any type-migrations; it will not change already defined types. This is to further enforce the coalition between our system and TypeScript's.

Additionally, our system was able to fill out generic types.

```
var sumFourDivisors = function (nums) {
  let res = 0;

  for (const e of nums) {
    const set = helper(e);
    if (set.size === 4) {
      for (const i of set) res += i;
    }
  }

  return res;

  function helper(num) {
    const set = new Set();
    const r = ~~(Math.sqrt(num) + 1);
    for (let i = 1; i < r; i++) {
      if (num % i === 0) {
        set.add(i);
        set.add(num / i);
      }
    }
    return set;
  }
};
```

to

```
var sumFourDivisors: (nums: number[]) => number = function (nums) {
  let res: number = 0;
  for (const e of nums) {
    const set: Set<number> = helper(e);
    if (set.size === 4) {
      for (const i of set) res += i;
    }
  }
  return res;
  function helper(num: number): Set<number> {
    const set: Set<number> = new Set();
    const r: number = ~~(Math.sqrt(num) + 1);
    for (let i = 1; i < r; i++) {
      if (num % i === 0) {
        set.add(i);
        set.add(num / i);
      }
    }
    return set;
  }
};
```

while TypeScript's inference couldn't give us a type-correct answer:

```
7:28 - error TS2365: Operator '+=' cannot be applied to types 'number' and 'unknown'.

7       for (const i of set) res += i;
                             ~~~~~~~~
```